Verifying Functional Equivalence Between C and Fortran Programs

Author: Wenhao Wu, **Advisor**: Stephen F. Siegel

UNIVERSITY OF DELAWARE

Introduction

The proposed prototype is an innovative equivalence verifier for ***Fortran*** and ***C*** programs by extending a **C**oncurrent **I**ntermediate **V**erification **L**anguage framework (*CIVL*). This prototype contains a *Fortran* front-end for parsing, a transformer for converting parsing tree into CIVL abstract syntax tree (AST), and an equivalence verifier.

Additionally, a set of extensive experiments has been conducted, and the results show that the performance of this prototype is satisfactory.

Motivation

- **Complex HPC projects involve both Fortran and C programs.**
  - HPC code evolution from the legacy code to the new code.
  - Component replacements for maximizing performance or minimizing energy cost.
- **Difficult for human to assess correctness of complex HPC implementations**
  - Parallelism and a-synchronization in HPC programs
  - Massive numerical computations and matrix operations
  - Efficiency on verifying all possible situations.
- **Use equivalence verification to assess correctness for HPC programs.**
  - Static analysis for property-proof (i.e., divided-by-zero, , out-of-bound indexes, deadlock, etc.)

```
SUBROUTINE MXM(A,N1,B,N2,C,N3)
  REAL A(N1, N2), .. , C(N1, N3)
  DO J=1,N3
    DO I=1, N1
      C(I,J) = 0.0
      DO K=1, N2
        C(I,J) = C(I, J) + A(I,K)*b(K,J)
      END DO
    END DO
  END DO
  RETURN
END ! Basic matrix Multiply in Fortran
```

```
void MXM(double **a, int n1, .. ,int n3){
  ..
  #pragma omp parallel for private(k, i, j)
  for (int j = 1; i <= n3, i++)
    for (int i = 1; i <= n1; i++ ){
      c[i-1][j-1] = 0.0
      for (int k = 1; k <=n2; k++)
        c[i-1][j-1] += a[i-1][k-1]*b[k-1][j-1];
    }
  return;
} // Basic matrix multiply in C
```

```
SUBROUTINE MXM(A,M,B,K,C,N)
! Mat Multiply with a 4x4 pencil
  ..
  MRESID = IAND(M, 3)
  NRESID = IAND(N, 3)
  M1 = M - MRESID + 1
  N1 = N - NRESID + 1
    DO I=1, M-MRESID, 4
      DO J=1, N-NRESID, 4
        ..
      END DO
      IF (NRESID .NE. 0) THEN
        IF (NRESID .EQ. 1) THEN
          ..
        END IF
      END DO
  IF (MRESID .EQ. 0) THEN
    ..
  END IF
  RETURN
END
```

Are they equivalent?[1]

Objective

- **Parsing Fortran programs correctly according to the Fortran 2008 standard.**
  - An *Antlr* Fortran front end based on the *Fortran 2008 standard*
  - An extension of Fortran grammar to process *OpenMP* pragma and *CIVL* primitives.
  - A set of test cases for each language feature (i.e., grammar rule and specific behaviors).
- **Translating parallel Fortran and C programs into a common intermediate representation (IR) — CIVL-IR**
  - A transformer converting a Fortran program into CIVL IR
    **Fortran behaviors not in the C standard are simulated.*
- **Performing equivalence verification on the CIVL-IR**
  - A comparator wraps two input program sets and compares two set with the input range, the output range, and the labeled execution structure (e.g., loops and branches).

Approach

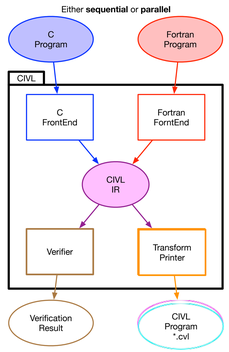

Figure.1 The high-level structure of comparing C and Fortran Programs

The implementation is an extension of ***CIVL***, which is a verification platform for concurrent *C* programs that use the *MPI*, *OpenMp*, *CUDA*, and *PThreads* APIs.

The **CIVL-IR**s are represented as sets of Abstract Syntax Trees constructed from each input *C or Fortran* source file and linked into one result AST by a Java-based *C* front-end.

CIVL traverses the state space in terms of the required properties (such as deadlock) to search for violations in all reachable states.

A ***Fortran* parser** has been integrated, which is derived from *Open Fortran Parser* (OFP)[2] and implements interfaces provided by the CIVL verifier. Furthermore, to adapt the AST structure used by *CIVL*, an **additional procedure** of transforming a Fortran parsing tree into CIVL-IR structure has been applied.

Figure.2 shows a abbreviated shortcut of CIVL-IR, which is generated from the code example shown on the left part. **Figure.3** shows the CIVL-C code generated by CIVL based on the Fortran program shown in the light green block.

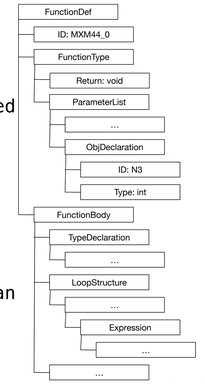

Figure.2 An example of the AST used in CIVL

Experimentation

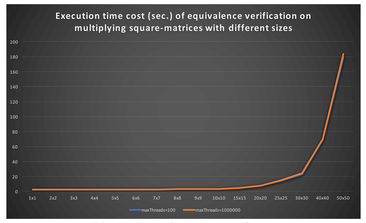

Figure.4 Time cost chart

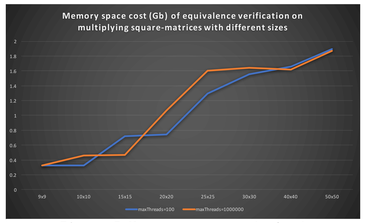

Figure.5 Memory space cost chart

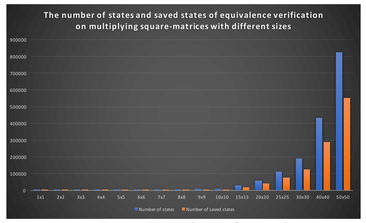

Figure.6 The number of total/saved states

Equivalence verification:

1). In the first set, CIVL compares two small piece of Fortran and C programs. Both of them calculate the modulo of 8 for random bounded input integers but with different methods. (i.e., one program uses a bit-wise *AND* operation with 7, and the other program uses integer division.)

2). In the second set, both Fortran and C programs will calculate a harmonic value for a given integer in a same arithmetical procedure. The interesting point is that those math functions are statements in Fortran but function calls requiring 'math.h' in C.

3). The last but the most interesting experiment compares a sequential Fortran program with two C parallel OpenMp implementations — a correct one and an incorrect one.

The functionality is to multiply matrices. Also, this piece of code is truncated from a real world Fortran program — Nekbone[1].
Some related statistics and results are shown in this section.

```
//CIVL-C code generated by CIVL based onthe Fortran code shown in light green block
void MXM44_0(double  A[][], int* N1, double  B[][], int* N2, double  C[][], int* N3) {
  int K;
  int I;
  int J;
  for (J = 1; J <= (*N3); J = J + 1) {
    for (I = 1; I <= (*N1); I = I + 1) {
      C[I - 1][J - 1] = 0.0;
      for (K = 1; K <= (*N2); K = K + 1) {
        C[I - 1][J - 1] = (C[I - 1][J - 1]) + ((A[I - 1][K - 1]) * (B[K - 1][J - 1]));
      }
    }
  }
  return;
}
```

Figure.3 Time cost chart

```
CIVL v1.11.1+ of 2017-08-08 -- http://vsl.cis.udel.edu/civl
...
=== Command ===
civl compare -input_omp_thread_max=10000000 -spec
mxmdriver_f.c ex1a.F -impl mxmdriver_c.c ex1c.c

=== Stats ===
   time (s)            : 7.46
   memory (bytes)      : 609746944
   max process count   : 1
   states              : 31301
   states saved        : 20510
   state matches       : 0
   transitions         : 31300
   trace steps         : 20509
   valid calls         : 193909
   provers             : cvc4, z3
   prover calls        : 32

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

Figure.7 CIVL output of comparing the sequential Fortran code and the parallel OpenMP C code (shown in the left part)

Results:

Those **correct** C implementations can be verified successfully without violations detected.
And **incorrect** implementations will result in an assertion violation (i.e., the assertion declares both outputs from the specification and the implementation outputs are exactly equivalent).
Most verification operations can be finished in one minute. For the last experiments of multiplying matrices, the cost increases exponentially.
All results are satisfactory and possible to be improved in the future.

```
CIVL v1.11.1+ of 2017-08-08 -- http://vsl.cis.udel.edu/civl

Violation 0 encountered at depth 6220:
.. (kind: ASSERTION_VIOLATION, certainty: PROVEABLE)
at mxmdriver_f.c:15.0-31
$output double OUT[mSize][mSize];
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Context: 0 <= SIZEOF(<75>) - 1 && 0 <= SIZEOF_REAL - 1
Assertion violated: $equals(&(OUT_spec), &(OUT))
Evaluation:
           &(OUT_spec)=&<d0>OUT_spec
           &(OUT)=&<d0>OUT
Result:
           <d0>OUT_spec=
| [0]={[0]=285, [1]=240, .. , [9]=-120}
 ..
           <d0>OUT=
| [0]={[0]=285, [1]=330, .. , [9]=690}
 ..

Input:
  mSize=10
  _omp_thread_max=10
Context:
  0<=(SIZEOF(75)-1)
  0<=(SIZEOF_REAL-1)
Call stacks:
process 0:
  main at mxmdriver_f.c:15.0-6 "$output"

 ..
Terminating search after finding 1 violation.

=== Command ===
civl compare -enablePrintf=false -inputmSize=10 -spec
mxmdriver_f.c ex1a.F -impl -input_omp_thread_max=10
mxmdriver_c.c ex1c.c

=== Stats ===
   time (s)              : 3.37
   memory (bytes)        : 432537600
   max process count     : 1
   states                : 9351
   states saved          : 6345
   state matches         : 0
   transitions           : 9351
   trace steps           : 6219
   valid calls           : 67496
   provers               : cvc4, z3, cvc3
   prover calls          : 2

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

Figure.8 CIVL output of detecting equivalence verification violation.

Conclusion

Goals achieved by current system are listed below:

- **Verifying basic sequential Fortran programs**
  - Program Units (Program, Subroutine, Function, Data Block)
  - Structures (DO-Loop, IF-branche, and Computed-GOTO)
  - Statements
    > Explicit/Implicit type declarations, CALL statements, PRINT statement, Bit-wise statements, mathematical statements, etc.
  - Expressions, Operators, Identifiers.
- **Comparing a sequential Fortran program with a concurrent C program.**
  - Equivalence verification performed by analyzing the ranges of the inputs and the outputs.
- **Transform Fortran programs into CIVL-C programs**
  - Both C and Fortran programs are transformed into CIVL-IR, which can be presented as a CIVL-C program.

Future Work

- **Completely implemented Fortran front-end:**
  - Develop the methods for transforming each Fortran behavior into CIVL-IR.
- **Completely support external APIs (e.g., openMP)**
  - Based on the current Fortran grammar, extend the front end to process external APIs
- **Integrate CIVL primitives into Fortran grammar**
- **Extend the current verification system for newly introduced Fortran Features**
  - Some Fortran behaviors are not existed in C, so that the current verification system need to be improved for processing those new properties defined in Fortran standard.
- **Extend the current system for advanced equivalence verification**
  - Based on the current equivalence verification system for both inputs and outputs, a stronger verification procedure will be developed for the program structure and used memory space.

Further Info

Scan the square code to visit the website of CIVL for more information!

References

[1] Paul Fischer and Katherine Heisey. NEKBONE: Thermal Hydraulics Mini-Application. Argonne National Laboratory,Oct. 2013.
[2] Open Fortran Parser, Craig E Rasmussen and Matt Sottile. Open Fortran Project.

Acknowledgement

The whole project is funded by:
National Science Foundation (CCF-1319571)