Verifying a Sparse Matrix Algorithm Using Symbolic Execution

Alexander C. Wilton
Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716, USA
`awilton@udel.edu`

Scientific software is, by its very nature, complex. It is mathematical and highly optimized which makes it prone to subtle bugs not as easily detected by traditional testing. We outline how symbolic execution can be used to write tests similar to traditional unit tests while providing stronger verification guarantees and apply this methodology to a sparse matrix algorithm.

1 Introduction

Scientific software has become ubiquitous across almost every field of science due to continuous advancements in computing. This type of software is usually designed to take on massive problems whose results are often used in critical decisions, making its correctness paramount.

However, this is not always easy. Scientific software is typically very mathematical and highly optimized which leaves room for subtle bugs that are not easily caught with traditional testing [11].

Lightweight static analysis tools can be used but they are typically too inaccurate to be useful in complex settings [10]. Heavyweight approaches to verification such as mechanized proofs can work in theory, but they involve an enormous amount of specialized effort to create and maintain [15].

An intermediate approach to verification is through *symbolic execution* [1, 3, 6, 13]. This is a technique which simulates execution of a program using symbolic expressions for its values instead of concrete values. This allows for a potentially infinite number of inputs to be reasoned about at once.

In this paper we outline how symbolic execution can be used in scientific software similarly to unit testing while providing stronger verification guarantees. We demonstrate this on a matrix-vector multiplication algorithm adapted from [12] using the CIVL model checker and symbolic execution tool [4, 18]. The tricky part is that this algorithm works on matrices stored in the widely used *compressed row storage* (CRS) format, while the vector is dense which means it is just stored as an array. The CRS format is more commonly referred to as the *compressed sparse row* (CSR) format. However, we opted to use this less common name to match the language used in the paper [12] that the multiplication algorithm is borrowed from.

In the rest of the paper we provide a brief overview of symbolic execution and CIVL in section 2. Then we outline our approach to verifying the CRS multiplication algorithm in Section 3 with an emphasis on the general approach to using CIVL in this way. Finally we give concluding remarks in Section 4.

Submitted to:
VSS 2025

2 Symbolic Execution

Symbolic execution is a well known verification technique in which a full simulation of the input program is executed with a "symbolic" semantics: each variable's value is represented by a symbolic expression instead of a concrete value. This allows for representing and reasoning over a potentially infinite number of inputs.

Branching and non-deterministic behavior is supported via backtracking so that the entire space of reachable states is exhaustively searched for errors. Each state includes a hidden boolean variable called the *path condition* which stores the set of assumptions and branches taken in the current execution. If a particular execution reaches a point in which the path condition is unsatisfiable, then the branch is deemed *infeasible* and is pruned from the search.

Because all feasible branches are checked, symbolic execution can easily end up running indefinitely if there is some loop in the program whose termination depends on a symbolic value. To remedy this, such inputs need to be given small bounds by the user. In practice, these bounds are usually placed on the size of some data structure while the data this structure actually holds is left unbounded. While this may appear to be a serious limitation, experience supports the *small scope hypothesis* [8,9]. This posits that almost all bugs will appear on inputs within some small bounds when a system is properly parameterized.

The path condition is also used when checking assertions. If an assertion is reached and it is determined that the current path condition does not imply the assertion then an error is reported to the user. To make these validity/unsatisfiability checks, external SMT (Satisfiability Modulo Theories) solvers are often used.

2.1 CIVL

CIVL is a symbolic execution tool bundled with its own intermediate language CIVL-C. It has a front-end that currently supports C and FORTRAN programs as input. CIVL-C offers programmers a familiar syntax and semantics because it is a large subset of standard C with additional language features supporting concurrency, specification and verification.

The SMT solvers CIVL currently supports are CVC4 [2] and Z3 [7]. An internal symbolic reasoner is also present which allows for many of these external SMT queries to be simplified or avoided entirely.

CIVL is also a model checker [5] which naturally extends its symbolic execution framework to support concurrent programs. This includes the use of several concurrency dialects such as OpenMP or CUDA-C. However, for better clarity and focus on symbolic execution, we will be restricting our attention to sequential programs.

There are other symbolic execution tools [3,16,17] which could just as easily be used in the way we present in this paper. The reason for choosing CIVL in this case simply comes down to the author's familiarity with the tool and its direct support for C programs.

2.1.1 Modeling Floating Point Numbers as Reals

Floating point numbers in CIVL are modeled as real numbers. While floating point properties are important, they add significant complexity to both specification and verification. We argue that specifications based on real numbers are often more appropriate as an initial verification target.

```
#ifndef SPARSE_H
#define SPARSE_H

struct crs_matrix {
  double *val;
  unsigned *col_ind;
  unsigned *row_ptr;
  unsigned rows, cols;
};

void crs_matrix_vector_multiply(struct crs_matrix *m, double *v, double *p);

#endif
```

Figure 1: `sparse.h`

```
#include "sparse.h"
void crs_matrix_vector_multiply (
       struct crs_matrix *m,
       double *v, double *p) {
  unsigned i, rows=m->rows;
  double *val = m->val;
  unsigned *col_ind = m->col_ind;
  unsigned *row_ptr = m->row_ptr;
  unsigned next=row_ptr[0];
  for (i=0; i<rows; i++) {
    double s=0.0;
    unsigned h=next;
    next=row_ptr[i+1];
    for (; h<next; h++) {
      double x = val[h];
      unsigned j = col_ind[h];
      double y = v[j];
      s = x*y+s;
    }
    p[i]=s;
  }
}
```

Figure 2: `sparse.c` : `m` and `v` are input parameters and `p` is an output

```
#ifndef MATRIX_H
#define MATRIX_H

typedef struct $mat {
  int n, m; // num rows, columns;
  double data[][];
} $mat;

void $mat_vec_mul($mat mat, double * v, double *p);

#endif
```

Figure 3: `matrix.cvh`

```
#include "matrix.cvh"
void $mat_vec_mul($mat mat, double * v, double *p) {
  int n = mat.n, m = mat.m;
  for (int i=0; i<n; i++) {
    double s = 0.0;
    for (int j=0; j<m; j++)
      s += mat.data[i][j]*v[j];
    p[i] = s;
  }
}
```

Figure 4: `matrix.cvl`

The primary reason for this is that they are independent of implementation specific details such as the order that floating point operations are performed. This is what allows for the "code as specification" approach highlighted in this paper. For example, Strassen's algorithm is not bit-level equivalent to naive matrix multiplication but it is equivalent using real numbers.

Additionally, violations of a specification based on reals represent logical errors which are usually more pressing and common. After these bugs are ironed out, another tool can be used to analyze floating point properties if desired. Ideal real models of arithmetic are complementary to approaches focused on floating points and generally offer a quicker initial verification pass.

3 Verifying a Multiplication Algorithm

Verifying algorithms with CIVL is often much like writing a traditional unit test. The general workflow consists of writing a program, called a *driver*, which:

1. Generates inputs for the test;
2. Executes the algorithm being tested;
3. Calculates the expected result using some trusted source;
4. Compares the results;
5. Performs any tear-down/cleanup.

Because scientific software is usually built upon a foundation of mathematical libraries, there are many software components which are amenable to this kind of verification.

To demonstrate this we will apply CIVL in this way to a multiplication algorithm between a CRS matrix and a dense vector. It is extracted from the source code located at [14] for the paper [12].

The CRS format for a matrix is a common way to efficiently represent sparse matrices. The C structure used to represent a CRS matrix is declared in a header file `sparse.h` shown in Figure 1. The fields `rows` and `cols` store the number of rows and columns of the matrix. The field `val` stores all the non-zero entries of the matrix as a single array in row-order. Array `col_ind` has the same length as `val` and stores which column each corresponding entry of `val` is located. The array `row_ptr` has size `rows` + 1 and is monotonically increasing. The ith entry of `row_ptr` holds the index of `val` and `col_ind` where the ith row starts.

The multiplication algorithm we wish to verify is implemented in the source file `sparse.c` shown in Figure 2. It takes as input a CRS matrix `m`, a (dense) vector `v`, and a pointer `p` which will point to the result of the multiplication when the call returns.

3.1 Specification

When CIVL executes code it will check for many different types of errors such as dividing by zero or accessing an array out of bounds. However we are also interested in the functional correctness of the multiplication algorithm. This requires determining what it means for the algorithm to be "correct."

We know what a matrix is as a mathematical concept and what it means to multiply it with a vector. The CRS structure is a way to *represent* a matrix. So to say this multiplication algorithm is correct means that the result of executing the function `crs_matrix_vector_multiply` is the same as performing the mathematical operation on the standard matrix that the input CRS structure represents.

So the key to specifying any algorithm involving a CRS structure is to describe exactly how such a structure represents a standard matrix. This is done by defining a *representation function.*

In the paper [12] that this example is taken from, a representation relation is used in a similar way. The difference is that in [12], a declarative approach is used, whereas here we use executable code to make this correspondence.

We created a header file `matrix.cvh` shown in Figure 3 that contains the CIVL-C structure which represents a standard matrix. The multiplication function for this matrix type is in the source file `matrix.cvl` shown in Figure 4.

The representation function for a CRS matrix is implemented on lines 39–52 of our driver `driver.cvl` presented in Figure 5. We kept this function in the driver for simplicity but in practice it may be better to separate this out into its own library for specifying and verifying algorithms related to CRS matrices.

```
#include <stdlib.h>
#include <pointer.cvh>
#include "sparse.h"
#include "matrix.cvh"

$input unsigned N_B = 3, M_B = 3, N, M;
$assume (1<=N && N<=N_B && 1<=M && M<=M_B);
$input double V[M], A[N*M];

/* Fills in p[0],...,p[len-1] with a strictly increasing sequence
   of integers in [0,max].  Precondition: 0 <= len <= max+1 */
void strict_inc(unsigned * p, unsigned len, unsigned max) {
  for (int i=0; i<len; i++) {
    unsigned a = (i == 0 ? 0 : p[i-1]+1), b = max - len + i + 1;
    p[i] = a + $choose_int(b-a+1); // choose in a..b
  }
}

struct crs_matrix make_crs(unsigned n, unsigned m) {
  unsigned * row_ptr = malloc((n+1)*sizeof(unsigned));
  row_ptr[0] = 0;
  for (int i=1; i<=n; i++)
    row_ptr[i] = row_ptr[i-1]+$choose_int(m+1);
  unsigned NZ = row_ptr[n];
  unsigned * col_ind = malloc(NZ*sizeof(unsigned));
  for (int i=0; i<n; i++)
    strict_inc(col_ind+row_ptr[i], row_ptr[i+1]-row_ptr[i], m-1);
  double * val = malloc(NZ*sizeof(double));
  for (int i=0; i<NZ; i++) val[i] = A[i];
  return (struct crs_matrix){ val, col_ind, row_ptr, n, m };
}

void destroy_crs(struct crs_matrix mat) {
  free(mat.val);
  free(mat.col_ind);
  free(mat.row_ptr);
}

$mat $mat_crs(struct crs_matrix crs) {
  unsigned n = crs.rows, m = crs.cols;
  $mat mat;
  mat.n = n;
  mat.m = m;
  mat.data = (double[n][m])$lambda(int i,j) 0.0;
  for (int i=0; i<n; i++) {
    unsigned r = crs.row_ptr[i],
      rnxt = crs.row_ptr[i+1];
    for (int k=r; k<rnxt; k++)
      mat.data[i][crs.col_ind[k]] = crs.val[k];
  }
  return mat;
}

int main() {
  double v[M], actual[N], expected[N];
  for (int i=0; i<M; i++) v[i] = V[i];
  struct crs_matrix mat = make_crs(N, M);
  crs_matrix_vector_multiply(&mat, v, actual);
  $mat dense = $mat_crs(mat);
  $mat_vec_mul(dense, v, expected);
  $assert($equals(actual, expected));
  destroy_crs(mat);
}
```

Figure 5: `driver.cvl`

3.2 Generating Inputs

Just like a unit test, our driver file must generate inputs for the algorithm. Lines 6–31 of our driver presented in Figure 5 are responsible for this.

CIVL-C provides a type qualifier `$input` for global variables which marks them as read-only and initializes them with an arbitrary value of their type. This is used on lines 6 and 8.

The variables `N` and `M` represent the number of rows and columns, respectively. These need to be bounded because otherwise the state space will be infinite since we usually loop over these variables.

CIVL allows for `$input` variables to be given specific concrete values from the command line. This could be used for verification with a specific number of rows and columns. However it is more convenient to check a range of values for `N` and `M`. So instead we add two additional `$input` variables `N_B` and `M_B` which are used to provide an upper bound on the values of `N` and `M` using the `$assume` statement on line 7. By default these bounds are initialized to 3 but can be overridden from the command line.

The `$input` array `V` is used by the `main` function of our driver to fill out an array that represents our input vector. Generating the CRS matrix itself is trickier.

It is possible to use `$input` on a global variable of type `struct crs_matrix` but there is a problem with this. Not every possible object of such a struct actually describes a valid CRS matrix. For instance `val` and `col_ind` must both point to arrays with the same length. The sequence `row_ptr[0]`, `row_ptr[1]`, … must be monotonically increasing. These are examples of data structure invariants that many functions which consume CRS structures require of their inputs.

We could describe these invariants using the `$assume` statement provided by CIVL-C. This offsets a lot of complex reasoning to the SMT solvers which can result in unsolved queries causing spurious reports of failure. Alternatively we can non-deterministically construct an arbitrary (valid) input with code similar to how a unit test might randomly generate its inputs. This is usually more intuitive for programmers inexperienced with formal logic, but it can blow up the state space due to an increased number of branches. We take the latter approach. This is implemented by the function `make_crs` seen on lines 19–31 which we will briefly explain.

After allocating space for `row_ptr` on line 20, it is filled with monotonically increasing data representing each row's starting index in `val` and `col_ind` (lines 22–23). This is done using CIVL-C's `$choose_int(int n)` expression which non-deterministically returns an integer value between 0 (inclusive) and `n` (exclusive). A similar process is used to fill out the values of `col_ind` using a helper function `strict_inc`. Finally, `val` is filled out using the `$input` array `A` declared earlier.

Under symbolic execution, when this function returns, the actual entries of the matrix will all be symbolic. The values of `row_ptr` and `col_ind` will contain concrete values chosen non-deterministically. To get a sense for this, we inserted print statements (not shown) at the end of `make_crs`. Because all different choices get explored in symbolic execution, this results in all explored inputs to be printed. Here is a small snippet of this output:

```
--------
n: 2 m: 3
val: [ X_A[0] X_A[1] X_A[2] ]
col_ind: [ 0 1 2 ]
row_ptr: [ 0 0 3 ]
--------
```

```
--------
n: 1 m: 3
val: [ ]
col_ind: [ ]
row_ptr: [ 0 0 ]
--------

--------
n: 3 m: 3
val: [ X_A[0] ]
col_ind: [ 0 ]
row_ptr: [ 0 1 1 1 ]
--------
```

A value of the form `X_A[i]` represents the `i`$^{\text{th}}$ index into the symbolic constant `X_A` assigned to the global `$input` variable `A` on initialization. Since this array is unconstrained, each of these symbolic values represent an arbitrary real number. Therefore, if CIVL reports the algorithm is bug-free then the program is correct for all possible CRS matrices with up to `N_B` rows and `M_B` columns.

3.3 The Driver

The `main` function of our driver can be seen on lines 54–63 of Figure 5. The array variable `v` represents our input vector and is initialized as such with the values of our global `$input` variable `V` on line 56. The function `make_crs` used to generate our input CRS matrix is called on line 57. It initializes a `crs_matrix` variable `mat`.

With the inputs generated, we call `crs_matrix_vector_multiply` using the array `actual` to store the results. Then, as described earlier, we create the matrix `dense` which our CRS matrix `mat` represents. We use it to calculate the expected result with the function `$mat_vec_mul` on line 60. The result of this call is stored in the array `expected`.

Next we assert that the two results stored in `actual` and `expected` are in fact equal on line 61. For convenience, we use a built-in CIVL-C primitive `$equals` which performs a deep equality between two arrays.

Finally we free any allocated memory made for the test. Cleanup is important in the context of verification because memory leaks are checked by CIVL. In this case, cleanup simply involves freeing the memory allocated by `make_crs`. A simple helper method `destroy_crs` is provided on lines 33–37 which does this.

3.4 Output

To run the driver we use the command

```
civl verify driver.cvl matrix.cvl sparse.c
```

This links the three source files together effectively into one single program for CIVL to analyze. Running this command results in the following output:

```
=== Source files ===
driver.cvl  (driver.cvl)
sparse.h  (sparse.h)
matrix.cvh  (matrix.cvh)
matrix.cvl  (matrix.cvl)
sparse.c  (sparse.c)

=== Command ===
```

```
civl verify driver.cvl matrix.cvl sparse.c

=== Stats ===
   time (s)           : 9.36        transitions  : 108128
   memory (bytes)     : 4.194304E8  trace steps  : 78911
   max process count : 1            valid calls  : 281776
   states             : 78239       provers      : cvc4, z3, why3
   states saved       : 113029      prover calls : 19
   state matches      : 673

=== Result ===
All errors marked with '+' are absent on all executions.
 + Dereference errors                 + Functional equivalence violations
 + Internal errors                    + Library loading errors
 + Other errors                       + Assertion violations
 + Communication errors               + Writes to constant variables
 + Absolute deadlocks                 + Division by zero
 + Writes to $input variables         + Invalid casts
 + Malloc errors                      + Memory leaks
 + Memory management errors           + MPI usage errors
 + Out of bounds errors               + Reads from $output variables
 + Pointer errors                     + Process leaks
 + Sequence errors                      Non-termination
 + Use of undefined values            + Union errors
```

Using a 2020 MacBook Air with a 1.2 GHz Quad-Core Intel Core i7, CIVL was able to verify this program is free of any of the checked errors for all CRS matrices with up to three rows and three columns in under ten seconds. Note that non-termination is not checked by default since this is not always in error in some programs.

We can push CIVL further by increasing the bounds of our matrix from the command line. Recall that the `$input` variable `M_B` bounded the number of columns of our matrix. We can rerun the example while setting `M_B` to a value of 4 with the command

```
civl verify -inputM_B=4 driver.cvl matrix.cvl sparse.c
```

CIVL is still able to succeed but it takes around fifty seconds to do so because the number of states start to explode.

We can see how CIVL works when there is a bug by intentionally introducing one. One easy-to-make mistake in this algorithm would be to accidentally switch the order of lines 12 and 13 in the multiplication algorithm (Figure 2). Running CIVL on this produces the following output:

```
Violation 0 encountered at depth 113:
CIVL execution violation in p0
(property: ASSERTION_VIOLATION, certainty: PROVEABLE) at
$assert($equals(actual, expected))
driver.cvl:61:2-35 | $assert($equals(actual, expected))
```

Additional information is printed which we have omitted for space. When an error is found, CIVL has several capabilities to assist in pinpointing what went wrong. States and transitions taken can be printed in full detail and CIVL can attempt to find a minimal execution for the error. Another useful tactic is to insert printing statements into the code and then use the `replay` command which re-executes the path the program took when reaching the error.

4 Conclusion

Scientific software often contains many highly optimized components for which unit testing does not suffice. We argue that a symbolic execution tool like CIVL can remedy this. We make this argument by outlining how to write a verification driver similarly to a unit test but with the ability to provide stronger verification guarantees. We apply this methodology to an example from the paper [12] and show that it was effective in verifying it for inputs within a small set of bounds.

While the techniques here provide much stronger correctness than traditional testing, the approach taken in the original paper is much more comprehensive. It provides an analysis of floating-point precision and constructs a formal proof. However the two approaches are not mutually exclusive. Using symbolic execution is a more lightweight approach that can be applied to quickly find bugs, even as the code is being developed. Once symbolic execution no longer finds any bugs, a more elaborate deductive approach can then be taken.

References

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu & Irene Finocchi (2018): *A Survey of Symbolic Execution Techniques.* *ACM Computing Surveys* 51(3), pp. 50:1–50:39, doi:10.1145/3182657.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds & Cesare Tinelli (2011): *CVC4.* In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *Computer Aided Verification*, Springer, Berlin, Heidelberg, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.

[3] Cristian Cadar, Daniel Dunbar & Dawson Engler: *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.*

[4] *CIVL: The Concurrency Intermediate Verification Language.* Available at `https://civl.dev`. Accessed 1 Feb 2025.

[5] Edmund M. Clarke, William Klieber, Miloš Nováček & Paolo Zuliani (2012): *Model Checking and the State Explosion Problem.* In Bertrand Meyer & Martin Nordio, editors: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 1–30, doi:10.1007/978-3-642-35746-6_1.

[6] L.A. Clarke (1976): *A System to Generate Test Data and Symbolically Execute Programs.* *IEEE Transactions on Software Engineering* SE-2(3), pp. 215–222, doi:10.1109/TSE.1976.233817.

[7] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver.* In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Berlin, Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[8] Daniel Jackson (2019): *Alloy: a language and tool for exploring software designs.* *Commun. ACM* 62(9), pp. 66–76, doi:10.1145/3338843.

[9] Daniel Jackson & Craig A. Damon (1996): *Elements of style: analyzing a software design feature with a counterexample detector.* *SIGSOFT Softw. Eng. Notes* 21(3), p. 239–249, doi:10.1145/226295.226322. Available at `https://doi.org/10.1145/226295.226322`.

[10] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill & Robert Bowdidge (2013): *Why Don't Software Developers Use Static Analysis Tools to Find Bugs?* In: *2013 35th International Conference on Software Engineering (ICSE)*, pp. 672–681, doi:10.1109/ICSE.2013.6606613.

[11] Upulee Kanewala & James M. Bieman (2014): *Testing Scientific Software: A Systematic Literature Review.* *Information and Software Technology* 56(10), pp. 1219–1232, doi:10.1016/j.infsof.2014.05.006.

[12] Ariel E. Kellison, Andrew W. Appel, Mohit Tekriwal & David Bindel (2023): *LAProof: A Library of Formal Proofs of Accuracy and Correctness for Linear Algebra Programs.* In: *2023 IEEE 30th Symposium on Computer Arithmetic (ARITH)*, pp. 36–43, doi:10.1109/ARITH58626.2023.00021.

[13] James C. King (1976): *Symbolic Execution and Program Testing.* *Communications of the ACM* 19(7), pp. 385–394, doi:10.1145/360248.360252.

[14] *LAProof.* Available at `https://github.com/VeriNum/LAProof/tree/main`. Accessed 1 Feb 2025.

[15] Xavier Leroy (2009): *Formal Verification of a Realistic Compiler.* *Commun. ACM* 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[16] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster & Michael Hicks (2011): *Directed Symbolic Execution.* In Eran Yahav, editor: *Static Analysis*, Springer, Berlin, Heidelberg, pp. 95–111, doi:10.1007/978-3-642-23702-7_11.

[17] Corina S. Păsăreanu & Neha Rungta (2010): *Symbolic PathFinder: Symbolic Execution of Java Bytecode.* In: *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, Association for Computing Machinery, New York, NY, USA, pp. 179–180, doi:10.1145/1858996.1859035.

[18] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer & Michael S. Rogers (2015): *CIVL: The Concurrency Intermediate Verification Language.* In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, doi:10.1145/2807591.2807635.