Sparse

Tristan Dyer John Baugh

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

Libraries. Compilers. Specifying sparse matrix code. Verifying sparse matrix code. Higher order verification.

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.

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.

Diagram of refinement here.

Abstraction function and representation invariant discussion.

Continue example with refinement here.

IV. Sparse Matrix Models

In this section 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 that introduce structure that supports a sparse representation: the DOK format, the ELL format, and the Yale format. In each of these refinements we determine the appropriate representation invariants as well as the abstraction functions and show that the refinement is valid.

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" that we can use to populate matrices in our models.

```
sig Matrix {
  rows, cols: Int,
  values: Int → Int → Value
}
```

Fig. 1. Abstract Matrix Model.

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

A. Abstract Sparse Matrix

What is the specification we are modeling? 2D matrix of numerical values: zero or more rows, zero or more columns, a numerical value at every row, column index location.

Our abstract model of a matrix, sparse or otherwise, is defined by the Matrix signature, shown in Fig. 1. 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 alone is an underspecified representation of a matrix as it allows, e.g. negative values for the number of rows and columns. We must include a representation invariant, defined as follows, in order to restrict the allowable set of objects to those that adhere to the specification of a two-dimensional matrix.

- The number of rows and columns are each greater than or equal to zero
- All index values fall within bounds
- The total number of values in the matrix is $nrows \times ncols$
- Every i, j index pair appears in the matrix

This completes our abstract model of a matrix. It can be used to represent any two-dimensional matrix that contains values. Being our highest level of abstraction, it makes no assertions about what values may be included in the matrix or how those values must be stored.

Show an instance here, using visualizations from Alloy Instances preferably.

B. DOK Format

The dictionary of keys (DOK) format makes use of dictionaries, or associative arrays, to store key, value pairs. An associative array is a collection of key, value pairs in which each possible key may appear only once.

sig DOK {
  rows, cols: Int,
  dict: Int → Int → Value
}

Fig. 2. DOK Model.

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], [11] 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 is almost identical to the Matrix signature. The $rows$ field represents the integer number of rows in the matrix and the $cols$ 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 $Int \to Int$ relation represents integer pairs used as keys in the dictionary.

In order to limit the scope of instances to only those that accurately represent the properties of an associative array that stores a sparse matrix, we have created the following representation invariant:

- The integer value $rows \geq 0$
- The integer value $cols \geq 0$
- There are no zeros stored as values
- All row, column pairs fall within bounds
- All row, column pairs may appear only once

The abstraction function:

- $nrows$ equal to number of rows in matrix
- $ncols$ equal to number of columns in matrix
- All $(i, j), v$ key, value pairs map to corresponding values in the matrix
- All (i, j) pairs *not* in DOK map to zeros in the matrix

C. ELL Format

ELL, coming soon.

D. Yale Format

The Yale, or frequently, Compressed Sparse Row (CSR) format, uses three arrays to store a sparse matrix.

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

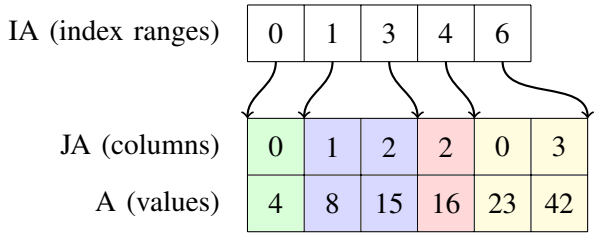

Fig. 3. Visual description of the Yale format.

sig Yale {
  rows, cols: Int,
  A: **seq** Value,
  IA, JA: **seq** Int,
}

Fig. 4. Yale Model.

The array IA stores indices into the other two arrays. Each consecutive pair of indices in the IA array represents the range of values in the other two arrays that store the data for a single row in the matrix. So, for example, the first two values in the IA array represent the range of values in the other two arrays that contain data for the first row of the matrix. The A array contains values while the JA array contains the corresponding column in which each value of the A array falls.

Description of where and why it is used... scientific software, good memory locality for sparse matrix-vector multiplication.

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

The abstraction function:

- $nrows$ equal to number of rows in matrix
- $ncols$ equal to number of columns in matrix
- For all integers $i \in [0, nrows)$ and $j \in [0, ncols)$:
  - The value of the "get" predicate at (i, j) is equal to the value at location (i, j) in the matrix.

The representation invariant:

- The integer value $nrows \geq 0$
- The integer value $ncols \geq 0$

- There are no zeros stored as values
- All values in IA $\leq nrows \times ncols$
- All values in JA $< ncols$
- The first value in IA is zero
- The last value in IA is the length of A
- The last value of IA is not repeated
- A and JA are the same length
- The max length of A is $nrows \times ncols$
- The max length of IA is $nrows + 1$
- The values in IA are monotonically increasing
- The difference between any two consecutive values in IA must be $\leq ncols$
- Values in JA must be unique per row

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