Bounded Verification of Sparse Matrix Computations

ACM Reference Format:
. 2019. Bounded Verification of Sparse Matrix Computations. In *Proceedings of Third International Workshop on Software Correctness for HPC Applications (Correctness 2019).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Sparse matrix computations are central to many applications in scientific computing. Sparse matrix data formats can be used to compress large matrices with a small number of non-zero elements into a more efficient representation. With roots in scientific computing, their usage has become adopted across a wide range of applications, including big data and machine learning. As such, there is a plethora of data formats capable of representing a sparse matrix, each presenting unique characteristics in terms of performance and usability. Commonly found representations include Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), Coordinate (COO), and Dictionary of Keys (DOK). New representations and modifications to existing ones are continually being created in order to take advantage of new hardware characteristics (cite) and new architectures such as GPUs (cite).

Coincident to sparse matrix representations are the operations that are performed on sparse matrices. These range from basic operations such as matrix addition and matrix-vector multiplication to more complex ones, such as column sorting and format conversion. Many of these operations are key components in libraries used across a wide range of applications. For example, sparse matrix-vector multiplication is a vital operation for solving linear systems of equations or computing eigenvalues.

Our own interests arose when optimizing ADCIRC, a popular large-scale ocean model, trying to assure that changes preserved correctness. Difficult to perform manually due to long run-times... testing is difficult and incomplete.

Introduce Alloy, a declarative language with automatic analysis, can perform simulations and checks... scientific software not a common application for state-based formal methods.

Our first use of Alloy was in subdomain modeling, can make incremental changes to storm surge models and perform simulations at an incremental computational cost, giving huge performance gains.

We are continuing work on ADCIRC++, working on internal algorithms, using new solvers, etc. and need to be able to reason about the particular representations required by solvers/algorithms.

Assembly from a finite element mesh works directly on sparse matrix data structures for performance reasons, sometimes hard to ensure correct in all cases, corner cases not missed.

Scope and organization of paper: Alloy and Bounded Verification, Sparse Matrix Representations, Sparse Matrix Operations, Matrix-Vector Multiplication, Format Translation, Matrix Transpose, Related Work, Conclusions.

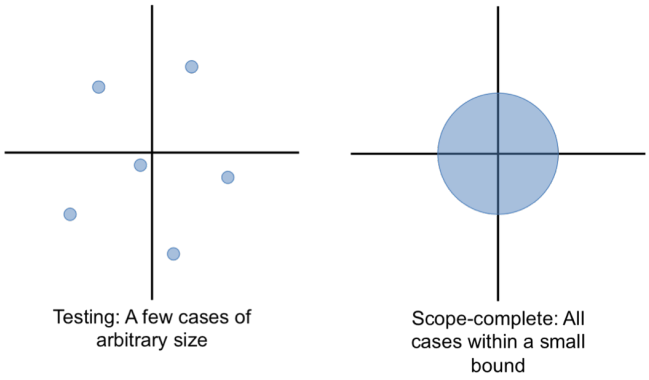

Figure 1: Illustration of Small Scope Hypothesis (cite)

2 ALLOY AND BOUNDED VERIFICATION

The tool used in our approach is Alloy, a declarative modeling language combining first-order logic with relational calculus and associated quantifiers and operators, along with transitive closure. It offers rich data modeling features based on class-like structures and an automatic form of analysis that is performed within a bounded scope using a SAT solver. For *simulation*, the analyzer can be directed to look for instances satisfying a property of interest. For *checking*, it looks for an instance violating an assertion: a counterexample. The approach is *scope complete* in the sense that all cases are checked within user-specified bounds. Alloy's logic supports three distinct styles of expression, that of predicate calculus, navigation expressions, and relational calculus. The language used for modeling is also used for specifying properties of interest and assertions. Alloy supports expressions with integer values and basic arithmetic operations.

Alloy cannot check all possible cases, but instead relies on the small-scope hypothesis, refer to Figure 1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Correctness 2019, November 2019, Denver, CO USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$Value = \{Zero, Value_0, Value_1, \ldots, Value_n\}$$

Figure 2: The set of abstract values.

Implicitness in specification; Alloy generates what might be viewed as input or data for operations. This means, e.g., arbitrary sparse matrices are generated; contrast this with unit testing in which inputs must be created manually. For numerical software, it is hard to perform 'parts' of a large-scale simulation by simply extracting code... have to create 'valid' data for input, populate data structures. In alloy we describe properties required of data and Alloy generates it for us, so in some ways this is easier than testing a component of a large-scale program.

Proof obligations: mathematical formula to be proven in order to ensure that a component is correct.

3 SPARSE MATRIX REPRESENTATIONS IN ALLOY

In this section we introduce some basic modeling elements used throughout the study and describe three models of sparse matrices. First we describe an abstract model of a mathematical matrix, leaving out all implementation details and notions of sparsity. Then we describe two refinements of this model which introduce structure that supports a sparse representation: the ELL format and the CSR format. Finally, we introduce the method of refinement and use it to show that the CSR format is a correct abstraction of the abstract data format.

By design, Alloy provides no means of working with floating point values. It includes integers, but in a limited scope, and so they are not useful when attempting to work with the complete integer set, as might be the case when considering values that could be stored in a sparse matrix. This is inconsequential, however, as we only aim to reason about the structural complexities of sparse matrix formats. Therefore, it is sufficient to create an abstract distinction between zero and non-zero values. Our models employ a Value signature, representing any numerical value, and a Zero signature, an extension of Value, that represents the value zero. These signatures can be found in Figure 3. Depending on the scope, this creates an abstract set of values, shown in Figure 2, that we can use to populate matrices in our models.

Throughout these models we make use of predicate overloading to define various predicates that can be applied universally. Overloading is enabled by Alloy's type system and type checker, which allow expressions to share a name as long as there is no ambiguity when resolving types. For example, the `range` predicate is used to generate a set of integers within some range. It takes two forms, one which accepts a single integer, n, and generates the set of integers $[0, n-1]$, and one which accepts two integers, i and j, and generates the set of integers $[i, j]$.

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

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

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

Figure 3: Abstract Matrix Model

3.1 Abstract Sparse Matrix

We begin with an abstract model of a two-dimensional mathematical matrix, sparse or otherwise, defined by the Matrix signature as shown in Figure 3. There are three fields defined on the Matrix signature representing (1) the number of rows in the matrix, (2) the number of columns in the matrix, and (3) the values and their locations in the matrix. This model makes no assumptions about the contents or structure of the matrix, and will be the arbiter of correctness when determining if a refinement is sound.

The representation invariant is specified in the predicate `repInv`. It states that the number of rows and columns in the matrix are each greater than or equal to zero, and that the set of index pairs used to store values is the complete mapping of row indices to column indices.

3.2 ELL Format

The ELL format, name after the ELLPACK library from which it originates, stores data in two arrays, each of size $rows \times maxnz$ where $maxnz$ is the maximum number of non-zero values that appear in any single column. The values array stores non-zero values and the columns array stores the column indices into which those values fall. Values and associated column indices stored at the same array index in each of the arrays. Each row of the matrix is allotted the same amount of space within the two arrays, and rows are indexed by multiplying the row index by the values of $maxnz$. Rows with fewer non-zero values than $maxnz$ are padded using a special value, negative one in our case. Although it is capable of storing any sparse matrix, the ELL format is most efficient when every row has the same number of non-zero values.

Our model of the ELL format is shown in Figure 4. The integer values `rows`, `cols`, and `maxnz` represent the number of rows, column, and the maximum number of non-zero values, respectively. The `vals` and `cids` fields, representing the arrays of values and column indices, are each represented using sequences. Sequences are an Alloy construct that allow for the declaration of a field as an ordered sequence of atoms.

```
sig ELL {
  rows, cols, maxnz: Int,
  vals: seq Value,
  cids: seq Int
}

pred repInv [e: ELL] {
  e.rows ≥ 0
  e.cols ≥ 0
  e.maxnz ≥ 0
  e.maxnz ≤ e.cols
  let sz = mul[e.rows, e.maxnz] |
    #e.cids = sz and #e.vals = sz
  all j: e.cids.elems |
    gte[j, -1] and lt[j, e.cols]
  all i: range[e.rows], j: range[e.cols] |
    let s = mul[i, e.maxnz],
        t = sub[add[s, e.maxnz], 1] |
      #e.cids.subseq[s, t].indsOf[j] ≤ 1
  all i: Int |
    e.cids[i] = -1 ⇔ e.vals[i] = Zero
}
```

Figure 4: The ELL Model.

This idiom is a convenient way to represent arrays, and is used throughout our models as such.

The representation invariant for the ELL format is specified in the `repInv` predicate. It states that an ELL instance is a valid ELL representation if the number of rows, columns, and maximum non-zero values are greater than or equal to zero, the maximum number of non-zero values is less than or equal to the number of columns, the value and column arrays are the same length, the column array contains valid indices or padding values, no column indices are repeated within a single row, and the value zero is stored at all columns containing a placeholder.

3.3 CSR Format

The Compressed Sparse Row (CSR) format, a description of which is shown in Figure 5, uses three arrays to store a sparse matrix, one for values and two for integers. The value array contains non-zero values of the matrix while column indices associated with those values are stored in a separate array. A third array stores the location in the values and column index arrays that begin each row. We adopt the convention that the values array is named A, the column array is named JA, and the row index array is name IA.

The CSR format, making no assumptions about the sparsity structure of the matrix, is a general format capable of compressing any sparse matrix. The storage savings for this approach is significant, requiring $2nnz + n + 1$ storage locations[1] rather than $n \times m$. Memory locality is improved for

[1] where n is the number of rows, m is the number of columns, and nnz is the number of non-zero values

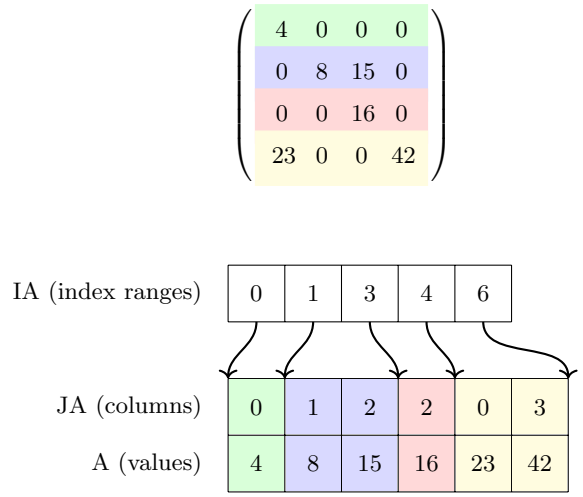

Figure 5: The CSR format.

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

pred repInv [c: CSR] {
  c.rows ≥ 0
  c.cols ≥ 0
  c.IA[0] = 0
  c.IA.last = #c.A
  #c.A = #c.JA
  #c.IA = add[c.rows, 1]
  Zero not in c.A.elems
  all i: range[c.rows] |
    c.IA[i] ≤ c.IA[add[i, 1]]
  all j: c.JA.elems | j in range[c.cols]
  all i: range[c.rows] |
    let a = c.IA[i], b = c.IA[add[i, 1]] |
      !c.JA.subseq[a, sub[b, 1]].hasDups
}
```

Figure 6: The CSR Model.

row access over the DOK format, but the indirect addressing steps can have an impact on performance [13].

Our model of the CSR format is shown in Figure 6. The integer values `rows` and `cols` represent the number of rows and columns, respectively. The `A` sequence of abstract values represents the value array, while the `JA` and `A` sequences of integers represent the column array and indexing array, respectively.

The representation invariant for the CSR format is specified in the `repInv` predicate. It states that a CSR instance is a valid representation of a sparse matrix if the following are true: (1) the number of rows and columns is greater than or

```
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[i.add[1]]] |
          c.JA[k] = j } |
            one k ⇒ v = c.A[k]
              else v = Zero
  }
}

assert repValid {
  all c: CSR, m: Matrix |
    repInv[c] and alpha[c, m] ⇒ repInv[m]
}
```

Figure 7: The CSR refinement check.

equal to zero, (2) the first value of the IA array is 0 and the last value of the IA array is equal to the number of values in the A array, (3) the number of values in the JA array is equal to the number of values in the A array, (4), the number of values in the IA array is equal to one more than the number of rows, (5) the value zero is not stored in the matrix, (6) the values in the IA array are monotonically increasing, (7) all column indices are in range, (8) and there are no repeated column indices within any single row.

3.4 Correctness of Sparse Matrix Formats

Both the ELL and CSR models provide representation invariants that define the properties of a valid ELL or CSR matrix. The question remains, however, whether these sparse formats are correct representations of an abstract matrix. By correct, we mean that these concrete representations exhibit all of the same properties as our abstract matrix. This notion of correctness, introduced by Hoare [] in 1972, allows one to reason about the correctness of a program that utilizes matrices without having to reason about implementation details associated with sparse matrix formats. Furthermore, the method is employed as part of the data refinement used to develop models of sparse matrix operations in Section 4.

The notion of conformance used in the Hoare checks for the correctness of a data refinement is one of inclusion, meaning that all concrete representations must map to an abstract one. This mapping, called an abstraction function, need not be a one-to-one mapping, and in many cases is many-to-one. While this method of determining the correctness of a concrete data type is sound, meaning that if an abstraction function exists it follows that the concrete representation conforms to the abstract one, the inverse it not necessarily true.

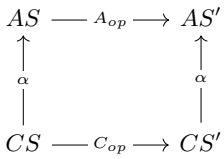

Figure 8: Data Refinement

The abstraction function, alpha, for the CSR matrix format is defined in Figure 7. It equates the number of rows and columns in the CSR representation to the number of rows and columns in the abstract representations and builds the set of row→column→values using set comprehensions. The use of set comprehensions here is deliberate. By determining exactly the set of values in the field, we are not required to include any frame conditions. Our experience in developing these models has been that as a model increases in complexity, so do the frame conditions, and that incorrectly implemented frame conditions are harder to catch than incorrectly implemented set comprehensions. Therefore, the use of set comprehensions for this type of operation is less error-prone.

To check within scope that the abstraction function is valid and that the CSR format is a correct refinement of the abstract matrix format, the repValid assertion states that for all CSR matrices, c, and all abstract matrices, m, if c is a valid CSR representation and maps to m through the abstraction function alpha, then it follows that m is a valid abstract matrix.

4 SPARSE MATRIX OPERATIONS IN ALLOY

Sparse matrix operations are used extensively in scientific computation for things like solving systems of equations and calculating eigenvalues.

While sparse matrix representations are used to reduce required storage space, operations on sparse matrices are designed to be efficient in time, often packaged in libraries written in low-level imperative languages.

To validate the correctness of a sparse matrix operation, we use data refinement. To model a data refinement in Alloy we relate the pre- and post-states of a concrete operation to the pre- and post-states of the abstract operation through the abstraction function, α. The diagram in Figure 8 demonstrates this relationship. Conceptually, this means that if we begin with some concrete representation, CS, we can choose to follow the arrows of the diagram in either direction and we will always end up at the same abstract state, AS'. We either perform the concrete operation and apply the abstraction function to the result or we first apply the abstraction function and then perform the abstract operation. Both paths around the diagram are equivalent.

Algorithms that use sparse matrices typically have nested loop structures due to presence of array indirections embedded within sparse matrix formats, e.g. CSR.

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \times \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{A_0 \cdot x_0} \\ A_1 \cdot x_1 \\ A_2 \cdot x_2 \end{Bmatrix}$$

(a)

$$\boldsymbol{A_0 \cdot x_0} = A_{00} * x_0 + A_{01} * x_1 + A_{02} * x_2$$

(b)

0	A_{00}	x_0
1	A_{01}	x_1
2	A_{02}	x_2

(c)

Figure 9: (a) a matrix-vector multiplication, (b) the components of the first dot product in (a), and (c) the relational form of the same dot product.

This type of structured loop presents unique challenges from a modeling perspective. We have developed a new idiom for handling the type of bounded iteration and evolving state typical of these algorithms, which is realized in three different approaches:

(1) Set comprehensions for simple cases that are easier to reason about, handling iteration "en masse"
(2) For internal state, such as indexing values or arrays, that does not evolve in a predictable manner due to either branching or some property of the looping values, using the `some` quantifier to generate the sequence of values subject to the appropriate constraints.
(3) For nested dependencies in which the iterating indices of the inner loop are dependent on the iterating indices of the outer loop, generate an i→k→t table subject to the appropriate constraints.

The following sections make use of each of these methods in order of growing complexity. Matrix-vector multiply has no time-varying state, format translation requires only the first time-varying method, and transpose makes use of the second time-varying method.

5 MATRIX-VECTOR MULTIPLICATION

In this section we model abstract matrix-vector multiplication and CSR sparse-matrix vector multiplication (SpMV) and demonstrate the refinement check needed to verify its correctness.

The result of the matrix-vector multiplication $\boldsymbol{A}x = b$ is a densely populated vector, y, in which each value is the dot product of a single row of the matrix A with the vector x. A dot product is simply a sum of products in which each product is of two values located at the same index within two vectors, as shown in Figure 9b.

```
sig SumProd {
  vals: Int→lone Value→Value
} {
  all i: vals.univ.univ | i ≥ 0
}
```

Figure 10: SumProd Signature

In order to compare the result of an abstract matrix-vector multiplication with the result of a concrete one, we must be able to compare arrays of dot products. Rather than compare numerical values, we wish to compare the vectors dot products as a composition of the values. To do so we introduce the `SumProd` signature, shown in Figure 10. To represent a sum of products, `SumProd` contains a single field, `values`, that maps unique integers to `Value` pairs. Through this field, the `SumProd` signature is able to represent a sum of products as a table in which each row contains a product of two values, and the summation is composed of all rows, as shown in Figure 9c. Additionally, the integer value associated with each row indicates the index from which the values in that row originate. To represent the vectors b and x in the matrix-vector multiplication $\boldsymbol{A}x = b$, we use a sequence of `SumProd`s and a sequence of `Value`s, respectively.

The matrix-vector multiplication models for both the abstract and CSR format are found in Figure 11. In both cases, the dimensions of the matrix and vectors are tested for compatibility in a matrix-vector multiplication, and dot products are generated for the solution in a row-wise fashion. Note that the `dotProd` predicate does not include products that will evaluate to zero in the resulting SumProd. As such, the dot products will be equivalent regardless of whether the vector originates from a dense or sparse representation. Further note that the same `dotProd` predicate is used in both the abstract and CSR cases. This is enabled by the `getrow` helper function which extract sets of column→value pairs in a row-wise fashion from a CSR matrix.

The refinement check is performed by the `refines` predicate. Alloy has shown this refinement to be correct for matrices up to 5×5 with 10 unique values.

The nature of the refinement check here is similar to the square diagram above, but need to show it precisely as it varies slightly.

This same method of data refinement has been used to model SpMV for the ELL format as well.

6 SPARSE MATRIX FORMAT TRANSLATIONS

The translation of a sparse matrix from one format to another may be necessary in a number of scenarios. For example, it is common practice to assemble a sparse matrix using a format that is efficient for incremental assembly and to translate the fully assembled matrix to a format that is efficient for

```
pred MVM [A: Matrix, x: seq Value,
          b: seq SumProd] {
  A.rows = #b
  A.cols = #x
  all i: range[A.rows] |
    dotProd[A.vals[i], x, b[i]]
}

pred MVM [C: CSR, x: seq Value,
          b: seq SumProd] {
  C.rows = #b
  C.cols = #x
  all i: range[C.rows] |
    let row = getrow[C, i] |
      dotProd[row, x, b[i]]
}

pred dotProd [row: Int→Value,
              x: seq Value,
              b: SumProd] {
  b.vals = { i: row.univ, j, k: Value-Zero |
              j = row[i] and k = x[i] }
}

fun getrow [c: CSR, row: Int]: Int→Value {
  let cols = rowcols[c, row],
      vals = rowvals[c, row] | ~cols.vals
}

pred refines [n: Int] {
  all C: CSR, M: Matrix, x: seq Value,
      Cb, Mb: seq SumProd |
    C.rows = n and C.cols = n and
    mulBoth[C, M, x, Cb, Mb] ⇒ vecEqv[Cb, Mb]
}

pred mulBoth [C: CSR, M: Matrix, x: seq Value,
              Cb, Mb: seq SumProd] {
  repInv[C] and repInv[M] and alpha[C, M]
    and MVM[C, x, Cb] and MVM[M, x, Mb]
}

pred vecEqv [a, b: seq SumProd] {
  #a = #b
  all i: a.inds |
    a[i].vals = b[i].vals
}
```

Figure 11: Matrix-Vector Multiplication for Abstract and CSR.

use in a solver. In the context of ADCIRC++, we wish to compare the performance of a number of solvers in a number of different storm surge scenarios. Due to differences in solution technique in combination with floating point error propagation, the results generated by various solvers may not be exactly equivalent. Therefore, in order to ensure that the same problem is being solved in each test case, a comparison of solutions is not sufficient. Rather, we must show that the same matrix–i.e., the same system of equations–is being used across solvers, regardless of format.

ADCIRC and ADCIRC++ both use the ELL format in conjuction with the ITPACKV solver. The format most commonly used in the solvers we wish to test (TODO: ITPACK, Pardiso, etc.) is the Compressed Sparse Row (CSR) format, and so we will model the ELL to CSR translation in this section. A similar approach is to be taken for other formats, including the Block Compressed Sparse Row (BSR) used in the cuSPARSE library, and the Compressed Sparse Column (CSC) format used in the Armadillo library.

```
1 kpos = 0
2 for i in range(nrows):
3 |   for k in range(maxnz):
4 |   |   idx = i * maxnz + k
5 |   |   if cols[idx] != -1:
6 |   |   |   A[kpos] = vals[idx]
7 |   |   |   JA[kpos] = cols[idx]
8 |   |   |   kpos = kpos + 1
9 |   IA[i + 1] = kpos
```

Algorithm 1: The ELL to CSR translation algorithm.

An ELL to CSR translation algorithm is provided by the SPARSEKIT [14] library, and a Pythonic listing of this algorithm is found in Algorithm 1. The algorithm works by iterating over each stored column of each row in the ELL data structure. If it is non-zero, the current value and its column index are copied in to the next available location in the corresponding CSR arrays, and the ending IA index for that row is incremented.

The translation from ELL to CSR in Alloy is performed by quantifying the state of the algorithm at every step in the innermost loop. As seen in Algorithm 1, there are three variables used for indexing, i, k, and *kpos*, that need to be quantified. The values of i and k are easily generated in Alloy using an **all** statement, but the value of *kpos* is less straightforward. It can optionally increase by one on line 8, at the end of the innermost loop, depending on whether or not the current matrix value is nonzero. As a result, the value of *kpos* at line 9, where an IA value is set, may not be the same as the value of *kpos* at lines 6 and 7, where values of A and JA are set. Therefore, we model the value of *kpos* as a sequence of integers in which the value at time t is the value of *kpos* before line 8 and the value at time $t + 1$ is the value of *kpos* after line 8.

This sequence is generated using the `some` quantifier, as seen in `ellcsr` predicate in Figure 12.[2] Then, at any instant

[2]The *kpos* sequence is generated using **some**, which here is a higher-order quantification. Alloy cannot perform higher-order quantification, but is able to eliminate it in some cases, such as this one, by using skolemization. We expect there to be only a single possible sequence

```
pred ellcsr [e: ELL, c: CSR] {
  e.rows = c.rows
  e.cols = c.cols
  c.IA[0] = 0
  #c.IA = c.rows.add[1]
  some kpos: seq Int {
    kpos[0] = 0
    #kpos = e.rows.mul[e.maxnz].add[1]
    #c.A = kpos.last
    #c.JA = kpos.last
    all i: range[e.rows] {
      all k: range[e.maxnz] {
        let kp = i.mul[e.maxnz].add[k],
            kp' = kp.add[1] {
          rowcols[e, i][k] != -1 ⇒ {
            c.A[kp] = rowvals[e, i][k]
            c.JA[kp] = rowcols[e, i][k]
            kpos[kp'] = kpos[kp].add[1]
          } else {
            kpos[kp'] = kpos[kp]
          }
        }
      }
      c.IA[i.add[1]] = kpos[i.add[1].mul[e.maxnz]]
    }
  }
}

assert translationValid {
  all e: ELL, c: CSR |
    repInv[e] and ellcsr[e, c] ⇒ repInv[c]
}
```

Figure 12: ELL to CSR Translation Model

the value of *kpos* in the next instant be determined within the loop based on the conditionals, which have been modeled using implication. With the sequence generated, the value of *kpos* at any instance during the execution of the algorithm can now be determined using the values of i and k.

For a translation to be considered valid, the final state must preserve the representation invariant of the target format as well as represent the same abstract matrix as the original format. Intermediate states, however, do not need to preserve the representation invariant. The Alloy assertion used to model this relationship is shown in Figure 12, which states that if the ELL matrix e is a valid representation of the abstract matrix m and the translation is applied, giving the CSR matrix c, then it follows that c is a valid CSR matrix that represents the abstract matrix m. The scope used for this check shows that the translation is valid for matrices up to 6×6.

of *kpos* values for any given translation, and so the **one** quantifier should be used instead. This quantification cannot be eliminated using skolemization, and so we have verified this property using Alloy*, an extension of Alloy that permits higher-order quantification [15].

TRANSPOSE PSEUDOCODE

Figure 13: CSR Transpose

7 MATRIX TRANSPOSE

Abstract matrix transpose can be performed "en masse" using set comprehension. The CSR transpose algorithm, on the other hand, involves nested loops with dependencies, shown in Figure 13.

Have modeled transpose using a method similar to the `kpos` method used in ELL to CSR translation that does, but have also demonstrated a way to generate a table of iteration steps – still in development and method is being refined.

The algorithm consists of four distinct steps: (1) counting of columns, (2) calculation of pointers, (3) copying of values, and (4) shifting of pointers. An array, *iao*, evolved continuously through this process and is used in all four steps. We are mostly concerned with the third step, as it contains an inner loop that has a dependency on the outer loop, seen in Algorithm 2.

```
for i in range(n):
    for k in range(ia[i], ia[i + 1]):
        j = ja[k]
        nxt = iao[j]
        ao[nxt] = a[k]
        jao[nxt] = i
        iao[j] = next + 1
```

Algorithm 2: The copy values step in CSR transpose

The first and fourth steps make use of set comprehensions, the second step uses a common modeling paradigm to generate values, but the third step requires the final method introduced in this paper. Rather than generate a sequence of integers, as in kpos method, generate a sequence of arrays. This method demonstrated in the `copyvals` predicate in Figure 14. May move this method of generating a sequence of arrays to be included with the kpos method, depending on the final form of the table-of-iterations method.

8 RELATED WORK

Arnold et al. [11] design a functional language and proof method for the automatic verification of sparse matrix codes. Their "little language" (LL) can be used to specify sparse codes as functional programs in which computations are sequences of high-level transformations on lists. These models are then automatically translated into Isabelle/HOL where they can be verified for full functional correctness. The authors use this automated proof method to verify a number of sparse matrix formats and their sparse matrix-vector multiplication operations. Their approach is purely functional, relying on typed λ-calculus and Isabelle libraries to perform proofs using Isabelle/HOL. LL, a shallow embedding in Isabelle/HOL, provides simple, composable rules that can be used to fully describe a sparse matrix format, removing the

```
pred copyvals [c, c': CSR, iao, iao': seq Int] {
  #iao' = #iao
  some ia: Int→Int→Int {
    -- init ia and array lengths
    0→iao in ia
    #ia.Int.Int = add[#c.A, 1]
    all i: ia.Int.Int | #ia[i] = #iao
    -- iterate
    all i: range[c.rows] {
      all k: range[c.IA[i], c.IA[i.add[1]]] {
        let t = k, t' = t.add[1],
            j = c.JA[k], nxt = ia[t][j] {
          ia[t'] = ia[t] ++ j→nxt.add[1]
          c'.JA[nxt] = i
          c'.A[nxt] = c.A[k]
        }
      }
    }
    iao' = ia[c.IA.last]
  }
}
```

Figure 14: CSR Transpose

burden of directly writing proofs. In quantifying rule reuse, the authors 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, a significant level of reuse. Even of the rules needed for more complex formats (CSC and JAD), only up to a third are format-specific." [11] The method enables a significant amount of rule reuse, but does not entirely prevent one from having to write some amount of λ-calculus. Filling in these gaps may prove difficult for those without a strong background in functional programming and theorem proving. Our approach, on the other hand, may appeal to an audience of scientists and engineers, who are accustomed to working with models anyway, and with the kind of automatic, push-button analysis supported by Alloy, those who develop software can focus on modeling and design instead of theorem proving.

Arnold approach does not support modeling of assembly of sparse matrices. Relational approach in combination with state change allows us to model the assembly process as well as updates to matrix values. Relational approach also allows for modeling of format conversions.

9 CONCLUSIONS AND FUTURE WORK

Future work to include:

- Newer, more complex formats, hybrid formats–citing "A hybrid format for better performance of sparse matrix-vector multiplication on a GPU"
- Parallelization of algorithms
- Include mesh representations in order to model the assembly process, for the purposes of optimization
- Complexities of GPU problems

REFERENCES

[1] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, VictorEijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc Web page," https://www.mcs.anl.gov/petsc, 2019. [Online]. Available: https://www.mcs.anl.gov/petsc

[2] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[3] A. Bik and H. Wijshoff, "Advanced compiler optimizations for sparse computations," *Journal of Parallel and Distributed Computing*, vol. 31, no. 1, pp. 14 – 24, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731585711410

[4] A. J. C. Bik and H. A. G. Wijshoff, "Automatic data structure selection and transformation for sparse matrix computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 2, pp. 109–126, Feb 1996.

[5] V. Kotlyar, K. Pingali, and P. Stodghill, "A relational approach to the compilation of sparse matrix programs," in *Euro-Par'97 Parallel Processing*, C. Lengauer, M. Griebl, and S. Gorlatch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 318–327.

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

[7] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1512–1542, 1994.

[8] J. Dingel and T. Filkorn, "Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving," in *Computer Aided Verification: 7th International Conference, CAV '95 Liège, Belgium, July 3–5, 1995 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 54–69. [Online]. Available: https://doi.org/10.1007/3-540-60045-0_40

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

[10] J. Woodcock, *Using Z : specification, refinement, and proof*. London New York: Prentice Hall, 1996.

[11] G. Arnold, J. Hlzl, A. Sinan Kksal, R. Bodik, and M. Sagiv, "Specifying and verifying sparse matrix codes," *ACM SIGPLAN Notices*, vol. 45, pp. 249–260, 09 2010.

[12] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[13] J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, Z. Bai, Ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000.

[14] Y. Saad, "Sparskit: a basic tool kit for sparse matrix computations - version 2," 1994.

[15] A. Milicevic, J. P. Near, E. Kang, and D. Jackson, "Alloy*: A general-purpose higher-order relational constraint solver," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 609–619.

[16] T. Dyer, "Article github repository," 2019. [Online]. Available: https://github.com/atdyer/alloy-lib