

Sparse

Tristan Dyer

John Baugh

I. INTRODUCTION

Computations on sparse matrices are prevalent in scientific and engineering software. Sparse matrix data formats are able to compress large matrices with a small number of non-zero elements into a more efficient representation. The development of software that makes use of sparse matrices is a tedious and often error-prone task because of the myriad data formats and complexities involved in tuning the operations on these formats to achieve an efficient implementation.

Characterize sparse matrix codes...written in imperative languages like C and Fortran...structure of sparse format interspersed with computation.

A number of approaches have been taken in order to address these issues. Object-oriented libraries such as PETSc [?] and Eigen [?] 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 [?], [?] allow developers to work with dense matrices in code, generating a sparse matrix program at compile time. Others [?] 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.

Rather than hiding the complexities of sparse matrix formats by providing data abstractions in code or automatic optimizations in compilers, we propose an approach that allows developers to reason about these complexities. Elements of this approach include declarative modeling and automatic, push-button analysis using the Alloy Analyzer [?], 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 [?], including data abstraction [?] and predicate abstraction [?], data and functional refinement [?], 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. 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.

Value Model.

Fig. 1. Abstract Model of Values

C. Data Refinement

The process of data refinement involves removing non-determinism, or uncertainty, from an abstract model [?]. 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.

III. 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 CSR 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.

A. Abstract Sparse Matrix

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. The model in Fig. 1 contains a Value signature, representing any numerical value. The Zero signature, an extension of Value, 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.

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

- Number of rows and columns greater than or equal to zero
- All index values fall within bounds
- Total number of values in the matrix is $nrows \times ncols$

- Every i, j index pair appears in the matrix

B. DOK Format

The dictionary of keys (DOK) format makes use of dictionaries, or associative arrays, to store key, value pairs. 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 [?], [?] 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.

- There are no zeros stored as values
- All row, column pairs fall within bounds
- All row, column pairs may appear only once

C. ELL Format

ELL.

D. CSR Format

CSR.

- 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