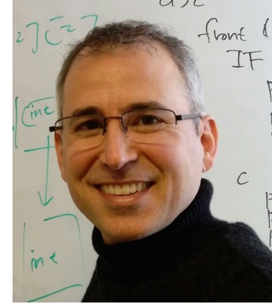
CIVL Solutions to VerifyThis 2016 Challenges

Stephen F. Siegel, University of Delaware



The VerifyThis 2016 program verification competition was held in April, 2016. The competition presented three verification challenges: the first dealt with matrix multiplication, including the correctness of Strassen's algorithm; the second dealt with Morris' tree traversal algorithm, and the third required verification of a multithreaded tree barrier. In this paper I present solutions using the CIVL (Concurrency Intermediate Verification Language) verifier. CIVL is able to verify each program within small but non-trivial bounds. The solutions are relatively simple, and are presented in full.

1. INTRODUCTION

This paper presents solutions to the VerifyThis 2016 challenges using the CIVL verifier. The fifth event in an annual series, the competition took place on April 2, 2016, as part of the European Joint Conferences on Theory and Practice of Software (ETAPS). The competition proper took place in one day, and consisted of three challenges, each lasting 90 minutes. Each challenge consisted of a natural language description of a problem, usually with some pseudocode, and a description of properties expected to hold. Working in teams of one or two members, participants attempted to implement, specify, and formally verify the system described, using any languages and verification frameworks of their choosing. The definition of *formal verification* is broad and the competition included deductive verification approaches as well as bounded model checking. The organizers evaluated each solution for correctness, completeness, and elegance. The 14 teams used 9 different tools: CIVL, Dafny, Why3, KIV, KeY, VerCors, Viper, mCRL2, and VeriFast. A second day was used to present and discuss solutions. The complete challenge statements and other information are available on the competition web site [Huisman et al. 2016a]. A post-competition report summarizes the results [Huisman et al. 2016b].

After the competition, participants and others could submit completed and cleaned up versions of their solutions to a public Wiki accessible from the competition web page. The solutions shown here are the revised versions. In each case, I point out any significant differences from my original solutions.

CIVL. The CIVL verifier [Siegel et al. 2015; CIVL 2017] uses symbolic execution and model checking techniques to verify sequential or parallel C programs. The parallel programs may use MPI [Message-Passing Interface Forum 2015], OpenMP [OpenMP 2017], CUDA [NVIDIA 2017], or Pthreads [IEEE 2004], or even combinations of those APIs. The framework is built around an intermediate language called CIVL-C. CIVL-C is an extension of (sequential) C which includes a number of primitives facilitating specification as well as basic concurrency primitives. The CIVL framework consumes a C program using those concurrency dialects and transforms it into a pure CIVL-C program which can be consumed by the verifier. Users can also write in CIVL-C directly, and this was the approach I took in the competition.

The CIVL verifier checks a number of safety properties, including absence of assertion violations, deadlocks, division by 0, illegal pointer dereferences, out-of-bound

array indexes, memory leaks, improper uses of malloc/free, and reads of uninitialized memory. The assertion language is richer than regular C—e.g., it includes first-order quantifiers and predicates for deep equality of objects.

Typically, CIVL is used to verify properties within some bounded region of the input space. Bounds are usually placed on the size of input data structures, on parameters that control the number of loop iterations, and on the number of processes or threads. Within these bounds, properties are verified exhaustively: for all possible values of the inputs, for all thread interleavings, for all choices available to the concurrency APIs (such as wildcard matchings in MPI), and so on.

The effectiveness of this verification approach relies on the *small scope hypothesis*—the belief that defects in a parameterized system almost always manifest themselves in some instance of the system with small parameter values. For example, if a program to multiply matrices behaves correctly on all matrices with size $n \leq 10$, most people would take this as strong evidence that the program behaves correctly for all n. Of course, it is *possible* to construct a solution which fails only for $n = 11$—but this tends not to occur "in nature."[1]

All other things being equal, bounded verification is not as good as approaches based on deductive reasoning which can prove claims without bounds. However, deductive approaches generally require significantly more effort and skill on the part of the user. I hope this paper adds to our understanding of this tradeoff. In particular, I believe the solutions to be sufficiently simple that a programmer of moderate skill, with some additional training, could use CIVL in a similar way.

Repeatability. All of the solutions, original and revised, as well as a Makefile and the CIVL output, can be downloaded from http://vsl.cis.udel.edu/verifythis2016. CIVL is open source software released under the GNU Public Licence. The results reported here were obtained using CIVL 1.7, the same version used in the competition, and which is available at http://vsl.cis.udel.edu/lib/sw/civl/1.7/latest/release/.[2] The CIVL verifier is a Java program and should run on any Java 8 virtual machine. Its only dependencies are three automated theorem provers, which should be installed and in the user's `PATH` the first time CIVL is run. The results reported here were obtained using Z3 [de Moura and Bjørner 2008] version 4.5.1, CVC4 [Barrett et al. 2011] version 1.4, and CVC3 [Barrett and Tinelli 2007] version 2.4.1, run on a 2014 MacBook Air with a 1.7 GHz Intel Core i7 CPU and 8GB RAM, the same laptop used in the competition.

2. OVERVIEW OF CIVL

2.1. Architecture

The CIVL framework comprises four components:

(1) SARL: the Symbolic Algebra and Reasoning Library, used to create and manipulate symbolic expressions, and to prove the validity of formulas. SARL uses external automated provers such as Z3 and CVC4;
(2) ABC: the ANTLR-based C front end, used to parse and represent C programs, including those using OpenMP, CUDA-C, and CIVL-C; produces a CIVL-C Abstract Syntax Tree;
(3) GMC: Generic Model Checking framework, implements standard model checking algorithms, such as depth-first search of a state-transition system; and

[1]There is an implicit assumption in the small scope hypothesis that "magic numbers" such as "11" do not occur in the program code. Such numbers should be replaced by parameters.
[2]The SHA1 checksum of `civl-1.7_3157.jar` is `c1e7d31663b8588459c9275ff81bbdf4ace160fc`.

(4) CIVL proper, which uses the above 3 components. Translates a CIVL-C AST into a lower-level model which defines a state-transition system in which states map program variables to symbolic expressions. Uses GMC to perform reachability analysis of that system, and SARL to check assertions.

2.2. SARL and Symbolic arithmetic

SARL provides services usually associated with two different fields: symbolic algebra and automated theorem proving. The language supported by SARL is essentially a typed ("many-sorted") first-order logic. The types include the real numbers, integers, booleans, and characters, as well as tuple, array, and function types. The terms in this logic are *symbolic expressions* and formulas are symbolic expressions of boolean type. Variables are *symbolic constants*. The function and predicate symbols are *symbolic operators*. There are methods to build expressions, to ask if a formula is valid, to get various kinds of information about an expression, to perform substitution, to simplify an expression or compute an interval approximation of a numeric expression under a given context, and so on. SARL solves many validity queries itself through its simplification process, but if that process does not suffice it invokes a sequence of external provers.

Additional features of SARL include the following:

— There are *complete* and *incomplete* array types. Both specify an element type E, but a complete type additionally specifies a length l, which is a symbolic expression of integer type. The domain of the complete type consists of all sequences of E of length l. The incomplete type's domain consists of all finite sequences of E; it is a supertype of every complete type with element type E. This differs from other systems in which arrays are infinite. SARL's approach is convenient for representing arrays in languages such as C.
— Operators that are commutative and associative (including addition, multiplication, *logical and*, and *logical or*) consume a finite sequence of arguments of any length.
— Two additional types are the *Herbrand integers* and *Herbrand reals*. These are like the usual integers and reals, but numeric operations involving them are treated as uninterpreted functions. These could be used for "bit-precise" reasoning, though currently CIVL does not use them.
— SARL also supports operations on bit vectors, which are represented as arrays of booleans.

Like most computer algebra systems, SARL attempts to transform symbolic expressions into a standard form[3] and to simplify expressions. There are multiple reasons for doing so: (1) a standard form means two equivalent expressions are likely to be represented in the same way, which speeds up equality testing, (2) in SARL, the Flyweight pattern is used on symbolic expressions, so using a standard form reduces the total number of expressions and therefore the memory footprint, (3) simpler expressions tend to be smaller, which further decreases the memory footprint, and (4) the speed and precision of most algorithms consuming symbolic expressions decreases with the size and complexity of the expressions, so simpler expressions generally make these algorithms more effective.

In the SARL semantic model, operations do not have to be total, i.e., there can exist values on which a symbolic expression is *undefined*. We write $t_1 \rightsquigarrow t_2$ if t_2 evaluates to the same value as t_1 in every valuation for which t_1 is defined. For example, if X is a

[3] I use *standard form* instead of *canonical form*, because the latter technically means a unique representative of an equivalence class. Transforming expressions to a true canonical form is often prohibitively expensive and not even possible for certain classes of expressions; cf. [Caviness 1970].

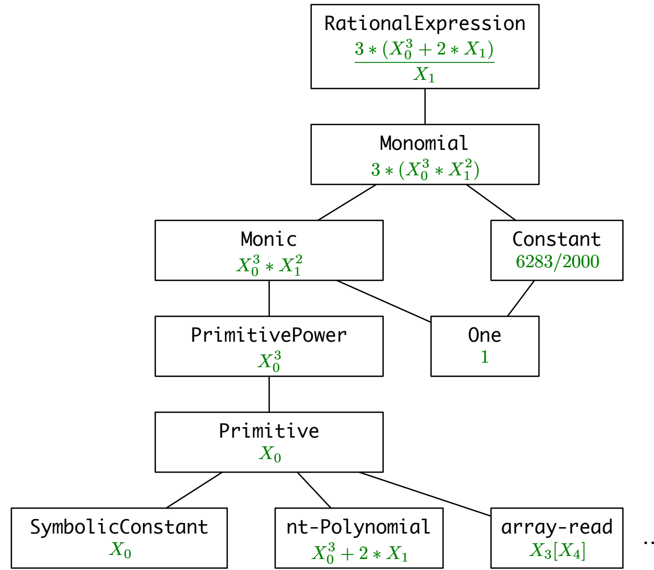


Fig. 1. Classes of expressions used to define the standard form for symbolic expressions of real type, with examples. An edge indicates that the bottom class is a sub-class of the class above.

symbolic constant of real type, $X/X \rightsquigarrow 1$, but $1 \not\rightsquigarrow X/X$, as X/X is not defined at 0. For most applications, t_1 can be safely replaced by t_2.

Every SARL method that returns a symbolic expression guarantees that expression will be in *standard form*. The form requires a total order on symbolic expressions: the symbolic constants and operators are ordered, and these extend to a total order on all expressions. For example, the standard form for boolean expressions is conjunctive normal form, in which the operands of the *and* and *or* operators occur in increasing order.

For expressions of real or integer type, the situation is more complicated. The specification of the standard real form, for example, requires defining several classes of expressions. The class hierarchy is depicted in Figure 1 and the definitions are as follows:

— The real *primitives* are a class of expressions that play the role of "variables" in polynomial forms. Most non-constant real expressions in which the operator is not addition, multiplication, subtraction, or division are primitives. This includes real symbolic constants, array reads for which the element type is real, function applications for which the return type is real, tuple selections for which the selected component is real, and power expressions in which the exponent is not a concrete integer. There is another type of real primitive, the *nt-polynomial*, defined below.
— A *real constant* is a concrete rational number. These are represented as quotients of (unbounded) integers with greatest common divisor 1, and for which the denominator is positive.
— A *primitive power* is a primitive or a power expression in which the base is a primitive and the exponent is a concrete positive integer greater than or equal to 2.
— A *monic* is the constant 1, a primitive power, or a product of two or more primitive powers with distinct primitives. The factors of a monic are ordered by their primitives.

— A *monomial* is a constant, a monic, or the product of a non-0 constant and a monic that is not one.
— An *nt-polynomial* ("non-trivial" polynomial) is a sum of two or more monomials, all of which have distinct monics. The terms occur in order of increasing monic and the leading coefficient is 1. Furthermore, there is no primitive which occurs as a factor in every term.

A monomial is *not* an nt-polynomial. Instead, all nt-polynomials are primitives, so can be used as "variables" in more complicated monics. The reason for this approach, as opposed to the more natural approach of declaring monomials to be polynomials, is that *expansion* is an expensive operation. Rather than requiring every operation to expand polynomials, expansion can be carried out only when necessary. For example, $(X_0 + X_1)^{100}$ is in standard form: it is a primitive power in which the primitive base is the nt-polynomial $X_0 + X_1$.

Finally, the standard form for any real expression is the *rational expression*:

— A *rational expression* is a monomial, or the quotient of a monomial and a monic which is not 1. The monic of the numerator and the denominator can have no primitive factor in common.

The following are the standard forms for numeric comparison operators:

— $0 < m$
— $0 \leq m$
— $m < 0$
— $m \leq 0$
— $0 = p$
— $0 \neq p$

where p is a primitive and m is a monic in which all factors are primitives (i.e., the maximum power of each factor is 1).

Standard properties of the real numbers allow every real or comparison expression t_1 to be transformed to a standard form t_2 such that $t_1 \rightsquigarrow t_2$. I omit the details, but give a few examples:

(1) $2 * X_0 + X_1$ is not in standard form. It is not an nt-polynomial because the leading coefficient is not 1. Its standard form is $2 * (X_0 + (1/2) * X_1)$, which is a monomial in which the constant is 2 and the monic is the nt-polynomial $X_0 + (1/2) * X_1$.
(2) $\dfrac{X_0 + X_1}{2 * (X_0 + X_2)}$ is not in standard form because the denominator is not a monic. Its standard form is $\dfrac{(1/2) * (X_0 + X_1)}{X_0 + X_2}$.
(3) $(X_0 + X_1)^2$ is in standard form, as is the equivalent expression $X_0^2 + 2 * (X_0 * X_1) + X_1^2$.
(4) $0 = X_0 * X_1^2$ is not in standard form. Its standard form is $0 = X_0 \vee 0 = X_1$.
(5) $0 < X_0^2 * X_1^3$ is not in standard form. Its standard form is $0 \neq X_0 \wedge 0 < X_1$.
(6) $0 \leq X_0^6$ is not in standard form. Its standard form is *true*.

2.3. Symbolic execution

I briefly review the main concepts from symbolic execution in a simple context; refer to [Siegel and Zirkel 2011] for details.

2.3.1. States. A program defines some set V of typed variables. A *concrete state* of the program consists of a value for the *program counter*, which specifies the current control location of the program, and a *valuation*, which is a type-respecting map from

V to some set Val of *values*. For example, a program with two integer variables `u` and `v` has a concrete state

$$\langle l_0; \mathtt{u} \mapsto 6, \mathtt{v} \mapsto 2\rangle \tag{1}$$

in which control is at some program location l_0, `u` has the value 6, and v has value 2.

A *symbolic state* consists of a program counter; a boolean symbolic expression called the *path condition*; and a *symbolic valuation*, which is a type-respecting map from V to the set of symbolic expressions. A symbolic state represents a (possibly infinite) set of concrete states—all concrete states obtained by substituting concrete values for the symbolic constants in such a way that the path condition evaluates to *true*. The program mentioned above has a symbolic state

$$\langle l_0; X_0 > X_1; \mathtt{u} \mapsto 2 * X_0, \mathtt{v} \mapsto X_1\rangle$$

in which control is at l_0, the path condition is $X_0 > X_1$, `u` is assigned the symbolic expression $2 * X_0$, and `v` is assigned X_1. The set of concrete states represented by this symbolic state contains concrete state (1), as the assignment $X_0 = 3, X_1 = 2$ causes the path condition to evaluate to *true*.

2.3.2. Symbolic execution for program verification. Symbolic execution entails a reachability analysis of a state-transition sytem in which the states are symbolic states and transitions correspond to program statements. Each program statement must be interpreted on the symbolic level in a way that is compatible with its concrete semantics. For the most part, this is straightforward: e.g., an assignment `x=e;` is executed by evaluating `e` symbolically, assigning the resulting symbolic expression to `x` in the new state, and updating the program counter appropriately.

At a branch on a boolean expression c, two outgoing transitions are enabled: one for the case where c is *true* and one for the case where c is *false*. To execute the *true* transition, c is added to the path condition, i.e., the new value of the path condition is the conjunction of the old value and the result of evaluating c. For the *false* transition, $\neg c$ is added to the path condition. Hence the path condition keeps track of all the choices made at branch points along an execution path.

Assertions are verified by checking the validity of the asserted formula ψ under the assumption that the path condition ϕ holds, i.e., by asking whether $\phi \Rightarrow \psi$ is valid. This validity implies that the assertion holds for all concrete states represented by the symbolic state. If validity can be established for all reachable symbolic states, the assertion holds on all reachable concrete states [Siegel and Zirkel 2011, Theorem 8].

2.3.3. State simplification. Two symbolic states are *equivalent* if they represent the same set of concrete states. As with symbolic expressions, it is desirable to transform symbolic states into an equivalent standard and/or simpler form. It can reduce the number of symbolic states searched, because a state is more likely to be recognized as equivalent to a state seen before. It can also improve the effectiveness of the validity checking.

The first step is to use standard form for all symbolic expressions occurring in the state; as discussed above, this is always the case when using SARL. However, simplification of symbolic states can go much further. The techniques used by SARL include the following:

(1) An *interval analysis* is applied to all monics in the path condition. This information is then used to simplify occurrences of that monic throughout the state. In some case this analysis obtains a single concrete value for the monic.
(2) When a concrete value is obtained for a symbolic constant, the value is substituted for the symbol everywhere in the state, and the symbol is completely eliminated.
(3) The set of all numeric equalities derived from the path condition is treated as a linear system of equations in which the "variables" are the monics. Gaussian

elimination is applied to this system to obtain further concrete values for monics and to eliminate some monics if they can be expressed as linear combinations of the remaining monics.
(4) When a sequence of array-write operations over an array is *complete* (i.e., there is a write to every cell), the entire expression is replaced by a concrete array.
(5) Since symbolic constants can appear and disappear from the state (e.g., due to dynamic memory allocation, or control leaving a scope), at certain points they are renamed in a canonical way (e.g., X_0, X_1, ...).

The state simplification process is one of the most expensive operations performed by the CIVL verifier. However, it is quite effective and often the vast majority of validity queries are resolved by simplification alone. This greatly reduces the number of calls to external provers.

3. CHALLENGE 1: MATRIX MULTIPLICATION

The first challenge involved matrix multiplication of square matrices. There were three tasks to the challenge.

3.1. Task 1: Specification and verification of a “naïve” algorithm

Task 1 presented pseudocode for a “naive” version of matrix multiplication:

```
int[][] matrixMultiply(int[][] A, int[][] B) {
  int n = A.length;
  // initialise C
  int[][] C = new int[n][n];
  for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
      for (int j = 0; j < n; j++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
  return C;
}
```

The task was to “[p]rovide a specification to describe the behaviour of this algorithm, and prove that it correctly implements its specification.”

My solution is given in Figure 2. The function `matrixMultiply` in the solution is nearly identical to the pseudocode, with some small changes, e.g., a pointer to the output matrix has been made a parameter to the function, in a typical C style. The remaining code sets up a general environment, provides some auxiliary functions for the specification, and implements a driver with an assertion to test the correctness of the output.

Several of the global variable declarations use the CIVL-C `$input` type qualifier.[4] An input variable represents a program input. The variable is initialized using the following protocol: first, if a value for that variable is specified on the command line, that value is used. For example,

```
civl verify -inputBOUND=10 mmp1.cvl
```

[4]Like all CIVL-C language primitives not in standard C, the name of this primitive starts with `$`. Also, C and CIVL-C keywords appear in `blue` text in this article.

```
1 #include <civlc.cvh>
2 $input int BOUND; // upper bound on N
3 $input int N; // the size of the matrices
4 $assume(1<=N && N<=BOUND);
5 $input float A0[N][N], B0[N][N], C0[N][N]; // arbitrary input matrices
6 void matrixMultiply(int n, float C[][], float A[][], float B[][]) {
7   for (int i=0; i<n; i++)
8     for (int j=0; j<n; j++)
9       C[i][j] = 0.0;
10   for (int i=0; i<n; i++)
11     for (int k=0; k<n; k++)
12       for (int j=0; j<n; j++)
13         C[i][j] += A[i][k] * B[k][j];
14 }
15 float dot(int n, float u[], float v[]) {
16   float sum = 0;
17   for (int i=0; i<n; i++) sum += u[i]*v[i];
18   return sum;
19 }
20 // gets the index-th column of matrix mat:
21 float * column(int n, float result[], float mat[][], int index) {
22   for (int i=0; i<n; i++) result[i] = mat[i][index];
23   return &result[0];
24 }
25 int main() {
26   float actual[N][N], tmp[N];
27   matrixMultiply(N, actual, A0, B0);
28   for (int i=0; i<N; i++)
29     for (int j=0; j<N; j++)
30       $assert(dot(N, A0[i], column(N, tmp, B0, j))  == actual[i][j]);
31 }
```

Fig. 2. Challenge 1, part 1: "naïve" matrix multiplication (`mmp1.cvl`)

specifies a value of 10 for variable `BOUND`. If no value is specified on the command line but an initializer is present, the initializer is used; this is a good way to provide a default value. Finally, if neither the command line value nor an initializer is present, the variable is assigned an arbitrary, unconstrained value of its type; for symbolic execution, this is accomplished by assigning the variable a fresh symbolic constant. All input variables are read-only.

In the solution, `BOUND` is used in an assumption to place an upper bound on `N`. Hence the command above will verify the program for all matrices of size 10 or smaller. If no concrete value of `BOUND` is provided, the verifier will run forever (or until it runs out of memory).

The idea for the specification is that the (i, j)-th entry of the product should be the dot product of the i-th row of A and the j-th column of B. To do this I implemented a function to extract a column from a matrix and a function to compute the dot product of two vectors. The assertion checks this for all (i, j). There is no difficulty in verifying this: all of the assertions follow immediately from the standard form. One can insert a `printf` statement between lines 29 and 30 to see the symbolic expressions; the output begins

```
C[0][0] =
  X_A0[0][1]*X_B0[1][0]+X_A0[0][2]*X_B0[2][0]+X_A0[0][3]*X_B0[3][0]+
  X_A0[0][4]*X_B0[4][0]+X_A0[0][5]*X_B0[5][0]+X_A0[0][6]*X_B0[6][0]+
  X_A0[0][7]*X_B0[7][0]+X_A0[0][8]*X_B0[8][0]+X_A0[0][9]*X_B0[9][0]+
  X_A0[0][0]*X_B0[0][0]
```

The final output gives the result and several statistics:

```
CIVL v1.7 of 2016-03-31 -- http://vsl.cis.udel.edu/civl
```

```
1  ... declaration of inputs and definition of matrixMultiply from Figure 2 ...
2 void main() {
3   $atomic {
4     float T1[N][N], T2[N][N], R1[N][N], R2[N][N];
5     matrixMultiply(N, T1, A0, B0); // T1 <- A0*B0
6     matrixMultiply(N, R1, T1, C0); // R1 <- T1*C0
7     matrixMultiply(N, T2, B0, C0); // T2 <- B0*C0
8     matrixMultiply(N, R2, A0, T2); // R2 <- A0*T2
9     $assert($equals(&R1, &R2)); // check deep equality of two objects
10   }
11 }
```

Fig. 3. Challenge 1, part 2: associativity of matrix multiplication (`mmp2.cvl`)

```
=== Command ===
civl verify -inputBOUND=10 mmp1.cvl

=== Stats ===
   time (s)            : 4.74
   memory (bytes)      : 163053568
   max process count   : 1
   states              : 35127
   states saved        : 11747
   state matches       : 0
   transitions         : 35126
   trace steps         : 11746
   valid calls         : 190978
   provers             : cvc4, z3, cvc3
   prover calls        : 37

=== Result ===
The standard properties hold for all executions.
```

The output indicates that the verification time is just under 5 seconds. The "standard properties" include all of the safety properties listed in Section 1 and in particular indicate that the assertion can never be violated.

3.2. Task 2: Verifying associativity

The second task was to "[s]how that matrix multiplication is associative, i.e., the order in which matrices are multiplied can be disregarded: $A(BC) = (AB)C$. To show this, you should write a program that performs the two different computations, and then prove that the result of the two computations is always the same."

The solution is given in Figure 3 and solves the problem exactly as specified. Verification time is 10 seconds for $\texttt{BOUND} = 10$. The solution uses a special CIVL-C function `$equals`, which accepts two pointers to C objects and tests for "deep equality." The definition is what you would expect: two arrays are "equal" if they have the same length and corresponding elements are "equal", two floats are "equal" if they are the same rational number, etc. Again, the assertion is completely resolved by the standard form for nt-polynomials, in particular the unique ordering for arguments to $+$ and $*$.

The body of the `main` function is placed in an `$atomic` block. This primitive is typically used in concurrent programs and prevents other processes from executing while one process is in the atomic region. A side-effect of CIVL's `$atomic` is that it reduces the number of state canonicalizations, which can make a significant performance difference, even in sequential programs. I have used it in the revised solutions because it decreases the verification time.

3.3. Task 3: Equivalence of the naïve and Strassen algorithms for matrix multiplication

The third challenge dealt with Strassen's matrix multiplication algorithm, which is real-equivalent to the naive algorithm but requires fewer multiplications and has slightly smaller asymptotic time complexity. The challenge statement referred participants to [Wikipedia 2016] for general information on the algorithm, and to [Thoma 2013] for code implementing the algorithm in several languages. The challenge was to prove the equivalence of the naive and Strassen's algorithm for arbitrary square matrices with size a power of 2.

My solution is given in Figure 4. Verification time is 4 seconds for $\texttt{BOUND} = 8$, and 93 seconds for $\texttt{BOUND} = 16$. Function `strassenR` is a direct transliteration of the Java code from the given web page. The environment creates arbitrary input matrices as before, but restricts to sizes which are a power of 2. `LEAF_SIZE` is another parameter used in [Thoma 2013]; it is the threshold below which ordinary matrix multiplication is used. It wasn't immediately clear to me what the appropriate values for this parameter were, but verification succeeded using the range $2..\texttt{N}$, and I have kept this range in the revision. A lower bound of 1 will also work, though verification time goes up to 8s for $\texttt{BOUND} = 8$ and 375s for $\texttt{BOUND} = 16$.

The solution uses CIVL's `$elaborate` function, which consumes a positive integer n. Semantically, this function is a no-op, but it signals the verifier to use a different search strategy. The verifier generates a sequence of transitions, all departing from the current state. The first assumes $n = 0$, the next $n = 1$, and so on. The verifier must be able to determine a concrete upper bound on n from the current path condition for this to succeed. The effect is to eliminate a symbolic constant from the state by exploring separate cases for all of its possible concrete values. This can increase the number of states, but simplify the symbolic expressions and prover queries. It is difficult to predict whether this strategy will be a net win, so CIVL leaves it up to the user. In this case it decreases verification time significantly.

3.4. Discussion

The differences between the revised solutions presented here and my original submission are relatively minor. Besides formatting and stylistic changes, the original solution was in one file, which I broke up into three separate files here in order to get more precise timings. I also added some atomic statements for better performance.

I missed the fact that the problem called for matrices with integer entries, and used `floats` instead. It would be trivial to change the `floats` to `ints`. However this brings up an important issue, which is the difference between floating-point and real number semantics. CIVL currently interprets all floating point operations as real arithmetic, which is why it declared these programs correct. With floating-point semantics, the associativity of matrix multiplication (task 2) fails to hold, since floating-point multiplication is not associative. Similarly, Strassen's algorithm (task 3) is not floating-point-equivalent to the naive algorithm. These discrepancies are not due to bugs—it is well known that Strassen's algorithm, for example, has floating-point characteristics that differ from that of the naive algorithm, but it is expected that it—or any "correct" matrix multiplication algorithm—be real-equivalent to the naive one. Clearly, floating-point semantics are useful for specifying some properties of numerical programs, and real semantics are useful for specifying others. We need tools that can handle both.

The specification in task 1 was essentially accomplished using more C code. This worked, but arguably the new C code is at least as complex as the original code being verified. While there can be advantages to using the programming language as the specification mechanism, in this case I feel a better solution could be constructed if there were additional specification primitives, in particular a "sum" operator, as in

```
1  ... declaration of inputs and definition of matrixMultiply from Figure 2 ...
2  $input int LEAF_SIZE; // threshold below which ordinary matrix multiplication is used
3  $assume (2 <= LEAF_SIZE && LEAF_SIZE <= N);
4  void add(int n, float C[][], float A[][], float B[][]) {
5    for (int i=0; i<n; i++) for (int j=0; j<n; j++) C[i][j] = A[i][j] + B[i][j];
6  }
7  void subtract(int n, float C[][], float A[][], float B[][]) {
8    for (int i=0; i<n; i++) for (int j=0; j<n; j++) C[i][j] = A[i][j] - B[i][j];
9  }
10 // Strassen algorithm from https://martin-thoma.com/strassen-algorithm-in-python-java-cpp/
11 void strassenR(int n, float C[][], float A[][], float B[][]) { // requires n is power of 2
12   if (n <= LEAF_SIZE) {
13     matrixMultiply(n, C, A, B);
14   } else {
15     int m = n/2;
16     float a11[m][m], a12[m][m], a21[m][m], a22[m][m];
17     float b11[m][m], b12[m][m], b21[m][m], b22[m][m];
18     float aResult[m][m], bResult[m][m];
19     for (int i=0; i<m; i++)
20       for (int j=0; j<m; j++) {
21         a11[i][j] = A[i][j];      a12[i][j] = A[i][j + m];
22         a21[i][j] = A[i+m][j];    a22[i][j] = A[i + m][j + m];
23         b11[i][j] = B[i][j];      b12[i][j] = B[i][j+m];
24         b21[i][j] = B[i+m][j];    b22[i][j] = B[i+m][j+m];
25       }
26     add(m, aResult, a11, a22); add(m, bResult, b11, b22);
27     float p1[m][m]; strassenR(m, p1, aResult, bResult); add(m, aResult, a21, a22);
28     float p2[m][m]; strassenR(m, p2, aResult, b11); subtract(m, bResult, b12, b22);
29     float p3[m][m]; strassenR(m, p3, a11, bResult); subtract(m, bResult, b21, b11);
30     float p4[m][m]; strassenR(m, p4, a22, bResult); add(m, aResult, a11, a12);
31     float p5[m][m]; strassenR(m, p5, aResult, b22); subtract(m, aResult, a21, a11);
32     add(m, bResult, b11, b12);
33     float p6[m][m]; strassenR(m, p6, aResult, bResult); subtract(m, aResult, a12, a22);
34     add(m, bResult, b21, b22);
35     float p7[m][m]; strassenR(m, p7, aResult, bResult);
36     float c12[m][m]; add(m, c12, p3, p5);
37     float c21[m][m]; add(m, c21, p2, p4); add(m, aResult, p1, p4);
38     add(m, bResult, aResult, p7);
39     float c11[m][m]; subtract(m, c11, bResult, p5); add(m, aResult, p1, p3);
40     add(m, bResult, aResult, p6);
41     float c22[m][m]; subtract(m, c22, bResult, p2);
42     // Grouping the results obtained in a single matrix:
43     for (int i=0; i<m; i++)
44       for (int j=0; j<m; j++) {
45         C[i][j]   = c11[i][j]; C[i][j+m]   = c12[i][j];
46         C[i+m][j] = c21[i][j]; C[i+m][j+m] = c22[i][j];
47       }
48   }
49 }
50 _Bool isPowerOf2(int n) {
51   while (n>1) {
52     if (n%2 != 0) return $false;
53     n = n/2;
54   }
55   return $true;
56 }
57 int main() {
58   $atomic {
59     $elaborate(N); // hint to verifier: iterate over concrete values of this variable
60     $assume(isPowerOf2(N));
61     float R1[N][N], R2[N][N];
62     matrixMultiply(N, R1, A0, B0);
63     strassenR(N, R2, A0, B0);
64     $assert($equals(&R1, &R2));
65   }
66 }
```

Fig. 4. Challenge 1, part 3: equivalence of Strassen's algorithm (mmp3.cvl)

$\sum_{i=a}^{b}$. (The ACSL specification language [ACSL 2016] specifies such an operator.) One could even imagine adding a number of standard operators from functional programming languages, like *fold*, *reduce*, and *filter*. These could lead to shorter and more intuitive specifications. On the other hand, the use of CIVL-C's `$equals` primitive was extremely useful for comparing two matrices, avoiding the need for iteration over every entry.

The careful reader may have noticed a little twist in the naive algorithm: the last two loops have been transposed from their usual order. At the discussion session, it was clear this complicated deductive approaches—in particular, the construction of appropriate loop invariants. In contrast, it makes no difference at all to the symbolic execution approach used here. I don't believe I even noticed this twist until after the competition.

4. CHALLENGE 2: BINARY TREE TRAVERSAL

The second challenge dealt with a binary tree

```
class Tree {
  Tree left, right, parent;
  bool mark;
}
```

with the properties

— "following a child pointer (left or right) and then following a parent pointer brings us to the original node"
— "the parent pointer of the root is null"
— "It has at least one node, and each node has 0 or 2 children"
— "We do not know the initial value of the mark fields"

and gave an algorithm [Morris 1979] for traversing the nodes in linear time but constant space:

```
void markTree(Tree root) {
  Tree x, y;
  x = root;
  do {
    x.mark = true;
    if (x.left == null && x.right == null) {
      y = x.parent;
    } else {
      y = x.left;
      x.left = x.right;
      x.right = x.parent;
      x.parent = y;
    }
    x = y;
  } while (x != null);
}
```

The tasks are to prove:

(1) "upon termination of the algorithm, all mark fields are set"
(2) "the tree shape does not change"
(3) "the code does not crash"
(4) "the code terminates"
(5) "the nodes are visited in depth-first order" (bonus).

```
1 #include <civlc.cvh>
2 #include <stdbool.h>
3 #include <stdlib.h>
4 $input int DB; // depth bound
5 typedef struct _tree { struct _tree *left, *right, *parent; _Bool mark; } *Tree;
6 void markTree(Tree root) { // mark every node, using only constant space
7   Tree x=root, y;
8   do {
9     x->mark = true;
10    if (x->left == NULL && x->right == NULL) { y = x->parent; }
11    else { y = x->left; x->left = x->right; x->right = x->parent; x->parent = y; }
12    x = y;
13  } while(x != NULL);
14 }
15 Tree makeTree(Tree left, Tree right, _Bool mark) { // makes a new tree
16   Tree result = (Tree)malloc(sizeof(struct _tree));
17   result->left = left; result->right = right; result->mark = mark; result->parent = NULL;
18   if (left != NULL) left->parent = result;
19   if (right != NULL) right->parent = result;
20   return result;
21 }
22 Tree makeArbitrary(int depth) { // construct an arbitrary tree with height < "depth"
23   if (depth == 0) return NULL;
24   // nondeterministic choice: 0: return tree with 0 nodes, 1: return tree with 2 nodes
25   if ($choose_int(2) == 0) return makeTree(NULL, NULL, false);
26   return makeTree(makeArbitrary(depth - 1), makeArbitrary(depth - 1), false);
27 }
28 Tree copyTree(Tree root) { // deep copy a tree, ignoring marks
29   if (root == NULL) return NULL;
30   return makeTree(copyTree(root->left), copyTree(root->right), false);
31 }
32 void freeTree(Tree root) { // free memory allocated by making a tree
33   if (root != NULL) { freeTree(root->left); freeTree(root->right); free(root); }
34 }
35 _Bool allMarked(Tree t) { // checks every node in Tree t is marked
36   if (t != NULL) {
37     if (!(t->mark)) return false;
38     if (!allMarked(t->left)) return false;
39     if (!allMarked(t->right)) return false;
40   }
41   return true;
42 }
43 _Bool wellFormed(Tree t) { // all non-leaf nodes u: u==u->left->parent==u->right->parent
44   if (t->left != NULL) {if (t->left->parent!=t || !wellFormed(t->left)) return false;}
45   if (t->right != NULL) {if (t->right->parent!=t || !wellFormed(t->right)) return false;}
46   return true;
47 }
48 _Bool isomorphic(Tree t1, Tree t2) { // given two well-formed trees, are they isomorphic?
49   if (t1 == NULL) return t2 == NULL;
50   if (t2 == NULL) return false;
51   if (!isomorphic(t1->left, t2->left)) return false;
52   return isomorphic(t1->right, t2->right);
53 }
54 int main() {
55   $atomic {
56     Tree t1 = makeArbitrary(DB), t2 = copyTree(t1); // create arbitrary tree, save a copy
57     $assume(t1 != NULL);
58     markTree(t1); // markTree should mark every node and not change the shape
59     $assert(allMarked(t1)); $assert(t1->parent == NULL); $assert(wellFormed(t1));
60     $assert(isomorphic(t1, t2)); // check that the shape of t1 has not changed
61     freeTree(t1); freeTree(t2);
62   }
63 }
```

Fig. 5. Challenge 2: binary tree traversal (`tree.cvl`)

My solution is shown in Figure 5. The program will explore all trees for which the number of edges along any path from the root to a leaf is less than `DB`. Verification takes about 3s for $\texttt{DB} = 4$ and 77s for $\texttt{DB} = 5$.

Function `markTree` is virtually identical to the code given in the challenge. To this I added a constructor `makeTree`, and a function `makeArbitrary` which uses nondeterministic choice to create an arbitrary tree satisfying the assumptions given in the challenge. The function `$choose_int` consumes an integer n and returns an integer in $0..n-1$; when verifying a program, all possible choices are explored. The remaining functions are used to specify the desired properties of the algorithm.

Function `allMarked` checks that all the mark fields are set. It recurses over all nodes and returns `false` if any unmarked node is found.

To check that the shape of `t1` does not change, a deep copy `t2` of the original tree is made using `copyTree`. This is compared with `t1` after `markTree(t1)`. The function `isomorphic` consumes two tree-or-nulls and checks that either both are null or the corresponding children are isomorphic. In my original solution, I thought this sufficed, but the organizers pointed out it would declare the following two structures to have the same shape:

In the revision, I have corrected this by checking that in `t1`, the root has null parent, and for any node u, the parent of a child of u is u. (These necessarily hold for `t2`.) This implies that every node is a child of at most one node, and the root is a child of no node. It follows that `isomorphic` establishes a structure-preserving bijection between the nodes of the two trees.

The only way that the code of `markTree` could crash is from an invalid or null pointer dereference. The CIVL verifier checks every dereference for these anomalies, so the code cannot crash for any tree within the specified bound.

CIVL does not have a specific mechanism to verify termination or other liveness properties. It does verify absence of deadlocks, but a program can fail to terminate without deadlocking because of an infinite loop. There are, however, a few "tricks" that can sometimes be used to verify termination. The first is to run the verifier with the flag `-saveStates=false`. This instructs the verifier not to save seen states as it searches the state space. If there is a cycle in the state space, this will cause the verifier to run forever. Hence if the verifier returns and reports all properties hold (in particular, absence of deadlock), every execution must terminate. Doing this for `tree.cvl` with $\text{DB} = 4$ leads the verifier to return after 47s. This represents a significant slow-down, due to the large number of states that are traversed multiple times, but establishes termination for trees within this bound. I did not try this for $\text{DB} = 5$.

The second trick is to modify the program by adding counters to loops. I inserted `int count=0;` after line 7 and `count++;` after 8. If the `do..while` loop could loop forever, the verifier would never return (even with saving of states), because every iteration yields a new state. Running this modified program, the verifier returned for both $\texttt{DB} = 4$ and $\texttt{DB} = 5$ with no noticeable slow-down.

I never attempted the bonus task on depth-first order.

4.1. Structure of the state: recursion, pointers, and dynamically-allocated memory

As this example illustrates, CIVL has no problem verifying programs with recursive functions, dynamically allocated data structures, and pointers, as long as the call stack and heap cannot grow without bound. A CIVL state has a complex and dynamic structure, as illustrated in Figure 6. New *dynamic scopes* are added whenever control enters

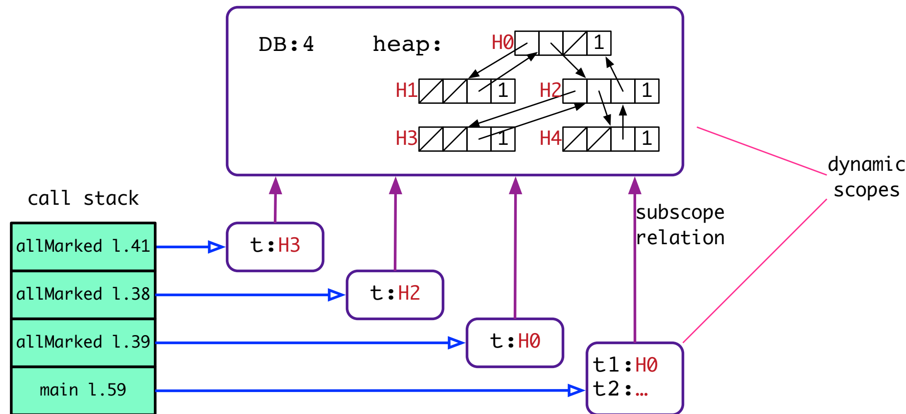


Fig. 6. Structure of a state for `tree.cvl`

a static scope, and are removed when control exits the static scope. Each dynamic scope stores the (symbolic) values of variables declared in that scope. In the pictured state, the large dynamic scope at the top is the global scope, which also contains the heap. Because of recursion, there can be multiple instances of a single static scope: three instantiations of the function scope for `allMarked` in this example. The call stack consists of a sequence of frames, each of which includes a reference to a dynamic scope and a program counter value.

A pointer is represented as a triple (d, i, r), where d is the ID number of a dynamic scope, i is the ID of a variable within that scope, and r is a *reference expression*. A reference expression is a symbolic expression that specifies a "path" to an internal point of a compound value, such as "array element 5 of tuple component 3 of tuple component 0." These expressions are supported by SARL, which provides methods to build and manipulate them, and a method to "apply" a reference expression to a symbolic expression of the appropriate compound type in order to extract the specified subexpression. This last method is used to implement pointer dereferencing. CIVL also supports most, but not all, forms of pointer arithmetic permitted under the C Standard.

5. CHALLENGE 3: TREE BARRIER

The third challenge was a concurrency problem: verification of a tree barrier. The number N of threads is fixed. There is also a fixed binary tree in which the nodes correspond to threads:

```
class Node {
  final Node left, right;
  final Node parent;
  boolean sense;
  int version;
  ... methods ...
}
```

This node class has a *barrier* method which is called by each thread to create a global synchronization point:

```
void barrier()
      requires !sense
      ensures  !sense
{
      // synchronization phase
```

```
        if (left != null) while (!left.sense) { }
        if (right != null) while (!right.sense) { }
        sense = true;  // assume this statement and the next to execute
        version++;     // simultaneously (that is, in one step)
        // wake-up phase
        if (parent == null) sense = false;
        while(sense) { }
        if (left != null) left.sense = false
        if (right != null) right.sense = false
}
```

When a thread enters the barrier, it waits for a signal from its children indicating that they and their descendants have entered the barrier. Then the thread signals its parent. When the root receives this signal, it knows all threads have arrived, and the signal then travels down the tree, freeing threads to leave the barrier.

The challenge continues: "Assume a state in which `sense` is *false* and `version` is zero in all nodes. Assume further that no thread is currently executing `barrier()` and that threads invoke `barrier()` only on their nodes. The number of threads (and, thus, the number of nodes in the tree) is constant, but unknown" and specifies two tasks: to prove

(1) the following invariant holds in all states: If `n.sense` is *true* for any node `n` then `m.sense` is *true* for all nodes `m` in the subtree rooted in `n`.
(2) for any call `n.barrier()`, if the call terminates then there was a state during the execution of the method where all nodes had the same version.

My solution is given in Figure 7. Verification time is 13s for $\mathtt{N} = 4$ and 173s for $\mathtt{N} = 5$.

The solution uses several CIVL-C concurrency primitives. The `$spawn` expression (line 63) wraps a function call, and creates a new *process* (or thread) to execute the call. The new process has its own call stack and the single frame on that stack will point to a new dynamic scope corresponding to the function body—the same scope that would be created by an ordinary function call. Evaluation of the `$spawn` expression returns immediately with a reference to the new process, an object of type `$proc`. Figure 8 shows the structure of a typical state in `barrier.cvl`.

In the solution, I have added a field `p` to the node structure (line 6). The value returned by `$spawn` is assigned to that field in the node corresponding to the new process. This is used later as an argument to the `$wait` function (line 64), which blocks until the referenced process terminates. This is the only place where the field is used, so the modification to the node structure could have been avoided by storing the `$proc` values in a local array instead. Even better: CIVL-C provides the `$parfor` primitive, which captures the common pattern of lines 63–64; those lines can be replaced with

```
$parfor (int i : 0 .. N-1) thread(theNodes[i]);
```

In addition to `$wait`, synchronization takes place though *guarded commands*, which have the form `$when (`*expr*`)` *stmt*. This command blocks until *expr* evaluates to *true*, then executes *stmt*; the first atomic sub-step of *stmt* executes atomically with the testing of the guard expression. This basic concurrency primitive can be used to implement semaphores, locks, and other higher-level concurrency constructs. For the guarded commands in the solution, the *stmt* is just a no-op. These are used in place of the pseudocode's busy-wait loops.

Another CIVL-C feature not in standard C is the ability to nest function definitions. In the solution, function `barrier` is placed inside function `thread`. The body of `thread` (line 39) may invoke `barrier`, but code outside of `thread` may not. Moreover, `barrier` may read and modify the local variables of `thread`, e.g., `myNode`. These internal func-

```
1 #include <civlc.cvh>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 $input int N; // the number of nodes (tried with N=1,2,3,4,5)
5 typedef struct _node { // a node in a binary tree
6   $proc p; // the process to which this node is associated
7   struct _node *left, *right *parent; // left child, right child, and parent
8   _Bool sense; // am I in the barrier?
9   int version; // number of times I have gone through barrier
10 } * Node;
11 Node theNodes[N]; // array storing all nodes created so far
12 int count = 0; // number of nodes created so far
13 void checkDescendantsSensesTrue(Node u) { // check all senses in u and descendants are true
14   if (u != NULL) {
15     $assert(u->sense);
16     checkDescendantsSensesTrue(u->left); checkDescendantsSensesTrue(u->right);
17   }
18 }
19 void checkAncestorSensesFalse(Node u) { // check all senses in u and ancestors are false
20   while (u != NULL) { $assert(!u->sense); u = u->parent; }
21 }
22 void thread(Node myNode) { // the function each thread will run
23   void barrier() { // the barrier function
24     if (myNode->left != NULL) $when (myNode->left->sense); // wait for left child's sense
25     if (myNode->right != NULL) $when (myNode->right->sense);
26     $atom {
27       myNode->sense = true;
28       checkDescendantsSensesTrue(myNode); // check invariant
29       myNode->version++;
30       if (myNode->parent == NULL) // root: check everyone has same version
31         for (int i=0; i<N; i++) $assert(theNodes[i]->version == myNode->version);
32     }
33     if (myNode->parent == NULL) myNode->sense = false;
34     $when (!myNode->sense); // wait until my sense is false
35     $atom { checkAncestorSensesFalse(myNode); } // check invariant
36     if (myNode->left != NULL) myNode->left->sense = false;
37     if (myNode->right != NULL) myNode->right->sense = false;
38   }
39   for (int i=0; i<3; i++) barrier(); // driver: run around the barrier 3 times...
40 }
41 Node makeTree(Node left, Node right) { // make a tree from given children
42   Node result = (Node)malloc(sizeof(struct _node));
43   result->left = left; result->right = right; result->sense = false; result->version = 0;
44   if (left != NULL) left->parent = result;
45   if (right != NULL) right->parent = result;
46   result->parent = NULL;
47   return result;
48 }
49 Node makeArbitraryTree(int numNodes) { // create an arbitrary tree with numNodes nodes
50   if (numNodes == 0) return NULL;
51   int leftSize = $choose_int(numNodes);
52   Node leftTree = makeArbitraryTree(leftSize);
53   Node rightTree = makeArbitraryTree(numNodes - leftSize - 1);
54   Node result = makeTree(leftTree, rightTree);
55   theNodes[count] = result; count++;
56   return result;
57 }
58 void freeTree(Node tree) { // free all nodes in the tree
59   if (tree != NULL) { freeTree(tree->left); freeTree(tree->right); free(tree); }
60 }
61 int main() {
62   Node theTree = makeArbitraryTree(N);
63   $atomic { for (int i=0; i<N; i++) theNodes[i]->p = $spawn thread(theNodes[i]); }
64   for (int i=0; i<N; i++) $wait(theNodes[i]->p);
65   freeTree(theTree);
66 }
```

Fig. 7. Challenge 3: tree barrier (`barrier.cvl`)

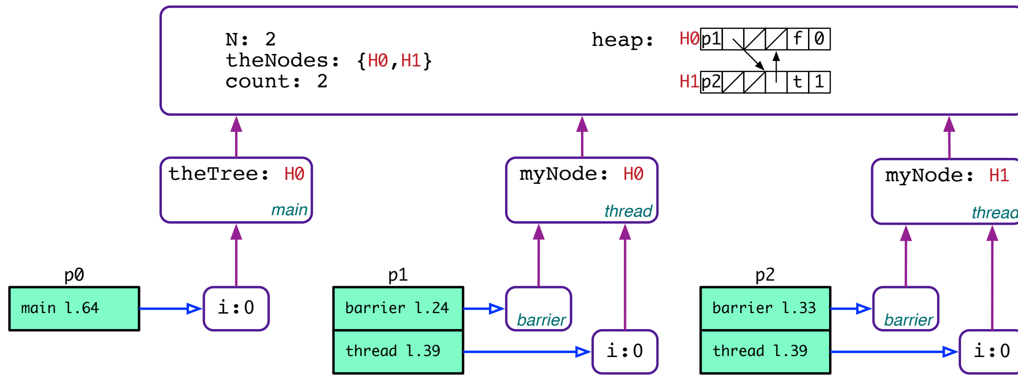


Fig. 8. A state of `barrier.cvl`

tions can be spawned just like any other function. While this feature is not used in the challenges, it is used extensively in other applications of CIVL, specifically to represent so-called “hybrid” parallel programs that use multiple levels of concurrency, such as multi-threaded MPI programs, or CUDA-C programs.

The `thread` function invokes the barrier three times. This use of 3 is arbitrary; it should be another parameter to the model. Some bound must be used, because the `version` field is incremented with each call and can therefore increase without bound—i.e., there would be an infinite number of states.[5]

The exact number of nodes `N` is specified as an input, and an arbitrary binary tree with `N` nodes is constructed. Verification time for $\texttt{N} = 4$ is 15s; $\texttt{N} = 5$ takes 203s.

CIVL checks automatically that the program does not deadlock, though the challenge did not specifically ask this.

The first task is to prove an invariant—a property that holds at every state. CIVL does not provide a way to specify invariants, but I was able to approximate this using assertions. The claim is that at any state, for any node n, if n.`sense` holds then m.`sense` holds for all descendants m of n. At the point just after `sense` is set to *true*, I inserted a function that checks that all descendants of `myNode` have *true* sense. Moreover, at the point just after waiting for `sense` to become *false*, I inserted a function that checks that all ancestors of `myNode` have *false* sense (this function was added in the revision). Putting these together, one has something very close to the requested invariant.

The second task is to show there is a state at which all nodes have the same `version`. It seemed to me that such a state would occur just as the root increments its `version`. I inserted code at this point to check this assertion. However, in the original statement of the challenge, the comment about the two statements `sense=true` and `version++` executing simultaneously did not appear, and my draft solution therefore did not include those two statements within an `$atomic` block. CIVL reported a violation of the assertion.

One of the purported advantages of model checking is its ability to generate a counterexample execution *trace* when a property is violated. This was certainly the case here. I have also found that the ability to produce *minimal* counterexamples is extremely useful for defect understanding. So I set $\texttt{N} = 2$ and ran the verifier with the `-min` flag, which instructs the verifier to find a path of minimal length to a violating state. The file `barrier_bad.cvl`, included in the experimental archive, demonstrates

[5]In other barrier examples distributed with CIVL, the driver is actually an infinite loop, yet the verifier still converges because the number of states is finite, so the loop will eventually reach states seen before.

this strategy. The minimal counterexample consists of 49 trace steps, the first 36 of which deal with the initialization phase of the program. After finding the counterexample, the trace can be *replayed* using the command

```
civl replay -showTransitions barrier_bad.cvl
```

to show the step-by-step sequence leading to the violation. An excerpt of the final part of the trace follows:

```
Step 43: Executed by p1 from State 43:
  53->54: (*(&<d0>heap.malloc0[0][0])).sense=true ...
Step 47: Executed by p2 from State 47:
  53->54: (*(&<d0>heap.malloc0[1][0])).sense=true ...
Step 48: Executed by p2 from State 48:
  54->55: ENTER_ATOMIC [_atomic_lock_var:=p2, p2.atomicCount:=1]
  55->56: (*(&<d0>heap.malloc0[1][0])).version=0+1
  56->57: TRUE_BRANCH_IF (guard: (void*)0==((struct _node)*)0)
  57->58: i=0 at barrier_bad.cvl:38.6-12 "int i=0"
  58->59: LOOP_BODY_ENTER (guard: 0<2) at barrier_bad.cvl:38.15-17 "i<N"
Error 0:
CIVL execution violation in p2 (kind: ASSERTION_VIOLATION, certainty: PROVEABLE)
at barrier_bad.cvl:38.25-72 "$assert(theNodes[i]->version  ... )":
Assertion: ((*((theNodes)[i])).5==(*(myNode)).5)
        -> 0==1
        -> false
Step 49: Trace ends after 49 trace steps.
Violation(s) found.
```

In this trace, `p1` sets its `sense` to *true*, then the root process `p2` intervenes, enters the barrier, sets its sense to *true*, and increments its version from 0 to 1. At this point, `p2` checks the assertion and discovers that `p1`'s value of 0 differs from its value of 1.

By examining this trace, it became clear that the two statements should happen atomically. I notified the organizers, as did one other participant (Bart Jacobs), and the organizers quickly fixed the statement. I was not able to complete the second task in time, but did so shortly after the competition. With the two statements inside an `$atomic` block, CIVL was able to verify the complete program for $\mathtt{N} \leq 5$.

6. CONCLUSION

I have presented CIVL solutions to the VerifyThis 2016 challenges that address almost all of the tasks. In each case, CIVL was able to verify the specified properties within small but non-trivial bounds, e.g., 16×16 matrices for Strassen multiplication, binary trees with height at most 4 for Morris' tree traversal algorithm, and up to 5 threads for the tree barrier. The verification runtimes range from a few seconds to a few minutes.

I believe the solutions provide some evidence for the usability of CIVL by programmers of moderate skill. The solutions are short, and do not require much knowledge beyond basic C. A few new primitives and concepts are needed, but these are minimal and do not require great conceptual leaps. The bulk of the work involves setting up a general environment and driver, which is similar to the work involved in ordinary program testing.

Detailed comparisons with other tools are beyond the scope of this paper, but a look at some of the other solutions can give a feel for the differences between deductive approaches and that of CIVL.

CIVL was the only tool to solve the Strassen task during the competition. Afterwards, three participants used Why3 [Filliâtre and Paskevich 2013] to construct a complete solution to challenge 1. The solution includes an implementation and proof

of a version of Strassen's algorithm for square matrices of any size, not just powers of 2; see [Clochard et al. 2016a] and [Clochard et al. 2016b]. The solution consists of over 1000 lines of WhyML code, including function contracts, predicate, type, and function definitions, invariants, axioms, lemmas, and theories—including an axiomatic theory of matrix arithmetic which could be re-used in other contexts. All of the verification conditions generated from this system are discharged by fully automatic theorem provers. This is a landmark achievement, but certainly required a level of sophistication beyond that of a typical programmer, as well as a good deal of effort.

Solutions to challenges 2 and 3 using VeriFast are also notable [Jacobs 2016]. These consist of Java code with annotations in a language based on separation logic. The annotations are used to construct a proof which is automatically checked by VeriFast. The solution to challenge 2 is complete (excluding the bonus task on depth-first order) and includes 92 non-whitespace annotation lines in addition to the implementation code. The solution to challenge 3 does not address the second task on the consistency of the version number, but is otherwise complete; it uses over 220 non-whitespace annotation lines.

The cognitive process is also different. Construction of a proof requires a deep understanding of the algorithm. This was evident in the post-competition discussion and in the solutions mentioned above. The user who succeeded in constructing a proof ended up understanding exactly *why* the Morris algorithm works, *why* the Strassen algorithm computes the product of the two matrices, and so on. In contrast, the user of CIVL can almost treat the algorithm as a black box. The exception is when something goes wrong, such as the missing atomicity requirement in challenge 3. In that case, CIVL's ability to generate a minimal counterexample proved very useful in understanding the defect.

Several improvements to CIVL would help it solve problems like those discussed here. These include:

— reporting the presence or absence of cycles in the state space
— additional specification primitives such as a `$sum` operator
— a way to specify invariants (properties expected to hold at every state), or even general temporal properties
— a mechanism to help generate arbitrary trees and other dynamic data structures
— the ability to specify loop invariants and to use them to verify programs without bounding input sizes.

The CIVL project is actively working on these and other improvements.

ACKNOWLEDGMENTS

I am grateful to Dr. Manchun Zheng and Ziqing Luo for their leading roles in the development of CIVL.

Funding for the CIVL project is provided by the U.S. National Science Foundation under awards CCF-1319571, CCF-1346769 and CCF-0953210.

This material is also based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number DE-SC0012566.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

REFERENCES

ACSL 2016. ACSL: ANSI/ISO C Specification Language. http://frama-c.com/acsl.html. (2016). Accessed Aug. 25, 2016.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 171–177. http://dl.acm.org/citation.cfm?id=2032305.2032319

Clark Barrett and Cesare Tinelli. 2007. CVC3. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science)*, Werner Damm and Holger Hermanns (Eds.), Vol. 4590. Springer, 298–302.

B. F. Caviness. 1970. On Canonical Forms and Simplification. *J. ACM* 17, 2 (April 1970), 385–396. DOI:http://dx.doi.org/10.1145/321574.321591

CIVL 2017. CIVL: Concurrency Intermediate Verification Language. https://vsl.cis.udel.edu/civl. (2017). Accessed Mar. 14, 2017.

Martin Clochard, Léon Gondelman, and Mário Pereira. 2016a. The Matrix Reproved (Verification Pearl). In *Verified Software. Theories, Tools, and Experiments: 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers*, Sandrine Blazy and Marsha Chechik (Eds.). Springer, 107–118. DOI:http://dx.doi.org/10.1007/978-3-319-48869-1_8

Martin Clochard, Léon Gondelman, and Mário Pereira. 2016b. Solutions — VerifyThis 2016. http://toccata.lri.fr/gallery/verifythis2016.en.html. (2016). Accessed March 8, 2017.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340. DOI:http://dx.doi.org/10.1007/978-3-540-78800-3_24

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3: Where Programs Meet Provers. In *Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP'13)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer-Verlag, Berlin, Heidelberg, 125–128. DOI:http://dx.doi.org/10.1007/978-3-642-37036-6_8

Marieke Huisman, Rosemary Monahan, and Peter Müller. 2016a. VerifyThis Verification Competition to be held at ETAPS 2016, Saturday, 2 April 2016. http://etaps2016.verifythis.org. (2016). Accessed March 8, 2017.

Marieke Huisman, Rosemary Monahan, Peter Müller, and Erik Poll. 2016b. *VerifyThis 2016: A Program Verification Competition*. Technical Report TR-CTIT-16-07. Centre for Telematics and Information Technology, University of Twente, Enschede. http://eprints.eemcs.utwente.nl/27060/01/sttt-summary.pdf.

IEEE. 2004. IEEE POSIX 1003.1c Standard. http://www.unix.org/version3/ieee_std.html. (2004). Accessed Oct. 30, 2015.

Bart Jacobs. 2016. Partial Solutions to VerifyThis 2016 Challenges 2 and 3 with VeriFast. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs (FTfJP'16)*. ACM, New York, NY, USA, Article 7, 6 pages. DOI:http://dx.doi.org/10.1145/2955811.2955818

Message-Passing Interface Forum. 2015. MPI: A Message-Passing Interface Standard, Version 3.1. http://www.mpi-forum.org/docs/docs.html. (4 June 2015).

Joseph M. Morris. 1979. Traversing Binary Trees Simply and Cheaply. *Inf. Process. Lett.* 9, 5 (1979), 197–200. DOI:http://dx.doi.org/10.1016/0020-0190(79)90068-1

NVIDIA. 2017. CUDA Zone. https://developer.nvidia.com/cuda-zone. (2017). Accessed March 14, 2017.

OpenMP 2017. OpenMP web site. http://www.openmp.org. (2017). Accessed March 14, 2017.

Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: The Concurrency Intermediate Verification Language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, Article 61, 12 pages. DOI:http://dx.doi.org/10.1145/2807591.2807635

Stephen F. Siegel and Timothy K. Zirkel. 2011. TASS: The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science* 5, 4 (2011), 395–426.

Martin Thoma. 2013. Part II: The Strassen algorithm in Python, Java and C++. https://martin-thoma.com/strassen-algorithm-in-python-java-cpp/. (Jan. 2013).

Wikipedia. 2016. Strassen algorithm. https://en.wikipedia.org/wiki/Strassen_algorithm. (2016).