

ABSTRACT

BENAVIDES FORERO, JUAN DAVID. Refinement of State-Based Specifications in Scientific Computing. (Under the direction of John Baugh).

In developing high performance computing and machine learning software, engineers and scientists make use of techniques, such as parallelism and sparse data structures, that are difficult to reason about and debug. Here we explore the role of data refinement, a correct-by-construction approach, in verifying scientific computing applications via bounded model checking. We show how single program, multiple data (SPMD) parallelism can be modeled in Alloy, a declarative specification language, and describe common issues that arise when performing scope-complete refinement checks in this context. We also show how this type of reasoning is naturally extendable to problems in machine learning, specifically neural networks, for which verification is an important concern due to their "black-box" like behavior.

© Copyright 2023 by Juan David Benavides Forero

All Rights Reserved

Refinement of State-Based Specifications in Scientific Computing

by
Juan David Benavides Forero

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Civil Engineering

Raleigh, North Carolina
2023

APPROVED BY:

Kumar Mahinthakumar

Emily Berglund

John Baugh
Chair of Advisory Committee

BIOGRAPHY

Juan David Benavides Forero was born in Bogotá, Colombia. He completed his undergraduate studies at the University of Texas at Austin in 2020, receiving degrees in both Geology and Civil Engineering. Prior to attending graduate school, he worked at a consulting engineering firm in Austin, TX. In 2023 he completed his Master's degree in Civil Engineering at North Carolina State University. During his graduate studies he focused on formal verification, high performance computing, and machine learning, as part of the computing and systems degree concentration.

ACKNOWLEDGEMENTS

Many thanks to my kind and patient advisor Dr. John Baugh and to all my family and friends.

TABLE OF CONTENTS

List of Figures	vi
Chapter 1 Introduction	1
1.1 State-Based Formal Methods	2
1.2 Applications in Scientific Computing	3
1.3 Related Work	5
Chapter 2 Refinement Checking in Alloy	7
2.1 Refinement Checking	8
2.2 Alloy Language and Analyzer	14
2.2.1 Relational Logic	14
2.2.2 Predicates and Assertions	16
2.3 Set Refinement in Alloy	18
2.4 Small Scope Hypothesis	20
Chapter 3 Laplace Solver using Jacobi Iteration	22
3.1 Parallelism in Coarray Fortran	24
3.2 Representing Numerical Computations	25
3.3 Refinement of the Jacobi Computation	27
Chapter 4 Bounded Verification	35
4.1 Total Functions	35
4.1.1 Unbounded Universal Quantifiers	36
4.1.2 Generator Axioms	37
4.1.3 Matrix Generator Axioms	37
4.1.4 Indirection Approach	38
4.2 Bounded Integers	41
Chapter 5 Future Directions	44
5.1 Interleaving Specifications	44
5.2 Other Possible Extensions	49
Chapter 6 Conclusions	50
References	52
APPENDIX	54
Appendix A Alloy Models	55
A.1 Base Model	55
A.2 Adequacy	62
A.3 Functional	62
A.4 Total	62

A.5	Correctness	63
A.6	Progress	63

LIST OF FIGURES

Figure 1.1	Convolutional neural network from Gu et al. (2018)	4
Figure 1.2	Calculating a single value in Jacobi iteration (left), an average pool layer (right)	4
Figure 2.1	Abstraction relation for integer sets from Liskov and Guttag (2000) .	9
Figure 2.2	Data refinement	10
Figure 2.3	Commuting diagram example for integer sets	11
Figure 2.4	Abstraction relations with invariants	12
Figure 2.5	Representation invariant for integer sets	13
Figure 2.6	A dense matrix (a), its representation in the Alloy visualizer (b), and the textual relations of the Alloy instance (c)	16
Figure 2.7	Instance of the show predicate in the Alloy analyzer	19
Figure 2.8	Checking the correctness of the integer set refinement in Alloy	20
Figure 2.9	Coverage of traditional testing (left) and the small scope hypothesis (right)	21
Figure 3.1	Mapping coarray images to a matrix (left) and enforcing invariants (right)	23
Figure 3.2	Overview of a CAF program with images organized in a 1D grid	24
Figure 3.3	Coarray (concrete type) In Alloy	25
Figure 3.4	Average of 4-nearest neighbors calculation (a) and its representation in Alloy (b)	26
Figure 3.5	Inspecting all possible size combinations for alpha	30
Figure 4.1	Relative size difference causing integer overflow	42
Figure 5.1	Transition instance by projecting over state	47

CHAPTER

1

INTRODUCTION

Complex software systems, especially in the context of high-performance computing (HPC) and machine learning (ML), are difficult to reason about and debug, often due to their use of techniques such as parallelism and sparse data structures. Typically, these systems undergo after-the-fact simulation and testing to measure performance and detect bugs. However, for systems that are highly complex or safety critical, more rigorous verification is desirable to find unexpected corner cases that may be missed through simulation. Moreover, after-the-fact testing may reveal conceptual design flaws that would be best addressed in earlier stages of development. This is especially true with the rising complexity of HPC applications and popularity of machine learning models, which have "black-box" like behavior.

We explore the role that formal methods can serve both to perform rigorous, scope-complete verification of programs as well as to aid in the design process, through correct-by-construction approaches. We make use of an abstraction-refinement framework, a stepwise approach for reasoning about software that describes the relationship between abstract and concrete specifications. Often, in HPC, an abstract mathematical specification is continually refined to a final concrete implementation through the stepwise addition of features such as parallelism and sparse data structures. Similarly, neural networks are often developed through incremental trial and error approaches, where different network shapes and hyper-

parameters are refined until an acceptable level of performance is achieved on a test data set. In neither case are the intermediate steps explicitly captured or formally verified, leaving much room for bugs and unanticipated corner cases. By rigorously exploring the refinement process with formal methods tools, we show that lightweight approaches can be practical and helpful in both design and verification, and may result in more robust programs with less time spent fixing bugs.

Scope and Organization. In what follows, we describe a lightweight modeling approach promoted by Jackson and Wing (1996) for reasoning about the structure and behavior of scientific software. The approach is lightweight in the sense that there is *partiality in modeling*—a focused application of the method—and *partiality in analysis*, since the verification being performed is bounded. We propose abstraction and refinement principles to manage sources of complexity, such as those introduced to meet performance goals, including sparse structure and parallelization. Elements of the approach include declarative models that are automatically checked with (Boolean satisfiability) SAT solvers, akin to the analysis approaches used in traditional engineering domains, so manual theorem proving is not required. The approach is bounded and therefore incomplete, but we appeal to the *small scope hypothesis*, which suggests that most real bugs have small counterexamples.

We begin by describing the types of scientific software we focus on, those typically falling in the categories of HPC or ML, and then give an overview of related work pertaining to verification, which this study directly or indirectly builds upon. In Chapter 2, we introduce fundamental concepts of refinement checking and Alloy, the primary language and tool we use. Following this, Chapter 3 describes the verification of a parallel HPC program using our Alloy refinement approach. Chapter 4 looks more deeply at modeling considerations that are unique to bounded model-checking, and Chapter 5 describes possible future extensions to this type of modeling. Finally, we provide conclusions in Chapter 6.

1.1 State-Based Formal Methods

For specifying software systems, de Roever et al. (1998) illustrate two main techniques. The first is an algebraic one, with equations characterizing data types and operations in an abstract way. The algebraic method is implementation independent and at times the more elegant approach. The second is a model oriented or *state-based* approach that characterizes programs through an imperative style using state transformations. This approach

is more scalable and adaptable to complex programs where concurrency is involved. We adopt a state-based approach, as it is more appropriate for the programs and systems we focus on and for the modeling language and tool we use.

1.2 Applications in Scientific Computing

The types of problems we are most interested in are those that fall under the category of scientific computing. This area spans many domains and applications, from meteorological models of the atmosphere to structural analysis software. In practice, this often takes the form of solving large linear algebra systems. As modern computing capabilities have increased, so have the complexity of models and the amount of available data. To keep pace, engineers and scientists adopt practices in High Performance Computing (HPC) such as parallelizing their code to run on advanced computing clusters or by taking advantage of sparsity to reduce computational requirements. Added complexity comes at a cost, with studies showing concerns over reliability and reproducibility of scientific software (Baugh and Dyer 2018).

In addition, with the rise of machine learning (ML) algorithms, there is an increasing amount of work focused on using techniques such as neural networks in scientific computing domains. The introduction of NNs brings about added complexity and makes verification even harder, due to the opaque nature of the models. However, we see many parallels between the types of computation often used in traditional HPC and that in NNs. In both areas, similar linear algebra operations such as matrix-vector and matrix-matrix multiplication are fundamental computations. Likewise, computations in traditional iterative methods such as the Jacobi iteration parallel those of important NN computations like convolutional (Fig. 1.1) and average-pooling (Fig. 1.2) layers.

On top of the complexity that is inherent to HPC domains and techniques, studies have pointed to deep issues with quality and reproducibility of scientific code (Storer 2017), as well as concerns over productivity (Faulk et al. 2009). Some of this is attributed to the context in which scientific code is developed, where domain experts are motivated most by pushing performance and finding novel results quickly. This comes at the cost of neglecting software engineering best practices such as robust testing and developing code that is maintainable, well documented and extendable over a long lifetime.

Although formal methods have received relatively less attention as a response to these concerns, successes in both academic and industry settings have shown they are a promis-

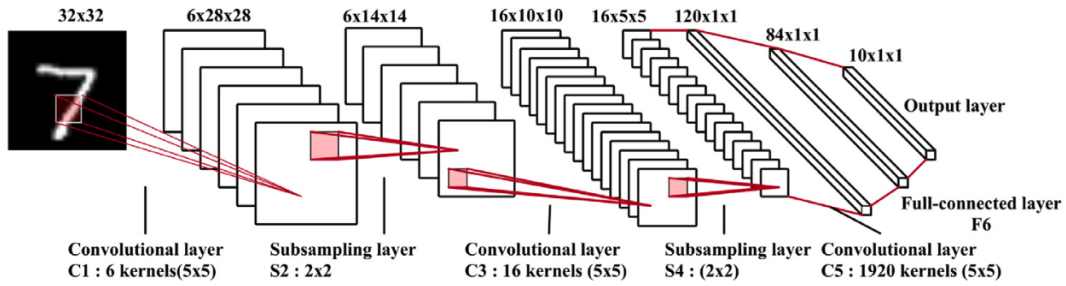


Figure 1.1: Convolutional neural network from Gu et al. (2018)

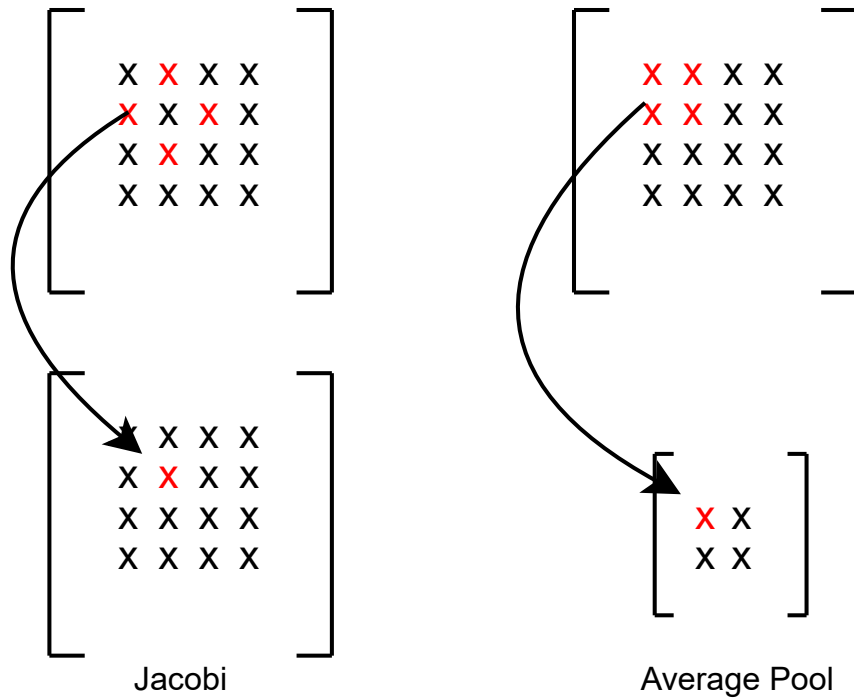


Figure 1.2: Calculating a single value in Jacobi iteration (left), an average pool layer (right)

ing tool for scientific software. Clarke and Wing (1996) describe the positive outlook for formal methods with the advent of specification notations and state of the art model checking tools. A number of notable examples are also showcased, including verification of software protocols and microprocessor architectures. While formal methods are most familiar to computer scientists, Elseaidy et al. (1997) set a precedent for using formal methods tools outside of traditional domains by verifying an active structural control system. In a

more recent survey, Woodcock et al. (2009) describe notable results of industrial projects making use of formal methods. They conclude that a vast majority of respondents reported positive results from using formal methods, and highlight a number of examples from electronic payment systems to verification of railway and train controls.

1.3 Related Work

Formal methods is an extensive field that we do not intend to survey. Instead, we present examples of related work that are most relevant to scientific computing. These studies set a precedent for the framework we present, highlighting refinement, lightweight model-finding, and verification in scientific computing.

Dyer et al. (2019) explore the use of Alloy to model and reason about the structure and behavior of sparse matrices, which are central to scientific computing. Examples of sparse matrix-vector multiplication, transpose, and translation between ELLPACK and compressed sparse row (CSR) formats illustrate the approach. To model matrix computations in a declarative language like Alloy, a new idiom is presented for bounded iteration with incremental updates. The study considers the subset of refinement proof obligations that can be formalized as safety properties—and are thus easier to check—in Alloy.

Baugh and Altuntas (2018) describe a large-scale hurricane storm surge model used in production and verification of an extension using Alloy. To explore implementation choices, abstractions are presented for relevant parts of the model, including the physical representation of land and seafloor surfaces as a finite element mesh, and an algorithm that allows for the propagation of overland flows. Useful conclusions are drawn about implementation choices and guarantees about the extension, in particular that it is equivalence preserving.

Martin (2019) shows how a data refinement approach can be used to formally specify parallel programs using a Coarray Fortran (CAF) implementation of an iterative Jacobi routine. At an abstract level, a mathematical description of a step in the iteration is given, and at the concrete level, the corresponding operation is defined for parallel coarray images; an abstraction function relates the two levels. Since it focuses on specification, the roles of state-space invariants and other refinement proof obligations needed for verification are not addressed.

Singh et al. (2020) describe a new approach for synthesizing correct-by-construction neural networks using formal specifications written in Alloy. Singh et al. point out that that the reliability and safety of systems powered by neural networks is an open problem.

They propose using relational specifications and an incremental synthesis algorithm for generating binary classifiers that are guaranteed to be correct. The algorithm is powered by SAT solvers and resolves both a network shape (number of layers and neurons) and parameters (weights and biases) that are guaranteed to produce correct predictions for the entire input space. Although the approach is limited to small network sizes, the study sets precedent for using Alloy and correct-by-construction approaches in the context of neural networks.

Katz et al. (2019) and Katz et al. (2017) describe using optimization and SMT-based techniques to verify properties of neural networks by posing them as a constraint-satisfaction problems. For networks using affine transformations and ReLU activations (Rectified Linear Unit), a modified simplex algorithm, ReLuplex, is used to provide bounds on the network output. The authors test their approach on a number of networks used in a real aircraft collision avoidance system.

Siegel et al. (2008) describes using the model checking tool Spin to verify parallel numerical programs using symbolic execution. Although the refinement framework is not specifically used, a sequential version of a program is used as the specification against which to compare the parallel one, which mirrors the perspective taken in this thesis.

Note on published work. Some of the content in this thesis is taken directly from work that was published in the conference proceedings of the *35th International Workshop on Languages and Compilers for Parallel Computing* (Benavides et al. 2023).

CHAPTER

2

REFINEMENT CHECKING IN ALLOY

Matrix operations, especially those involving sparsity or parallelism are central to both HPC and neural network applications. In HPC, tools such as MPI or OpenMP are often used to solve large sparse systems on traditional computing clusters. In machine learning, tools such as CUDA are used to process highly parallel operations on graphical processing units (GPUs) both for training and inference. Sparsity in neural networks is common and introduced in various ways, such as pruning weak connections between layers after training.

In order to work toward these themes, much of the work presented here builds on previous work by Dyer et al. (2019), which formalizes an Alloy approach for verifying sparse matrix operations. Their work follows an abstraction-refinement approach to prove correctness properties about sparse matrix formats, such as Compressed Sparse Row (CSR), when primary operations such as matrix transpose and matrix-vector multiplication are applied.

The work in Dyer et al. (2019) takes the view that dense matrix operations can be thought of as the abstract level, which can be related to sparse operations at the concrete level, through abstraction predicates written in Alloy. Safety properties on these predicates are then verified through scope-complete checks, to confirm that the concrete operations are correct refinements of the abstract ones. We use the Alloy structure from Dyer et al. (2019)

for representing matrices and extend the work in the direction of parallel HPC programs. We take the view, as Martin (2019) does, that sequential specifications are the abstract level, that get refined to a parallel implementation. The refinement framework is then used to systematically verify that the parallel concrete implementation does not introduce unwanted behaviors that are not part of the abstract specification.

Both the modeling and bounded checking are done in Alloy, which uses a simple declarative modeling language that allows building up expressions using a relational logic. Unlike imperative programming languages that engineers are most accustomed to using, there is no notion of control flow or mutation, and there is limited support for numerical concerns outside of bounded integer arithmetic. Therefore, the focus is on proving correctness of structure and abstract behavior, rather than concerns that would typically fall under numerical analysis.

2.1 Refinement Checking

We make use of a refinement framework to focus on "design-thinking" and to encourage the writing of programs that are correct by construction:

"Designing a large and complex program usually involves application of some refinement method providing a way to gradually transform an abstract program, possibly a specification, into a concrete implementation. The main principle of such a method is that if the initial abstract program is correct and the transformation steps preserve correctness, then the resulting implementation will be correct by construction" (de Roever et al. 1998).

The notion of refinement is relative. It makes use of an upper abstract level and a lower level concrete one. The levels are related to each other through an abstraction relation (α) that describes how the concrete state space corresponds to the abstract one. The abstract level may be an initial mathematical specification or an intermediate code implementation that is known to be correct. Likewise, the concrete level may correspond to a final complex implementation (with parallelism, sparsity, etc.) or simply a small intermediate step away from the abstract. Whatever levels are being compared, ultimately the goal is to ensure correctness of the concrete operation:

Definition: Semantic Implementation Correctness de Roever et al. (1998). Given two programs, one called concrete and the other called abstract, the concrete

program *implements* (or *refines*) the abstract program correctly whenever the use of the concrete program does not lead to an observation which is not also an observation of the abstract program.

For example, we can consider the implementation of an integer set as in Fig. 2.1. At the abstract level we have the mathematical description of a set; a collection of distinct integer objects. At the concrete level, we have a code-level implementation of a set datatype: an integer array. Consequently, the abstraction relation (α) maps concrete arrays to abstract sets.

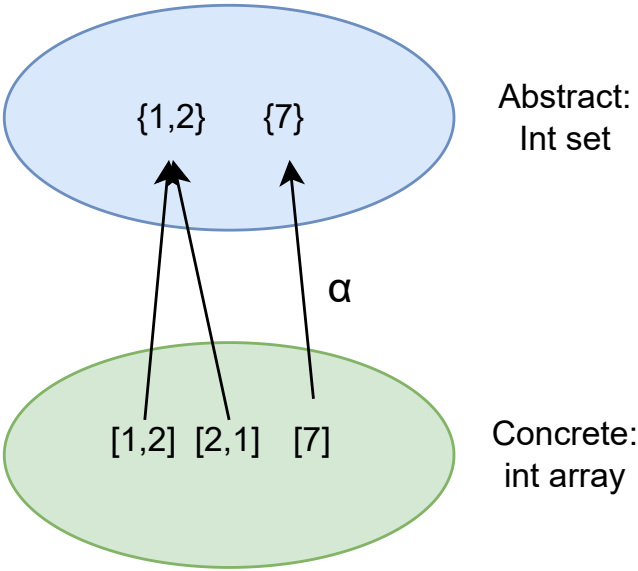


Figure 2.1: Abstraction relation for integer sets from Liskov and Guttag (2000)

We seek to show that this concrete representation, and operations defined on it (such as appending an item to the array), are correct refinements of the abstract specification and the corresponding abstract operations (such as adding an item to the set). In this case α is a many-to-one relation, as different concrete arrays can represent the same set. We show later why it is important to understand the structure of the abstraction relation and characteristics such as whether it is many-to-one, functional, total, etc.

To ensure that a refinement step such as this one preserves correctness in a formal,

machine-checkable manner, proof obligations must be met and discharged; their articulation and promotion begins with the work of Hoare (1972) and thereafter proceeds along both relational and predicate transformer lines; de Roever et al. (1998) summarizes and contrasts a variety of modern approaches.

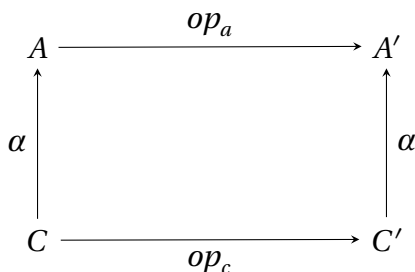


Figure 2.2: Data refinement

Refinement as inclusion, above, is a global criterion. To be made practical, a local criterion with a finite number of verification conditions can be obtained by defining a *simulation* in terms of abstraction relations and commutativity diagrams.

In Fig. 2.2, the commutativity diagram shows the concrete (C) and abstract (A) states related by an abstraction relation α , together with concrete and abstract operations, op_c and op_a , respectively, that define transitions from the non-primed to primed states. Fig. 2.3 shows the same commuting diagram applied to the integer set example, where the abstract operation (op_a) adds an item to the set, and is refined by the concrete operation (op_c), which appends an item to the array.

There are four different technical notions of simulation that correspond to ways in which commutativity can be defined in terms of the diagram (de Roever et al. 1998). When α is both total and functional, the four types of simulation coincide and some of the proof obligations simplify, including the condition for *correctness* of the concrete operation op_c :

$$\forall a, a': A, c, c': C \mid \alpha(c, a) \wedge op_c(c, c') \wedge \alpha(c', a') \Rightarrow op_a(a, a') \quad (2.1)$$

That is, starting from a concrete state in which the corresponding abstract precondition holds, the final concrete state must represent a possible abstract final state. Such a criterion implies inclusion, i.e., that programs using some concrete data type C have (only) observable behaviors of programs using a corresponding abstract data type A . Diagrams

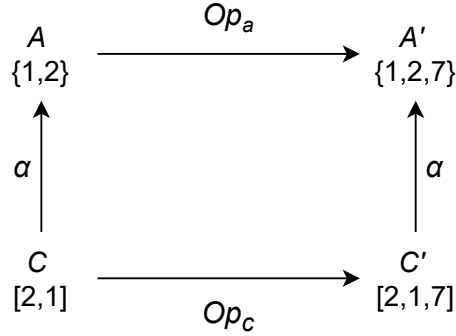


Figure 2.3: Commuting diagram example for integer sets

satisfying such properties are said to commute *weakly*, whereas strong commutativity would be expressed with material equivalence instead of implication in Eq. 2.1.

Summarizing the set of proof obligations for data refinement in a state-based formalism (de Roever et al. 1998), we have the following conditions:

1. *Adequacy* – every abstract state must have a concrete counterpart.
2. *Correspondence of initial states* – every concrete initial state must represent an abstract initial state.
3. *Applicability of the concrete operation* – the precondition for the concrete operation should hold for any concrete state whose corresponding abstract state satisfies the abstract precondition.
4. *Correctness of the concrete operation* – as we describe above.

Not every condition applies in every situation, and in some cases a condition may require a special interpretation for the given context. For instance, an adequacy check for a refinement from an abstract sequential program may be satisfied trivially in the one-processor case, but one might rather show adequacy for an n -processor case.

To draw sound conclusions from these, the structure of α is clearly important. If the correctness condition of Eq. 2.1 is to apply, for instance, it must be shown to be both functional (Eq. 2.2) and total (Eq. 2.3):

$$\forall a_1, a_2: A, c: C \mid \alpha(c, a_1) \wedge \alpha(c, a_2) \Rightarrow \text{equal}(a_1, a_2) \quad (2.2)$$

$$\forall c : C \mid \exists a : A \mid \alpha(c, a) \quad (2.3)$$

Often, representation invariants (Inv) must be defined to describe what constitutes valid concrete and/or abstract states. As in Fig. 2.4 this helps to constrain the abstraction relation to be functional and total, by restricting it to only those regions where its behavior is defined.

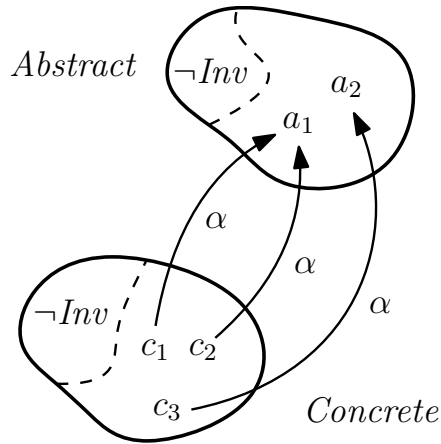


Figure 2.4: Abstraction relations with invariants

In the integer set example, abstract sets contain only unique values, so a representation invariant on concrete arrays limits them to having no duplicate values (as in Fig. 2.5). Such invariants are implemented at the code level and should be maintained by any concrete operation. Often, for more complicated programs, the model-finding nature of Alloy helps to us determine what proper invariants should be, which can be done by examining counterexamples produced by the tool when checking correctness.

For integer sets, it is straightforward to confirm that the abstraction relation is total and functional. However, for more complex implementations this is not always the case. What would it mean to check these in Alloy? The three equations above are all expressions of first order logic, and yet they present different levels of difficulty to the Alloy Analyzer, the model-finding tool supporting the formalism. Eq. 2.3 in particular, which checks whether or not a relation is total, is problematic because its SAT encoding results in an unbounded universal quantifier (Jackson 2012).

Similar checks are required if we have concerns, as we should, about *progress* properties:

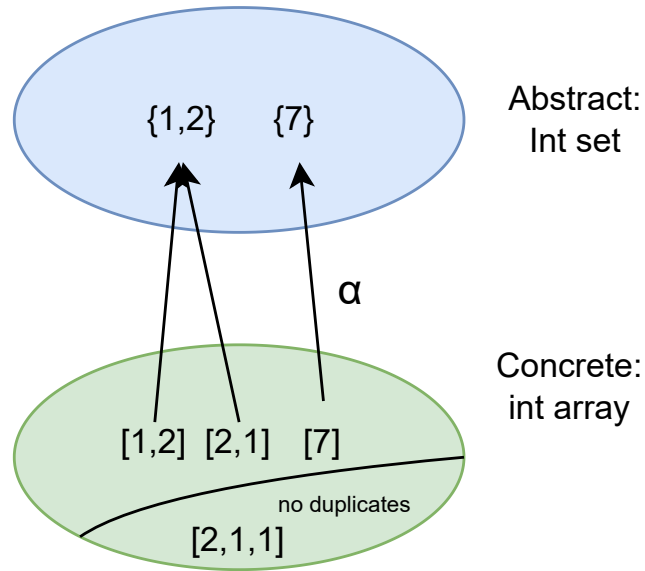


Figure 2.5: Representation invariant for integer sets

a concrete operation op_c can “do nothing” and satisfy the correctness check vacuously, e.g., when the term $op_c(c, c')$ is false in Eq. 2.1, as it might be due to an inadvertently buggy specification. So we add to the set of proof obligations a progress check:

5. *Progress of the concrete operation* – with respect to the operation, every initial state satisfying the concrete precondition must have a corresponding final state.

As with Eq. 2.3, which requires that α be total, this kind of check introduces an unbounded quantifier in its formulation, and is again problematic for Alloy.

All this points to a limitation of finite instance finding. Various approaches have been devised to try and circumvent it, including the definition of generator axioms (Jackson 2012), though they are sometimes difficult to come by or too computationally expensive to employ, as we later show. We later describe a new, simpler approach for performing these and other checks in Alloy and do so in the context of HPC.

2.2 Alloy Language and Analyzer

2.2.1 Relational Logic

In the Alloy language, all models are made up of *atoms* and *relations*. Atoms are a primitive type and are indivisible, uninterpreted, and immutable. Relations are sets of tuples that denote how different atoms are related to each other. The main language construct, *signatures*, introduce both a type and a set of atoms, and may introduce *fields* that define relations over them. Because reals or floating-point values are not supported, we model matrix elements as a basic signature:

```
sig Value {}
run {} for 4 Value
```

This fragment constitutes a valid Alloy model. Running it will produce all possible instances within the model scope: in this case the scope is up to four value atoms. An instance will be represented by an atom from the set $\{Value0, Value1, Value2, Value3\}$.

Matrices are composed of indexed values, so for modeling matrices, a signature is declared with fields representing this relation. Matrix size is specified with two *int* fields, for number of rows and columns. In this case, *rows* then represents a two-way mapping, represented by the tuple $m \rightarrow n$ when matrix m is mapped to integer n . Likewise, *vals* represents a four way mapping, with the tuple $m \rightarrow i \rightarrow j \rightarrow v$ representing a matrix m mapped to row index i , column index j and value v . The *lone* keyword in Alloy is used to indicate the multiplicity of the relation: at most one value will exist for any i, j pair.

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

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

CSR, a sparse matrix format, is thought of as one of the concrete types that refines the dense matrix object. A CSR matrix stores only non-zero values in a 1-d array, along with two indexing arrays to capture row and column extents. In this case, the *A* field represents

the three way mapping of the tuple $c \rightarrow n \rightarrow v$ from CSR matrix c , to integer n and value v , while the IA and JA relations are represented by tuple $c \rightarrow i \rightarrow j$ from matrix c , to integers i and j .

The Alloy analyzer can be used to explore instances that satisfy model constraints. For example, the Alloy fragment below (*showDense*) can be run to generate all possible instances where there is at least one 2×2 matrix. We use the dot notation to enforce the matrix size through the row and column fields, so the expression *m.rows* represents the mapping from matrix to integer for matrix *m*. We also specify that this matrix must satisfy some predicate *Inv*, an invariant which is described later. Any run or check of an Alloy model is scoped. In this example, at most one dense matrix and two values are allowed in any generated instance. Fig 2.6 is one possible instance that Alloy produces.

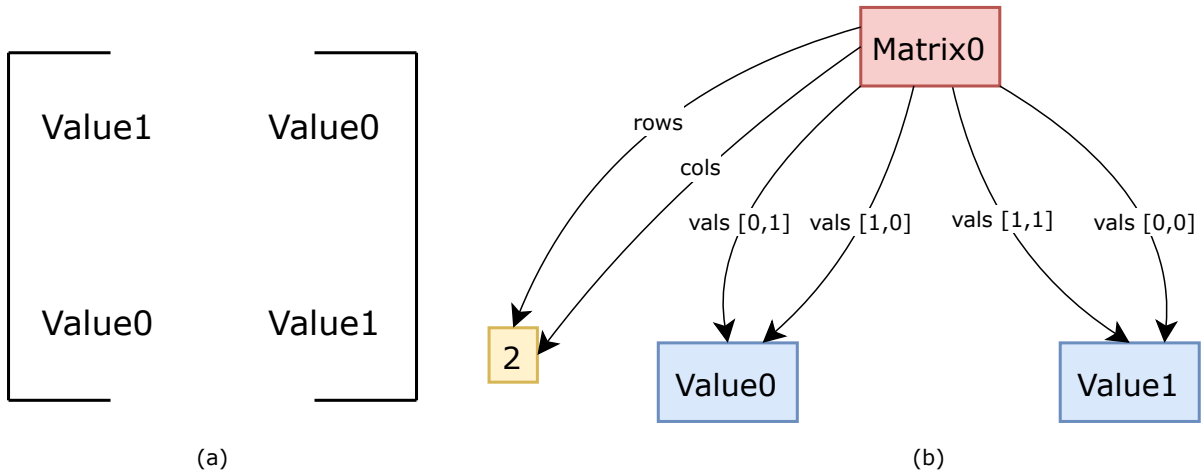
```
run showDense {
  some m: Matrix {
    m.rows = 2
    m.cols = 2
    Inv[m]
  }
} for 1 Matrix, 2 Value
```

Instances can be inspected both through a visual interface, as in Fig 2.6 (b), as well as through table, tree, and textual forms. Figure 2.6 (c) shows the same instance through the textual format, clearly showing what relations make up the instance.

As stated, Alloy relations relate atoms and are sets of tuples: sequences of atoms. They can also be thought of as a table, where every entry is an atom. The number of rows in the table is the *size* of the relation, while the number of columns is its *arity* (Jackson 2002). The Alloy evaluator can be used to inspect elements of traces. For the instance shown in Fig. 2.6, typing "vals" into the evaluator will produce the following table, with size four and arity four.

Matrix0	0	0	Value1
Matrix0	0	1	Value0
Matrix0	1	0	Value0
Matrix0	0	1	Value1

When inspecting textual instances in Alloy, such as Fig. 2.6 (c) these tuples will be represented using the arrow product syntax (\rightarrow). The domain restriction operator ($\langle :$) is used to identify which named relation the tuples belong to (in the case of the table above, it belongs the *Matrix-vals* relation).



```

this/Value = {Value$0, Value$1}
this/Matrix = {Matrix$0}
this/Matrix<:rows = {Matrix$0->2}
this/Matrix<:cols = {Matrix$0->2}
this/Matrix<:vals = {
Matrix$0->0->0->Value$1,
Matrix$0->0->1->Value$0,
Matrix$0->1->0->Value$0,
Matrix$0->1->1->Value$1}

```

(c)

Figure 2.6: A dense matrix (a), its representation in the Alloy visualizer (b), and the textual relations of the Alloy instance (c)

2.2.2 Predicates and Assertions

Predicates are used to evaluate conditions between Alloy sets and signatures. For instance, the matrix invariant predicate below is used to specify conditions that must be met for a Matrix signature to be considered a valid representation of a dense matrix (such as that the rows and columns must be non-negative). Predicates can be thought of as Boolean functions, which evaluate to true or false, but do not alter the state of any objects passed as parameters, as atoms are immutable.

```

pred Inv [m: Matrix] {
  m.rows ≥ 0
}

```

```

    m.cols ≥ 0
    m.vals.univ = range[m.rows] → range[m.cols]
}

```

Central to the approach are abstraction predicates (such as alpha below) that relate concrete and abstract state spaces. In this case, alpha establishes the relation between CSR representations (concrete) and dense matrices (abstract):

```

pred alpha [c: CSR, m: Matrix] {
    m.rows = c.rows
    m.cols = c.cols
    m.vals = {
        i: range[c.rows], j: range[c.cols], v: Value |
        let k = { k: range[c.IA[i], c.IA[add[i, 1]]] | c.JA[k] = j } |
        one k ⇒ v = c.A[k] else v = Zero
    }
}

```

Again, we note that predicates such as alpha are not operational as in imperative programming languages. It is typical to see abstraction relations represented as transformations from the concrete state space to the abstract state space (as in Fig. 2.4), but in declarative models, there is no notion of *pre* and *post* states. Predicates like alpha simply compare two distinct objects and check whether the predicate holds.

To verify properties of signatures and predicates, Alloy *assertions* can be used to carry out bounded model checking. When checking an assertion, Alloy will try to find a counterexample that violates the specification, within the given scope. For example, we might be interested in confirming that any valid CSR matrix is only related through alpha to valid dense matrices. In the assertion below, Alloy will search the state space for an instance where a valid CSR matrix is related through alpha to a dense matrix whose invariant conditions are not satisfied. If no counterexamples are found, the predicate is valid in the specified scope:

```

assert alphaValid {
    all c: CSR, m: Matrix |
        Inv[c] and alpha[c, m] ⇒ Inv[m]
}

```

Dyer et al. (2019) define several matrix representations in Alloy and use the assertion mechanism to verify that several concrete sparse operations are correct refinements of their dense counterparts. In subsequent chapters we describe how we build on this framework to verify parallel programs, with an emphasis on an iterative Jacobi routine described by Martin (2019).

2.3 Set Refinement in Alloy

For the integer set refinement example, we show a complete Alloy model below to illustrate the approach, in this case using abstract *items* instead of integers. The abstract type is modeled as a signature *A* with a single field relation to a set *Items*. The concrete type is modeled as a signature *C* with a single field relation to an Alloy sequence, which is equivalent to a 1-d array of *Items*. The concrete invariant, *Inv*, enforces the requirement that no duplication within the concrete type is allowed. An abstraction predicate, *alpha*, maps concrete to abstract states through the Alloy range function (*ran*). Two operations are then defined, an abstract one (*addA*) and a concrete one (*addC*).

```
sig Item {}

sig A {
  s: set Item
}

sig C {
  s: seq Item
}

fact noDups { all c: C | Inv[c] }

pred Inv [c:C] {
  not c.hasDups
}

pred alpha [c: C, a: A] {
  ran[c.s] = a.s
}

pred addC [c, c": C, i: Item] {
  c".s = c.s.add[i]
}

pred addA [a, a": A, i: Item] {
  a".s = a.s ++ i
}
```

Often, before checking correctness, a predicate such as *show*, below, can be run to visually inspect instances of the model in the Alloy analyzer. In this case, *show* specifies that a model should contain some related *a* and *c* initial states as well as *a'* and *c'* post

states, where an item is being added to each through their respective operations (*AddA* and *AddC*). The run is also scoped so that, in this case, instances have at most two *A* objects, two *C* objects, and two *Items*. Fig. 2.7 shows an instance of this run in the Alloy Analyzer.

```

run show {
  some a, a": A, c, c": C, disj i, i": Item |
    i in a.s and i" not in a.s and
      alpha[c, a] and alpha[c", a"] and addA[a, a", i]
}
for 2 A, 2 C, 2 Item

```

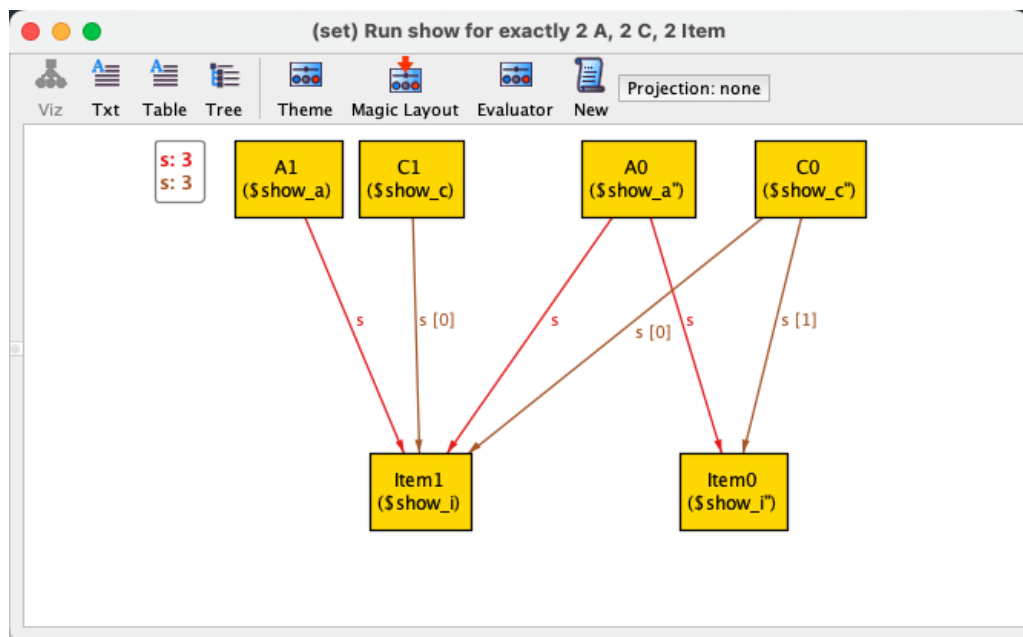


Figure 2.7: Instance of the show predicate in the Alloy analyzer

To verify that the concrete specification is a correct refinement of the abstract one, we use the correctness check from Eq. 2.1:

```

check correct {
  all a, a": A, c, c": C, i: Item |
    (alpha[c, a] and addC[c, c", i] and alpha[c", a"])
      => addA[a, a", i]
}

```

Fig 2.8 shows the result of running this predicate in Alloy. With no counterexamples pro-

```
Executing "Check correct"  
Solver=minisat(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch  
1073 vars. 69 primary vars. 2338 clauses. 176ms.  
No counterexample found. Assertion may be valid. 56ms.
```

Figure 2.8: Checking the correctness of the integer set refinement in Alloy

duced we conclude the safety check has passed. Other proof obligations of refinement, such as checking whether alpha is total, are discussed later.

2.4 Small Scope Hypothesis

The expressiveness and automatic model-finding capabilities of Alloy come at the cost of *bounding* any verification so that it is complete up to a given scope. Since problems tend to surface in very small contexts, due to Alloy’s exhaustive analysis, this does not appear to be a practical limitation in the studies we performed. This finding is consistent with the *small scope hypothesis* (Jackson 2012), which argues that most bugs can be found by exhaustively testing a system within some small scope. Rather than manually or randomly choosing some number of tests within a larger scope, a more effective strategy for finding corner cases and having strong safety guarantees is to test all possible inputs within a (reasonably) small scope. This approach lends itself well to working with tools like Alloy that are powered by SAT solvers that excel at exhaustively searching a problem space, as long as the scope is small enough to be computationally feasible.

Using this kind of reasoning in Alloy has been successful in past work by Dyer et al. (2019). By keeping model scopes small, Alloy is able to produce counterexamples instantaneously or in reasonable amounts of time so that insights about program behavior can be quickly gathered. Because of Alloy’s model-finding capabilities, only program fragments of interest need be specified, allowing spot-checks to be isolated and performed on them instead of on large-scale systems in their entirety through conventional means of testing. The approach has led to discovering documentation errors, and misstated properties that do not align with the actual software.

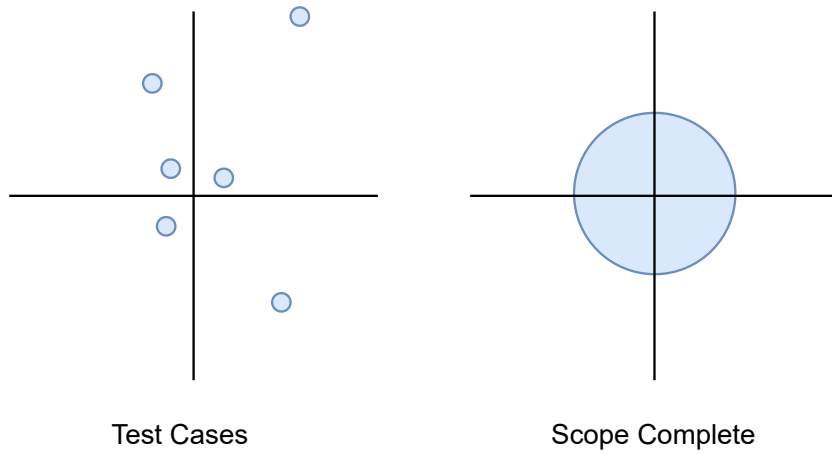


Figure 2.9: Coverage of traditional testing (left) and the small scope hypothesis (right)

CHAPTER

3

LAPLACE SOLVER USING JACOBI ITERATION

Data refinement is a correct-by-construction approach for the stepwise development of programs and models from a higher-level abstract specification to lower-level concrete ones (de Roever et al. 1998). The HPC field lends itself well to a data refinement approach as most programs begin with a mathematical specification — often as a theory report — that serves as a guide, to one extent or another, in the implementation of high performance code.

To extend the work of Dyer et al. (2019) in a way that builds on verifying HPC and NN applications, we consider a parallel matrix computation described by Martin (2019). Martin introduces a parallel Jacobi iteration routine written in Coarray Fortran (CAF) and sketches a specification approach within a refinement framework. We propose modeling approaches in Alloy for capturing the specification in a way that is machine-checkable in Alloy.

Iterative Jacobi computation. We consider the numerical solution to Laplace’s equation over a rectangular domain, with fixed values on the boundary, using the technique of Jacobi

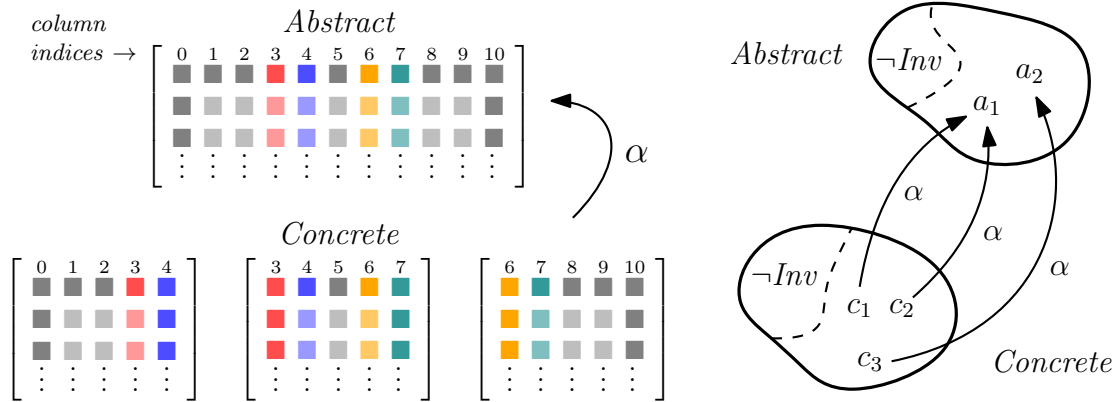


Figure 3.1: Mapping coarray images to a matrix (left) and enforcing invariants (right)

iteration. The example is implemented in Coarray Fortran (CAF), a single program, multiple data (SPMD) extension to the language. In CAF, designated variables are extended with a parallel dimension, so that each is shared across copies of the same program (images) using the Partitioned Global Address Space (PGAS) model.

At each iteration, the algorithm averages the four nearest neighbors of all interior elements of a matrix. Because it updates or “displaces” all of the elements at the same time, the Jacobi method is sometimes called the *method of simultaneous displacements*, which contrasts with the Gauss–Seidel method, whose elements are successively “updated in place.” As a result, extra storage is needed in the Jacobi method to take a step, but, afterward, the previous step’s storage can be reused if we swap matrix storage locations at each iteration.

To specify the parallel program, Martin takes a refinement perspective, defining a step in a sequential Jacobi iteration as the abstract level, and a step in a parallel CAF implementation as the concrete level. The abstraction relation maps coarrays at the concrete level (a sequence of image matrices) to a single abstract matrix. Duplicating columns at the image interfaces allows computation and then communication to proceed in separate “stages” in the CAF program.

Fig. 3.1 shows the column mapping between coarray images and the abstract matrix (left) and the role of invariants (right) which we use to tighten the abstraction relation α so that it is total. In the abstract space, the invariant ensures basic matrix index and bounds checking, while the concrete invariant ensures behavior specific to coarray matrices.

3.1 Parallelism in Coarray Fortran

We use Alloy to model CAF programs, which employ single program multiple data parallelism (SPMD). In CAF, variables are extended with a coarray dimension and replicated across copies of the same program (images). For instance, the fragment below declares a real 3×3 array, U , that exists on each image. $U(i, j)$ then refers to element (i, j) on the local U array, while $U(i, j)[1]$ refers to element (i, j) on the U array of image 1. The memory model allows any image to access data on any other. CAF provides constructs for synchronizing processes, such as the *sync all* statement in the fragment below that enforces an explicit synchronization barrier, so all images must reach this point before advancing further in the program.

```
REAL :: U(3,3)[*]  
IF (this_image() .EQ. 1) THEN  
    U(1,1) = 1  
END IF  
SYNC ALL  
U(1,1) = U(1,1)[1]  
END
```

It is typical to see images indexed on a 1D grid, as in Fig. 3.2, however other configurations are possible.

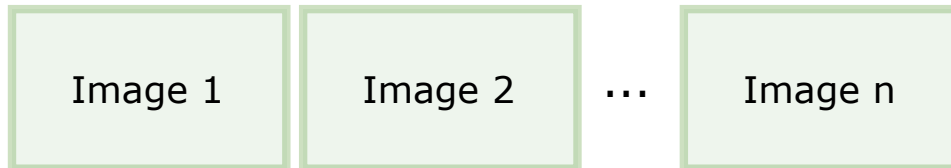


Figure 3.2: Overview of a CAF program with images organized in a 1D grid

For programs dealing with CAF matrices organized in a 1D grid, we formalize a coarray in Alloy as a *sequence* of matrices. The *seq* expression here is equivalent to an $Int \rightarrow Matrix$ relation, but includes predefined Alloy constructs for dealing with sequences, such as functions to return all elements or indices of a sequence. Fig 3.3 shows an Alloy instance of a Coarray with three images.

```
sig Coarray {  
    mseq: seq Matrix
```

}

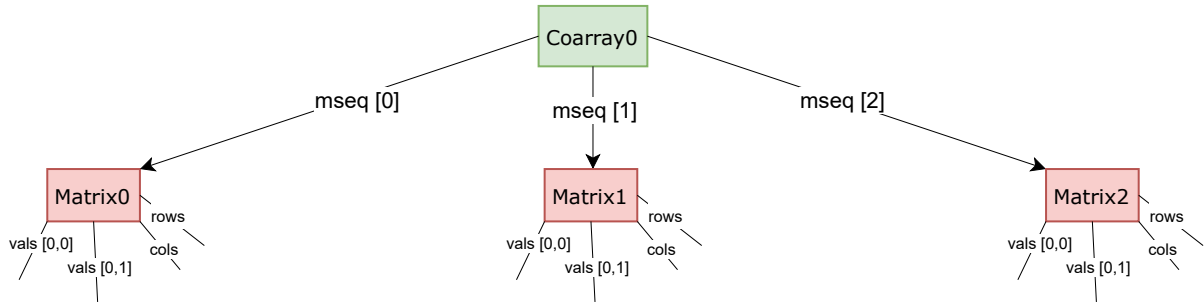


Figure 3.3: Coarray (concrete type) In Alloy

In HPC and NN, it is common for large sequential matrix operations to be decomposed into smaller parallel operations. Thus, in the refinement framework, a dense matrix continues to be the abstract type, which is refined to a sequence of matrices as the concrete type. Likewise, the sequential implementation is thought of as the abstract operation, and the parallel implementation as the concrete one.

3.2 Representing Numerical Computations

Numerical computations are central to both HPC and NN programs. For example, in a Jacobi iteration (Fig. 3.4a), each interior matrix element is replaced by the average of its four nearest neighbors:

```
DO J = 1, JMAX
  DO I = 1, IMAX
    V(I, J) = 0.25 * (U(I-1, J) + U(I+1, J)
      + U(I, J-1) + U(I, J+1))
  END DO
END DO
```

This type of operation is common in HPC applications and is similar to the type of computations used in NN convolutional layers, where a filter with learned parameters is convolved with patches of an input matrix to produce a smaller output matrix, or in pooling layers, where the average or max value of an input patch is calculated to perform a down-sampling.

However, because Alloy does not support real arithmetic, numerical computations like this are not represented directly. For instance, to model the Jacobi iteration presented above, we introduce a signature, *Neighbors*, with four relations mapping to the Values that make up the average. A Neighbors signature is itself also a value (as it *extends* Value) and could be mapped to by another Neighbors signature, in subsequent iterations of a Jacobi routine. Fig. 3.4 illustrates how the neighbors signature abstractly represents the average of four values.

```
sig Neighbors extends Value {
  up, down, left, right: Value
}
```

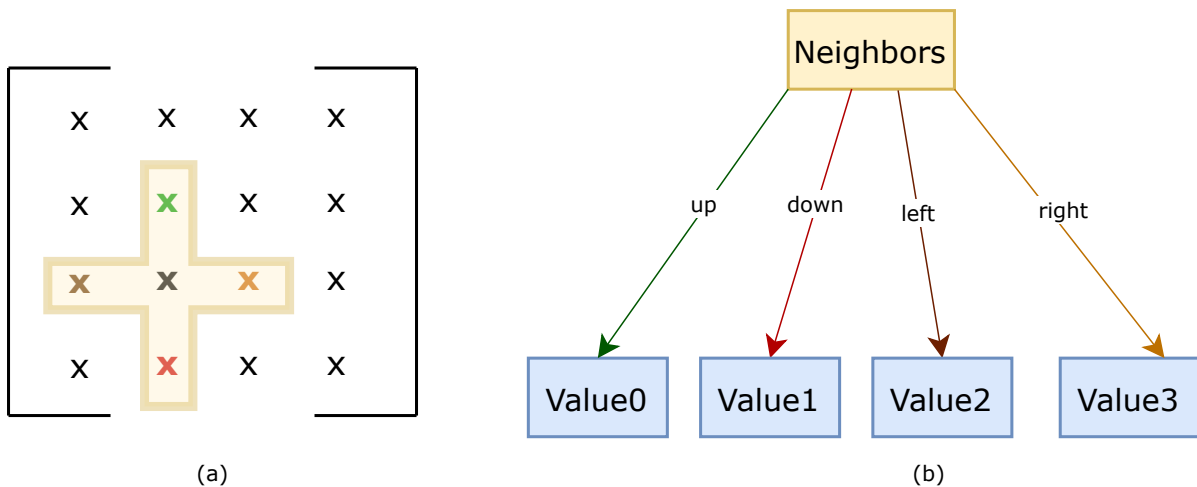


Figure 3.4: Average of 4-nearest neighbors calculation (a) and its representation in Alloy (b)

Alloy functions can be used to return sets of signatures. In this case, given a Matrix and an (i, j) index pair as parameters, the neighbors function returns the corresponding Neighbors signature representing the average value:

```
fun neighbors[U: Matrix, i: Int, j: Int]: one Neighbors {
  { n: Neighbors |
    n.up = U.vals[plus[i, 1], j] and
    n.down = U.vals[minus[i, 1], j] and
    n.left = U.vals[i, minus[j, 1]] and
```

```

    n.right = U.vals[i, plus[j, 1]]
  }
}

```

The JacobiStep predicate below is then the Alloy representation of the nested Fortran loop above (one step in a Jacobi iteration):

```

pred JacobiStep [U, V: Matrix] {
  sameShape[U, V]
  V.vals.univ = range[V.rows] → range[V.cols]
  V.vals =
    { i: range[U.rows], j: range[U.cols], x: Value |
      let boundary = (j = 0 or j = minus[U.cols, 1] or
                    i = 0 or i = minus[U.rows, 1]) |
        x = (boundary ⇒ U.vals[i, j] else neighbors[U, i, j]) }
}

```

This JacobiStep predicate relates two dense matrices; in the refinement framework, it corresponds to the abstract operation.

3.3 Refinement of the Jacobi Computation

Here we describe the main components of the Alloy model for the parallel Jacobi routine as well as the full set of correctness checks that were performed to verify the refinement.

Concrete type. To model coarrays we use the inbuilt Alloy sequence to represent the 1D indexing of images used in the Jacobi program. The concrete invariant (*Inv*) enforces interface conditions between neighboring coarrays, i.e., the duplication of columns necessary for halo or border exchanges, and it ensures that the matrices corresponding to a given coarray variable all have the same dimensions in each image, as dictated by CAF semantics.

```

sig Coarray {
  mseq: seq Matrix
}

pred Inv [c: Coarray] {
  all m1, m2: c.mseq.elems | sameShape[m1, m2]
  all i: allRows[c], q, p: c.mseq.indxs |
    let Lc = lastCol[c] {
      q = minus[p, 1] ⇒
        c.mseq[q].vals[i, Lc] = c.mseq[p].vals[i, 1]
      q = plus[p, 1] ⇒

```

```

        c.mseq[q].vals[i, 0] = c.mseq[p].vals[i, minus[Lc, 1]]
    }
}

```

With this structure, element (i, j) in the matrix of image q , of coarray c , is referred to by:

```
c.mseq[q].vals[i, j]
```

Abstraction relation. The *alpha* predicate relates concrete coarrays and abstract matrices by mapping values from the larger input matrix to each image, including the necessary overlaps. Helper predicates are used to neatly abstract other necessary numeric computations. For example, *totCols* enforces the size relation between the number of columns in an abstract matrix and each coarray image.

```

sig Coarray {
  mseq: seq Matrix
}

-- abstraction relation (alpha)
pred alpha [c: Coarray, m: Matrix] {
  totRows[c, m.rows]
  totCols[c, m.cols]
  all i: range[m.rows], j: range[m.cols] {
    -- 1st column of m is the 1st column of the 1st image of c
    j = 0 ⇒ m.vals[i, j] = c.mseq[0].vals[i, 0]

    -- last column of m is the last column of the last image of c
    j = lastCol[m] ⇒
      let mi = sub[#c.mseq, 1],          -- matrix index
          ci = lastCol[c.mseq[mi]] | -- column index
          m.vals[i, j] = c.mseq[mi].vals[i, ci]

    -- mapping of middle image columns
    j != 0 and j != lastCol[m] ⇒
      let mi = div[sub[j, 1], sub[c.mseq[0].cols, 2]],
          ci = add[1, rem[sub[j, 1], sub[c.mseq[0].cols, 2]]] |
          m.vals[i, j] = c.mseq[mi].vals[i, ci] }
}

```

Within the predicate, the various arithmetic operations imply relative size and mapping rules, so care must be taken to ensure bounded integer operations are still valid. For example, the size relation between abstract matrix and concrete coarray columns is as follows:

$$(c - 2) = (m - 2) / i$$

where c is the number of columns in each coarray matrix, m is the number of columns in the larger abstract matrix, and i is the number of images.

Although the equation can be used directly in Alloy, since integer arithmetic is supported, we rearrange it as follows to reduce the potential for integer overflow:

$$m = (c - 2)i + 2$$

The variables m , c , and i are then used in an Alloy predicate *rel* that relates them; it includes special cases for abstract matrices with fewer than four columns.

```

pred totRows [c: Coarray, r: Int] {
  let i = #c.mseq |
    r = (i > 0 ⇒ c.mseq[0].rows else 0)
}

pred totCols [c: Coarray, m: Int] {
  let i = #c.mseq |
    rel[m, i > 0 ⇒ c.mseq[0].cols else 0, i]
}

pred rel [m, c, i: Int] {
  m ≥ 0 and c ≥ 0 and i ≥ 0
  m = 0 ⇒ c = 0 and i = 0
  m > 0 and m < 4 ⇒ c = m and i = 1
  m ≥ 4 ⇒ m = add[mul[i, sub[c, 2]], 2]
}

```

To explore alpha, we can use the model-finding abilities of Alloy to verify which abstract and concrete size combinations work to satisfy the abstraction relation for any number of images. We strip the problem down the arithmetic operations and introduce a signature *Eqn*, with fields representing the number of columns in the abstract matrix (nm), number of columns in each concrete matrix (nc), and the number of images (ni). A signature fact ensures that the three fields satisfy *rel*. The show predicate is then run to find every combination (up to the given integer scope) that constitutes a valid instance.

```

sig Eqn {
  nm, nc, ni: Int
}{
  rel[nm, nc, ni]
}

run show {
  all m,c,i:range[12] | rel[m,c,i] ⇒ some e:Eqn |
    e.nm = m and e.nc = c and e.ni = i
}

```

this/Eqn	nm	nc	ni
Eqn ⁰	11	11	1
Eqn ¹	10	10	1
Eqn ²	4	4	1
Eqn ³	8	4	3

Eqn ² 1	Eqn ² 2	Eqn ² 3	Eqn ² 4	Eqn ² 5
Eqn ² 1	10	6	2	
Eqn ² 2	6	6	1	
Eqn ² 3	8	5	2	
Eqn ² 4	11	5	3	
Eqn ² 5	5	5	1	

Figure 3.5: Inspecting all possible size combinations for alpha

```
} for 26 but 5 Int
```

Fig. 3.5 shows how the table interface of the Alloy visualizer can be used to inspect this instance.

Concrete operation. The concrete operation refines the abstract operation, in this case by introducing parallelism. We model this in Alloy through a *JacobiStep* predicate between two coarrays. Like the Fortran implementation, we organize the predicate into two main sections, a computation phase and a communication phase. In the computation phase, each image calculates new values for its interior elements, using the same *neighbors* mechanism as the abstract operation. In the communication phase, interface column values are shared between adjacent images to prepare for the next iteration.

```
pred JacobiStep [u, v: Coarray] {
  sameShape[u, v]
  all i:allRows[u], j:allCols[u], q, p: u.mseq.inds |
    let Lc = lastCol[u], Lr = lastRow[u] {
      Inv[v.mseq[q]]
      simpleMatrix[u.mseq[q]]

      -- COMPUTATION PHASE
```

```

q = p and j != 0 and j != Lc =>
  (i != 0 and i != Lr =>
    v.mseq[q].vals[i, j] = neighbors[u.mseq[q], i, j]
    else v.mseq[q].vals[i, j] = u.mseq[q].vals[i, j])
v.mseq[0].vals[i, 0] = u.mseq[0].vals[i, 0]
q = u.mseq.lastIdx =>
  v.mseq[q].vals[i, Lc] = u.mseq[q].vals[i, Lc]

-- COMMUNICATION PHASE
q = minus[p, 1] and j = Lc =>
  v.mseq[q].vals[i, j] = v.mseq[p].vals[i, 1]
q = plus[p, 1] and j = 0 =>
  v.mseq[q].vals[i, j] = v.mseq[p].vals[i, minus[Lc, 1]]
}
}

```

The communication step makes a subtle but important design choice that is reflected in the predicate above. A coarray image shares the values it computes in interface columns before the next iteration begins, yet no synchronization barrier is needed. Instead of “pulling” values—which may or may not have been computed—from adjacent matrices, an image “pushes” its computed values by writing to its neighbors. Doing so guarantees the absence of race conditions, and eliminates the need for interleaving-style specifications.

Although we can and have used interleaving to detect race conditions in simple CAF models (described in depth in Chapter 5) it is interesting to ask what happens in applications like that of Martin, which are not written in an update-in-place style, and where there is nevertheless interference, such as the inadvertent overwriting of values by processes due to a bug. Can we find it? In such a case, overwriting produces a contradiction in the antecedent of the correctness check, so it appears safe. Therefore, one needs both safety and progress checks, which we include. That is to say, interference of this kind manifests as lack of progress, which is detectable.

Adequacy. To tackle the proof obligations for refinement introduced in Chapter 2, we first check adequacy. For every valid abstract state (dense matrix), there should exist a valid concrete state (coarray). In first order logic, we have:

$$\forall a: A \mid \exists c: C \mid \alpha(c, a) \quad (3.1)$$

This equation introduces an unbounded universal quantifier and cannot be checked verbatim in Alloy as it contradicts the semantics of signatures, which denote just some set

of values (Jackson 2002). Instead, we reformulate the assertion as follows:

```

sig P {
  con: lone Coarray,
  abs: Matrix
} {
  some con  $\Rightarrow$  #con.mseq > 1
}

check adequacy { all p:P | alpha[p.con,p.abs]  $\Rightarrow$  some p.con }

```

Intuitively, the check confirms that for any signature P , there exists a valid concrete state in the *con* field when the abstraction relation holds between it and the abstract state in the *abs* field. Because any particular abstract object can map to multiple concrete ones, we take care to exclude the trivial case where a dense matrix is mapped to a single image coarray.

The form of this check differs from the first order logic in Equation 3.1, but is equivalent, and constitutes a novel approach—which we refer to as the *indirection approach* for contending with unbounded universal quantifiers in Alloy. We discuss this contribution of our research in depth in Chapter 4.

Alpha is total. We verify that the abstraction relation, *alpha*, is total. That is, for every valid concrete state there should be a corresponding abstract state:

$$\forall c: C \mid \exists a: A \mid \alpha(c, a) \quad (3.2)$$

We again use the indirection approach to express an equivalent check in Alloy:

```

sig P {
  con: Coarray,
  abs: lone Matrix
}

check isTotal { all p:P | alpha[p.con,p.abs]  $\Rightarrow$  some p.abs }

```

The verifier confirms that for every instance of P , a valid abstract matrix can be found to satisfy *alpha* and populate the *abs* field.

Alpha is functional. We verify that alpha is a functional relation: any two abstract states that map to the same concrete state must be equivalent.

$$\forall a_1, a_2: A, c: C \mid \alpha(c, a_1) \wedge \alpha(c, a_2) \Rightarrow \text{equal}(a_1, a_2) \quad (3.3)$$

Alloy has a rich notion of scope that allows users to bound each signature separately to accommodate scope sizes that are appropriate for a problem, down to each assertion. With matrices, however, their sizes are naturally determined by row and column dimensions, a numerical quantity that is bound by a single bitwidth specification in Alloy, which sets the scope of all integers (Milicevic and Jackson 2014).

While otherwise not a concern, when a model calls for matrices of relatively different sizes — like coarray images that correspond to a larger abstract matrix — some checks may produce spurious counterexamples. This occurs when a coarray matrix that fits within the integer scope corresponds to an abstract matrix that does not. For this check, we quantify the concrete state space over a subset of coarrays (called *CoarraySmall*) — one in which the necessary integers are guaranteed to be in scope to avoid integer overflows. We describe working with bounded integer arithmetic in detail in Chapter 4.

```

check isFunctional {
  all a1,a2: Matrix, c:CoarraySmall |
    (alpha[c,a1] and alpha [c,a2]) ⇒ equivalent[a1,a2]
}

```

Because simple equality (=) would not be sufficient to check for equivalence of the two abstract states, helper predicates are designed to specify what conditions need be met:

```

pred equivalent [u, v: Coarray] {
  sameShape[u, v]
  all i: u.mseq.indxs | equivalent[u.mseq[i], v.mseq[i]]
}

pred equivalent [U, V: Matrix] {
  U.rows = V.rows
  U.cols = V.cols
  U.vals = V.vals
}

```

Progress. To ensure checks involving the concrete operation are not vacuously true, we verify progress of the concrete operation. For every valid concrete pre-state there must be a valid concrete post-state:

$$\forall c:C \mid \exists c':C \mid op_c(c,c') \tag{3.4}$$

As this is a progress check with an unbounded universal quantifier, we again use the induction approach:

```

sig P {
  c: Coarray,
  c": lone Coarray
}

```

```

check progress { all p: P | JacobiStep[p.c, p.c"]  $\Rightarrow$  some p.c" }

```

The check ensures that for any concrete pre-state (c), some concrete post-state (c'') satisfies the concrete operation (*JacobiStep*) and populates the $p.c''$ field.

Correctness. Finally, having satisfied the proof obligations on *alpha*, as well as adequacy and progress, we proceed to check correctness of the concrete operation.

$$\forall a, a': A, c, c': C \mid \alpha(c, a) \wedge op_c(c, c') \wedge \alpha(c', a') \Rightarrow op_a(a, a') \quad (3.5)$$

We again use the *CoarraySmall* quantifier to avoid integer overflows.

```

check correctness {
  all c, c": CoarraySmall, a, a": Matrix |
    (alpha[c, a] and alpha[c", a"] and JacobiStep[c, c"])
       $\Rightarrow$  JacobiStep[a, a"]
}

```

Using this approach we were able to verify the specification originally proposed by Martin (2019) automatically in Alloy.

The feedback via counterexamples is useful for constraining invariants and other properties that are not explicitly defined by Martin but that are necessary for formal verification based on refinement. With small but reasonable scopes, running the Alloy models is relatively fast, with some of the longer checks returning in tens of minutes. Because of the bounded approach, as noted above, there are three key considerations to contend with, which we discuss more fully in Chapters 4 and 5.

CHAPTER

4

BOUNDED VERIFICATION

Some issues unique to bounded verification arise when checking data refinement in the Alloy Analyzer. The refinement approach as defined in predicate logic does not always translate directly in the context of a finite scope. Here we describe two main issues we encountered: checking the totality of functions and dealing with bounded integer arithmetic, as well as our approaches for addressing them.

4.1 Total Functions

Many of the proof obligations in the refinement framework are of the form “for all x there exists some y such that ...”. For instance, we are interested in knowing whether a relation is total, which serves as a progress check. We begin with some basic definitions of relations and their structure.

Definitions. A relation over sets X and Y is any subset of $X \times Y$. If f is a relation over X and Y , then we define the domain of f to be $domain(f) \triangleq \{x \in X \mid (\exists y \in Y \cdot (x, y) \in f)\}$ and the range of f to be $range(f) \triangleq \{y \in Y \mid (\exists x \in X \cdot (x, y) \in f)\}$. A relation over X and Y is

total over X if $domain(f) = X$ (Lynch and Vaandrager 1995).

Using a functional notion, we might write more simply:

$$\forall x: X \mid \exists y: Y \mid f(x) = y \tag{4.1}$$

to check whether a relation is total.

Such a property should be verified along all edges of a commuting diagram to avoid spurious results. In this chapter we show how assertions of this form are challenging in Alloy, describe the approaches taken in previous studies, and propose a new, simpler approach for tackling the problem.

4.1.1 Unbounded Universal Quantifiers

Existential assertions with unbounded universal quantifiers are difficult to reason about in model checkers such as Alloy.

"The problem arises when a signature is intended to represent all possible values of a composite structure. This contradicts the semantics of Alloy, in which a signature denotes just some set of values. The contradiction only becomes apparent when universal quantification is used in a particular way" (Jackson 2002).

Matrices and other objects can be represented in Alloy, but they will behave in Alloy as "just some set of values," as Jackson describes, not having all the properties that are intuitively expected of them when existential quantifiers are nested within a universal quantifier.

For example, every matrix has a corresponding transpose, and one might try to verify this in Alloy with the assertion as below:

```
assert { all m: Matrix | some m': Matrix | transpose[m, m'] }
```

Running this in Alloy, however, will result in counterexamples where the satisfying transpose is never populated. The model finding nature of Alloy might suggest that the above assertion should pass, but in any particular instance there is no requirement for a satisfying transpose matrix to be populated according to Alloy semantics. Unless specified outright, it chooses only an arbitrary set of atoms, up to the given scope. For this reason, an alternate approach is needed when verifying the proof obligation checks discussed in Chapter 3.

4.1.2 Generator Axioms

Jackson (2002) discusses the issue of unbounded universal quantifiers and suggests the use of generator axioms to force Alloy to populate every possible instance of an object, within a bounded scope, to allow such existential assertions to be checked. As an example, he proposes a generator axiom to populate the universe of all possible *Set* signatures, in order to verify that, for any two sets, there exists a third that contains the elements of both. As expected, running the model below will always yield counterexamples where the satisfying set *s3* is never populated

```
sig Set {
  elements: set Element
}

sig Element {}

check Closed{
  all s1, s2: Set | some s3: Set |
    s3.elements = s1.elements + s2.elements
}
```

unless a generator axiom is introduced that forces all possible sets to exist:

```
fact SetGenerator {
  some s: Set | no s.elements
  all s: Set, e: Element |
    some s": Set | s".elements = s.elements + e
}
```

In this case, a generator axiom seems straightforward, but they are not always as easy to formulate or computationally feasible.

4.1.3 Matrix Generator Axioms

As the complexity of models increases, generator axioms quickly become harder to work with. For matrices, verifying that some computation such as matrix transpose is total requires a generator axiom to populate a matrix for every unique combination of *Value* signatures in the scope. This can be accomplished in Alloy by ensuring that every matrix be distinct and fixing the scope to include the known number of combinations. Below, we use a small scope to illustrate the approach. We restrict the size of matrices to only those that are 2×2 and allow only two distinct values (*v1* and *v2*). This results in 16 possible combinations, which we ensure at the scope level. It is also possible to ensure this at the

predicate level (by specifying $\#Matrix = 16$) but this necessitates a larger *Int* scope, which increases solution time significantly.

```

sig Matrix {
  rows, cols: Int,
  vals: Int→Int→lone Value
}
{
  rows = 2 and cols = 2
  vals.univ = range[rows]→range[cols]
}

pred transpose [a, b: Matrix] {
  b.vals = { j, i: Int, v: Value | i→j→v in a.vals }
}

pred Generator {
  no disj m1, m2: Matrix | m1.vals = m2.vals
}

fact { Generator }

check {
  all m: Matrix | some m": Matrix | transpose[m, m"]
}
for 2 but exactly 16 Matrix, 3 Int

```

This works as expected and the check passes. However, because the axioms produce a combinatorial number of instances, i.e., $\mathcal{O}(v^{n \times n})$ for $n \times n$ matrices whose elements each have v possible values, the approach is impractical. For an only slightly larger scope of 4 distinct values and 2×2 matrices there are 256 matrix permutations, which is already outside the translation capacity of the Alloy analyzer. Further, for more complex signatures, constraining a generator axiom by counting unique combinations may be infeasible. All this points to the need for a simpler approach that scales to larger problems.

4.1.4 Indirection Approach

To overcome the unbounded universal quantifier issue in Alloy, we propose a novel way to check existential assertions without using generator axioms by introducing an additional level of indirection. We do this in the form of a signature with relations mapping to the objects being checked. For a general assertion of the form

$$\forall x: X \mid \exists y: Y \mid f(x) = y \quad (4.2)$$

we can formulate a check on whether f is total in Alloy as below:

```

sig P {
  x: X,
  y: lone Y
}

pred f [x: X, y: Y] { ... }

check isTotal { all p: P | f[P.x, P.y]  $\Rightarrow$  some P.y }

```

Here the keyword *lone* indicates a multiplicity of one or none, so the y field is allowed to be empty. This construction forces Alloy to check all possible instances of P where $f[p.a, p.b]$ holds, yet the field for y is empty, indicating that f is undefined for some particular x .

Instead of using a generator axiom as before, we check to see if any matrix has a corresponding transpose using this approach:

```

sig P {
  x: Matrix,
  y: lone Matrix
}

check { all p: P | transpose[p.x, p.y]  $\Rightarrow$  some p.y }

```

The check passes without necessitating the use of generator axioms or any additional effort on the part of the modeler to constrain the exact required scope.

We go a step further and show that this approach is equivalent, through bi-implication, to the generator axiom shown previously. Below, *isTotal* represents the unbounded universal quantifier assertion restructured as a predicate. Similarly, *isTotal2* is equivalent to the new indirection approach.

```

one sig p in P {}
one sig m in Matrix {}

fact { m = p.x }

pred isTotal { all mT: Matrix | not transpose[m, mT] }

pred isTotal2 { transpose[p.x, p.y] and all mT: Matrix | p.y != mT }

check equivalent { isTotal2  $\Leftrightarrow$  (isTotal and Generator) }

```

Together, *isTotal* and *Generator* are equivalent to *isTotal2*, giving confidence that the new approach is simpler, yet equivalent to using a generator axiom.

We also illustrate the approach with the set example from Jackson (2002) shown previously. Below, the signature P contains three *lone* relations that map to sets, and we reformulate the existential check to include an implication on the existence of $P.S3$:

```

sig P {
  S1: lone Set,
  S2: lone Set,
  S3: lone Set }

pred closed [s1, s2, s3:Set] {
  s3.elements = s1.elements + s2.elements
}

pred nonEmpty[s: Set] { some s.elements}

check allClosed {
  all p: P |
  nonEmpty [p.S1] and closed[p.S1, p.S2, p.S3]  $\Rightarrow$  some p.S3
}

```

A predicate ensuring that at least one of $P.S1$ or $P.S2$ must be non-empty is needed, or Alloy will produce a counterexample where both are empty, and thus no $P.S3$ is needed to satisfy the closed predicate. For a check like this to work as intended, predicates such as *Closed* should not enforce existence of the object — for instance, with a clause such as *some s3*. This property is important, otherwise the antecedent in the assertion would imply existence of an object and the check would pass vacuously.

To further confirm that this type of check works as intended, we introduce tests that cause the approach to fail on purpose and inspect the counterexamples produced. For the set example, we specify a *ClosedBroken* predicate, which is deliberately undefined whenever the special signature *brokenElem* belongs to set $s1$.

```

sig brokenElem extends Element {}

pred ClosedBroken [s1, s2, s3:Set] {
  brokenElem not in s1.elements  $\Rightarrow$ 
  s3.elements = s1.elements + s2.elements
}

check allClosedBroken {
  all p: P |
  nonEmpty[p.S1] and ClosedBroken[p.S1, p.S2, p.S3]  $\Rightarrow$  some p.S3
}

```

As expected, checking this assertion produces counterexamples where *brokenElem* causes *P.S3* to be empty, showing that the relation is not total.

This novel approach makes refinement checking practical in Alloy, as all proof obligations can be easily checked. For example, checking whether an abstraction relation is total:

```

sig P {
  c: Concrete,
  a: lone Abstract
}

pred alpha [c: Concrete, a: Abstract] { ... }

check isTotal {
  all p: P | alpha[p.c, p.a]  $\Rightarrow$  some p.a
}

```

We also use this approach for checking adequacy and progress of the concrete operation. Outside of refinement, this approach is general enough for other applications, where generator axioms were previously unavoidable.

4.2 Bounded Integers

Alloy is a structural modeling language, and its emphasis is on object models in non-numerical domains, where integer arithmetic is at most a minor concern. However, for models of HPC applications, which must capture concepts like array indexing, integer arithmetic cannot be avoided. Modelers must carefully inspect refinement checks to make sure counterexamples are not spurious, as a result of integer scope, while also avoiding large scopes that would undermine performance.

We illustrate these issues through an example from the HPC application described in Chapter 3:

```

pred rel [m, c, i: Int] {
  m  $\geq$  0 and c  $\geq$  0 and i  $\geq$  0
  m = 0  $\Rightarrow$  c = 0 and i = 0
  m > 0 and m < 4  $\Rightarrow$  c = m and i = 1
  m  $\geq$  4  $\Rightarrow$  m = add[mul[i, sub[c, 2]], 2]
}

```

The *rel* predicate shown above relates the number of columns in a larger abstract matrix (*m*), the number of columns in each individual, smaller, coarray matrix (*c*), and the number

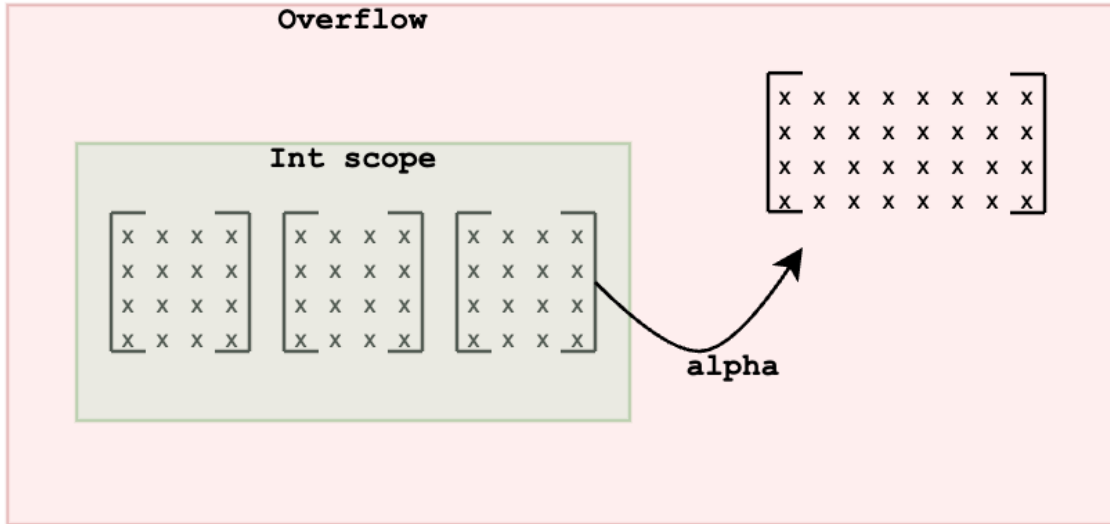


Figure 4.1: Relative size difference causing integer overflow

of images (i). This predicate makes up part of the abstraction relation *alpha* and causes issues with integer scope due to the relative difference in size between the concrete and abstract spaces. As shown in Fig. 4.1, a concrete state may be in scope while its abstract counterpart may not be.

When checking if *alpha* is functional,

```

check isFunctional {
  all a1, a2: Matrix, c: Coarray |
    (alpha[c, a1] and alpha[c, a2])  $\Rightarrow$  equivalent[a1, a2]
}
for 1 Coarray, 2 Matrix, 4 Int

```

Alloy produces spurious counterexamples where the antecedent to the implication is false (that is, instances where both $\alpha[c, a1]$ and $\alpha[c, a2]$ are false). Inspecting these counterexamples further, we see that Alloy chooses coarrays of such size that the corresponding larger abstract matrices are not representable within the specified integer scope. A modeler may try increasing the integer scope but with every increase, Alloy drives the counterexample up in size to produce the same (just larger) case. One might also try reformulating the check as an Alloy *run* in its negated form, as below, and be surprised when no instances (analogous to counterexamples) are found. This suggests inconsistencies in how the Alloy analyzer translates these arithmetic checks into SAT.

```

run isFunctionalAlt {
  some a1, a2: Matrix, c: Coarray |
    alpha[c, a1] and alpha[c, a2] and not equivalent[a1, a2]
}
for 1 Coarray, 2 Matrix, 4 Int

```

To avoid overflow in correctness checks, we propose limiting the size of objects in either the concrete or abstract space through a replacement quantifier. For the Jacobi program, we limit the size of the concrete space, as coarray matrices are smaller than their abstract matrix counterparts. A robust way to do this in Alloy is by quantifying assertions over the set of "small enough" coarrays, rather than all coarrays. An Alloy function (*CoarraySmall*) can be used to return the set of coarrays for which there exists an integer (within scope) that satisfies the *rel* predicate:

```

fun CoarraySmall : set Coarray {
  { c: Coarray | some m: Int | rel[m, c.mseq[0].cols, #c.mseq] }
}

```

This ensures that Alloy will not run into integer overflow issues when checking assertions quantified over this set, such as when checking if *alpha* is functional:

```

check isFunctional { all a1, a2: Matrix, c: CoarraySmall | ...

```

Outside of the parallel Jacobi program, this type of approach is applicable to other models where relative size differences involving integer computations are of importance.

CHAPTER

5

FUTURE DIRECTIONS

The work presented in this thesis details an approach for verifying scientific programs, specifically those making use of parallelism, through refinement checking in Alloy. We see various avenues to expand on this research in future work. The main avenue of interest focuses on data race issues, which, although not a concern for the Jacobi routine modeled in Chapter 3, are important when dealing with most parallel programs. Other directions of interest include extending this work to apply more directly to neural networks, and using Alloy to build symbolic libraries for common, reusable mathematical expressions — such as the matrix-vector multiplication model by Dyer et al. (2019).

5.1 Interleaving Specifications

An important concern with HPC applications is the non-determinism that is inherent to parallel computing. When modeling in Alloy, a decision must be made as to whether some or all processes should be modeled atomically through interleaving in order to capture interference or data race issues. In the CAF Jacobi iteration code modeled in this thesis, interleaving is not necessary due to the nature of communication between images. In the

Fortran code by Martin (2019), a computation step is followed by a communication step where images share the computed values at their interfaces. No synchronization barrier is needed before communication because Images push their computed values onto others, rather than pulling values (that may or may not yet be computed) from neighboring images:

```

IF (IMAGE .GT. 1) THEN
  DO I = 1, IMAX
    V(I, PARTITION_WIDTH + 1)[IMAGE-1] = V(I, 1)[IMAGE]
  END DO
END IF
IF (IMAGE .LT. N_IMAGES) THEN
  DO I = 1, IMAX
    V(I, 0)[IMAGE+1] = V(I, PARTITION_WIDTH)[IMAGE]
  END DO
END IF
SYNC ALL

```

This is an important distinction, as the reverse case (pull communication) would require an Alloy model with interleaving to capture the potential data race issues. Subtle design choices such as this may not be captured correctly in Alloy and result in verifying a specification that is not true to the original code. Further, some programs may make use of complicated synchronization techniques where it is not obvious whether interference is a concern. Because of this, exploring the potential for modeling interleaving processes in Alloy is a worthwhile direction.

Small programs can be readily checked in Alloy for data races using state transition relations. A process may be modeled as a signature with $var \rightarrow State$ relations that model program counters and local variables. For example, we consider the simple CAF program below, where a single integer variable (num) is replicated across all images. Image 1 modifies its local num and all other images pull this new value to update their own copy:

```

integer :: num[*]
A if (this_image() .eq. 1) &
B   num = 1
C sync all
D num = num[1]
end

```

Without the synchronization barrier that appears in line *C*, the result of the program is non-deterministic, as some images can reach line *D* before Image 1 has set its value. The example is trivial, but modeling it in Alloy illustrates how stateful relations can be used to detect data races.

In the Alloy code below, we treat each Image as a signature, with relational fields for

each coarray variable (in this case just *num*) and a program counter tracking which phase (line of code) the image is computing.

```

open util/ordering[Image] as io
open util/ordering[State] as so

sig State {}
abstract sig Phase {}
one sig A, B, C, D extends Phase {}

sig Image {
  ph: Phase one → State,
  num: Int one → State
}

```

Separate predicates are defined to handle each state transition. The *step* predicate below ensures that for each change in state (from *s* to *s'*) any arbitrary (*some*) number of the images will make a transition. The *ph* field is used to determine which transition predicate is available for a particular image — one is defined for every line of code in the original program, including the synchronization barrier.

```

pred init [s: State] {
  all i: Image | i.ph.s = A and i.num.s = 0
}

pred step [s, s': State] {
  (some i: Image | Assign[i, s, s'] or copyFrom[i, s, s']) or
  finish[s, s'] or synch_all[s,s']
}

pred Assign [i: Image, s, s': State] {
  i.ph.s = A
  i.ph.s' = B
  i.num.s' = (i = io/first ⇒ 1 else 0)
  unchanged[Image - i, s, s']
}

pred synch_all [s, s': State] {
  all i: Image | i.ph.s = B and
  i.ph.s' = C and i.num.s' = i.num.s -- stutter
}

pred copyFrom [i: Image, s, s': State] {
  i.ph.s = C
}

```

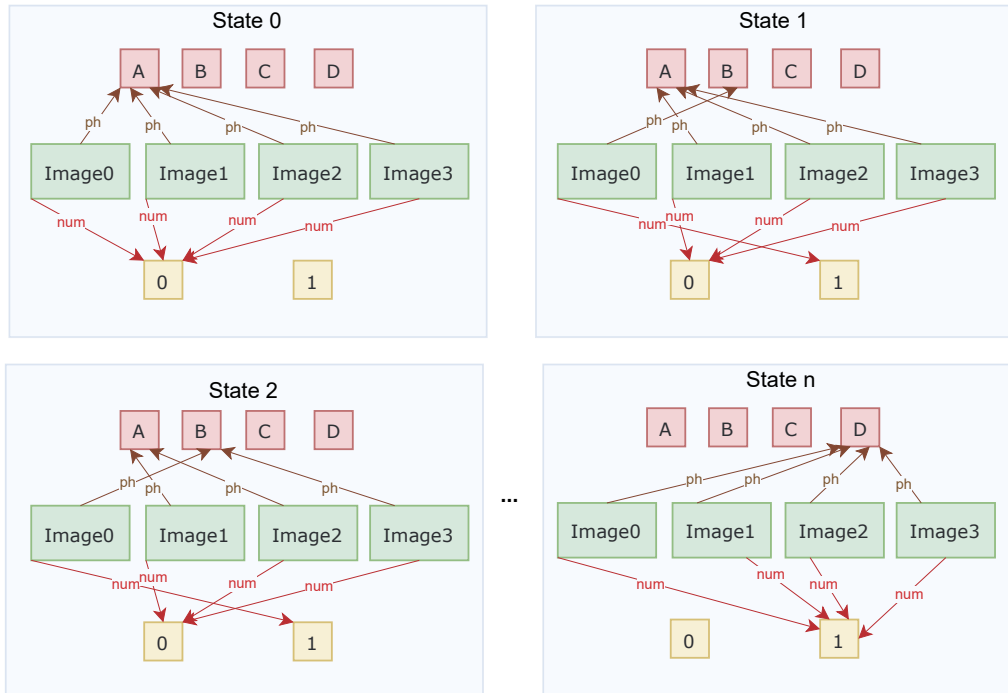


Figure 5.1: Transition instance by projecting over state

```

i.ph.s" = D and i.num.s" = io/first.num.s
unchanged[Image - i, s, s"]
}

pred finish [s, s": State] {
  all i: Image | i.ph.s = D and
  i.ph.s" = i.ph.s and i.num.s" = i.num.s -- stutter
}

pred unchanged [images: set Image, s, s": State] {
  all i: images | i.ph.s" = i.ph.s and i.num.s" = i.num.s
}

```

The predicate *show_synched* is run to generate valid instances:

```

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

```

Projecting an instance of this model over State in the Alloy analyzer allows the user to navigate through the state transitions of the program, where variable assignments are

captured for each step, as in Fig. 5.1.

A separate predicate can then be defined to model the program without the explicit synchronization barrier by removing *synch_all* as a possible transition and skipping over phase *C* in *copyFrom2* below:

```
pred step_race [s, s": State] {
  (some i: Image | Assign[i, s, s"] or copyFrom2[i, s, s"])
  or finish[s, s"]
}

pred copyFrom2 [i: Image, s, s": State] {
  i.ph.s = B
  i.ph.s" = D and i.num.s" = io/first.num.s
  unchanged[Image - i, s, s"]
}

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

Finally, to detect race conditions in either case (with or without synchronization), assertions are checked to determine whether, at termination, each image has the same value for *num*.

```
check all_one {
  show_synched ⇒
  all i: Image | i.num.so/last = 1
} for 4 but 20 State

check all_one_race {
  show_race ⇒
  all i: Image | i.num.so/last = 1
} for 4 but 20 State
```

As expected *all_one* passes but *all_one_race* does not, and produces a counterexample where Image 1 modifies its value for *num* after other images have already pulled it.

For this trivial program, Alloy is able to detect race conditions easily, and the assertions are checked in a matter of seconds. For larger, more complex programs, this type of modeling in Alloy is an area for future work. It is not clear what level of complexity can be handled for interleaving on realistic programs before scope explosion makes the models computationally infeasible.

5.2 Other Possible Extensions

Much of the work presented here, although mainly in the context of HPC, proposes ideas that are extensible to neural networks. Many of the operations and types of parallelism common to HPC applications have parallels in NNs.

One future direction of interest is extending the work by Singh et al. (2020), which uses Alloy specifications to create correct-by-construction neural networks, rather than through more traditional means of training with data sets. Although they do not use a refinement framework, further work could bridge the gap between their study and the work in this thesis.

Siegel et al. (2008) use Spin, a model checking tool, to verify small parallel programs by building up symbolic expressions and checking them for equivalence. They use sequential specifications to confirm that parallel implementations are correct — mirroring the approach taken in this thesis. An area of interest is recreating this work in Alloy, with the aim of producing a library of reusable symbolic math operations (such as matrix multiply).

Srivastava et al. (2020) describe MatRaptor, a parallel sparse-sparse matrix multiplication accelerator. The method makes use of a row-wise product and a novel sparse storage format that overcomes certain limitations of CSR. The authors report significant performance benefits, both in terms of speed, power consumption and area. The approach is beneficial to many domains, including compressed neural network computations — which we are interested in. Future work could involve verifying their approach to test the Alloy refinement framework on a modern ML-motivated sparse parallel matrix algorithm.

CHAPTER

6

CONCLUSIONS

We present an approach for checking data refinement in Alloy to verify correctness properties of scientific software. Unlike attempts at after-the-fact verification, our emphasis is on “design thinking,” an inherently iterative process that, with tool support, may help scientists and engineers gain a deeper understanding of the structure and behavior of the programs they create. Tangible artifacts from the process include representation invariants that must be maintained by concrete implementations—in languages like Fortran, C/C++, and Julia—and abstraction relations that define and document how they should be interpreted.

Although we believe this to be a promising approach for verifying scientific programs, the work presented in this thesis also highlights some of the challenges of finite instance finding in dealing with existential quantifiers, integer bounds, and scope explosion. Practitioners will likely encounter these issues themselves, so developing a common approach for tackling them is necessary. Further work is needed to understand the best Alloy formulations for typical refinement checks as well as addressing technical limitations of the Alloy Analyzer in scientific applications.

Acknowledgments.

This work was funded by NSF under the Formal Methods in the Field (FMitF) program, awards #2124205 (NCSU) and #2124100 (Utah).

REFERENCES

- Baugh, J. and Altuntas, A. (2018). Formal methods and finite element analysis of hurricane storm surge: A case study in software verification. *Science of Computer Programming*, 158:100–121.
- Baugh, J. and Dyer, T. (2018). State-based formal methods in scientific computation. In *ABZ 2018: Abstract State Machines, Alloy, B, TLA, VDM, and Z: 6th International Conference*, pages 392–396. Springer.
- Benavides, J., Baugh, J., and Gopalakrishnan, G. (2023). An HPC practitioner’s workbench for formal refinement checking. In Mendis, C. and Rauchwerger, L., editors, *Languages and Compilers for Parallel Computing, LCPC 2022*, pages 64–72. Springer, Cham. Lecture Notes in Computer Science, vol. 13829.
- Clarke, E. M. and Wing, J. M. (1996). Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643.
- de Roever, W.-P., Engelhardt, K., and Buth, K.-H. (1998). *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press.
- Dyer, T., Altuntas, A., and Baugh, J. (2019). Bounded verification of sparse matrix computations. In *Proceedings of the Third International Workshop on Software Correctness for HPC Applications, Correctness’19*, pages 36–43. IEEE/ACM.
- Elseaidy, W. M., Cleaveland, R., and Baugh, J. W. (1997). Modeling and verifying active structural control systems. *Science of Computer Programming*, 29(1):99–122. COST 247, Verification and validation methods for formal descriptions.
- Faulk, S., Loh, E., Van De Vanter, M. L., Squires, S., and Votta, L. G. (2009). Scientific computing’s productivity gridlock: How software engineering can help. *Computing in Science & Engineering*, 11(6):30–39.
- Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., Liu, T., Wang, X., Wang, G., Cai, J., et al. (2018). Recent advances in convolutional neural networks. *Pattern recognition*, 77:354–377.
- Hoare, C. A. R. (1972). Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281.
- Jackson, D. (2002). Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256 – 290.
- Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.
- Jackson, D. and Wing, J. (1996). Lightweight formal methods. *Computer*, 29(4):21.

- Katz, G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. (2017). Reluplex: An efficient SMT solver for verifying deep neural networks. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, pages 97–117. Springer.
- Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., et al. (2019). The Marabou Framework for verification and analysis of deep neural networks. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*, pages 443–452. Springer.
- Liskov, B. and Guttag, J. (2000). *Program Development in Java: Abstraction, Specification, and Object-oriented Design*. Addison-Wesley.
- Lynch, N. A. and Vaandrager, F. (1995). Forward and backward simulations, Part I: Untimed systems. *Information and Computation*, 121(2):214–233.
- Martin, J. M. R.. (2019). Testing and verifying parallel programs using data refinement. In *Communicating Process Architectures 2017 & 2018*, pages 491–500. IOS Press.
- Siegel, S. E., Mironova, A., Avrunin, G. S., and Clarke, L. A. (2008). Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–34.
- Singh, S., Vasic, M., and Khurshid, S. (2020). Designing neural networks using logical specs. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 160–171. IEEE.
- Srivastava, N., Jin, H., Liu, J., Albonesi, D., and Zhang, Z. (2020). MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780. IEEE.
- Storer, T. (2017). Bridging the chasm: A survey of software engineering practice in scientific programming. *ACM Computing Surveys (CSUR)*, 50(4):47:1–47:32.
- Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. (2009). Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4).

APPENDIX

APPENDIX

A

ALLOY MODELS

A.1 Base Model

```
module jacobi

/*
 * Verification of a parallel Laplace equation solver on a rectangular
 * domain from an implementation in Coarray Fortran
 *
 * Authors: Juan Benavides, John Baugh, and Ganesh Gopalakrishnan
 * Alloy Analyzer 6.1.0
 *
 * For a detailed description, see:
 *
 * An HPC practitioner's workbench for formal refinement checking
 *
 * LCPC 2022: 35th International Workshop on Languages and Compilers
 * for Parallel Computing (to appear)
 */

// Abstract level: start with original matrix structure from Dyer et al.
```

```

sig Value {}
one sig Zero, One extends Value {}

-- Dense matrices of symbolic (atomic) values, zero and non-zeros

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

fact AbstractInv { all m: Matrix | Inv[m] }

-- abstract invariant: enforce as a fact
pred Inv [m: Matrix] {
  m.rows ≥ 0
  m.cols ≥ 0
  m.rows = 0 ⇔ m.cols = 0 -- just one shape for an empty matrix (both 0)
  m.vals.univ = range[m.rows]→range[m.cols]
}

-- the set [0, n-1]
fun range [n: Int]: set Int {
  { i: Int | 0 ≤ i and i < n }
}

-- two matrices have the same dimensions
pred sameShape [a, b: Matrix] {
  a.rows = b.rows
  a.cols = b.cols
}

-- two matrices are equivalent
pred equivalent [a, b: Matrix] {
  a.rows = b.rows
  a.cols = b.cols
  a.vals = b.vals
}

-- index of last column of a matrix
fun lastCol [m: Matrix]: one Int {
  { i: Int | i = minus[m.cols, 1] }
}

-- index of last row of a matrix
fun lastRow [m: Matrix]: one Int {
  { i: Int | i = minus[m.rows, 1] }
}

```

```

}

// Adding expressions

// Instead of building up and comparing general arithmetic
// expressions, provide a way to record values of the four neighbors
// in a matrix that are used to compute an average (the only type of
// arithmetic computation being performed in a Jacobi iteration).

sig Neighbors extends Value {
  up, down, left, right: Value
}

fun neighbors [m: Matrix, i: Int, j: Int]: one Neighbors {
  { n: Neighbors |
    n.up = m.vals[plus[i, 1], j] and
    n.down = m.vals[minus[i, 1], j] and
    n.left = m.vals[i, minus[j, 1]] and
    n.right = m.vals[i, plus[j, 1]]
  }
}

-- no extraneous neighbors: keep only those that appear somewhere in a matrix
fact { all n: Neighbors | n in Matrix.vals[Int][Int] }

-- A step in a Jacobi iteration (abstract)
pred JacobiStep [U, V: Matrix] {
  sameShape[U, V]
  V.vals.univ = range[V.rows]→range[V.cols] -- populate elements (neighbors)
  V.vals =
    { i: range[U.rows], j: range[U.cols], x: Value |
      let boundary = (j = 0 or j = minus[U.cols, 1] or
                    i = 0 or i = minus[U.rows, 1]) |
      x = (boundary ⇒ U.vals[i, j] else neighbors[U, i, j]) }
}

// Generate some instances to look at

pred show {
  some a, b: Matrix |
    relevant[a] and simpleMatrix[a] and JacobiStep[a, b]
}

run show for 5 but 2 Matrix, 0 Coarray

-- show only "interesting" matrices

```

```

pred relevant [m: Matrix] {
  m.rows ≥ 3
  m.cols ≥ 3
}

-- a matrix with only "atomic" values for elements (no Neighbors)
pred simpleMatrix [m: Matrix] {
  no m.vals[Int][Int] & Neighbors
}

// Concrete level: a coarray is a sequence of matrix images

sig Coarray {
  mseq: seq Matrix
}

-- coarrays have at least one image, and they all have the same
-- the number of images (basic Coarray Fortran semantics)
fact numImages {
  all a: Coarray | #a.mseq > 0 and all b: Coarray | #a.mseq = #b.mseq
}

fact ConcreteInv { all c: Coarray | Inv[c] }

-- concrete invariant: enforce uniform shape of all images and
-- overlapping columns at the interfaces for border exchanges.
pred Inv [c: Coarray] {
  all disj a, b: c.mseq.elems | sameShape[a, b]
  all i: allRows[c], q, p: c.mseq.inds |
    let Lc = lastCol[c] {
      -- if q is left of p, last column of q is the second column of p
      q = minus[p, 1] ⇒
        c.mseq[q].vals[i, Lc] = c.mseq[p].vals[i, 1]
      -- if q is right of p, first column of q is next to last column of p
      q = plus[p, 1] ⇒
        c.mseq[q].vals[i, 0] = c.mseq[p].vals[i, minus[Lc, 1]]
    }
}

-- abstraction relation (alpha)
pred alpha [c: Coarray, m: Matrix] {
  totRows[c, m.rows]
  totCols[c, m.cols]
  all i: range[m.rows], j: range[m.cols] {
    -- 1st column of m is the 1st column of the 1st image of c
    j = 0 ⇒ m.vals[i, j] = c.mseq[0].vals[i, 0]
  }
}

```

```

-- last column of m is the last column of the last image of c
j = lastCol[m] ⇒
  let mi = sub[#c.mseq, 1],      -- matrix index
      ci = lastCol[c.mseq[mi]] | -- column index
      m.vals[i, j] = c.mseq[mi].vals[i, ci]

-- mapping of middle image columns
j != 0 and j != lastCol[m] ⇒
  let mi = div[sub[j, 1], sub[c.mseq[0].cols, 2]],
      ci = add[1, rem[sub[j, 1], sub[c.mseq[0].cols, 2]]] |
      m.vals[i, j] = c.mseq[mi].vals[i, ci]
}

-- number of rows (every image has the same number of rows)
pred totRows [c: Coarray, r: Int] {
  r = c.mseq[0].rows
}

-- number of matrix columns when coarray images are combined
pred totCols [c: Coarray, m: Int] {
  rel[m, c.mseq[0].cols, #c.mseq]
}

// rel: relationship between the number of abstract matrix columns,
//      coarray matrix columns, and images

// a: number of columns in the abstract matrix
// c: number of columns in EACH coarray matrix
// i: number of images

pred rel [a, c, i: Int] {
  a ≥ 0 and c ≥ 0 and i > 0
  a < 4 ⇒ c = a and i = 1
  a ≥ 4 ⇒ a = add[mul[i, sub[c, 2]], 2] // arranged to minimize int size
}

-- A step in a Jacobi iteration (concrete, predicate 2 from Martin)
pred JacobiStep [u, v: Coarray] {
  sameShape[u, v]
  all i: allRows[u], j: allCols[u], q, p: u.mseq.indcs |
    let Lc = lastCol[u], Lr = lastRow[u] {
      -- start with simple matrices for now
      simpleMatrix[u.mseq[q]]

      -- COMPUTATION PHASE

```

```

-- internal region: set non-boundary elements in v to neighbors in u
-- top and bottom rows (except first and last cols):
-- copy into v the elements from u
q = p and j != 0 and j != Lc =>
  (i != 0 and i != Lr =>
    v.mseq[q].vals[i, j] = neighbors[u.mseq[q], i, j]
    else v.mseq[q].vals[i, j] = u.mseq[q].vals[i, j])
-- first column of first image: copy into v the elements from u
v.mseq[0].vals[i, 0] = u.mseq[0].vals[i, 0]
-- last column of last image: copy into v the elements from u
q = u.mseq.lastIdx => v.mseq[q].vals[i, Lc] = u.mseq[q].vals[i, Lc]

-- COMMUNICATION PHASE
-- if q is left of p, set last column of q to the second column of p
q = minus[p, 1] and j = Lc =>
  v.mseq[q].vals[i, j] = v.mseq[p].vals[i, 1]
-- if q is right of p, set first column of q to next to last column of p
q = plus[p, 1] and j = 0 =>
  v.mseq[q].vals[i, j] = v.mseq[p].vals[i, minus[Lc, 1]]
}
}

-- for iterating over columns of a coarray image
fun allCols [c: Coarray]: set Int {
  range[c.mseq[0].cols]
}

-- for iterating over rows of a coarray image
fun allRows [c: Coarray]: set Int {
  range[c.mseq[0].rows]
}

-- two coarrays have the same dimensions
pred sameShape [u, v: Coarray] {
  all i: range[#u.mseq] |
    u.mseq[i].rows = v.mseq[i].rows and u.mseq[i].cols = v.mseq[i].cols
}

-- two coarrays are equivalent
pred equivalent [u, v: Coarray] {
  sameShape[u, v]
  all i: u.mseq.inds | equivalent[u.mseq[i], v.mseq[i]]
}

-- index of last column of a coarray image
fun lastCol [c: Coarray]: one Int {

```

```

    { i: Int | i = minus[c.mseq[0].cols, 1] }
}

-- index of last row of a coarray image
fun lastRow [c:Coarray]: one Int {
  { i: Int | i = minus[c.mseq[0].rows, 1] }
}

// Generate some instances to look at

pred show2 {
  some U, V: Matrix, u, v: Coarray |
    relevant[v, V] and alpha[u, U] and alpha[v, V] and
      simpleMatrix[U] and JacobiStep[u, v] and JacobiStep[U, V]
}

run show2 for 2 Coarray, 7 Matrix, 4 Value

-- show only "interesting" coarray-matrix combinations
pred relevant [c: Coarray, m: Matrix] {
  #c.mseq ≥ 2
  m.cols ≥ 3
  m.rows ≥ 3
}

/*
default scope is 3 (except for integers, which have a default bitwidth of 4)

max integer for "n Int" = 2^(n-1) - 1

  n   max   min
--   ---   ---
 10   511  -512
  9   255  -256
  8   127  -128
  7    63   -64
  6    31   -32
  5    15   -16
  4     7    -8   <- default
  3     3    -4
  2     1    -2
*/

```

A.2 Adequacy

```
-- adequacy: every abstract state must have a concrete counterpart

open jacobi

sig P {
  con: lone Coarray,
  abs: Matrix
}{}
  (some con and abs.cols ≥ 4) ⇒ #con.mseq > 1 // multiple images
}

// the trivial case (with just one image that is identical to the
// abstract matrix) is uninteresting, so check abstract matrices that
// are large enough to have multiple images (above)

assert isAdequate { all p: P | alpha[p.con, p.abs] ⇒ some p.con }

check isAdequate for 7 but 1 P, 4 Int
```

A.3 Functional

```
-- functional: show that alpha is functional

open jacobi

-- restrict coarray sizes to prevent overflow
fun CoarraySmall : set Coarray {
  { c: Coarray | some m: Int | rel[m, c.mseq[0].cols, #c.mseq] }
}

assert isFunctional {
  all a1, a2: Matrix, c: CoarraySmall |
    (alpha[c, a1] and alpha [c, a2]) ⇒ equivalent[a1, a2]
}

check isFunctional for 7 but 4 Int
```

A.4 Total

```

-- total: show that alpha is total

open jacobi

sig P {
  con: Coarray,
  abs: lone Matrix
}

assert isTotal { all p: P | alpha[p.con, p.abs] ⇒ some p.abs }

check isTotal for 7 but 1 P, 4 Int

```

A.5 Correctness

```

-- correct: show that the refinement is correct (a safety check)

open jacobi

-- restrict coarray sizes to prevent overflow
fun CoarraySmall : set Coarray {
  { c: Coarray | some m: Int | rel[m, c.mseq[0].cols, #c.mseq] }
}

assert correct {
  all c, c2: CoarraySmall, a, a2: Matrix |
    (alpha[c, a] and alpha[c2, a2] and JacobiStep[c, c2]) ⇒ JacobiStep[a, a2]
}

check correct for 7 but 4 Int

// using Lingeling on an M1 MacBook Air takes about 36 minutes

```

A.6 Progress

```

-- progress: show that the concrete operator (JacobiStep) is total

open jacobi

sig P {
  pre: Coarray,
  post: lone Coarray
}

```

```
}  
  
assert progress {  
  all p: P | JacobiStep[p.pre, p.post] ⇒ some p.post  
}  
  
check progress for 7 but 4 Int
```