A MATLAB to Modelica Translator

Mohammad Jahanzeb[1], Arunkumar Palanisamy[1], Martin Sjölund[1], Peter Fritzson[1]

[1]PELAB – Programming Environment Laboratory
[1]Department of Computer and Information Science
Linköping University, SE-581 83 Linköping, Sweden
mjahanzeb@live.com, {arunkumar.palanisamy, martin.sjolund, peter.fritzson}@liu.se

Abstract

Matlab is a proprietary, interactive, dynamically-typed language for technical computing. It is widely used for prototyping algorithms and applications of scientific computations. Since it is a dynamically typed language, the execution of programs has to be analyzed and interpreted which results in lower computational performance. In order to increase the performance and integrate with Modelica applications it is useful to be able to translate Matlab programs to statically typed Modelica programs.

This paper presents the design and implementation of Matlab to Modelica translator. The Lexical and Syntax analysis is done with the help of the OMCCp (OpenModelica Compiler Compiler parser generator) tool which generates the Matlab AST, which is later used by the translator for generating readable and reusable Modelica code.

Keywords: Modelica, MetaModelica, Matlab, OMCCp, translation.

1 Introduction

The Matlab language is dynamically typed; it does not have explicit type declarations. A variable's type is implicit from the semantics of its operations and the type is allowed to dynamically change at runtime.

These features improves ease of use for prototyping and interactive use, but add heavy run-time overheads, such as runtime type checking, array bounds checking and dynamic resizing, to its interpretive execution. Therefore, Matlab programs often run slower than their counterparts which are written in conventional statically typed programming languages.

The main goal of this work is the development of a translator that accepts Matlab programs as input and generates Modelica code as output which is suitable for static compilation. Due to the complexity of the Matlab language, a realistic goal is to develop a translator for a subset of Matlab.

The translation task of Matlab to Modelica code mainly involves the front-end implementation of the Matlab to Modelica compiler. The OMCC (OpenModelica Compiler Compiler) compiler generation tool, which has been developed as a part of the OpenModelica project, can be used as a parser and translator generator extended with advanced error handling facilities. The tool is implemented in MetaModelica and integrated with the MetaModelica semantics specification language based on operational semantics for generating executable compiler and interpreter modules.

The OMCCp part of the OMCC tool makes the implementation of the first two stages of a compiler much easier and saves time. We have to just write the lexer and parser rules for the Matlab language and input them to OMCCp to generate the appropriate lexer and parser modules in MetaModelica. The generated parser builds the Abstract Syntax Tree (AST) for the Matlab source code that is parsed.

The Matlab AST is later used by the second phase of the Matlab-to-Modelica translator which performs a series of internal transformations and finally generates the Modelica-AST which is unparsed to readable Modelica code.

There exists a large body of computational algorithms which are programmed in the Matlab language. The need to have an easy way of using and incorporating such algorithms within Modelica models, as well as achieving improved performance, motivated the development of the Matlab to Modelica translator in the OpenModelica project.

This paper is structured as follows: Section 2 presents some related approaches. Section 3 describes the different steps involved in generating a compiler from

specifications in different specification formalism. Section 4 explains the design and implementation of the Matlab to Modelica translator with main focus on the translator. The translator description is for the subset of the Matlab language grammar for which translation to Modelica code is supported. Section 5 explains the type inference approaches. Section 6 describes the Modelica unparser. Section 7 presents Test results and performance measurements. Finally, Section 8 concludes the paper with a short discussion of achieved results.

2 Related Work

In this section we discuss a few related approaches for the compilation of Matlab code to statically typed languages.

The Matlab tamer is an extensible object-oriented framework for generation of static programs from dynamic Matlab programs implemented in Java. The Matlab Tamer supports a large subset of Matlab. It builds a complete call graph, transforms every function into a reduced intermediate representation, and provides typing information to aid the generation of static code [15]. In an earlier student project we tried to use the Matlab tamer framework directly for a translator to Modelica. However, that project was not completed.

MCFOR: A Matlab to Fortran95 compiler is designed for translating Matlab code to Fortran95. It generates readable and reusable Fortran95 code. MCFOR had very limited support for built-in functions. It showed that the numerical and matrix features of Fortran95 are a good match for the compiled Matlab, and that the static nature of the language, together with powerful Fortran95 compilers provides the potential for high performance [16].

3 Background

3.1 Generating Compiler Phases

The different phases of the compiler can be generated from a formal specification in different formalisms, as depicted in Figure 1.

Generally a compiler is divided into two parts, the front-end and the back-end. The scanner and parser constitute the front end phase whereas the optimization and code generator constitute the back-end phase of the compiler. In this paper we focus on the front-end parts of the compiler [1] [2], and use the existing Modelica unparser to generate the Modelica code from the internal Modelica AST

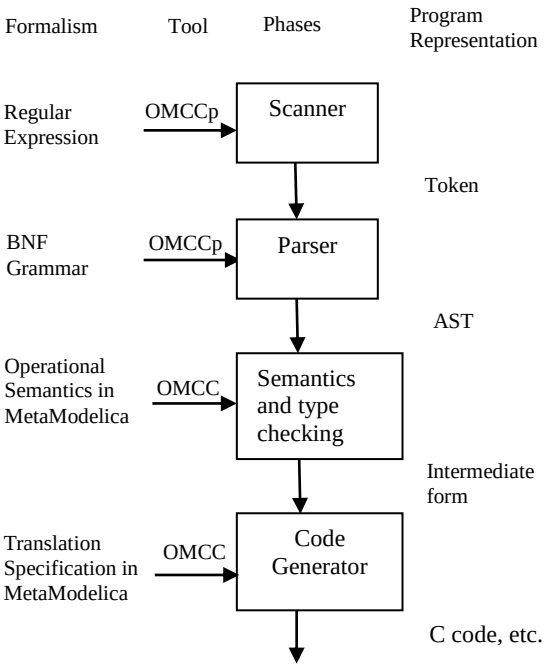

Figure 1. Structure of Compiler Generation using OMCC

3.2 Lexical Analysis

The Lexical analysis, performed by a scanner, is the first stage of the compilation process. It receives the source code as input and generates tokens. It also identifies the special tokens defined by the specified language making it simpler for the next phase of the compiler.

The structure of the tokens are usually specified by the use of regular expressions. There are several tools available that automate the labor of constructing the transition rules to identify the tokens for a scanner. We use the Flex tool for this purpose which generates C code; the generated C code containing tables is used by OMCCp to generate the appropriate lexer components in MetaModelica [1] [2].

3.3 Syntax Analysis

The syntax analysis, also called parsing, is the second stage of the compilation process. The parser takes the tokens generated by the lexer and determines whether the tokens are constructed according to the rules of the grammar. By doing this it creates the Abstract Syntax Tree (AST) if the input conforms to the defined grammar and otherwise reports an error message.

The AST is used as input to the back-end. The back-end uses the AST for type checking, optimization, and finally generates machine specific code. The grammar rules are usually specified in the form of BNF (Backus Normal Form). We use the BNF grammar rule format of the popular Bison tool for writing the grammar rules;

DOI 10.3384/ECP140961285

the generated C code contain parse tables that are used by OMCCp to generate appropriate parser components in MetaModelica [1][2].

3.4 Semantic Analysis

The semantic analysis is the phase in which the compiler adds semantic information to the parse tree and adds semantic information to the symbol table. This phase also 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 abstract syntax tree, meaning that this phase logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.

The translator which takes the Matlab AST as input performs a series of internal transformation to produce the corresponding translated Modelica AST which can be unparsed to the generated textual Modelica code [1] [2].

4 Design

The design architecture of the Matlab to Modelica translator is depicted in Figure2.

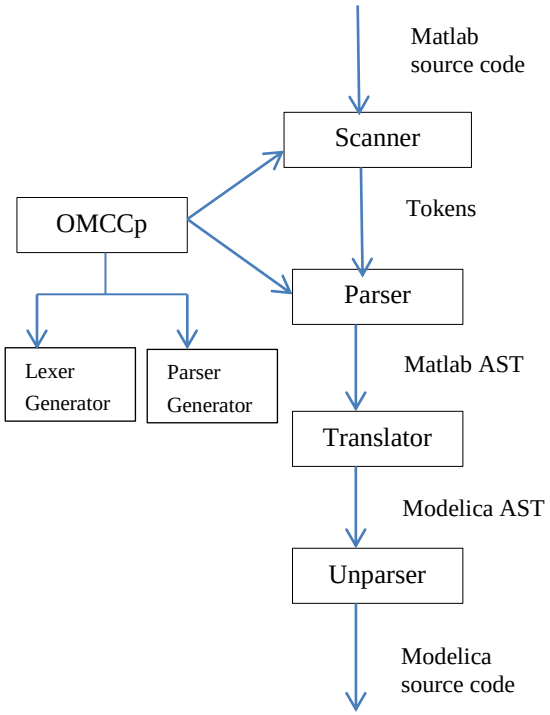

Figure 2. Structure of Matlab to Modelica translator

The translator contains a scanner and a parser for the Matlab source code, which build the Matlab abstract syntax tree during parsing. The scanner and the parser are implemented using the OMCCp tool which generates appropriate lexer and parser components in MetaModelica. The result of the parser is used as input to the translator which performs a series of internal transformations and generates a Modelica AST. The last stage is the Unparser where the Modelica AST is unparsed to generate the Modelica source code. The scanner, parser and the translator are implemented in MetaModelica. In this paper we focus on the Matlab to Modelica translator; for detailed information about OMCCp see [1] [2].

4.1 LexerGenerator

The LexerGenerator is the main package of the lexer which generates the necessary lexer subsystems in MetaModelica. This module generates three packages in MetaModelica namely LexerModelica.mo, LextableModelica.mo and LexcodeModelica.mo. For more information about these packages see [1] [2].

4.1.1 Lexer.mo

Lexer.mo is the main file which contains the calls to other functions in Lextable.mo and LexCode.mo that constitute the lexer. The main function of this package is to load the source code file and recognize all the tokens described by the grammar.
For recognizing a token the lexer.mo runs DFA (Deterministic Finite Automata) based on the transition arrays found in Lextable.mo. When it reaches an acceptance state it calls the function `action` in LexCode.mo which returns list of tokens that are input to the parser. The interface of the function which does this operation is given below.

```
function scan
  input String fileName "input source code
file";
  input list<Integer> program "source code
as a stream of Integers";
  input Boolean debug "flag to activate
the debug mode";
  output list<OMCCTypes.Token> tokens
"return list of tokens";
end scan
```

4.2 Parser Generator

The ParserGenerator package is the main package of OMCCp that generates the parser which performs the syntax analysis of the compiler. ParserGenerator generates four packages in MetaModelica namely TokenModelica.mo, ParserTable.mo, ParserCodeModeli

ca.mo and ParserModelica.mo. For more information about these packages see [1] [2].

4.2.1 Parser.mo

The main function of this package is to efficiently convert the list of tokens received by the Lexer into Abstract Syntax Tree (AST). The package also contains the implementation of the LALR algorithm. For performing this task the package uses the parse table located in the package ParseTable.mo, to perform the shift-reduce action calls it uses the package ParserCodeModelica.mo. The interface of the function which starts the construction of AST is given below.

```
function parse "realize the syntax
   analysis over the list of tokens and
   generates the AST tree"
  input list<OMCCTypes.Token> tokens "list
    of tokens from the lexer";
  input String fileName "file name of the
    source code";
  input Boolean debug "flag to output
    debug messages that explain the states
    of the machine while parsing";
  output Boolean result "result of the
    parsing";
  output ParseCode.AstTree ast "AST tree
    that is returned when the result
    output is true";
end parse;
```

4.3 Translator

The transformation *translator* is the third stage of the implementation which takes the Matlab AST as the input and performs a series of internal transformation and finally generates the Modelica AST. The translator package contains several functions which perform the above tasks by finding a possible mapping to a Modelica AST data structure. The implemented translator supports a particular subset of the Matlab language which is discussed in more detail in the following section.

4.4 Primary Function

A Matlab program consists of a list of files called M-files, which include a sequence of Matlab commands and statements. We can split the syntax of a Matlab function into three parts. First is the declaration part, which consists of the function name, the input and output formal parameters. The next part is the function body which consists of a list of statements which ends with the keyword `end`. A sample function in Matlab is given below.

```
function perfect = isperfect(value)
  sum = 0;
  for (divisor = 1 : value - 1)
    result = value / divisor;
    if (result == floor(result))
       sum = sum + divisor;
    end
  end
  if (sum == value)
     perfect = 1;
  else
     perfect = 0;
  end
end
```

The function name is `isperfect` which is defined in the declaration section. The `input` and `output` parameters are `value` and `perfect` respectively.

The body of function starts right after the declaration line and ends with keyword `end` which also ends the function. Due to the dynamic nature of the language there are no data type declarations in the Matlab code. All variables inside the function body are local to the `isperfect` function.

4.4.1 Translation of Matlab to Modelica function

A Modelica function consists of three parts. The first part is similar to a Matlab function which contains function name, input, and output formal parameters. Since the Modelica language has static typing we have to define each function formal parameter with its proper data type as well as`input` and `output` keywords.

The next part is `protected`. All local variables apart from `input` and `output` formal parameters have to be declared in the protected section with proper data types. Finally the body of function starts with the keyword `algorithm` and ends with `end` keyword. An example of the Matlab function `isperfect` translated to Modelica is presented below.

```
function isperfect
  input Real value;
  output Real perfect;
protected
  Integer sum;
  Real result;
algorithm
  sum:=0;
  for divisor in 1:value - 1 loop
      result:=value / divisor;
    if result == floor(result) then
      sum:=sum + divisor;
    end if;
  end for;
  if sum == value then
    perfect:=1;
  else
    perfect:=0;
  end if;
end isperfect;
```


DOI 10.3384/ECP140961285

4.5 Translation of function declaration statements

In the function declaration the translator assigns proper data types to the input and output formal parameters. The translator determines the data types of formal parameters once the whole function body has been traversed. An example of the translation is presented below.

Matlab

```
function perfect = isperfect(value)
```

Modelica

```
function isperfect
  input Real value;
  output Real perfect;
```

4.6 Identification and Translation of Local identifiers

All variables defined inside the Matlab function body are local variables. Modelica declares all local variables in the translated function under the protected section. Therefore all those local variables have to be assigned proper data types before declaration in the protected section.

For translation of this phase we have to identify all local variables first since the function body contains input and output formal parameters and Modelica does not support re-declaration of variables. The translator then assigns the proper data types. The process of identifying the local variables is depicted in Figure 3.

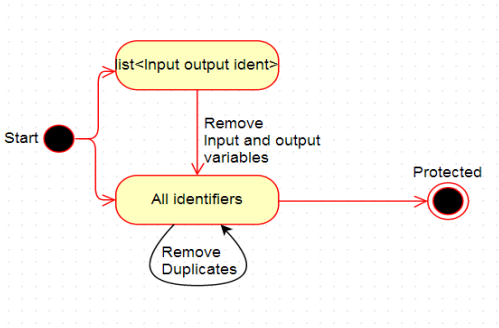

Figure 3. Declaration of variables

To perform this task we declare two lists of strings. Assume that there are two lists: a `param` and an `ident` list. The former contains input and output parameters and the latter contains all variables of the function including input and output parameters.

The translator compares both lists and removes all those variables from the `ident` list which were found in the `param` list. Now the `ident` list only contains local variables left but it might have duplicate variables. To remove the duplicates compare the list with itself and remove all duplicates.

4.7 Identification and Translation of constant variables

Matlab does not support constant declaration of variables as in Modelica with the prefix `constant` keyword. Here we mean identifiers whose right hand side is equal to any constant value, i.e., real, integer, logical or array not any variable. For example, `sum = 10` where 10 is an integer scalar. All these variables are declared under the `protected` heading with their relevant data types. Take a look at the following translation:

Matlab

```
function [sum] = add_scl_mat(scl1)
   real1 = 10.5;
   mat2 = [1,2,3;4,5,6];
   sum = mat2 + scl1 + real1;
end
```

Modelica

```
function add_scl_mat
  input Real scl1;
  output Real sum[:,:];
protected
  Real real1;
  Integer mat2[:,:] = [1,2,3;4,5,6];
algorithm
  real1:=10.5;
  sum:=mat2 .+ scl1 .+ real1;
end add_scl_mat;
```

The translator identifies two local variables. First the variable `real1` whose right hand side is a real number, i.e., 10.5. The next one is `mat2` whose right hand side is a two dimensional integer array, i.e., [1,2,3;4,5,6].

The translator identifies their respective data types from the right hand side value and then performs translation. The translated scalar identifiers only have type declarations in the protected section, the assignment of values is under the algorithm section whereas arrays have both declaration and assignment in the protected section.

4.8 Translation of Function Body

The function body of the Matlab function is translated into a Modelica algorithm section. The body is composed of a list of statements where every statement is translated to the relevant Modelica statement.

4.8.1 Identification and Translation of Function Call

The syntax for function call and array index operation is the same in Matlab using round parentheses, i.e. `sqrt(i)`. Thus `sqrt` can be a function call or an index operation. Matlab decides this during execution time. However, Modelica has different syntax for index operations. It uses brackets `[]` instead of parentheses (). Thus, `sqrt[i]` should be used if sqrt is an array that should be indexed. The function call has the same syntax as in Matlab, i.e., `sqrt(i)`.

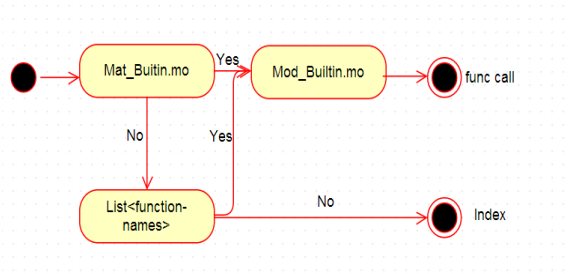

Figure 4. Process of Identifying function call

Figure 4 presents the process where a function call is translated. For correct translation the translator must differentiate between function call and index operation.

The function call in the original Matlab code can either be a built-in function in Matlab, a sub-function, or a recursive function. For proper generation of translated code the translator copies the names of main functions and sub-functions into a list of strings.

Secondly, the mostly used Matlab built-in functions are collected and placed in the file Mat_Builtin.mo**.** This file works as a dictionary. If the translator finds any function call in input Matlab code it compares the name of the function in both the dictionary and the list of collected function names.

If there is a match in any list then the translator looks for a similar function in the other file Mod_Builtin.mo where Matlab and Modelica functions were placed which have a similar purpose but have different names.

For example, the `disp('text here')` function is used in Matlab for print/display purpose is replaced to a `print("text here")` function in OpenModelica because it has a similar purpose. Similarly `rdivide()` to `div()`, `diag()` to `diagonal()` etc. If the translator does not find it, it interprets the identifier as an array that should be indexed and the translator replaces parenthesis `()` by brackets `[]`.

4.9 Identification and Translation of Anonymous function to sub-function

An anonymous function is just like a standard Matlab function. It is placed inside the body of a function and accepts inputs and return output as standard Matlab functions. However it contains only a single executable statement.

An example of a sub-function translation is presented below.

Matlab

```
function [result] = fnc(x)
  y = 2;
  sqrt= @(x) x^y;
  result = sqrt(x);
end
```

Modelica

```
function fnc //primary function
   input Real x;
   output Real result;
 protected
   Real y;
 algorithm
   y:=2;
   result:=sqrt(x, y);
 end fnc;

 function sqrt   //sub-function
   input Real x;
   input Real y;
   output Real sqrt;
 algorithm
   sqrt:=x ^ y;
 end sqrt;
```

Modelica does not support such functions. Therefore we handle them differently. The Translator converts an anonymous function to a sub-function in Modelica. The translator traverses the whole AST to find out whether the body of a function contains any anonymous function or not. If yes, then translator copies the whole anonymous expression from the original code and removes that node from the AST. Afterwards it converts it into a sub-function in Modelica. In a function call the translator replaces the input parameters with sub-function input formal parameters. As we see in the code `sqrt(x)` has only one formal parameter but in translated Modelica code we find two parameters, `x` and `y`.

4.10 Translation of looping statements

Both languages use similar syntax in for- and while-loop header expressions. The translator only translates the `=` sign to the `in` keyword and the postfix `loop` keyword in an header expression.

Also, the square brackets `[]` replaces the braces or curly brackets `{}`. Matlab ends a for loop with an `end`

DOI 10.3384/ECP140961285

`end for` keywords. Therefore translator adds the `end for` keywords at the end. The same ending procedure is followed for while loops and if statements as `end while`, `end if`. An example of the translation is presented below.

.**Matlab**

```
for m = 1:14
end
```

Modelica

```
for m in 1:14 loop
end for;
```

Matlab

```
while nFactorial < 100
end
```

Modelica

```
while nFactorial < 100 loop
end while;
```

4.11 Translation of if statements

The `if` statement is also similar in both languages. The translator adds the `then` keyword in the headers of if expressions and the `endif` keyword to end the expressions. An example of the translation is presented below.

Matlab

```
if(result == floor(result))
   sum = sum + divisor;
end
```

Modelica

```
if result == floor(result) then
   sum:=sum + divisor;
end if;
```

4.12 Translation of Switch Statement

Modelica does not currently support the `switch` statement. When the translator finds a switch statement in the original Matlab code it translates it into if-else statements with possible `elseif` branches. An example of the translation is presented below.

Matlab

```
switch mynumber
    case -1
        disp('negative one');
    case 0
        disp('zero');
    case 1
        disp('positive one');
  otherwise
   disp('other value');
end
```

Modelica

```
if mynumber == -1 then
  print("negative one");
elseif mynumber == 0 then
  print("zero");
elseif mynumber == 1 then
  print("positive one");
end if;
```

4.13 Sub-function

A Matlab file can contain more than one function. If it contains multiple functions, then the first function is the primary function which can be accessed in other M files and the rest of the functions are secondary functions which are called as sub-functions.

A sub-function is only accessed by its primary function. All sub-functions are translated into sub-functions in Modelica and this process follows the same procedure for translation as for the main/primary function.

5 Type Inference

We have adopted the MCFOR "A Matlab To Fortran95 Compiler" approach for evaluating the types in our translator. It starts by analyzing each assignment statement on the right hand side (RHS) of the expression and assign types [16].

5.1 Statement where Right Hand Side is a constant

Constant values are very precise sources for assessing data types. They are the starting points for our type inference process. If the translator finds an assignment statement whose right hand side is a constant (e.g.) `temp = 10`, in Modelica such an expressions will be declared as an integer data type.

Example

```
temp = 10  => Integer temp
temp = 10.5 => Real temp
temp = [1,2,3,4] => Integer temp[1,4]
"size is 1x4"
temp = [1,2;3,4] => Integer temp[2,2]
"size is 2x2"
```

5.2 Statement where Right Hand Side is Built-in function

Built-in functions in Matlab are well defined therefore these functions also provide enough information for assessing data types. If an assignment statement contains built-in function on right hand side translator will easily assess its type from database where we have a list of all built-in functions supported by Modelica with their return type. (e.g.) `zeros(n,m)` is an n-by-m ma-

trix of zeros. If n and m is equal to 2 then translator will declare left hand side identifier as a two dimensional Integer array.

Example

```
temp = zeros(2,2) => Integer temp[2,2]
"size is 2x2"
temp = ones(1,3) => Integer temp[1,3]
"size is 1x3"
```

5.3 Statement where Right Hand Side is a computational expression and contains relational operator

Relational operators always return the results of the logical type. (e.g.) `c = a < b`.

5.4 Statement where Right Hand Side is a computational expression and contain arithmetic operator (array)

Matlab operators not only provide the shape and size information for the result but also provide constraints on the type and shape of operand. The assignment statement where right hand side is matrix computational expression, `c = a*b` where `a & b` are matrices.

In order to evaluate such an expression in Matlab the inner dimensions must agree. If `a & b` have a shape of n-by-m and p-by-q, then m must be equal to `p`. The generated result will have the shape of `n-by-q`. For example if we have an expression `temp = a * b` where dimension size of `a` is `1x3` and `b` is `3x3` then `temp` have dimension `1x3` and translator declares as an `Integer temp[1,3]`.

5.5 Statement where Right Hand Side is a computational expression and contain arithmetic operator (scalar)

Consider the following expression `temp = a / b` where `a & b` are Integer scalar variables. To avoid data loss the translator promotes the left hand side variable as real data type.

5.6 RHS is not equal to LHS

In Matlab we can re-create any variable with different type assignment and every assignment can create a type conflict where left hand side variable is different from the type of the right hand side expression. In such cases the translator performs typecasting on right hand side expression and defines it with proper data type.

6 Unparser

When translating the code to an abstract syntax tree in Modelica we need to unparse the AST to get a readable output, i.e., Modelica source code. The unparsing is performed by the package Dump which is already present in the OpenModelica Compiler. The interface of the function which starts the unparsing is given below.

```
public function unparseStr
  "Pretty prints the Program, i.e. the
    whole AST, to a string."
  input Absyn.Program inProgram;
  input Boolean markup
  output String outString;
end unparseStr;
```

7 Testing

We tested a number of Matlab programs to demonstrate that the translator was working properly.

A sample Matlab function to calculate the area is presented below

```
function area_sum = area_inside(radius,
num_boxes)
  box_length = (2.0*radius)/num_boxes;
  box_rad = box_length*0.5;
  box_area = box_length*box_length;

  area_sum = 0;
  for (xi = 1 : num_boxes)
    xc = - 1 + box_rad + box_length*(xi -
        1);
   for (yi = 1 : num_boxes)
      yc = 1 + box_rad + box_length*(yi -
          1);

      dist = sqrt((xc*xc) + (yc*yc));
      if (dist < radius)
        area_sum = area_sum + box_area;
      end
    end
  end
end
```

The above input is translated to the following Modelica source code which gives the same result..

```
function area_inside
  input Real radius;
  input Real num_boxes;
  output Real area_sum;
protected
  Real box_length;
  Real box_rad;
  Real box_area;
  Real xc;
  Real yc;
  Real dist;
algorithm
  box_length:=2.0 * radius / num_boxes;
  box_rad:=box_length * 0.5;
  box_area:=box_length * box_length;
  area_sum:=0;
```


DOI 10.3384/ECP140961285

```
  for xi in 1:num_boxes loop
      xc:=-1 + box_rad + box_length *
          (xi - 1);
    for yi in 1:num_boxes loop
         yc:=1 + box_rad + box_length *
           (yi - 1);
      dist:=sqrt((xc * xc) + (yc * yc));
      if dist < radius then
        area_sum:=area_sum + box_area;
      end if;
    end for;
  end for;
end area_inside;
```

7.1 Performance Evaluation

In this section we present the performance of the Modelica code generated by the translator which is compared to the execution time with the corresponding Matlab code in Matlab. Before the comparison we made sure that both codes generate same results. The measurements are listed in Table 1 below.

Table 1. Time measurement of Matlab with Modelica

Model	Scalar/Array	Matlab(ms)	Modelica(ms)
Closure	2D-array	185.90	26.49
IsPerfect	scalar	198.60	11.65
Finite	2D-array	69.98	55.25
AreaInside	scalar	180.05	16.72
MySort	1D-array	218.45	26.49
SwapVector	1D-array	80.67	18.02
BubbleSort	1D-array	109.34	81.96

From Table 1 we can clearly see that the generated Modelica code has better performance than the Matlab code.

8 Conclusion

In this paper we have presented a Matlab to Modelica translator that works for a specific subset of Matlab. The translator generates readable and reusable Modelica code. It can handle the following Matlab constructs:

- *Functions*: Simple, Nested and Sub-functions.
- *Loops*: For, Nested For, While, Nested While, and break.
- *Statements*: If, If else, If elseif else and Switch.
- *Program Termination:* return.
- *Arithmetic Operators:* Plus +, Unary plus +, Minus -, Unary minus -, Matrix multiply *, Array multiply .*, Matrix power ^, Array power .^, Backslash or left matrix divide \, Slash or right matrix divide /, Left array divide .\, and Right array divide ./.
- *Relational Operators:* Equal ==, Not equal ~=, Less than <, Greater than >, Less than or equal <= and greater than or equal >=.
- *Logical Operators:* Element-wise logical AND &, Element-wise logical OR | and Logical NOT ~.

The generated code can then be statically compiled by a Modelica compiler which usually result in better runtime performance.

9 Acknowledgments

This work is done within the OpenModelica project. The OpenModelica work is supported by the Open Source Modelica Consortium.

References

[1] Edgar Alonso Lopez-Rojas. OMCCp: A MetaModelica based parser generator applied to Modelica. Master's-Thesis, Linköping University, Department of Computer and Information Science, PELAB- Programming Environment Laboratory, ISRN:LIU-IDA/LITH-EX-A--11/019--SE, May 2011.

[2] Arunkumar Palanisamy. Extended MetaModelica based Integrated Compiler generator. Master's-Thesis, Linköping University, Department of Computer and Information Science, PELAB-Programming Environment Laboratory, ISRN:LIU-IDA/LITH-EX-A--12/058--SE, October 2012.

[3] Peter Fritzson. Principles of Object-oriented modeling and simulation with *Modelica 2.1*. Wiley-IEEE Press, 2004.

[4] Peter Fritzson, Adrian Pop and Martin Sjölund. Towards Modelica4 Meta-Programming and Language Modeling with MetaModelica 2.0, Technical reports Computer and Information Science Linköping University Electronic Press, ISSN:1654-7233; 2011:10

[5] Peter Fritzson and Adrian Pop. Meta-Programming and Language Modeling with MetaModelica 1.0. Technical reports Computer and Information Science Linköping University Electronic Press, ISSN: 1654-7233. 2011:9.

[6] Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman. *Compilers Principles, Techniques, and Tools*, Second *Edition*. Addison-Wesley, 2006.

[7] Peter Fritzson et al. Compiler Construction laboratory assignments. Compendium, Bokakademin,

Linköping University, Department of Computer and Information Science, 2011.

[8] Michael Burke and G.A. Fisher Jr. A practical method for syntactic diagnosis and recovery. In *Proceedings of the 1982 SIGPLAN symposium on Compiler Construction*, 1982.

[9] Open Source Modelica Consortium. *OpenModelica System Documentation Version 1.6*, November 2010. http://www.openmodelica.org

[10] TheMathWorks,Inc.MATLAB http://www.mathworks.se/

[11] Luiz De Rose and David Padua. Techniques for the Translation of MATLAB Programs into Fortran 90, *ACM Transactions on Programming Languages and Systems*, Vol 21, March 1999.

[12] Luiz De Rose and David Padua. *A MATLAB to Fortran 90 Translator and its Effectiveness*. ISBN:0-89791-803-7, 1996.

[13] Pramod G. Joisha, Abhay Kanhere, Prithviraj Banerjee, U. Nagaraj Shenoy, Alok Choudhary. The Design and Implementation of a Parser and Scanner for the MATLAB Language in the MATCH Compiler, 1999.

[14] Pramod G. Joisha, U. Nagaraj Shenoy, Prithviraj Banerjee. *An Approach to Array Shape Determination in MATLAB*. Technical report no. CPDC–TR–2000–10–010, Oct 2010.

[15] Anton Dubrau and Laurie Hendren. *Taming Matlab*, Sable Technical Report No. sable-2011-04. McGill University, School of Computer Science, April, 2011.

[16] Jun Li. MCFOR: *A MATLAB TO FORTRAN 95 COMPILER*. McGill University, School of Computer Science, August, 2009.

10 Appendix

The Appendix presented below represents the AST data structures used for the construction of MATLAB source code.

Absynmat.mo

```
encapsulated package AbsynMat
public type Ident = String;
uniontype Start
  record START
    User_Function usr_fun;
    Separator sep;
    list<Statement> stmt_lst;
  end START;
end Start;

uniontype User_Function
  // Begin defining a function.
  record START_FUNCTION
    Ident fname;
    list<Parameter> prm;
    Option<Separator> sep;
    list<Statement> stmt_lst;
    Statement stmt_2nd;
  end START_FUNCTION;

// Finish defining a function.
  record FINISH_FUNCTION
    list<Decl_Elt> ret;
    User_Function usr;
  end FINISH_FUNCTION;
end User_Function;

uniontype Argument
  record ARGUMENT
    Expression exp;
  end ARGUMENT;

  record VALIDATE_MATRIX_ROW
    Argument arg_lst;
  end VALIDATE_MATRIX_ROW;
end Argument;

uniontype Command
  record TRY_CATCH_COMMAND
    Separator sep;
    list<Statement> stmt_lst1;
    list<Statement> stmt_lst2;
    list<Mat_Comment> m_cmd_lst;
    list<Mat_Comment> m_cmd_lst2;
  end TRY_CATCH_COMMAND;

  record UNWIND_PROTECCOMMAND
    list<Statement> stmt_lst1;
    list<Statement> stmt_lst2;
    list<Mat_Comment> m_cmd_lst;
  end UNWIND_PROTECCOMMAND;

  record DECL_COMMAND
    Ident identifer;
    list<Decl_Elt> decl_elt;
  end DECL_COMMAND;

  record BREAK_COMMAND
  end BREAK_COMMAND;

  record CONTINUE_COMMAND
  end CONTINUE_COMMAND;

  record RETURN_COMMAND
  end RETURN_COMMAND;

  record SWITCH_COMMAND
    Expression exp;
    Separator sep;
    tuple<list<Switch_Case>,
Option<Switch_Case>> swcse_lst;
    Option<Mat_Comment> m_cmd_lst;
  end SWITCH_COMMAND;

  record WHILE_COMMAND
    Expression exp;
    Option<Separator> sep;
    list<Statement>  stmt_lst;
    Option<Mat_Comment> m_cmd_lst;
  end WHILE_COMMAND;
end AbsynMat;
```

 DOI 10.3384/ECP140961285