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 [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] aim to provide the compiler with an abstract description of a sparse format, allowing it to 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 [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. 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 this research 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

Abstract Model.

Fig. 2. Abstract Sparse Matrix Model.

C. Data Refinement

The process of data refinement involves removing non-determinism, 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.

III. Sparse Matrix Models

In this section we describe four models of sparse matrices. First we describe an abstract model of a sparse matrix that refrains from including any implementation specific details; it does not directly describe any specific sparse matrix format. Then we describe three refinements of this model: the first is a model of the DOK format, the second is a model of the ELL format, and the third is a model of 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

Before modeling sparse matrices, we first create an abstraction of the values that they contain, as shown in Fig. 1. Because the distinction that separates sparse matrices from dense ones is the removal of zeros, we represent values as either zero or non-zero. The Value signature represents any numerical value and the Zero signature, and extension of Value, represents the value zero. So depending on the scope, the set of values available is

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

An abstract model of a sparse matrix is given in Fig. 2.

B. DOK Format

DOK.

C. ELL Format

ELL.

D. CSR Format

CSR.

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.