Sparse

I. Introduction

Sparse matrix data formats are able 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 that the development of software that makes use of sparse matrices is a tedious and often error-prone task.

Need to add something about how verification of sparse matrix codes is also difficult, and has only really been done by Arnold. This is one of our major contributions: sparse matrix codes are difficult to verify (see Arnold), and the development of representation invariants for a number of formats can make this easier. The representation invariants can be directly translated into code, and used during the development and debugging process to verify that operations on matrices never violate the representation invariants (a la Liskov).

A number of approaches have been taken in order to address these issues. Object-oriented libraries such as PETSc [1] and Eigen [2] provide data abstractions targeted towards specific classes of solvers. These libraries provide templates that allow developers to assemble sparse matrices without having to address the structural complexities that a specific format may present. These matrices can then be used in a variety of solvers, given that the format is supported.

Alternatively, there is a body of research that takes the approach of designing and building compilers capable of automatically making decisions about sparse matrix formats and operations. Some compilers [3], [4] allow developers to work with dense matrices in code, generating a sparse matrix 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 that accesses the sparse data.

We describe an approach that allows developers to reason about the inherent complexities of sparse matrix formats and operations and to determine invariants that can be used to verify implementations. 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 matrices, with their numerous representations and ...hard-to-get-right-implementation-details... are approached using abstraction based methods [7], including data abstraction [8] and predicate abstraction [9], data and functional refinement [10], and other techniques, manually, as part of a modeling process...

The benefits of this approach lie in its generality. By using Alloy to perform the modeling and analysis, the modeler may choose the programming language that best fits their needs when transitioning from model to implementation. Can reason about existing software. Can reason about design of new sparse matrix libraries. Can reason about compiler design.

II. Related Work

Verifying sparse matrix code. 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 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.

III. 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.

A. 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.

B. Data Abstraction

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

Start example here.

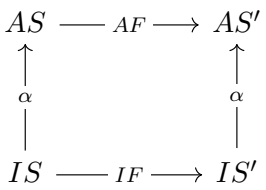

Fig. 1. TODO: An abstract state (AS) becomes new abstract state (AS') through an abstract function (AF). An implemented state (IS) becomes a new implemented state (IS') through an implemented function (IF). The implemented states are related to the abstract states through the abstraction function α.

C. 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.

IV. 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, 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

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

Fig. 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] {...}
```

Fig. 3. Predicate overloading of `rowInRange` and `colInRange`.

Fig. 1, 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 Fig. 2. 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`.

A. 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 Fig. 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 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` $\times$ `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
}
```

Fig. 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
}
```

Fig. 5. The abstract matrix initialization.

1) Matrix Initialization: The `init` predicate shown in Fig. **??** 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.

```
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
}
```

Fig. 6. The abstract matrix update.

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

Fig. 7. The DOK Model.

2) Matrix Update: The `update` predicate shown in Fig. 4 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.

B. 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 Fig. 5, 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.

1) Representation Invariant: The representation invariant for the DOK format, shown in Fig. **??**, 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.

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 Fig. **??**, 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.

3) Matrix Initialization: Initialization of the DOK matrix, shown in Fig. **??**, 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.

```
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
  }
}
```

Fig. 8. The DOK representation invariant.

```
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
}
```

Fig. 9. The DOK abstraction function

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

Fig. 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]
  }
}
```

Fig. 11. The DOK update predicate.

4) Matrix Update:

5) SpMV:

C. ELL Format

ELL, coming soon.

D. CSR Format

The Compressed Sparse Row (CSR) format, shown in Fig. 7, 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:

V. Sparse Matrix Code

Show how representation invariants can be used in code.

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

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

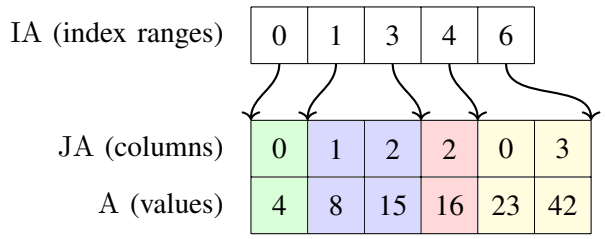

Fig. 12. The CSR format.

```
sig CSR {
  nrows, ncols: Int,
  A: seq Value,
  IA, JA: seq Int,
}
```

Fig. 13. The CSR model.

```
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]
      }
    }
  }
}
```

Fig. 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]
}
```

Fig. 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
      }
}
```

Fig. 16. The CSR representation invariant.

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.