Bounded Verification of Sparse Matrix Computations

Tristan Dyer
North Carolina State University
Raleigh, NC
atdyer@ncsu.edu

Alper Altuntas
National Center for Atmospheric Research
Boulder, CO
altuntas@ucar.edu

John Baugh
North Carolina State University
Raleigh, NC
jwb@ncsu.edu

***Abstract*—We show how to model and reason about the structure and behavior of sparse matrices, which are central to many applications in scientific computation. Our approach is state-based, relying on a formalism called Alloy to show that one model is a refinement of another. We present examples of sparse matrix-vector multiplication, transpose, and translation between formats using ELLPACK and compressed sparse row formats to demonstrate the approach. To model matrix computations in a declarative language like Alloy, a new idiom is presented for bounded iteration with incremental updates. Mechanical verification is performed using SAT solvers built into the tool.**

***Index Terms*—sparse matrix formats, state-based formal methods, mechanical verification.**

I. Introduction

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

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

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

Though important and challenging, static verification of sparse matrix software has received little attention. In a study addressing the problem, Arnold et al. [1] describe several attempts to do so, noting that they "failed to verify the functional correctness of even simple formats using several state-of-the-art tools," before creating a variable-free functional language in the style of FP to support verification with Isabelle/HOL.

In this paper, we develop and present a state-based approach for reasoning about sparse matrix computations and show how data abstraction and refinement principles can be used to check invariants and perform bounded verification of safety properties. We use Alloy [2], a lightweight formal method, to develop models that represent the structure and behavior of sparse matrices, and introduce a new idiom that supports the modeling of imperative loop structure, as is commonly found in scientific software.

To contrast our work with Arnold et al. [1], our models are intended to be more directly relatable to code in imperative programming languages like Fortran and C++; we rely on a formalism and tool whose application is more readily transferable to allied problems in scientific computing [3], allowing for economies of scale in their use; and because verification is bounded, our approach comes with push-button automation that does not require ingenuity in proving theorems.

The paper is organized as follows. Section II introduces the approach, the Alloy language, and notions of correctness and refinement. Sections III and IV show how matrix structure and behavior can be modeled and verified, with examples of ELL and CSR formats. Section V describes an idiom for bounded iteration and models for translation between sparse formats, matrix transpose, and matrix-vector multiplication. Section VI discusses scope, the ability to detect bugs, and limitations of the approach. Section VII describes related work, and Section VIII offers conclusions and directions for future research.

II. Approach

We make use of a state-based formalism called Alloy [2], a declarative modeling language that combines first-order logic and relational calculus, and includes associated quantifiers and operators, along with transitive closure. It offers rich data modeling features based on class-like structures and an automatic form of analysis that is performed within a bounded scope using a SAT solver. For *simulation*, the Alloy Analyzer can be directed to look for instances satisfying a property of interest. For *checking*, it looks for an instance violating an assertion:

978-1-7281-6015-3/19/$31.00 ©2019 IEEE
DOI 10.1109/Correctness49594.2019.00010

a counterexample. The approach is *scope complete* in the sense that all cases are checked within user-specified bounds. Alloy's logic supports three distinct styles of expression, that of predicate calculus, navigation expressions, and relational calculus. The language used for modeling is also used for specifying the properties to be checked.

Because Alloy is a structural modeling language it provides no means of representing real numbers or floating point values, and has only limited support for integers. In this study, we model zero and non-zero values relying only on the property that different numerical values are distinct, and checking for real equivalence of symbolic expressions when and as needed. In related work [3], our group uses predicate abstraction in Alloy models to factor out numerical concerns, allowing us to show that a performance enhancement made to a popular ocean model is equivalence preserving.

Instead of automatically generating verification conditions from code, we work with "abstract algorithms" that model the array indexing, mutation, and stateful behavior of programs written in imperative languages like Fortran and C++. Doing so makes the approach language agnostic and keeps verification tractable, since fine-grained control can be exercised over model details and scopes. The checks are not exhaustive, but we appeal to the *small scope hypothesis* [2], [4], which suggests that most bugs have small counterexamples.

A. Structure and Behavior

In state-based formalisms like Alloy, systems are described by defining what constitutes a state and the transitions between states. Though not an obvious choice for scientific software, such an approach is consistent with the perspective we advance: by separating concerns we can direct attention to structural and behavioral complexities that exist apart from the numerical ones [5].

With respect to structure, for instance, complex state is defined implicitly by declarative properties, in terms of sets, relations, and logical formulas. The Alloy Analyzer then serves as a *model finder* in the mathematical sense, finding models of logical formulas. What this means in practice is that fragments of scientific programs, existing or planned, can be put through their paces, with input state automatically generated to drive the model into its corner cases, should they exist. Such state might include, for instance, mesh topologies used in hurricane storm surge simulations, as developed in prior work [3], or sparse matrix formats, the subject of this paper.

In terms of behavior, operations are modeled as predicates that define transitions between states, which are also defined declaratively. Nondeterminism may be employed as a means of expressing concurrency [6] in models, which may be written in interleaving or noninterleaving styles. For "stateful" algorithms that rely on mutation, Alloy has no fixed idioms, but a common approach is to expand the arity of a "dynamic" relation by introducing a time column and imposing an ordering on time. We later introduce a complementary approach that works well for matrices and similar operations that rely on bounded iteration.

$$\begin{tikzcd}
A \arrow[r, "OP_A"] & A' \\
C \arrow[u, "\alpha"] \arrow[r, "OP_C"'] & C' \arrow[u, "\alpha"']
\end{tikzcd}$$

Fig. 1. Refinement commuting diagram.

B. Correctness and Data Refinement

Our notion of conformance is based on substitutability. A computer program written in terms of matrix computations, if correct, should remain correct if sparse matrix formats are used instead to improve performance. The historical origins of data refinement begin with Hoare [7] and proceed in two major veins—based on relational and predicate transformer semantics—with numerous representative examples including Reynolds' stepwise refinement of programs [8], Back and Morgan's refinement calculus [9], and Abadi and Lamport's refinement mappings [10]. More recently, Bolton [11] shows how data types in the Z notation can be translated into Alloy using an explicit encoding that is able to find refinement mappings automatically.

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

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

$$\begin{aligned} I(C) \wedge OP_C(C, C') \wedge \alpha(C, A) \wedge \alpha(C', A') \\ \Rightarrow OP_A(A, A') \end{aligned} \quad (1)$$

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

Abstract matrix operations, then, serve as a specification and are formulated declaratively using Alloy's set comprehension notation, as are abstraction functions. Sparse, concrete operations are often stateful, and for those we use a new idiom for bounded iteration that has intuitive appeal and an obvious relationship to code in imperative programming languages. Concrete invariants have a direct use as heap invariants [12] that must be satisfied by implementing programs.

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

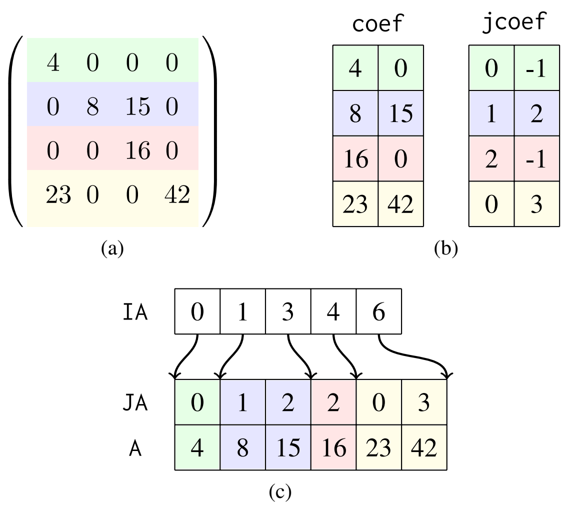

Fig. 2. A matrix in dense (a), ELL (b), and CSR (c) formats, with rows colored to show how elements are stored across formats.

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

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

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

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

Fig. 3. Matrix structure in Alloy: signatures for values and matrices in dense, ELL, and CSR formats.

In subsequent sections, we present portions of Alloy models to illustrate our approach, noting that complete models can be found online [13]. Also, although we introduce most major features of the language as we go, it may be helpful for those unfamiliar with it to consult the Alloy language reference, which is available online.[2]

III. Matrix Structure

Two commonly used sparse matrix formats are ELLPACK (ELL) and compressed sparse row (CSR), which we introduce here, looking first at their structure. Fig. 2 shows a matrix in dense, ELL, and CSR formats.

The ELL format, named for the ELLPACK library from which it originates, uses two two-dimensional arrays, `coef` and `jcoef`, as seen in Fig. 2b. The `coef` array stores matrix values and the `jcoef` array stores column indices for the corresponding values in `coef`. The dimension of each array is $rows \times maxnz$, where $maxnz$ is the maximum number of non-zero values in any single row of the matrix. Rows that contain fewer than $maxnz$ non-zero values are padded with placeholder values—negative one in the `jcoef` array and zero in the `coef` array.

The CSR format offers further compression of the ELL format by removing the values used for padding, as shown in Fig. 2c. To do so, the `coef` array is flattened into row-major order to produce the one-dimensional array `A`, and the same process is applied to the `jcoef` array to produce `JA`. To access individual rows, an indexing array, `IA`, contains the starting location of each row within the two arrays.

Turning to Alloy, matrix values and dense, ELL, and CSR representations are defined by the signatures shown in Fig. 3. A *signature* in Alloy introduces both a type and a set of uninterpreted atoms, and may introduce *fields* that define relations over them. In addition to defining a type, a signature's name can also be used within an Alloy expression to denote the set of elements it defines. Subtype signatures using *extends* introduce no new types but instead represent sets of elements that are subsets of their parents, and the *one* keyword denotes a singleton subset.

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

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

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

Abstract state, from a refinement perspective, is defined by the `Matrix` signature, which includes fields for the number of rows and columns and for dense storage. The `vals` field is a relation that denotes a two-dimensional array, mapping row and column indices to values; the multiplicity keyword *lone* (less than or equal to one) says there can be at most one such value for any index pair. The combination of Alloy's dot join and box join operators[3] means that, for a matrix `m`, the `i-j` element can be referred to as `m.vals[i][j]`, and if its value is `v`, the tuple `i→j→v` is a member of the `m.vals` relation.

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

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

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

the CSR format, `IA`, `JA`, and `A` fields denote their respective one-dimensional arrays.

At this point, the collection of signatures defined in Fig. 3 constitute a complete Alloy model. For validation, the Alloy visualizer can be used to step through and inspect instances, either textually or graphically, that are populated by atoms bounded by the individual scopes of the `Value`, `Matrix`, `ELL`, and `CSR` sets. When doing this, some of the instances produced correspond to valid formats, like those shown in Fig. 2, and some do not. For instance, some `Matrix` instances have `vals` with `i-j` indices out of bounds, some `ELL` instances have invalid column indices in `jcoef`, and so on.

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

IV. Matrix Behavior

To describe dynamic behavior in Alloy, operations are defined as predicates, or relations between states. Fig. 4 shows examples of some basic predicates, along with assertions to check their behavior.

The model fragment in the top half of Fig. 4 begins with a predicate `update`, a basic operation of the ELL format, which can be used to change a single value in the matrix—this might be a step in element assembly or matrix construction operations that are either provided by libraries or implemented by users. Parameters include pre- and poststate ELL matrices (`e`, `e'`), an index pair (`i`, `j`) specifying an element of the matrices, and a new value (`v`) for the element in the poststate.

Within `update`, the first line acts as a precondition, or guard, so that pre- and poststates are related only if indices are valid. An implication then makes use of helper predicates, depending on whether the new value for `v` is zero (shown) or non-zero (not shown). Finally, a frame condition [14] is defined by predicate `sameDimensions`: the number of rows and columns is unchanged in the transition. In `toZero`, the expression `k = e.jcoef[i].j` uses relational join to determine the column of `jcoef` that contains index `j` of row `i`, if it exists. The `setAt` predicate overrides the value of the `i-k` elements in `coef` and `jcoef`. To show that `update` preserves the concrete invariant for ELL matrices (not shown), an assertion `preservesInvariant` is provided.

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

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

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

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

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

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

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

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

Fig. 4. Matrix behavior in Alloy: the ELL update operation and invariant check (above), and ELL abstraction function and refinement check (below).

V. Matrix Computations

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

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

Fig. 5. Tabular pattern for nested loops defining an iteration table (`iter`) and time-indexed scalar variables (x, y), where ψ and ω define loop bounds.

A. An Idiom for Stateful Behavior

To accommodate stateful algorithms in declarative languages, idioms are often devised to address the so-called incremental update problem [15], such as state transformers [16] and lazy streams [17] in functional programming, array comprehensions and accumulators in dataflow and single assignment languages [18], [19], and a relational view taken of sparse matrix computations as database queries [20].

Although Alloy itself has no built-in notion of mutation, there are several techniques for modeling it [2], [21]. One common approach—a *local state* idiom—adds a column to a relation to make it "dynamic:" the addition serves as sort of a timestamp for the other columns in the relation. Then, using one of Alloy's built-in modules, a total ordering can be applied to the signature being used as a timestamp. Another approach more common to other state-based formalisms—a *global state* idiom—separates relations by placing them into different signatures based on whether they are considered static or dynamic, and using dynamic signatures for the pre- and poststate parameters of transition predicates.

For matrix computations, these approaches prove to be difficult to make work in practice because of complex interactions between nested loop structure, conditionals, and the exact scopes used by the ordering module. It is particularly important, then, to find idioms and design patterns for formal methods, where possible, that accommodate particular domain needs and broaden their range of applicability [22]. Here we describe a *tabular* idiom for the type of nested, bounded iteration commonly found in matrix computations.

Fig. 5 illustrates the pattern for two nested loops, with boilerplate that controls iteration over an innermost loop body. The existentially quantified expression binds an iteration table `iter` and some number of dynamic variables, such as x and y (of **`univ`** type, the universal set), which are indexed by an **`Int`** timestamp. The `table` predicate establishes the form and order of the table using an argument that defines loop bounds as a binary relation, and to which a time column is added. Time variables `t` and `t'` are used within the loop body to work with current and next values of the dynamic variables.

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

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

(a)

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

(b)

$$\begin{tikzcd}
 & m & \\
e \arrow[ur, "\alpha"] \arrow[rr, "ellcsr"'] & & c \arrow[ul, "\alpha"']
\end{tikzcd}$$

(c)

Fig. 6. ELL to CSR translation: (a) pseudo-code, (b) fragment of Alloy model, and (c) commuting diagram.

B. ELL to CSR Translation

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

To translate between ELL and CSR formats, as an example, recall from earlier descriptions that the CSR representation removes padding from rows that contain fewer than $maxnz$ values in the ELL format, as shown in Fig. 2. To allow for this, a third array containing the start location for each row, IA, is defined. The translation algorithm, shown in Fig. 6a, loops through the coef and jcoef arrays used in the ELL format, adding any non-zero values to the A and JA arrays used in the CSR format. A variable $kpos$ keeps track of the next available position in the CSR arrays. Once an inner loop completes, the value of $kpos$ is the starting location of the next row to be recorded in IA.

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

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

Using the tabular idiom, we distinguish between static and dynamic variables. The `A`, `JA`, and `IA` arrays are static since their elements are set just once. The variable kpos, however, is dynamic, so the following boilerplate is used ahead of the Alloy fragment shown in Fig. 6b:

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

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

As shown in Fig. 6c, the abstraction functions α for both the ELL and CSR formats are used to determine correctness, which we check as follows:

$$I(e) \land \mathit{ellcsr}(e, c) \Rightarrow (\alpha(e, m) \Leftrightarrow \alpha(c, m)) \tag{2}$$

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

C. CSR Transpose

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

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

which swaps i and j indices.

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

The algorithm uses two sets of arrays: A, JA, and IA arrays as input, and AO, JAO, and IAO arrays as output. Focusing on just the third phase, shown in Fig. 7a, the algorithm steps through rows of the input matrix, determines the current column j, finds the starting position nxt of that column in the output matrix using the IAO array, and updates those elements of the A and JA arrays. The starting location of that column is then incremented in the IAO array for the next iteration.

For the Alloy model, a fragment corresponding to phase 3 is shown in Fig. 7b, where we once again distinguish between static and dynamic variables. The j and nxt variables are temporary local variables, and the A and JA arrays are static since their elements are set just once. The iao array, however, is dynamic, since the j element is accessed and modified in each step of the inner loop. The following boilerplate is used:

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

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

(a)

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

(b)

$$\begin{tikzcd} m \arrow[r, "trans"] & m' \\ c \arrow[u, "\alpha"] \arrow[r, "trans"'] & c' \arrow[u, "\alpha"'] \end{tikzcd}$$

(c)

Fig. 7. CSR transpose, phase 3: (a) pseudo-code, (b) fragment of Alloy model, and (c) commuting diagram.

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

As illustrated in Fig. 7c, the concrete CSR transpose operation can be shown to conform to the abstract one, which we check as follows:

$$\begin{aligned} I(c) \land \mathit{trans}(c, c') \land \alpha(c, m) \land \alpha(c', m') \\ \Rightarrow \mathit{trans}(m, m') \end{aligned} \tag{3}$$

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

D. CSR Matrix-Vector Multiplication

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

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

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

(a)

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

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

(b)

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

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

(c)

$$\begin{tikzcd} & m \arrow[dr, "mvm_x"] & \\ c \arrow[ur, "\alpha"] \arrow[rr, "mvm_x"'] & & b \end{tikzcd}$$

(d)

Fig. 8. Matrix-vector multiplication: (a) sum of products for row i of matrix A and vector x, (b) dense dot product in Alloy, (c) CSR dot product in Alloy, and (d) commuting diagram.

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

The SumProd signature defines this particular kind of symbolic expression as a ternary relation of integers and value pairs, as shown below and illustrated in Fig. 8a.

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

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

To model matrix-vector multiplication, then, the result vector b is represented as a sequence of SumProds. A basic step in the algorithm that computes dot products is shown Figs. 8b and 8c for matrices in dense and CSR formats, respectively. In both cases, each product pair p-q is comprised of non-zero values (Value-Zero) to facilitate a comparison of expressions.

[5]See, for instance, the work of Siegel et al. [23], who build symbolic expression tables for checking real, IEEE, and Herbrand equivalence of general symbolic expressions using the Spin model checker.

As illustrated in Fig. 8d, the concrete CSR matrix-vector multiplication operation can be shown to conform to the abstract one, which we check as follows:

$$I(c) \wedge mvm(c, x, b) \wedge \alpha(c, m) \Rightarrow mvm(m, x, b) \qquad (4)$$

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

VI. Discussion

To perform the analyses, Alloy provides a number of SAT solvers. For simulation, we use MiniSat [24], an incremental SAT solver from Chalmers University of Technology, Sweden, and for checking, Lingeling [25] from Johannes Kepler University, Austria. All experiments are performed on a 3.5-GHz-Intel-Core-i7 desktop computer.

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

When using default scopes, simulations in Alloy are produced instantaneously, as are counterexamples when checking assertions, indicating, for instance, array referencing and indirection problems; matrices as small as 2×2 and smaller are typically sufficient. For successful checks, most are completed in a matter of seconds or under a minute, with the longest being the refinement check for CSR transpose, which takes a couple of hours. In practice, we often use smaller matrix sizes and larger numbers of distinct values.

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

$$I(c) \Rightarrow \exists\, c' \mid trans(c, c') \qquad (5)$$

which asserts that every CSR matrix c satisfying its invariant has a transpose. In practice, the applicability of operations can be spot-checked using Alloy's simulator and, for small scopes, generator axioms [2] can be used to populate terms in the poststate.

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

$$I(m) \wedge trans(m, m') \wedge trans(m, m'') \Rightarrow eqv(m', m'') \qquad (6)$$

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

VII. Related Work

Earlier we cite the studies of Arnold et al. [1] and Kotlyar et al. [20], both of which have compilation of sparse matrix codes as their primary objective. Beyond program synthesis, Arnold et al. also take up verification because of its potential role in discovering new formats via inductive synthesis. They write: "We are not aware of previous work on verifying full functional correctness of sparse matrix codes. We are not even aware of work that verified their memory safety without explicitly provided loop invariants."

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

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

VIII. Conclusions

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

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

Work with Alloy is proceeding in two major directions: applications and tool support. On the latter, we are developing a framework for sharing and visualizing Alloy instances that includes support for domain-specific customization, with spatial layouts that can accommodate planar embeddings of finite element meshes and matrices of various dimensions. We also imagine but have not implemented a layer of syntactic sugar that could reduce some of the boilerplate needed to express bounded iteration. With respect to applications, we are looking at more complex sparse matrix formats and parallelization, adding meshing and assembly concerns for hybrid and element-by-element solvers, and incorporating moving patches [26] and other types of adaptivity in finite element meshes.

IX. Acknowledgments

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

References

[1] G. Arnold, J. Hölzl, A. Sinan Köksal, R. Bodík, and M. Sagiv, "Specifying and verifying sparse matrix codes," *ACM SIGPLAN Notices*, vol. 45, pp. 249–260, 09 2010.

[2] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[26] A. Altuntas and J. Baugh, "Adaptive subdomain modeling: A multi-analysis technique for ocean circulation models," *Ocean Modelling*, vol. 115, pp. 86–104, 2017.