Modulo-X: A Simple Transformation Language for HPC Programs

Ferosh Jacob, *Department of Computer Science, University of Alabama* **Contact: fjacob@crimson.ua.edu**
Jeff Gray, *Department of Computer Science, University of Alabama*
Purushotham Bangalore, *Department of Computer and Information Sciences, University of Alabama at Birmingham*

Introduction

Software modularity, driven by the concept of separation of concerns, is a desired property for the development and evolution of software. For a community like High Performance Computing (HPC), where the programming models and architectures change constantly, modularity of the source code plays a pivotal role. Often, HPC programmers must rewrite existing programs to a new environment to deliver the optimum performance possible. There is a need for tools and techniques that can make such transformations easier and simpler to perform.

In this poster, we introduce a simple source transformation language named Modulo-X for FORTRAN and C (popular languages in the HPC community), through which modularity of HPC code can be improved. We follow a language-independent approach, such that programs in both C and FORTRAN are considered for modularity improvement.

Implementation Details of Modulo-X

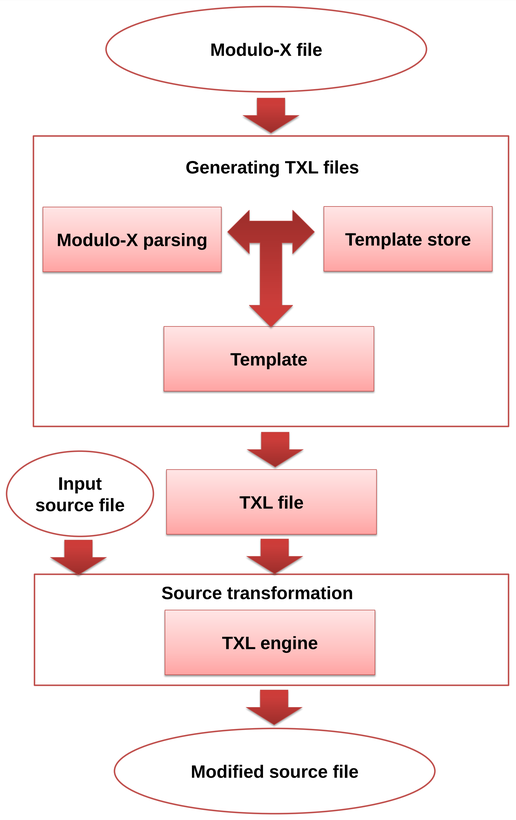
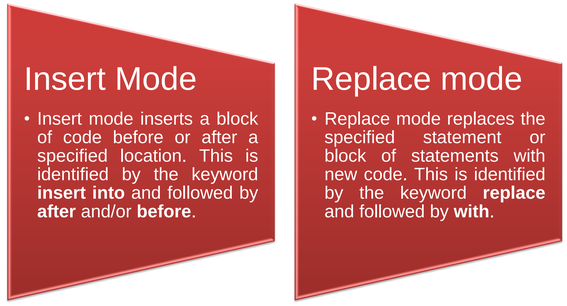


Figure 1. High level design diagram of Modulo-X

A high-level design diagram illustrating the translation process of Modulo-F is shown in Figure 1. As can be seen in the figure, a Modulo-F file is parsed to identify the template required for performing the specified action; the template is populated with values from the Modulo-X file and passed to the TXL engine along with the source FORTRAN/C file. The TXL engine transforms the input source file to the output source file using the newly created TXL file. Currently, there are four templates available for transformation. For a specified criterion, there are templates to:

1) Insert statements before or after the criterion (global scope)
2) Insert statements before or after the criterion (local scope)
3) Replace or remove statements (global scope)
4) Replace or remove statements (local scope)

Modulo-X Overview



Actions can be specified for a function (in the Insert example in Figure 2, the action occurs at the EMBAR FORTRAN procedure) or globally (in the Replace example in Figure 2, the action occurs at all the locations that match the criterion).

```
//Insert example (FORTRAN)
insert into EMBAR having {
  exprStatement: "Mops=2.d0**(m+1)/tm/1000000.d0"
}
after {
  call print_results( Mops)
}.
//Replace example (FORTRAN)
replace * having {
  //Linear congruential random  generator
  funcStatement: "randlc"
}
with {
  //Lagged Fibonacci random  generator
  t2 = randlf(t1, t1)
}.
```

Figure 2. Modulo-X Insert and Replace examples

Modulo-X is a transformation language targeted for procedural languages like FORTRAN and C. Two basic Modulo-X statements are shown in Figure 2.

The Insert example inserts the statement call `print_results(Mops)` inside the FORTRAN function named `EMBAR` after the FORTRAN statement `Mops=2.d0**(m+1)/tm/100000`.

The Replace example replaces all the occurrences of `randlc` function calls with `randlf(t1,t1)`.

In general, each **Modulo-X statement executes an action at a specified location of code.**

```
//Region definition using compound statements
region R1 {
  forStatement: "i=0;i<ihi;i++"
}
```

Figure 3. Defining region using for loop

```
//Region definition using simple statements
region R1 {
  start exprStatement: "solution_num=0"
  end funcStatement: "timer_read"
}
```

Figure 4. Defining region using start and end

- **Expression statement** — exprStatement
- **Function statement** — funcStatement
- **For statement** — forStatement
- **While statement** — whileStatement

- ❑ The **criterion** specifies the location where an action must occur.
- ❑ A criterion is specified using the *having* keyword.
- ❑ The criterion can point to a statement or a block of statements. Figure 2 shows the specification of a statement and Figures 3 and 4 shows the specification of a block of statements (named as region).
- ❑ Every criterion follows the syntax that has the statement type followed by the actual statement. This is to allow language-independent transformations. Modulo-X allows four types of statement as shown below.

Case Study: Satisfy problem

Parallel Implementation

The evaluation of satisfiability for a given set of variable assignments can be computed independently of another evaluation with a different set of variable assignments. Two common parallel implementations for the problem are:

1. Execute all evaluations in parallel, and if an assignment combination satisfies the expression, update that information to a shared memory that is accessible to all the parallel execution.
2. Divide the possible variable assignment combinations equally between the parallel executions and at the end of the execution, merge all the information gathered by the individual executions.

Defining transformation regions

The first step in converting the sequential version of the Satisfy program to a parallel version is to identify the parallelizable regions in the program. In this case, the transformation region is a for loop that occurs after the statement `solution_num=0` and before `timer_end()`. This can be defined in two different ways, as shown in Figures 3 and 4.

```
include <omp.h>
insert into main having R1
before {
      //OpenMP parallel directive
#pragma omp for schedule(dynamic,16) nowait
        shared ( ihi, ilo, n )
        reduction ( + : solution_num )
}.
```

Figure 5. OpenMP implementation using insert

OpenMP and MPI implementations

To convert the sequential code (identified as R1 or R2) to an OpenMP implementation of the program, OpenMP directives must be included before the statements. In the case of an MPI implementation, it is preferable to replace the sequential implementation with MPI code. The OpenMP and MPI versions of the Satisfy problem using Modulo-X are shown in Figures 5 and Figure 6, respectively. Note that the libraries required for the implementations can be added using the `include` keyword.

```
include <mpi.h>
replace main having R1
with {
  int id,ilo=0,ilo2, ihi2,solution_num_local=0;
  MPI_Init ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &id );
  MPI_Comm_size ( MPI_COMM_WORLD, &p );

  //Divide and execute code
  ilo2 = ((p-id)* ilo + (id)*ihi )/(p);
  ihi2 = ((p-id-1)* ilo + (id+1)*ihi)/(p);

  for ( i = ilo2; i < ihi2; i++ ){
    i4_to_bvec ( i, n, bvec );
    value = circuit_value ( n, bvec );
    if (value == 1){
      solution_num_local = solution_num_local + 1;
    }
  }
  MPI_Reduce (...);
  MPI_Finalize ( );
}.
```

Figure 6. MPI implementation using replace

Related Work

Aspect-Oriented Programming (AOP) [2] modularizes crosscutting concerns. In [3], an aspect language for FORTRAN was introduced. We believe Modulo-X is easier to use with a low learning curve in comparison to an aspect language.

Conclusion

While creating a parallel version of a sequential program, some code sections may be duplicated in the translated version, which can hinder the evolution of the newly created program. This can be prevented if parallel sections of a program can be separated from the sequential sections.

In this poster, we introduced a transformation language, called Modulo-X, which can make the parallel to sequential conversion task easier without modifying the original source code. A case study is included to show how sequential code can be converted to two leading parallel programming models (i.e., OpenMP and MPI) using Modulo-X.

References

[1] E. W. Dijkstra. On the role of scientific thought. *In* Selected Writings on Computing: A Personal Perspective, pages 60–66. Springer-Verlag, 1982.
[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming*, pages 220–242, Jyv¨askyl¨a, Finland, June 1997.
[3] S. Roychoudhury, J. Gray, and F. Jouault. A model-driven framework for aspect weaver construction. *Transactions on Aspect-Oriented Software Development*, 8:1–45, 2011.