Sparse Matrices Correctness 2019 Paper

ACM Reference Format:
. 2019. Sparse Matrices Correctness 2019 Paper. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Sparse matrix data formats can be used to compress large matrices with a small number of non-zero elements into a more efficient representation. Scientific and engineering software that make use of sparse matrix formats are often implemented in low-level imperative languages such as C++ and Fortran. The optimized nature of these software often means that the structural organization of sparse matrix formats and mathematical computations are heavily intertwined. Additionally, the myriad data formats and complexities involved in tuning the operations on these formats to achieve efficient implementations means the development of software that utilizes sparse matrices can be a tedious and error-prone task.

Often, the solvers provided by popular linear algebra libraries used in scientific computing require assembled sparse matrices as input, leaving the entire assembly process to the developer. Considering the problem domains in which they are used, this can quickly become a complex task. For example, sparse matrices are often used to represent the meshes used in the finite element and finite difference methods. In order to represent physical properites, these meshes can contain rich datasets, the values of which may depend on the complex spatial relationships of the mesh components. As a result, the matrix assembly process can quickly become intertwined with the mathematics required to determine the precise values that will be placed in the matrix. Furthermore, as performance improvements are made by, for example, exploiting parallelism in modern hardware, the assembly process and embedded calculations can become even more complex.

To address the difficulties in assembling a sparse matrix, object-oriented libraries such as PETSc [1] and Eigen [2] provide data abstractions targeted towards specific classes of solvers. These libraries allow developers to assemble sparse matrices without having to address the structural complexities that a sparse matrix format may present. However, the utility of these libraries may be limited due to solver limitations or other performance restrictions.

Alternatively, compiler support may help with the development and performance tuning of software that makes use of sparse matrices. Some compilers [3, 4] allow developers to work with dense matrices in code, generating a sparse program at compile time. Others [5] allow the user to provide the compiler with an abstract description of a sparse format, from which the compiler can make automatic optimizations in code.

To mitigate the introduction of errors into code during development, techniques such as unit testing and test-driven development are often employed. These methods aim to reveal bugs by testing the individual components of a piece of software during development. While useful, these methods are both tedious, requiring test cases to be written manually, and incomplete, meaning they are incapable of checking every scenario or combination of scenarios.

We describe an approach that allows developers to reason about the inherent complexities of sparse matrix formats and operations without limiting implementation choices. Elements of this approach include declarative modeling and automatic, push-button analysis using the Alloy Analyzer [6], a lightweight bounded model checking tool. Characteristics of sparse matrix formats and operations are modeled using abstraction based methods [7], including data [8] and predicate [9] abstraction, data and functional refinement [10], and other techniques, manually, as part of a modeling process.

The benefits of this approach lie in its generality. In contrast to code, Alloy allows for partial descriptions in which details are hidden behind abstractions. This allows us to, for example, reason about the complexities of building a sparse matrix without the burden of calculating the specific values that matrix will hold. Furthermore, in being able to freely choose the level of abstraction, we are able to model a wide range of —, from specific algorithms written in low-level imperative languages to higher level, abstract concepts.

- partial descriptions - level of abstraction - structure and behavior

The remainder of this paper is organized as follows. In Section 2 we discuss related work. In Section 3 we describe certain elements of lightweight formal methods and how they will be applied to our problem domain using Alloy. In Section 4 we model three matrix formats, including one abstract matrix and two sparse matrix formats. In Section 5 we model the translation of a sparse matrix from the ELL format to the CSR format, and in Section 6 we model a sparse matrix-vector multiplication algorithm for the CSR format. Finally, in Section **??** we conclude with future directions and closing remarks.

2 RELATED WORK

Arnold et al. [11] design a functional language and proof method for the automatic verification of sparse matrix codes.

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 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.

3 LIGHTWEIGHT FORMAL METHODS

... An additional aspect of lightweight formal methods is an incremental style of modeling, which tools like Alloy support by offering immediate feedback while models are being constructed: we start with a minimal set of constraints and “grow” them via conjunction.

3.1 Alloy

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.

$$\begin{tikzcd}
AS \arrow[r, "AF"] & AS' \\
IS \arrow[u, "\alpha"] \arrow[r, "IF"] & IS' \arrow[u, "\alpha"]
\end{tikzcd}$$

Figure 1: TODO: An abstract state (AS) becomes new abstract state (AS') through an abstract function (AF). An implemented state (IS) becomes a new implemented state (IS') through an implemented function (IF). The implemented states are related to the abstract states through the abstraction function α.

3.2 Data Abstraction

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

Start example here.

3.3 Data Refinement

The process of data refinement involves removing nondeterminism, or uncertainty, from an abstract model [10]. While an abstract model may omit certain design choices, a refinement can resolve some of these choices, removing uncertainty and approaching the level of a concrete implementation.

Abstraction function and representation invariant discussion.

Continue example with refinement here.

4 SPARSE MATRIX MODELS

In this section we introduce some basic modeling elements used throughout the study, and then we describe four models. First we describe an abstract model of a matrix that does not represent any specific data format, leaving out all implementation details and notions of sparsity. Then we describe three refinements of this model which introduce structure that supports a sparse representation: the DOK format, the ELL format, and the CSR format. For each matrix representation we model initialization, update, and matrix-vector multiplication. For each refinement, we provide representation invariants and abstraction functions to show that the refinement is sound.

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,

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

Figure 2: The set of abstract values.

```
pred rowInRange[d: DOK, row: Int] {...}
pred rowInRange[e: ELL, row: Int] {...}
pred rowInRange[c: CSR, row: Int] {...}
pred colInRange[d: DOK, col: Int] {...}
pred colInRange[e: ELL, col: Int] {...}
pred colInRange[c: CSR, col: Int] {...}
```

Figure 3: Predicate overloading of `rowInRange` and `colInRange`.

representing any numerical value, and a Zero signature, an extension of Value, that represents the value zero. 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 `rowInRange` and `colInRange` predicates are used to determine if a row or column index is within the bounds of some matrix. The predicate definitions for the three sparse matrix types considered in this study are shown in Figure 3. The `rowInRange` predicate evaluates to true if the value `row` is greater than or equal to zero and less than the number of rows in, for example, the DOK matrix `d`, false otherwise. Similarly, the `colInRange` predicate evaluates to true if the value `col` is greather than or equal to zero and less than the number of columns in the DOK matrix `d`. Additional usage of overloading found in these models includes the representation invariant, which is always named `repInv`, and the abstraction function, which is always named `alpha`.

4.1 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 4. 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 as a signature fact so that it is applied to every member of the Matrix signature. The representation invariant states that (1) the number of rows and columns in a matrix are each greater than or equal to zero, (2) all row, column indices fall within bounds, (3) the total number of values in the matrix is `rows` × `cols`, and (4) there is a value at every (i, j) index pair.

```
sig Matrix {
  nrows, ncols: Int,
  values: Int → Int → Value
} {
  nrows ≥ 0
  ncols ≥ 0
  all i, j: Int |
    i → j in values.univ ⇒
      0 ≤ i and i < nrows and
      0 ≤ j and j < ncols
  let nvals = mul[nrows, ncols] |
    #values = nvals and
    #values.univ = nvals
}
```

Figure 4: The abstract matrix model.

```
pred init [m: Matrix, rows, cols: Int] {
  m.nrows = rows
  m.ncols = cols
  let valset = m.values[univ][univ] |
    valset = Zero or no valset
}
```

Figure 5: The abstract matrix initialization.

```
pred update [m, m’: Matrix,
            row, col: Int,
            val: Value] {
  m.rows = m’.rows
  m.cols = m’.cols
  rowInRange[m, row]
  colInRange[m, col]
  let curr = m.values[row][col] |
    m’.values =
      m.values
      - row→col→curr
      + row→col→val
}
```

Figure 6: The abstract matrix update.

4.1.1 Matrix Initialization. The `init` predicate shown in Figure 5 is used to initialize an empty matrix. We consider the initialized state of a matrix to be one in which the number of rows and columns is defined and all values are zero.

4.1.2 Matrix Update. The `update` predicate shown in Figure 6 is used to perform an update. We consider a matrix update to be a transition in which a single value of a matrix is changed and the matrix does not change size. For all matrices, this includes four possible transitions: non-zero to non-zero, non-zero to zero, zero to non-zero, and zero to zero, or stutter.

```
sig DOK {
   nrows, ncols: Int,
   dict: Int → Int → Value
}
```

Figure 7: The DOK Model.

```
pred repInv [d: DOK] {
   Zero not in d.dict[univ][univ]
   all i, j: Int {
      i→j in d.dict.univ ⇒
         rowInRange[d, i] and
         colInRange[d, j]
   }
   all i, j: Int {
      rowInRange[d, i] and
      colInRange[d, j] ⇒
         lone v: Value | i→j→v in d.dict
   }
}
```

Figure 8: The DOK representation invariant.

4.2 DOK Format

The dictionary of keys (DOK) format uses a dictionary, or associative array, to store key, value pairs. An associative array is a collection of key, value pairs in which each possible key may appear only once. Sparse matrices are stored such that pairs of row, column indices are used as keys to reference stored values. The DOK format is frequently used [2, 12] in the assembly of sparse matrices because of the efficient $\mathcal{O}(1)$ access to individual elements provided by the associative array format. However, because associative arrays make no guarantees about memory locality, iterating over all values in the matrix can be inefficient. As a result, it is common to use this format to incrementally assemble the full matrix and convert it to another sparse matrix format that allows for more efficient operations on matrices.

The DOK signature, shown in Figure 7, resembles the Matrix signature. The `nrows` field represents the integer number of rows in the matrix and the `ncols` field represents the integer number of columns in the matrix. The `dict` field represents the dictionary of key, value pairs. It takes the same form as the values field of the Matrix signature, but the **Int** → **Int** relation represents integer pairs used as keys in the dictionary.

4.2.1 Representation Invariant. The representation invariant for the DOK format, shown in Figure 8, states that (1) there are no zeros stored as values, (2) all row, column indices used as keys fall within the bounds of the matrix, and (3) all row, column indices that fall within the bounds of the matrix may appear at most once as keys in the dictionary.

4.2.2 Abstraction Function. The abstraction function provides a mapping from the DOK representation of a matrix to our abstract representation of a matrix. The `alpha` predicate, shown in Figure 9, states that the DOK matrix d is a refinement of Matrix m if (1) d has the same number of rows and columns as m, (2) all row, column index pairs that map to a value in the d dictionary map to the same value at the same location in m, and (3) all in-range row, column pairs *not* used as keys in the d dictionary map to zeros in m.

```
pred alpha [d: DOK, m: Matrix] {
   m.rows = d.rows
   m.cols = d.cols
   all i, j: Int, v: Value |
      i→j→v in d.dict ⇒
         i→j→v in m.values
   all i, j: Int |
      rowInRange[d, i] and
      colInRange[d, j] and
      i→j not in d.dict.univ ⇒
         i→j→Zero in m.values
}
```

Figure 9: The DOK abstraction function

```
pred init [d: DOK, rows, cols: Int] {
   d.nrows = rows
   d.ncols = cols
   no d.dict
}
```

Figure 10: The DOK initialization predicate.

```
pred update [d, d’: DOK,
             row, col: Int,
             val: Value] {
   d.rows = d’.rows
   d.cols = d’.cols
   rowInRange[d, row]
   colInRange[d, col]
   let curr = d.dict[row][col] {
      ZtoZ[d, d’, curr, val] or
      ZtoNZ[d, d’, row, col, curr, val] or
      NZtoZ[d, d’, row, col, curr, val] or
      NZtoNZ[d, d’, row, col, curr, val]
   }
}
```

Figure 11: The DOK update predicate.

4.2.3 Matrix Initialization. Initialization of the DOK matrix, shown in Figure 10, requires setting the size of the matrix and creating an empty dictionary. Because the abstraction function maps indices that aren’t included as keys to zero, an empty dictionary represents a matrix in which all values are zero.

4.2.4 Matrix Update.

4.2.5 SpMV.

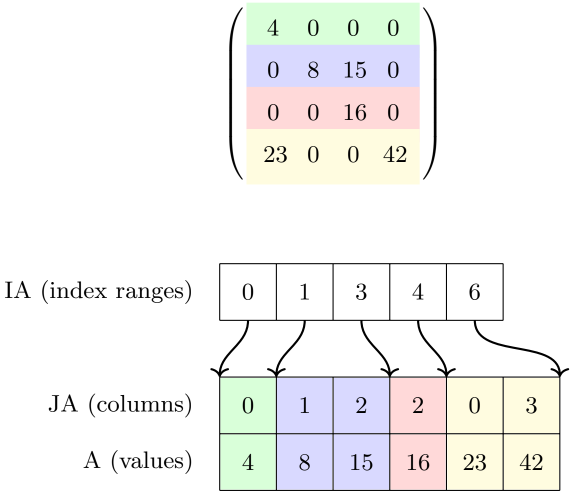

Figure 12: The CSR format.

```
sig CSR {
  nrows, ncols: Int,
  A: seqValue,
  IA, JA: seqInt,
}
```

Figure 13: The CSR model.

4.3 ELL Format

ELL, coming soon.

4.4 CSR Format

The Compressed Sparse Row (CSR) format, shown in Figure 12, 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 ordered as they are traversed in a row-wise fashion. Column indices of each value are stored in a separate array, while a third array stores the location in the values array that starts 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 row access over the DOK format, but the indirect addressing steps can have an impact on performance [13].

Need to describe the “get” predicate and why it is needed.

The abstraction function:

The representation invariant:

```
fun get [c: CSR, row, col: Int]: Value {    let a =
c.IA[row],
        b = c.IA[add[row, 1]] {
    (no a or no b or a = b) ⇒ Zero
    else {
      let j = c.JA.subseq[a, sub[b, 1]],
          v = c.A.subseq[a, sub[b, 1]],
          i = j.idxOf[col] {
        no i ⇒ Zero else v[i]
      }
    }
  }
}
```

Figure 14: The get function used in the CSR model.

```
pred alpha [c: CSR, m: Matrix] {
  m.nrows = c.nrows
  m.ncols = c.ncols
  all i, j: Int |
    rowInRange[c, i] and
    colInRange[c, j] ⇒
      m.values[i][j] = get[c, i, j]
}
```

Figure 15: The CSR abstraction function.

```
pred repInv [c: CSR] {
  Zero not in y.A.elems
  all i: y.IA.rest.elems |
    gte[i, 0] and
    lte[i, mul[y.rows, y.cols]]
  all j: y.JA.elems |
    gte[j, 0] and
    lt[j, y.cols]
  y.IA[0] = 0
  y.IA.last = #y.A
  #y.IA > 1 ⇒
      gt[y.IA.last, y.IA.butlast.last]
  #y.A = #y.JA
  lte[#y.A, mul[y.rows, y.cols]]
  lte[#y.IA, add[y.rows, 1]]
  all i: y.IA.inds |
    i > 0 ⇒
      let a = y.IA[sub[i, 1]],
      b = y.IA[i],
      n = sub[b, a] {
        b ≥ a
        n ≤ y.cols
        n = #y.JA.subseq[a, sub[b, 1]].elems
      }
}
```

Figure 16: The CSR representation invariant.

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

5 MODELING 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 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 Figure 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 index *it* is the value of *kpos* before line 8 and the value at index $it + 1$ is the value of *kpos* after line 8. This sequence is generated using the **genkpos** predicate found in Figure 17. With the sequence generated, the value of *kpos* at any location inside

```
pred genkpos [e: ELL, kpos: seq Int] {
    kpos[0] = 0
    #kpos = e.rows.mul[e.maxnz].add[1]
    all i: rowInds[e] {
        all k: indices[e.maxnz] {
            let it = i.mul[e.maxnz].add[k]{
                rowcols[e, i][k] != -1 ⇒ {
                    it < kpos.lastIdx ⇒
                        kpos[it.add[1]] = kpos[it].add[1]
                } else {
                    it < kpos.lastIdx ⇒
                        kpos[it.add[1]] = kpos[it]
                }
            }
        }
    }
}
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{
        genkpos[e, kpos]
        #c.A = kpos.last
        #c.JA = kpos.last
        all i: rowInds[e] {
            all k: indices[e.maxnz] {
                rowcols[e, i][k] != -1 ⇒
                let kp = kpos[i.mul[e.maxnz].add[k]] {
                    c.A[kp] = rowvals[e, i][k]
                    c.JA[kp] = rowcols[e, i][k]
                }
            }
            c.IA[i.add[1]] = kpos[i.add[1].mul[e.maxnz]]
        }
    }
}
assert transValid {
    all e: ELL, c: CSR, m: Matrix |
        repInv[e] and alpha[e, m] and ellcsr[e, c] ⇒
            repInv[c] and alpha[c, m]
}
check transValid
    for 4 Int, 7 seq, 1 Matrix, 1 ELL, 1 CSR, 2 Value
```

Figure 17: The ELL to CSR Model

of the algorithm can now be determined using the values of i and k.

The **ellcsr** predicate generates a sequence of *kpos* values using the **genkpos** predicate[2] and performs the translation using the same indexing pattern used in the algorithm. Indeed, the value of *kpos* could be generated directly within the **ellcsr** predicate, but we found that the separation made it easier to reason about two key components separately: (1) the nondeterministic evolution of the *kpos* value and (2) the correct indexing of values.

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 17, 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.

5.1 In-Place Translation

In SPARSEKIT [14], the ELL to CSR translation is a Fortran subroutine that receives five arrays as input: two arrays, COEF and JCOEF, representing the ELL format, and three arrays, A, JA, and IA, representing the CSR format. The target CSR arrays must be allocated before calling the subroutine, and must contain enough memory to store the sparse matrix. The only way to ensure that there is enough memory allocated is to either count the number of non-zero values in the ELL matrix before calling the subroutine or naively allocate enough memory to hold a densely populated ELL matrix (one in which there are no padding values). This presents two possible performance issues: (1) counting the number of non-zero values requires looping through the entire ELL matrix, and (2) depending on the size and sparsity of the matrix, allocating enough memory to perform the translation may not be possible on certain hardware.

An in-place translation from ELL to CSR could sidestep these issues, provided there is no requirement that the original ELL matrix be valid after the translation occurs. During the translation, all padding values are removed from COEF and JCOEF to produce A and JA, respectively, and so we can infer that in the case where there are no padding values present, the arrays will be equivalent. Therefore, we know that the maximum amount of memory required to store A and JA is equal to the amount required to store COEF and JCOEF. So if an in-place translation is possible, the number of non-zero values is not required and the only memory allocation required is for the IA array, the size of which is always one more than the number of rows in the matrix.

```
assert inPlace {
  all e: ELL, kpos: seq Int{
    repInv[e] and genkpos[e, kpos] ⇒ {
      all i: rowInds[e] {
        all k: indices[e.maxnz] {
          let idx = i.mul[e.maxnz].add[k] |
            kpos[idx] ≤ idx
        }
      }
    }
  }
}
check inPlace
  for 5 Int, 15 seq, 0 Matrix, 1 ELL, 0 CSR, 2 Value
```

Figure 18: Alloy assertion that the ELL to CSR translation can be performed in-place.

```
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             vals[kpos] = vals[idx]
7             cols[kpos] = cols[idx]
8             kpos = kpos + 1
9     IA[$i + 1$] = kpos
```

Algorithm 2: The in-place ELL to CSR translation algorithm.

Algorithm 2 shows the in-place translation. It is equivalent to Algorithm 1, except for lines 7 and 8 which are modified to perform the translation in-place. In order to determine if this algorithm performs a valid translation, we must show that once a value has been written to an index, that index is not read from in subsequent iterations. In this algorithm, the values used to index into the arrays, *kpos* and *idx*, are always increasing. At every iteration, the value of *idx* increases by exactly one and the value of *kpos* optionally increases by one. Therefore, we can conclude that *kpos* indices being used to overwrite existing values will always be less than or equal to *kpos* indices being used to read existing values. This means that once a value is written to an index, that index will never again be used by the algorithm to read that value, and so an in-place translation is possible.

This property can be formally verified using Alloy, as shown in Figure 18. The assertion used here states that for valid ELL matrices up to size 15×15, the value of *kpos* is less than or equal to the value of *idx* at every iteration of the translation algorithm. Indeed, the assertion holds and so we

[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 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].

have formally verified that within scope, the algorithm can be performed in-place. While visual inspection and careful reasoning about the algorithm led us to the same conclusion, the benefits of using Alloy here are twofold: (1) a formal verification provides confidence that our reasoning is sound, and (2) this type of modeling gives valuable insight into how to reason about the types of algorithms commonly found in scientific and engineering software, many of which may not be as easy to analyze analytically.

6 SPARSE MATRIX-VECTOR MULTIPLICATION

Sparse linear algebra libraries and the methods they employ are vitally important in the domain of scientific computing (TODO: why?). Of particular interest is the sparse matrix-vector product, which solves $y = \boldsymbol{A}x$ where $\boldsymbol{A}$ is a sparse matrix and the vectors y and x are dense. Because SpMV usage is often highly repetitive within, e.g., iterative solvers, performance improvements via novel algorithms and the exploitation of modern hardware are areas of ongoing research (TODO: citations).

...why is SpMV difficult? Things like array indirection, varying sparsity patterns, it's memory bound...

As modern hardware introduces higher levels of parallelism and SpMV algorithms become more complex, the likelihood of introducing subtle bugs becomes decidedly higher. In this section we present a method for reasoning about the structural complexities and verifying the correctness of SpMV algorithms. We first model an abstract matrix-vector multiplication and subsequently demonstrate that a CSR SpMV algorithm is a valid functional refinement.

6.1 Abstract Matrix-Vector Multiplication

In order to demonstrate that an SpMV algorithm is correct, we must first create a model of matrix-vector multiplication to act as the arbiter of correctness. The result of the matrix-vector multiplication $y = \boldsymbol{A}x$ 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 19b.

Thus far, our matrix and sparse matrix models are capable of representing the structure of the matrix and vectors, as well as the values contained in $\boldsymbol{A}$ and x. However, because we wish to model the *structural* behavior of the algorithm, the resulting vector, y, cannot simply contain abstract values. Given a solution generated by a matrix-vector multiplication, we need to be able to verify that the *composition* of a value is correct; that each value present in each dot product originates from the correct location. This is enabled by using the `SumProd` signature, introduced in Section 6.1.1, along with the following predicates: (1) to generate dot products, we introduce the `dotProd` predicate in Section 6.1.2, (2) to determine the equivalence of dot products we introduce the `valEqv` predicate in Section 6.1.3, and (3) to perform an abstract matrix-vector multiplication, we introduce the `MVM` predicate in Section 6.1.4.

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

(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 19: (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.

6.1.1 *Sum of Products Model.*

The `SumProd` signature found in Figure 20a represents the result of a dot product operation, a sum of products. Because each dot product evaluates to a single value that must be stored in one of our existing matrix models, the `SumProd` signature extends `Value`. Note that this evaluation is never actually realized, we merely wish to assemble values such that they represent a sum of products. To do so, `SumProd` contains a single field, `values`, that maps unique integers to `Value` pairs. We do not allow `SumProd`s to nest, as we only wish to model dot products of numerical values, not complex numerical expressions, and so `SumProd` is removed from the set of values that may be stored by using the expression `(Value-SumProd)`. Using the `values` 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 19c. Additionally, the integer value associated with each row indicates the index from which the values in that row originate.

6.1.2 *Dot Product Model.*

The `dotProd` predicate found in Figure 20b assembles the dot product of `row` and `x` into the `SumProd b`. These three arguments warrant an explanation, as their usage may not be immediately clear.

First, the relation `row` represents a single row from an abstract `Matrix`. Recall that a `Matrix` stores values using an `Int`→`Int`→`Value` relation, in which the first and second integer represent the row and column, respectively, in which the corresponding value falls. Therefore, when we reference all values in a single row, the first integer is removed, leaving us with a relation of the form required by `row`, which

```
sig SumProd extends Value {
    values: Int
        → lone (Value-SumProd)
        → (Value-SumProd)
} {
    all i: values.univ.univ | i ≥ 0
}
```

(a)

```
pred dotProd [row: Int→Value, x: Matrix, b: SumProd]
{
    all col: row.univ |
        b.values[col] = row[col]→x.values[col][0]
    all col: Int- row.univ |
        no b.values[col]
}
```

(b)

```
pred valEqv [x, y: Value] {
    (x = y)
    or (x in SumProd and isZero[x] and y = Zero)
    or (y in SumProd and isZero[y] and x = Zero)
    or (isZero[x] and isZero[y])
    or (x in SumProd and
        y in SumProd and
        removeZeros[x] = removeZeros[y])
}
```

(c)

Figure 20: (a) The `SumProd` signature, representing a summation of products, (b) the `dotProd` predicate, which assembles a dot product, and (c) the `valEqv` predicate, which determines if two values are equivalent.

is `Int→Value`. In this relation, the integer represents the column in which the corresponding value falls.

Second, the argument `x` must be a `Matrix`, representing the column vector that is being multiplied by the matrix in our matrix-vector multiplication. Note that we do not enforce certain properties of `x`, namely that it contain a single column and the same number of rows as the matrix has columns. It is assumed that these checks are performed in concert with the use of the `dotProd` predicate.

Finally, the argument `b` is a `SumProd`, representing the dot product of `row` and `x`. The set of values contained by this `SumProd` is fully enumerated within the predicate using the two `all` statements. The first one states that for each column index, `col`, in the current row of the matrix, the `col`-th value of the dot product summation is equal to the row value at index `col` times the vector value at (`col`, 0). The second states that any integer that is *not* a column index also does not show up in the set of `SumProd` values.

```
pred MVM [A, x, b: Matrix] {
    ...
    all i: rowInds[A] |
        dotProd[A.values[i], x, b.values[i][0]]
}
```

Figure 21: The abstract matrix-vector multiplication model.

6.1.3 Dot Product Equivalence. When we perform a matrix-vector multiplication, the result is a column vector in which each value is a single `SumProd`. So, in order to compare the results of two different matrix-vector multiplication algorithms, we need to be able to determine the equivalence of individual `SumProd`s. Additionally, we must be able to determine if a `SumProd` is equal to `Zero`, so as not to preclude an algorithm from skipping the assembly of a dot product should it know that a row of the matrix is empty. Finally, because differing algorithms may be operating on storage structures that contain differing numbers of zeros, we expect the composition of dot products to also contain differing numbers of zeros. However, we do expect all dot products to contain equivalent sets on nonzero values originating from the same locations. Therefore, to determine the equivalence of two values, the `valEqv` predicate, shown in Figure 20c, compares only nonzero values when it encounters a `SumProd` value. Note the use of a few helper functions, namely `isZero` and `removeZeros`. The `isZero` function evaluates to true if the `SumProd` contains no nonzero values, and the `removeZeros` function returns the set of nonzero values contained in a `SumProd`. These and other helper functions can be found in the supplemental repository [16].

6.1.4 Matrix-Vector Multiplication Model. Finally, we are able to model a matrix-vector multiplication using the `MVM` predicate shown in Figure 21. This predicate receives three matrices, `A`, `x`, and `b`. For brevity, a number of lines have been removed from the `MVM` predicate. These lines, which can be found in the supplemental repository [16], simply ensure that `x` and `b` each only contain a single column and that the dimensions of all three matrices are able to represent a valid matrix-vector multiplication. Additionally, we enforce that there are no `SumProd`s present in either `A` or `x`. As mentioned previously, we are only interested in modeling matrix-vector multiplications that consist of simple numerical values.

The `all` statement in the `MVM` predicate performs the actual matrix-vector multiplication. It simply states that the value located at row `i` in the solution vector `b` is equal to the dot product of row `i` of the matrix `A` with the vector `x`.

6.2 CSR Matrix-Vector Multiplication

In this section we model a CSR matrix-vector multiplication algorithm and demonstrate that it is a valid functional refinement of the matrix-vector multiplication algorithm modeled in Section 6.1.

```
pred MVM [C: CSR, x, b: Matrix] {
    . . .
    all i: rowInds[C] |
        let row = getrow[C, i] |
            dotProd[row, x, b.values[i][0]]
}
```

(a)

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

(b)

```
assert refines {
    all C: CSR, M, x, mb, cb: Matrix |
        . . .
        alpha[C, M] and
        MVM[C, x, cb] and
        MVM[M, x, mb] ⇒ matEqv[cb, mb]
}
```

(c)

Figure 22: (a) The CSR matrix-vector multiplication model, (b) the getrow helper function, which extracts the column→value pairs of a row in a CSR matrix, and (c) the predicate used to check for refinement.

Recall that the CSR format contains three arrays. The arrays A and JA are equal in length and contain values and the column indices at which those values appear, respectively. The third array, IA, contains integers used to index into A and JA. Each consecutive pair of indices represents a subset of the A and JA arrays corresponding to a single row of the matrix. The first pair of indices represents the first row, the second pair represents the second row, and so on.

```
1 for $i$ in range($nrows$):
2     for $j$ in range(IA[$i$], IA[$i+1$]):
3         val = A[$j$]
4         col = JA[$j$]
5         b[$i$] = b[$i$] + val * x[col]
```

Algorithm 3: The CSR MVM algorithm.

Algorithm 3 performs the matrix-vector multiplication of a CSR matrix with the vector x, resulting in the vector b. In this algorithm, the inner loop is responsible for iterating over consecutive pairs of indices in the IA array. Within this inner loop, the current value and column are extracted from the A and JA arrays, and the dot-product is accumulated at the appropriate location.

We model Algorithm 3 in the MVM predicate found in Figure 22a. This predicate shares much in common with the abstract model, still requiring that matrix and vector dimensions are valid for a matrix-vector multiplication. These lines, just as in the abstract model, have been removed for brevity.

The row loop in the CSR model is modeled in the same way as the row loop in the abstract model, by using an all expression. Before the inner loop, however, this model uses the getrow helper function to extract the columns and values from the CSR data structure. This helper function, found in Figure 22b, models the array dereferencing found in lines 3 and 4 of Algorithm 3. Additionally, it returns the full set of column→value pairs for a given row. This allows us to reuse the dotProd predicate found in Figure 20b. Note that the getrow function makes use of two additional helper functions, rowcols and rowvals, which extract the sequence of columns and values, respectively, for a given column. Both can be found in the supplemental repository [16].

Finally, we check that our algorithm correctly performs a matrix-vector multiplication in the refines assertion, found in Figure 22c. First, the lines that have been removed simply enforce the representation invariant for all of the arguments passed to the predicate. Next, two matrix-vector multiplications are assembled: $\boldsymbol{M} * x = m_b$ and $\boldsymbol{C} * x = c_b$. Additionally, the CSR matrix $\boldsymbol{C}$ and abstract matrix $\boldsymbol{M}$ are made equivalent using the abstraction function alpha. The assertion states that if $\boldsymbol{M}$ and $\boldsymbol{C}$ represent the same matrix and are multiplied by the same vector, x, then their respective solution vectors, m_b and c_b are equivalent.

This assertion has been tested for up to 5×5 matrices, and is found to be valid. Therefore, we can conclude that within scope Algorithm 3 is correct.

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