CIVL The Concurrency Intermediate Verification Language

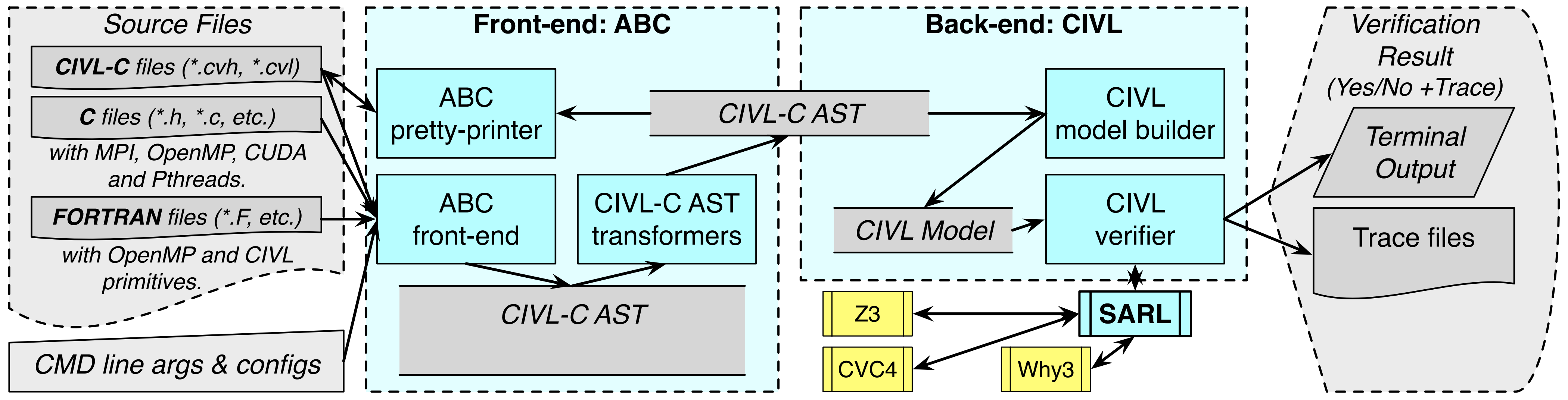
University of Delaware

Stephen F. Siegel, Ziqing Luo, Wenhao Wu, Paul D. Hovland and Jan Hückelheim

http://civl.cis.udel.edu/app/

1. Abstract

- CIVL: Concurrency Intermediate Verification Language
- Verify correctness of Scientific Computing applications
- Supported languages: **C, Fortran** (work in progress)
- Supported Parallelism: **OpenMP, MPI, CUDA, Pthreads**
- Everything is translated to the intermediate language **CIVL-C**
- Verification techniques: model checking, symbolic execution
- Compare two programs, for example:
  - before / after refactoring
  - before / after parallelization or performance tweaks
- Compare to formal specification
- Find inputs that trigger bugs, for example deadlocks, race conditions, assertion violations, illegal pointer dereferences, memory leaks, division by zero, out-of-bound array indexing
- Assumes real arithmetic, no model for roundoff errors. But: still useful to distinguish roundoff from other sources of error.



2. CIVL-C Language

- CIVL-C: intermediate language embedded in sequential C
- Concurrency and verification primitives
- Functions can be spawned as processes
- Nested function definitions in any scope
- With this, CIVL-C can represent semantics of message passing and shared memory parallelism

Examples of CIVL-C primitives:

$input: type qualifier for read-only global variables initialized with unconstrained values;
$output: type qualifier for write-only global variables as outputs;
$assume(*expr*) : statement informing the verifier to ignore the current execution unless *expr* holds;
$assert(*expr*) : checks that *expr* holds and reports error if it does not;
$forall {*T* i | *cond* } expr** : universal quantification; **$exists {*T* i | *cond* } *expr* is similar;
$choose {*stmt0 stmt1* … }: non-deterministic selection of one enabled statement;
$spawn *f* (*arg0*, …) : creates a new process executing function *f* ;
$when(*guard*) *stmt* : guarded command; enabled only when *guard* evaluates to true;
$atomic *stmt* : executes *stmt* without interleaving of other processes.

3. A CIVL Example

```c
// dot_product.c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int i, n;
  float a[8], b[8], sum;

  n = 8;

#pragma omp parallel for
  for (i = 0; i < n; i++) {
    sum = sum + (a[i] * b[i]);
    printf("loop %d\n", i);
  }
  printf("  Sum = %f\n”, sum);
}
```

- C/OpenMP program computing a dot product
- Data-race caused by missing reduction clause
- CIVL detects the data race on the variable sum
- The error message reported by CIVL shown below
- CIVL is able to generate a minimal counterexample for every defect found. which is useful for defects that are only triggered by a specific input.

```
CIVL v1.19 of 2019-02-08 – https://vsl.cis.udel.edu/civl
  ...
Thread 1 can not safely read memory location &<d5>sum, because thread 0 has
written to that memory location and hasn’t flushed yet.

Violation 0 encountered at depth 166:
CIVL execution violation in p2 (kind:ASSERTION_VIOLATION, certainty: PROVABLE)
  ...

Input:       ...
Context:     ...
Call stacks: ...

=== Source files ===
dot_product.c

=== Command ===
civl verify –input_omp_thread_max=2 dot_product.c

=== Stats ===
  ...

=== Result ===
The program MAY NOT be correct. See CIVLREP/dot_product_log.txt
```

4. CIVL Transformations

CIVL performs a sequence of transformations to build a pure CIVL-C AST merged from multiple translation units written in C, CIVL-C or Fortran language. Side-effect expressions and parallel dialects are transformed to functionally equivalent CIVL-C code. There is also a transformer that simplifies OpenMP programs using static analysis techniques.
Four transformation examples are given below:

4.1 MPI Transformation

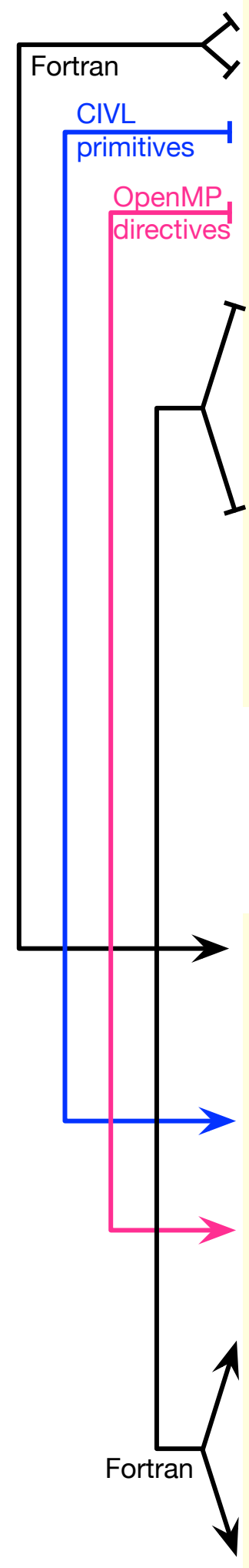
```c
$input int _mpi_nprocs;
$input int _mpi_nprocs_lo, _mpi_nprocs_hi;
$assume(_mpi_nprocs_lo <= _mpi_nprocs &&
      _mpi_nprocs <= _mpi_nprocs_hi);
$mpi_gcomm _mpi_gcomm =
      $mpi_gcomm_create($here, _mpi_nprocs);

void _mpi_process(int _mpi_rank) {
  MPI_Comm MPI_COMM_WORLD =
    $mpi_gcomm_create($here, _mpi_gcomm, _mpi_rank);
  /* ⟨ external–definitions ⟩ */
    ...
  int _gen_main(void) { ... } _gen_main(); //main func
  $mpi_gcomm_destroy(MPI_COMM_WORLD);
}

void main() {// This is the root process marked as p0
  $parfor (int i : 0 .. _mpi_nprocs–1)
    _mpi_process(i);
  $assert($forall (int i: 0..N–1)
    $mpi_gcomm_destroy(_mpi_gcomm);
}
```

A C/MPI program is automatically translated into the CIVL-C code shown above. The commented root process starts *`_mpi_nprocs`* CIVL processes. Each *`_mpi_process`* executes the original *`main`* function by calling *`_gen_main`*. Buffered messages are stored in a global communicator *`_mpi_gcomm`*.

```c
int main(void)
{ ... } //main func
```

4.3 Fortran Transformation

```fortran
      subroutine mxf1(a,n1,b,n2,c,n3)
        real a(n1,1), b(1,n3), c(n1,n3)
C$CVL $assert(n2 == 1);
C$OMP parallel default(shared) private(i,j,k)
C$OMP do
        do j=1,n3
          do i=1,n1
            c(i,j) = a(i,1)*b(1,j)
          ENDDO
        ENDDO
C$OMP end do
C$OMP end parallel
      end
```



ABC translates a Fortran program with OpenMP and CIVL-C primitives into a CIVL-C AST, which can be printed as a CIVL-C program.

```c
void MXF1(double A[][], int* N1, double B[][],
    int* N2, double C[][], int* N3) {
  int I; int J;
  $assert((*N2) == 1);
  #pragma omp parallel
  {
    #pragma omp for
    for (J = 1; J <= (*N3); J +=1 )
      for (I = 1; I <= (*N1); I += 1)
        C[J–1][I–1] = A[1–1][I–1]*B[J–1][1–1]
      }
    }
  }
}
```

4.2 OpenMP Simplification

CIVL combines several kinds of static analysis to determine if a parallelized OpenMP code is functionally equivalent to a sequentialized version of itself.
For two concurrent iterations `i` and `i′`, the OpenMP construct *`simd`* with *`safelen(8)`* clause shown below guarantees that:

$$i \neq i' \text{ and } |i - i'| < 8$$

```c
#include <assert.h>

int main() {
  int a[32], b[16];
  int *p0, *p1, *p2;

  p0 = a; p1 = p0+16; p2 = b;
#pragma omp simd safelen(8)
  for (int i=0; i<8; i++) {
    p0[i] = 1; p0[i+8] = 1;
    p1[i] = 1; p1[i+8] = 1;
    p2[i] = 2; p2[i+8] = 2;
  }
  // Check the correctness
  for (int i=0; i<32; i++)
    assert(a[i] == 1);
  for (int i=0; i<16; i++)
    assert(b[i] == 1);
}
```

Points-to Analysis

```
p → a[0],
q → a[16],
r → b[0]
```

Read/Write Analysis

read/write objects:

```
p[i]   : a[i],
p[i+8] : a[i+ 8],
q[i]   : a[i+16],
q[i+8] : a[i+24],
r[i]   : b[i],
 …
```

Dependency Analysis

```
a[i] indep-of a[i′],
a[i] indep-of a[i′+ 8],
a[i] indep-of b[i′],
 …
```

4.4 Comparison Combination

```c
//spec
$input int IN, K=3;
$assume(0 < IN);
$output int OUT;

int main() {
  OUT = IN*K;
}
```

CIVL proves the functional equivalence between a specification program (spec) and an implementation one (impl) by checking their output variables.
Two given programs are combined into a single one and the comparison is checked by equality assertions for each pair of output variables.

```c
//impl
$input int IN, K=3;
$output int OUT;

int main() {
  int sum = 0;
  int i = 0;
  for(; i < K; i++)
    sum = sum + IN;
  OUT = sum;
}
```

```c
$input int N, K=3;
$output int OUT_spec;
$output int OUT;

int $system_spec() {
  $assume(0 < IN);
  int $spec_main(void){
    OUT_spec = IN*K;
  }
  $spec_main();
}

int $system_impl() {
  int $impl_main(void){
    int sum = 0;
    int i = 0;
    for(; i < K; i++)
      sum = sum + IN;
    OUT = sum;
  }
  $impl_main();
}

int main() {
  $system_spec();
  $system_impl();
  $assert_equals(
      &OUT_spec, &OUT);
}
```

5. NUCLEI

```fortran
      do 140 loop=1,20
        do 110 i=1,lcx
          block(1)
  110   continue
        block(2)
        do 120 j=2,lcx+1
          block(3)
  120   continue
        block(4)
        if(loop.eq.1) then
          block(5)
        else
          if(cond(loop)) goto 170
          block(6)
        end if
  140 continue
  170 block(7)
```

```fortran
  flag = 0.0
  do loop=1,20
    do i=1,lcx
      block(1)
    end do
    block(2)
    do j=2,lcx+1
      block(3)
    end do
    block(4)
    if(loop.eq.1) then
      block(5)
    else
      if(cond(loop)) then
        flag = 1.0
      else
        if(flag.eq.0.0) then
          block(6)
        end if
      end if
    end if
  end do
  block(7)
```

- Nucleon matter code
- R. B. Wiringa, et al. Equation of state for dense nucleon matter. Phys. Rev. C, 38:1010–1037, 1988
- Uses several obsolescent Fortran features: shared do termination, statement functions, and fixed form source
- Original code was modified to avoid goto, exit and break statements to enable automatic differentiation using OpenAD
- Modifications were supposedly semantics-preserving
- CIVL found a bug: goto statement (left side) was incorrectly replaced with an if-block (right side)
- Currently manual Fortran to C conversion necessary.

6. FLASH

```fortran
subroutine Heat(blockCount, blockList, dt, time)
  #include "Flash.h"
  #include "constants.h"

  ! use statements
  ! type declarations ...
  ! some initializations etc.

!$omp parallel do private(n,blockID,k,j,i,solnData,dimSize, ...)
   do n = 1, blockCount
     blockID = blockList(n)
     ! Get the block distribution for each openMP thread
     call Grid_getBlkIndexLimits(blockID,blkLimits,blkLimitsGC)
     call Grid_getBlkPtr(blockID,solnData)
     dimSize(:)=blkLimitsGC(HIGH,:)–blkLimitsGC(LOW,:)+1
     do k = blkLimits(LOW,KAXIS), blkLimits(HIGH,KAXIS)
        do j = blkLimits(LOW,JAXIS), blkLimits(HIGH,JAXIS)
           do i = blkLimits(LOW,IAXIS), blkLimits(HIGH,IAXIS)
             ! Read from solnData(:,i,j,k)
             ! Calculate updates
             ! Write to solnData(:,i,j,k)
           enddo
        enddo
     enddo
     call Grid_releaseBlkPtr(blockID,solndata)
  enddo
end subroutine Heat
```

- Work in progress : verify OpenMP parallelization in Flash application.
- CIVL-C supports all features shown in this example.
- CIVL’s Fortran front-end needs to be extended further:
  - Fortran interfaces
  - Fortran pointers
  - Fortran optional arguments
  - Fortran array assignments.

U.S. DEPARTMENT OF ENERGY
Office of Science

This material is 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.