# Developing an Environment for Reconfigurable Mesh Algorithms using Multi-Core Computers

Authors:
Daniel Laevski ID: 307575472
Nadav Inbar ID: 039063466

Supervisor:
Esti Stein, M.Sc

**Abstract**: *The Reconfigurable Mesh (RM) serves as a theoretical model for massively parallel computing, but has recently been investigated as a practical architecture for many-cores with light-weight, circuit-switched interconnects. In order to advance the use of this technology there is a need for suitable and convenient working environment. As of today, there is a lack of programming environments, including languages, compilers, and debuggers for reconfigurable meshes. We will provide a graphical environment for writing and running algorithms for RM. The environment will support: compiling (translating) the program, and run it with a graphical interface for a better debugging and understanding the RM flow. We will build a suite of programs to solve the following tasks: define a programming language for our compiler (translator), write the compiler (translator), simulate the multi-threading execution of the program, and gave a graphical representation of the mesh during the running of the program. For building the compiler (translator) we will use the flex and yacc code generating tools.*

**Keywords**: reconfigurable mesh, process element, compiler, programming environment, multi-thread.

## 1. INTRODUCTION

A Reconfigurable Mesh consists of an array of processing elements and an inter-connect that can be switched to form different patterns of segmented buses. All processing elements execute cycles of bus configuration, communication, and constant-time computation in a lock-step. The Reconfigurable Mesh as a model for massively parallel computing has been quite popular in the 90s, a vast body of algorithms with very low runtime complexities have been developed for it. The practical exploitation of the low runtime complexities, however, has been hindered mainly by the unrealistic assumption that communication time is either constant or logarithmic in the number of processing elements. [1]

A Reconfigurable Mesh (RM) is one of models of parallel computation based on reconfigurable bus system and mesh of processors. Many results and efficient algorithms on the RM are known, such as sorting, graph related problems, computational geometry and arithmetic computation – e.g. the Convex Hull problem [2]. We make use of the channel of data flow (bus topology) to solve problems on the RM, but it is not easy to follow data since the bus topology changes dynamically during the execution of algorithm on the RM. For same reason, it is also hard to find collisions of data on bus. Therefore it is very hard to design, analysis, or understand the algorithms on the RM by hand. Algorithm visualization is very useful to understand the behavior of computational model. It also assists the user to design, analysis, and debug the algorithms. Normally, the user can specify the data and control the execution of algorithm via some user interfaces. [3]

To be able to write and examine, perceive the implications and visualize the algorithms which were mention above, it is necessary to translate the structure of the language to a module that can be executed and run. This task is performed by the compiler.

A compiler is fairly complex program, writing such a program, or even understanding it is not a simple task. And most computer scientist and professionals will never write a complete compiler. Nevertheless, compilers are used in almost all forms of computing, and anyone professionally involved with computers should know the basic organization and operation of a compiler. In addition, a frequent task in compiler applications is the development of command interpreters and interface program, which are smaller than compilers but which use the same techniques. A knowledge of this techniques is, therefore, of significant practical use. [4]

The rest of the paper is organized as follows. Section 2 reviews a theoretical background about the Reconfigurable Mesh model, and will explain abstract design of a compiler and its architecture and its aspects. Section 3 discuses about the previous designs for the programming environments of the Reconfigurable Mesh. Section 4 explains the specific construction of the RM language and its compiler. Section 5 will illustrate the software design document and the UI mockups.

## 2. THEORY

In this section the article will explain the general approach for the Reconfigurable Mesh model, the abstract design of its architecture and its aspects. Also, this part of the article will discuss about the basics for building a compiler.

### 2.1. Reconfigurable Mesh

Reconfigurable Mesh is One of the most interesting models of parallel computation is the Reconfigurable Mesh (RM). The RM consists of a mesh augmented by the addition of a dynamic bus system whose configuration changes in response to computational and communication needs. More precisely, a RM of size n x m consists of nm identical SIMD processors positioned on a rectangular array with n rows and m columns. It is assumed that every processor knows its own coordinates within the mesh: we let $PE_{i,j}$ denote the processor in row i and column j. Each processor $PE_{i,j}$ is connected to its four neighbors $PE_{i-1,j}$, $PE_{i+1,j}$, $PE_{i,j-1}$ and $PE_{i,j+1}$, provided they exist, and has four ports denoted by N(North), S(South), E(East) and W(West). Local connections within the processor between these ports can be established, under program control, creating a powerful bus system that changes dynamically to accommodate various computational needs. This yields a variety of possible bus topologies for the mesh, where each connected component is viewed as a single bus. [5]

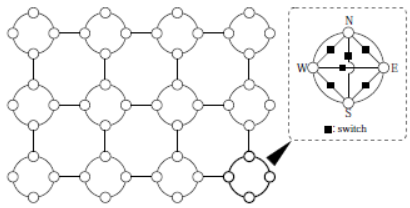


Fig. 1: A 3 x 4 reconfigurable mesh

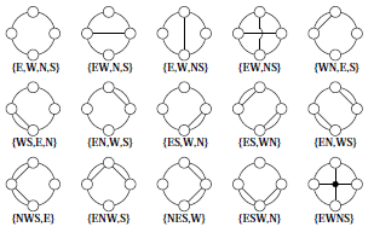


Fig. 2: Possible fifteen internal connections between the four I/O ports of a PE

All processors work synchronously and a **single step** of an RM consists of the following 4 phases:**, Read Phase:** receive data from a port (all processors connected to same sub-bus can receive same data sent at the previous phase) **Calculate Phase:** execute constant-time local computations**, Reconfigure Phase** change the configuration of inner buses, **Write Phase**: send data to a port (the data which is sent at this phase is transferred through a sub-bus).

### 2.2. Compilers

Compilers for high level programming languages are large, complex software systems. The development of a large software system should always begin with the decomposition of the overall system into subsystem (modules) with well defined and understood functionality. The division used should also involve sensible interface between the modules. This structuring task is in general not simple and frequently requires good intuition on the part of the engineer. Fortunately, compilers are very well understood software systems with tried and tested structuring principles which, with certain adaptation, can be applied to almost all high level programming languages. [6]

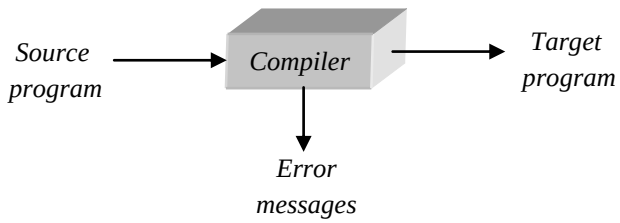


Fig. 3: A compiler

A compiler, as narrowly defined, consist of a series of phase that sequentially analyze giving form of a program and synthesize new ones, beginning with the sequence of characters constituting a source program to be compiled and producing ultimately, in most cases, a locatable object model that can be linked with others and loaded into machine's memory to be executed. As any basic text on compilers construction tells us, there are at least four phases in the compilation process as showed in figure 4.

i. **Lexical analysis**: breaks the source code text into small pieces called tokens. Each token is a single atomic unit of the language, for instance a keyword, identifier or symbol name. The token syntax is typically a regular language, so a finite state automaton constructed from a regular expression can be used to recognize it. This phase is also called lexing or scanning, and the software doing lexical analysis is called a lexical analyzer or scanner.
ii. **Syntax analysis**: involves parsing the token sequence to identify the syntactic structure of the program. This phase typically builds a parse tree, which replaces the linear sequence of tokens with a tree structure built according to the rules of a formal grammar which define the language's syntax. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.
iii. **Semantic analysis**: is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically proceeds the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.
iv. **Code generation**: that transforms the intermediate code into target code in the form of a relocatable object module or directly runnable object code. [7]

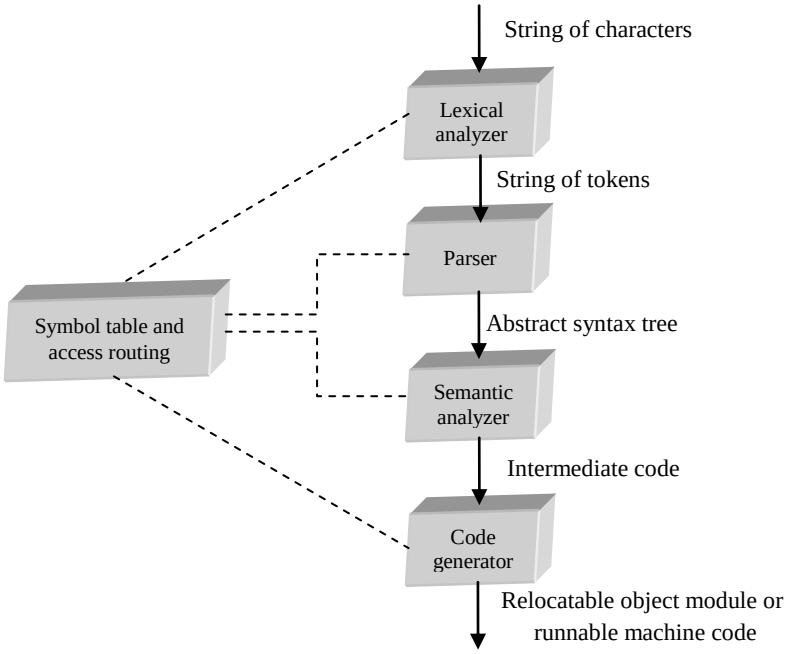


Fig. 4: High level structure of a simple compiler

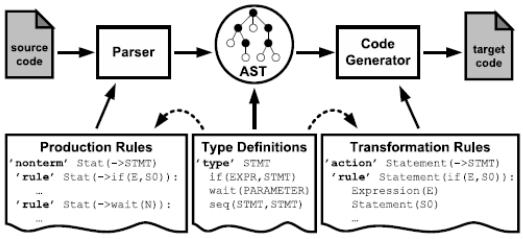


Fig. 5: Compiler specification

## 3. Related Work

As far as we know there are some tools to visualize algorithms on RM: the Simulator for the Reconfigurable Mesh Architecture by C. Steckel, M. Middendorf, H. ElGindy and H. Schmeck. [8], the java applet presented by K. Miyashita and R. Hashimoto [3], the Serial Simulation of RM by M. Murshed and R.P. Brent [9], and the ARMLang, Language and Compiler for Programming Reconfigurable Mesh by H. Giefers and M. Platzner [10].

The first one displays the behavior of algorithm graphically and the results of simulation are stored in a file. However the algorithm to be visualized in this simulator must be written in assembler language. Hence a sophisticated algorithm is implemented as a large file and there is more probability of existence of miss-codings or bugs. In addition, it cannot display the data that each processor holds neither statistic information.

The second, because the Java RM applet is implemented as an applet, it can run on any operating system using a standard web browser without re-compiling. However they are not dealing with multi-threating or multi core.

The third, called RPC (Reconfigurable Parallel C). can simulate a sub network of a 3 dimensional Reconfigurable Mesh known as mesh of meshes.

The last, discusses about the creation of an ARMLang compiler for the specification of lockstep programs on regular processor arrays, and a simulation environment that allows for debugging and visualization of the parallel programs.

## 4. Detailed Description

In this section the article will talk about the RM Language structure, statements, language flow control and specific methods of the Reconfigurable Mesh model. The section will also discuss regarding the code translation of the RM Language code to the target code we chosen as shown in the previous section.

### 4.1. The RM language

Unlike the general SIMD model on which the Reconfigurable Mesh bases, we do not impose the restriction to globally execute a single instruction. In our model, processing elements execute autonomously dedicated programs on local data while the overall algorithm proceeds synchronously in terms of steps.

Figure 6 shows an essential part of the BNF representation of RM language. The Program rule expresses the top-level structure of an RM language specification which comprises two parts. First, the RM language variables declaration. the programmer specifies the name and the dimensions of the mesh. For example, the RM language statement "Mesh xor[3][8];" defines a Mesh named xor for a 3 x 8 array of PE's, Consequently, the RM language compiler will generate 24 individual programs. Variables can be either scalar values or arrays. Following the Reconfigurable Mesh programming model, all data in RM language is completely distributed providing each processor with its own local copy of all variables. The second part of an RM language program is the program body, formed by a sequence of statements.

```
Program ⇒ variableDeclaration  functionDeclaration  Body
variableDeclaration ⇒ "Var" meshDefinition registerDefinition IOvectorDefinition
meshDefinition ⇒ "Mesh" meshIdentifier meshDimension3
meshDimension3 ⇒ "[" Integer "]" meshDimension2
meshDimension2 ⇒ "[" Integer "]" meshDimension1 | meshDimension1
meshDimension1 ⇒ "[" Integer "]" semicolon | semicolon
Body ⇒ "Program Begin" Step Functions "End"
Step ⇒   newline readPhase Block
         newline calculatePhase Block
          newline busPhase Block
          newline writePhase Block   |
         newline readPhase Block
          newline calculatePhase Block
          newline busPhase Block
          newline writePhase Block   Step
connectionConfiguration ⇒ "Connect" "(" switchConfiguration nextConfiguration ")"
nextConfiguration ⇒ "," nextConfiguration | switchConfiguration | semicolon
switchConfiguration ⇒ "VOID" | "NORTH-SOUTH" | "WEST-EAST" | "NORTH-WEST" |
         "NORTH-EAST" | "SOUTH-EAST" | "SOUTH-WEST" | "NORTH-WEST-SOUTH" |
         "NORTH-WEST-EAST" | "NORTH-EAST-SOUTH" | "EAST-SOUTH-WEST"|
         "NORTH-SOUTH-WEST-EAST" | "N0RTH" | "SOUTH" | "WEST" | "EAST"
statament ⇒ scanStatement | printStatement |executeStatement | assignRegisterStatement |
         assingDirectionStatement | connectionConfiguration
```

Fig. 6: BNF fragment of the RM language

RM language supports a variety of statements known from standard programming languages, but also some very distinctive statements specific to the Reconfigurable Mesh model. Among the familiar statements are assignments using the variables, a number of arithmetic operators and the assignment operator “=”. Control flow is coded with the familiar IF and IF-ELSE constructs or, for loops, with FOR statement. Line comments are started with double slashes. RM language’s more distinctive statements are described in the following:

Besides the normal use of control flow statement, RM language supports an additional use of conditional statement. Parameters such as the position of the PE, the overall number of processors (CORES), or the mesh dimensions (COL and ROW), can be integrated in those statement.

The interconnect of our Reconfigurable Mesh architecture bases on reconfigurable switches attached to the processing elements. To support communication, RM language includes a CONNECT statement which the processing element executes to set the switch pattern for its local switch, as well as Scan, Execute and Print statements for the actual communication. The CONNECT statement requires as parameter one of the allowed switch settings (switchConfiguration), which are shown in figure 6. For example, the pattern NORTH-WEST-SOUTH indicates that the NORTH, the WEST and the SOUTH port of the processing element need to be fused. Additionally, the processing element is connected to the resulting bus segment. Hence, a subsequent Scan operation reads in data that has passed the switch through the fused ports. The examples given in ANNEXES I, shows a simple sequence of a scan, execute and a print statement which allows the PEs to exchange values. Both connected processing elements can execute scan, execute and print simultaneously, as our architecture employs communication channels with directed links.

In addition, the ARM Language provides several assist functions as 1UN, 2UN, BIN and PST for processing the output from the mesh, as explained in ANNEXES II.

### 4.2. Compiling/Translating the RM code

The application will translate the RM code into C# code. Utilize the automated tools such as the "flex" and the "yacc" for parsing the RM language, the parser and the code generator will generate two programs, pe-code and mesh-code, the first will be the instructions (code) of each processing element. And the second program will be the code for the Mesh, in other words a code that supervise and synchronize all the phases of the RM, which are executed by each process element, with the pe-code program. This demonstrates how the SIMD model of the RM implements in the application.

The **lexical analysis**, or the scanning, phase of a compiler has the task of reading the source program as a file of characters and dividing it up into tokens. Tokens are like the words of a natural language: each token is a sequence of characters that represent a unit of information in the source program. Typical examples are **keywords** such as **if** and **Connect**, which are fixed string of letters, **identifier,** which are user defined strings, usually consisting of letters and numbers and beginning with a letter, **special symbols,** such as the arithmetic symbols " **+** " and " **%** ", as well as few multi-characters symbols, such as " < = " and " **==** ". In each case a token represents a certain pattern of characters that is recognized, or matched, by the scanner from the beginning of the remaining input characters. [4]

Methods of pattern of specification and recognition are primarily **regular expressions**, and **finite automate.** In our implementations this task is done by the automated tool, "flex", which his input is the regular expressions that we provide. (See Table 1).

| ASCII | Description |
|---|---|
| [A-Za-z0-9] | Alphanumeric characters |
| [A-Za-z0-9_] | Alphanumeric characters plus "_" |
| [^\w] | non-word character |
| [A-Za-z] | Alphabetic characters |
| [ \t] | Space and tab |
| [0-9] | Digits |
| [a-z] | Lowercase letters |
| [A-Z] | Uppercase letters |
| [A-Fa-f0-9] | Hexadecimal digits |

Table 1: Example of regular expressions

The **syntax analyzer** task is parsing or determining the syntax. The syntax of a programming language is usually given by the **grammar rules** of a **context-free grammar**, in a manner similar to the way the lexical

structure of the tokens recognized by the scanner is given by regular expressions. Indeed, a context-free grammar uses naming conventions and operations very similar to those of regular expressions. The major difference is that the rules of a context-free grammar are recursive. The consequences of this seemingly elementary change to the power of the representation are enormous. The class of structured recognizable by context-free grammars is increased significantly over those of regular expressions. The basic structure used is usually some kind of tree, called a **parse tree** or **syntax tree.** [4]

*sequence of tokens* $\xrightarrow{\textit{parser}}$ *syntax tree*

Fig. 7: The parsing process

In RM language the context-free grammar production rules as shown in figure 8:

Program ⇒ variableDeclaration functionDeclaration Body
variableDeclaration ⇒ "Var" meshDefinition registerDefinition IOvectorDefinition
registerDeclaration ⇒ "Register" identifiers
identifiers ⇒ registerIdentifier nextIdentifier
nextIdentifier ⇒ "," registerIdentifier nextIdentifier | semicolon
meshIdentifier ⇒ id
registerIdentifier ⇒ id

Fig. 8: Example of context-free grammar productions rules

For the full context-free grammar productions rules, see ANNEXES III.

The **semantic analysis computes** additional information needed for compilation once the syntactic structure of a program is known. This phase is referred to as semantic analysis because it involves computing information that is beyond the capabilities on context-free grammar and standard parsing algorithms and, therefore, is not regarded as syntax. The information computed is also closely related to the eventual meaning, or semantics, of the program been translated. Since the analysis perform by compiler is by definition static (it takes place prior to execution), such semantic analysis is also called static semantic analysis. In a typical statically typed language such as C, semantic analysis involves building symbol table to keep track of the meanings of names established in declaration and performing type inference and type checking on expressions and statements to determine their correctness within the type rules of the language. [4]

The **code generation** which task is to generate executable code for a target machine that is a faithful representation of the semantic of the source code. Code generation is the most complex phase of the compiler, since it depends not only on the characteristics of the source language but also on detailed information about the target architecture, the structure of the runtime environment, and the operating system running on the target machine.

## 5. PRELIMINARY SOFTWARE ENGINEERING DOCUMENTS

This section demonstrate the software design such as: Use case diagram, Class diagram, and UI mockups.

### 5.1. Use Case

This Use Case in figure 9 illustrates the interaction of the User with the Reconfigurable Mesh Development Environment, the programmer can open/save file, write an algorithm in the text editor, which is part provided with the environment as shown in figure 11. The Use Case is also shows the ability of the environment to compile, run and debug the algorithm as discussed above.

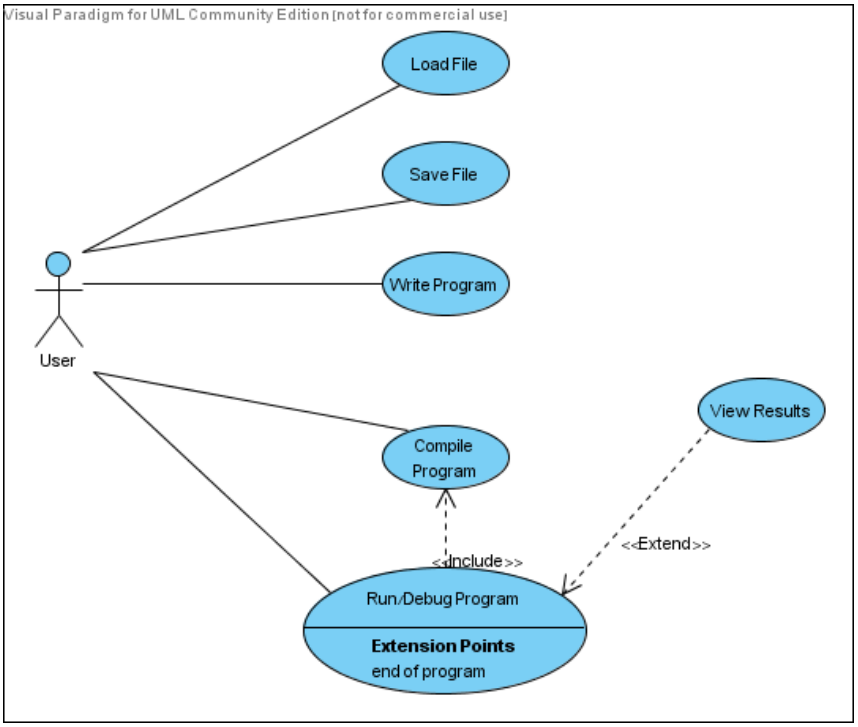


Fig. 9: Use Case diagram

### 5.2. Basic Data Structure

Representation of the RM Language Environment Data Structure on the UML model in figure 10. The processing element class contains the position, registers and Connection Configuration (Switch) of the PE. The Mesh class is responsible for the array of PEs that construct the Reconfigurable Mesh, and the busses that will be assembled during the execution of the algorithm.

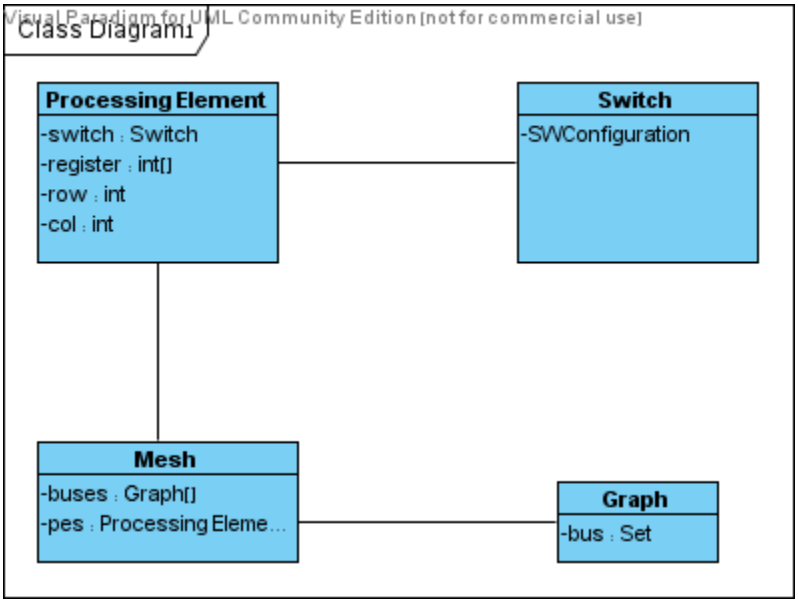


Fig. 10: Class diagram

### 5.3 GUI

The first UI mockup in figure 11 shows the text editor for writing algorithms in the RM language.

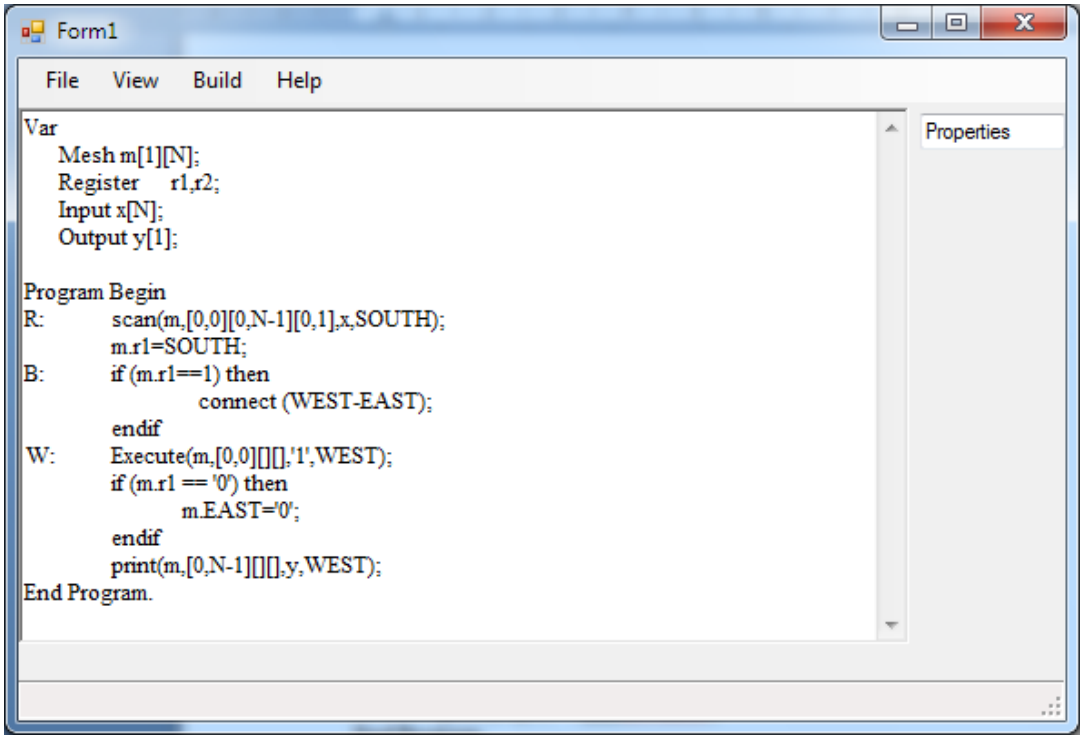


Fig. 11: RM language text editor.

Figure 12 shows the Mesh during an algorithm execution. Here you can see the internal bus and his connection configuration of each processing element and bus flow of the entire Mesh.

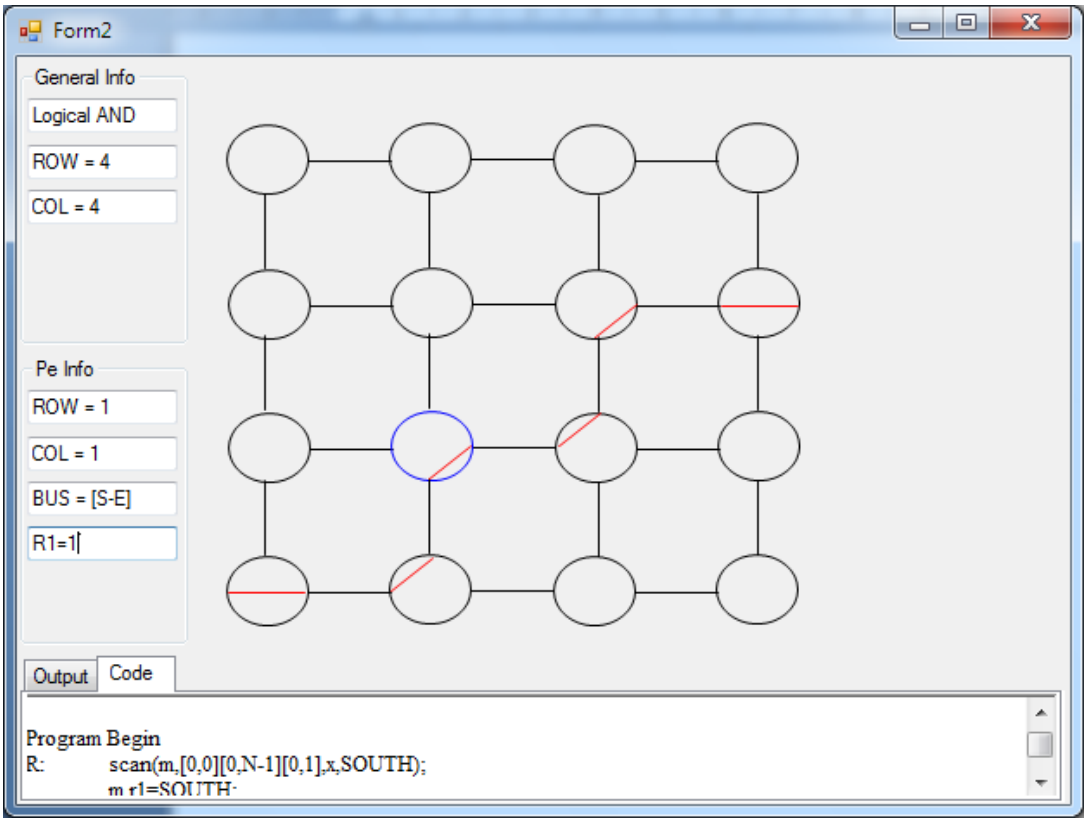


Fig. 12: RM language text editor.

### 5.4 Testing Plan

This section explains the testing procedure of the RM suite, and will give a few examples for testing plans.

**Testing the lexical module**: the test will enter a string of characters to lexical analyzer as an input, and will compare the returned tokens to the expected results. The test will be able to cope with both valid and invalid inputs, and return a suitable feedback. Table 2 shows a few examples.

| Input | Tokens - output | Test feedback |
|---|---|---|
| Mesh m[8][8]; | 'Mesh' 'identifier' '[' 'integer' ']' '[' 'integer' ']' 'semicolon' | OK |
| Register 5Mes; | 'register' ERROR | OK (This should be an error; an ID should not start with a digit). |
| m.r1 = SOUTH; | 'identifier$_1$' 'assignment statement' 'direction' 'semicolon' | OK |
| if (m.r1 == 1) then connect(WEST-EAST); | 'IF' '(' 'identifier$_1$ ' 'EQUAL' 'integer' 'THEN' 'CONNECT' '(' 'direction' '-' 'direction' ')' 'semicolon' | OK |
| m.r1 = m.r2 + m.r3; | 'identifier$_1$' 'assignment statement' 'identifier$_2$' '+' 'identifier$_3$' 'semicolon' | OK |
| m.r1 = + + + ; | 'identifier$_1$' 'assignment statement' '+' '+' '+' 'semicolon' | OK (although this is not a legal statement in the language, it's not lexical analyzer task to find this kind of an error). |

Table 2: Examples of inputs and outputs for the lexical module test.

**Testing the syntax analyzer module:** the test will enter a string of tokens to the syntax analyzer as an input, and will compare the returned parse tree to the expected results. Figure 13 shows an example.

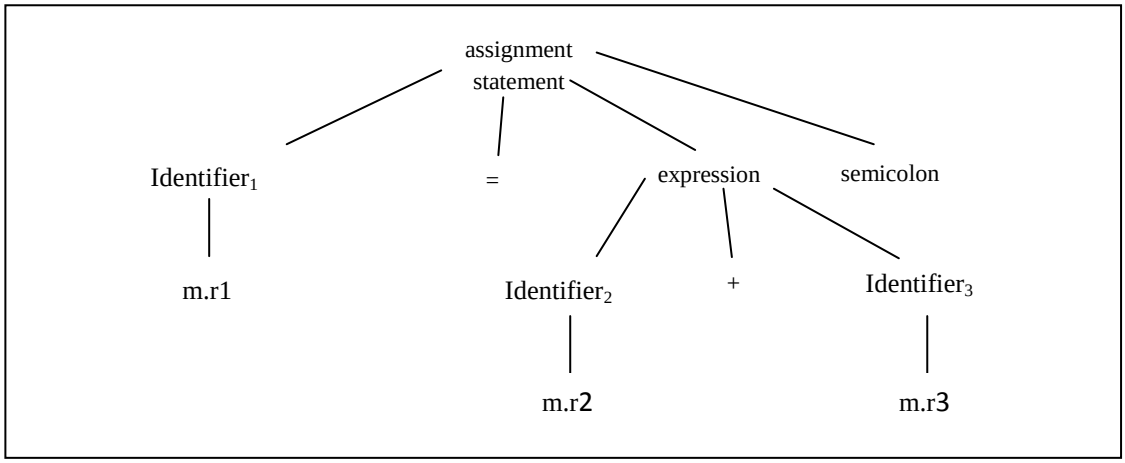


Fig. 13: Parsing tree for the expression m.r1 = m.r2 + m.r3;

**Testing the semantic and code generator module:** the test will enter a full algorithm of the RM language, e.g. in ANNEXE I, as an input, and will compare the returned code to the expected results. Naturally, when using the semantic module it contains the previous modules. Figure 14 shows an example.

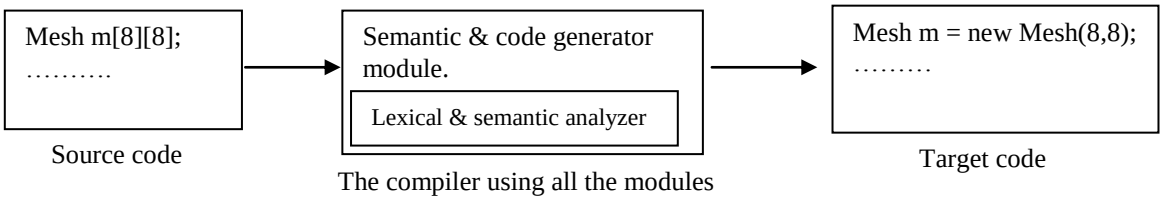


Fig. 14: Example of translating a RM language algorithm into target code.

**Testing the GUI module:** this test will be a manual test, we will run a RM language algorithm, and check his behavior to the expected behavior.

## References:


[1] H. Giefers and M. Platzner "Realizing Reconfigurable Mesh Algorithms on Softcore Arrays" appears in: "Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008. International Conference on", pp 41-48, July 2008.

[2] Reisis, D.I. "An efficient convex hull computation on the reconfigurable mesh", Parallel Processing Symposium, 1992. Proceedings, Sixth International, pp 142-145,

[3] K. Miyashita and R. Hashimoto, "A Java Applet to Visualize Algorithms on Reconfigurable Mesh", in Proc. of the 15th Workshops on Parallel and Distributed Processing, IPDPS ’00, 2000, pp. 137–142.

[4] K.C. Louden, "Compiler construction, principles and practice", PWS Publishing Company, pp 1-2, 31-138,1997.

[5] Y. Ben-Asher and E. Stein, "Basic Algorithms for the Asynchronous Reconfigurable Mesh", VLSI Design Volume 15 (2002), Issue 1, Pages 441-454.

[6] R. Wilhelm & D. Maurer, "Compiler Design", Wokingham, UK: Addison Wesley, pp 222 – 233, 1995.

[7] S. Muchnick, "Advanced compiler design and implementation", San Francisco, CA : Morgan Kaufmann, pp 1-3, 1997

[8] C. Steckel, M. Middendorf, H. ElGindy and H. Schmeck, "A Simulator for the Reconfigurable Mesh Architecture", "Parallel & Distributed Processing", pp 99-104, 1998.

[9] M. Murshed and R. Brent, “Serial simulation of reconfigurable mesh, an image understanding architecture”, Advances in Computer Cybernetics, vol. 5, pp. 92–97, 1998.

[10] H. Giefers and M. Platzner, "ARMLang: A Language and Compiler for Programming Reconfigurable Mesh Many-Cores", Parallel and Distributed Processing Symposium, International pp. 1-8, 2009 IEEE International Symposium on Parallel&Distributed Processing, 2009.

## ANNEXE I

**Examples of Algorithms in RM Language**

**<u>Logic AND</u>**

```
Var
        Mesh m[1][N];
        Register     r1, r2;
        Input x[N];
        Output y[1];

    Program Begin
R:  scan(m, [0,0][0,N-1][0,1], x, SOUTH);
    m.r1=SOUTH;
B:  if (m.r1==1) then
            connect (WEST-EAST);
    endif
W: execute(m, [0,0][][], '1', WEST);
    if (m.r1 == '0') then
        m.EAST='0';
    endif
    print(m, [0,N-1][][], y, WEST);
    End Program.
```

**<u>Left Most Determine</u>**

```
Var
    Mesh m[1][N];
    Register     r1, r2;
    Input x[N];
    Output y[1];

Program Begin
R: scan(m, [0,0][0,N-1][0,1], x, SOUTH);
   m.r1=South;
B: if (m.r1==0) then
            connect (WEST-EAST);
   endif
W: execute(m, [0,0][][],'0',WEST);
    if (m.r1 == '1') then
              m.EAST = '1';
    endif
R: if (m.r1 == 1) then
            m.r2=WEST;
            if (m.r2 == 0)
                NORTH = '1';
            else       NORTH = '0'
            endif
   endif
   print(m, [0,0][0,N-1][0,1], y, NORTH);
   pst(y);
  End Program.
```

## ANNEXE II

**Language Definition**

```
Var
        Mesh <name>[D1][D2][D3];
        register<name>,<name>,…,<name>;
        Input <name>[D1];
        Output <name>[D1];
Program
scan(<to>,[<start_D1>,<start_D2,<start_D3>][<end_D1>,<end_D2,<end_D3>][<jump_D1>,<jump_D2>,<jump
_D3>],
<from>,<Direction>);
print(<from>,[<start_D1>,<start_D2,<start_D3>][<end_D1>,<end_D2,<end_D3>][<jump_D1>,<jump_D2>,<ju
mp_D3>],
<to>,<Direction>);
execute(<to>,[<start_D1>,<start_D2,<start_D3>][<end_D1>,<end_D2,<end_D3>][<jump_D1>,<jump_D2>,<ju
mp_D3>],
<from>,<Direction>);
assign value to register:
<mesh_name>.<regname>=<direction>;
// note: The value will be assign to the all Mesh.
<mesh_name>[D1][D2][D3].<regname>=<direction>;
 // note : the value will be assigned to the dimension according to the number of dimensions
assign value to Direction:
<mesh_name>.<Direction> = <register_name>| <value>;
<mesh_name>[D1][D2][D3].<direction>=<register_name>| <value>;
```

**Special Assist Function;**

1UN(< Output_vector >) – sum the '1' in the Output Vector.

2UN(< Output_vector >) – sum the joined (to the left) '1' in the Output Vector.

BIN(< Output_vector >) – Decimal representation of the Output Vector.

PST(< Output_vector >) – Returns the location of the singular '1' in the Output vector.

## ANNEXE III

### The RM language BNF / production rules for the syntax analyzer

Program ⇒ variableDeclaration functionDeclaration Body

variableDeclaration ⇒ "Var" meshDefinition registerDefinition IOvectorDefinition

meshDefinition ⇒ "Mesh" meshIdentifier meshDimension3

meshDimension3 ⇒ "[" Integer "]" meshDimension2

meshDimension2 ⇒ "[" Integer "]" meshDimension1 | meshDimension1

meshDimension ⇒ "[" Integer "]" semicolon | semicolon

registerDeclaration ⇒ "Register" identifiers

identifiers ⇒ registerIdentifier nextIdentifier

nextIdentifier ⇒ "," registerIdentifier nextIdentifier | semicolon

meshIdentifier ⇒ id

registerIdentifier ⇒ id

Body ⇒ "Program Begin" Step Functions "End"

Step ⇒ newline readPhase Block
newline calculatePhase Block
newline busPhase Block
newline writePhase Block |

newline readPhase Block
newline calculatePhase Block
newline busPhase Block
newline writePhase Block  Step

readPhase ⇒ "R" ":"
calculatePhase ⇒ "C" ":"
busPhase ⇒ "B" ":"
writePhase ⇒ "W" ":"

connectionConfiguration ⇒ "Connect" "(" switchConfiguration nextConfiguration ")"
nextConfiguration ⇒ "," nextConfiguration | switchConfiguration | semicolon
switchConfiguration ⇒ "VOID" | "NORTH-SOUTH" | "WEST-EAST" | "NORTH-WEST" |
"NORTH-EAST" | "SOUTH-EAST" | "SOUTH-WEST" | "NORTH-WEST-SOUTH" |
"NORTH-WEST-EAST" | "NORTH-EAST-SOUTH" | "EAST-SOUTH-WEST"|
"NORTH-SOUTH-WEST-EAST" | "N0RTH" | "SOUTH" | "WEST" | "EAST"
basicDirection ⇒ | "N0RTH" | "SOUTH" | "WEST" | "EAST"
Block ⇒ statement | statement nextStatament

nextStatement ⇒
statament ⇒ scanStatement | printStatement |executeStatement | assignRegisterStatement |
assingDirectionStatement | connectionConfiguration

scanStatement ⇒"scan" "(" id "," algorithmCoordinates "," id "," switchConfiguration ")" semicolon
printStatement ⇒"print" "(" id "," algorithmCoordinates "," id "," switchConfiguration ")" semicolon
executeStatement ⇒"execute" "(" id "," algorithmCoordinates "," id "," switchConfiguration ")" semicolon
assingRegisterStatement ⇒
meshIndentefier Dimention "."registerndetefire "=" basicDirection semicolon
assingDirectionStatement ⇒ meshIndentefier Dimention "." switchConfiguration "=" value semicolon

algorithmCoordinates⇒ startCoordinates endCoordinates jumpCoordinates semicolon
StartCoordinates ⇒ Dim "," Dim | Dim "," Dim "," Dim
endCoordinates ⇒ Dim "," Dim | Dim "," Dim "," Dim
jumpCoordinates ⇒ Dim "," Dim | Dim "," Dim "," Dim
Dim ⇒ "["Integer "]"

value ⇒ Number | registerIdentifier