Modelica Based Parser Generator with Good Error Handling

Arunkumar Palanisamy[1], Adrian Pop[1], Martin Sjölund[1], Peter Fritzson[1]

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

Abstract

This paper describes the new OpenModelica Compiler-Compiler (OMCC) including a parser generator, OMCCp which is based on an LALR parser generator extended with advanced error handling facilities. It is implemented in the MetaModelica language with parsing tables generated by the tools Flex and Bison. It is integrated with the MetaModelica semantics specification language, based on operational semantics for generating executable compiler and interpreter modules.

The OMCCp parser generating part of OMCC is being used for the full Modelica language grammar as well as for the language extensions of MetaModelica, ParModelica, and Optimization specifications. The generated parsers have reasonable performance compared to other parser generators.

Keywords: Modelica, MetaModelica, Flex, Bison, ParModelica, Optimization, OMCCp

1 Introduction

The OpenModelica environment currently makes use of the tool ANTLR (Another tool for Language Recognition) [11] to generate the parser for the OpenModelica Compiler (OMC). In this paper we present an alternative to ANTLR, the new OpenModelica Compiler-Compiler parser generator (OMCCp), developed within the OpenModelica project. The tool is implemented in MetaModelica which is an extension of the Modelica language for modeling the semantics of languages. The work [9] [11] is integrated with the recently developed bootstrapped OpenModelica compiler (OMC) [14].

The ANTLR parser generator which has been used in the OpenModelica project for several years has well known disadvantages including memory overhead, bad error handling, and lack of type checking, Also it does not generate directly MetaModelica code for building the Abstract Syntax Tree (AST).

Since the AST nodes are initially generated by C functions (for later conversion into MetaModelica garbage collected memory space) without strong type checking in the C language, small errors in the semantic actions of the grammar are not detected at generation time and can give rise to hard-to-find errors in the generated code (even small errors in the grammar actions C code lead to segmentation faults).

When the semantic actions can be specified in MetaModelica and the AST builder directly generates the MetaModelica code, the above mentioned errors can be completely eliminated.

The need for good error handling as well as avoidance of certain AST-building errors has motivated the development of the Modelica based parser generator with good error handling [9][11].

This paper is structured as follows: Section 2 describes the different steps involved in designing a compiler from different specification formalism. Section 3 presents the error-handler features added to the tool and also illustrates the different types of error handler messages displayed to the user in the case of erroneous programs. Section 4 explains the main MetaModelica language constructs used in this implementation for generating the Abstract Syntax Tree (AST). Section 5 explains the OMCCp tool design and architecture and Section 6 presents test results and performance measurements. Finally Section 7 concludes the paper summarizing achieved results.

2 Background

2.1 Generating Compiler Phases

A compiler can be generated from a formal specification in different formalisms, as depicted in Figure 1.

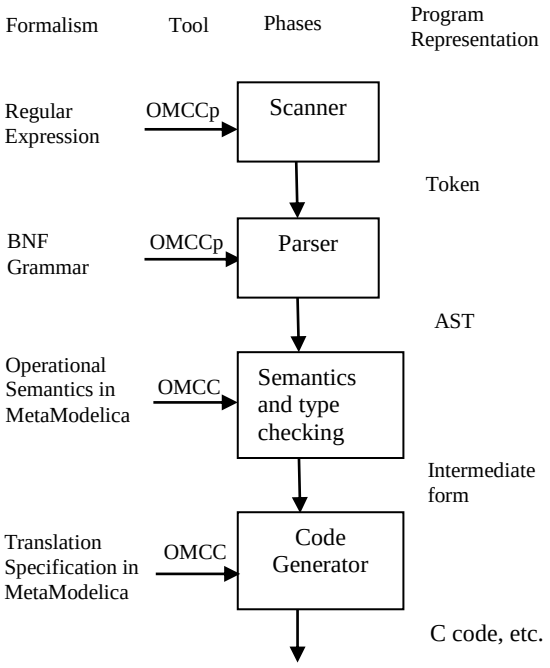

Figure 1. Design stages of a compiler from different specification formalism

Figure 1 describes the stages involved in generating a compiler from specifications in different formalisms. 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 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.

2.2 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 identifies the special tokens defined by a language making it simpler for the next phase of the compiler. The tokens are usually specified by using Regular Expressions.

There are several tools available that automate the process of constructing the transition rules to identify the tokens for a scanner. We use the Flex tool [13] for this purpose which generates C code; the generated C code is later used by OMCCp to generate the appropriate lexer components in MetaModelica [1] [9] [11].

2.3 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. During this process the Abstract Syntax Tree (AST) is created if the input conforms to the defined grammar. Otherwise an error message is reported.

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-Naur Form). We use the popular Bison tool for writing the grammar rules; the generated tables are used by OMCCp as part of the parser algorithm written in MetaModelica that interprets these parse tables [1][9][11].

3 Error Handling

The error handling techniques in the front-end are more relevant during the Syntax analysis phase than in the lexical analysis phase. Only a few errors can be detected by the lexical analysis, such as non-terminated comments, use of invalid characters, or unrecognized tokens. One possible error-recovery strategy implemented in a lexer is to ignore invalid characters from the input and continue the process.

The error handling techniques can be divided into two categories: Error recovery techniques and error message display. Error recovery techniques are concerned with how the parser can keep parsing after an erroneous token is found. Error message display concentrates on how to present useful hints for the developer in order to correct the source code. In this section we will present the two topics for error handling during the syntax analysis phase [1] [9] [11] [2] [4].

3.1 Error Recovery

Error recovery techniques try to improve the quality of the parser by different techniques such as primary recovery or secondary recovery. The first condition to start the recovery is to access the configuration obtained when the token preceding the error token was shifted onto the stack

Primary recovery techniques are related to single token modification from the list of tokens. Single modification is only possible when the error is classified as simple. Such a modification can be insertion, deletion, substitution, or merging.

DOI 10.3384/ECP14096567

Each attempt to perform a repair is known as a trial. A common technique for searching the trials is to attempt to repair the error token by performing one of these operations: *merging*, *insertion*, *substitution*, *scope recovery* and finally *deletion*. In the case of insertion or substitution a set of possible candidates should be generated and then from there a single candidate or none should be selected.

When the error requires more than a simple modification, the list of tokens needs to be reduced. This can be done by discarding tokens that precedes or follows the error token. This is known as secondary recovery [1] [9] [11] [2].

3.2 Error Messages

OMCCp uses a primary recovery technique to display all the possible recovery candidates to the developer. When an error is found, the parser fires the error handler function including the environmental variables that contain the actual configuration of the parser and the backed up configuration when the last token was shifted. This backup is used to start an extensive search for the possible valid tokens to be replaced. This allows the developer to better understand the messages and makes it easier to select the correct token.

OMCCp uses seven different kinds of error messages for the syntax analysis and one more for the lexical analysis. The error messages displayed in this implementation are discussed below.

3.2.1 Erase Token

The *erase token* message occurs when the parser finds same token repeated more than once. To test if a token can be erased, the parser is run on the remaining list of tokens ignoring the current token. If the test succeeds a proper error message is displayed to the user suggesting a possible solution for erasing the repeated token [9] [11]. An example error message display is given below.

```
[../../testsuite/omcc_test/error5.mo:10:20
-10:24:writable]
Error: Syntax error near: 'if x == 10 then
then', REPLACE token with '+' or '-' or
'.' or 'NOT', ERASE token ‘then’
```

3.2.2 Insert Token

To test if a token can be inserted before the error token, the parser is run on a modified list of the remaining tokens by placing the candidate token before the error token as current token. The candidate token is selected if the test succeeds and placed in the candidate list [9] [11]. If there are items in the candidate list we display proper message to the user. An example error message is given below.

```
[../../testsuite/omcc_test/error3.mo:9:3-
9:4:writable]
Error: Syntax error near: 'if x == 10',
INSERT token 'THEN'
```

3.2.3 Replace Token

The replace token is similar to insert token error message. The parser is run modifying the remaining list of tokens by placing the candidate token before the error token as current token. The candidate token is selected if the test succeeds and placed in the candidate list and a proper error message is displayed to the user [9] [11].

```
[../../testsuite/omcc_test/error2.mo:7:1-
7:6:writable]
Error: Syntax error near: 'while x <> 99
then','THEN',REPLACE token with 'Loop'
```

3.2.4 Insert Token at End

This message is used only at the end of the program, when no other token is available in the input of tokens and a non-finished acceptance state has been achieved. All the tokens are tested to verify if they can make the program to end in a valid acceptance state. If a token succeeds then the proper message is displayed to the user as a possible solution to fix the error [9] [11]. An example message is given below.

```
[../../testsuite/omcc_test/error6.mo:14:1-
14:15:writable]
Error: Syntax error near:'end error_test',
INSERT at the End token 'SEMICOLON'
```

3.2.5 Merge Token

Sometimes a space can be inserted by mistake between two tokens and make a keyword appear as two separate identifier tokens. In this case the error token and the token that follows it are processed again by the Lexer with their value concatenated.

If the lexer combines them as a valid token this token is tested to see if it satisfies the test and is a valid configuration for the parser [9] [11]. If it succeeds the system displays a proper message to the user. An example error message is given below.

```
[../../testsuite/omcc_test/error4.mo:10:9-
10:10:writable]
Error: Syntax error near:'if x = = 10
then', MERGE tokens '=' and '='
```

3.2.6 Generic Error

It is possible that no solution or candidate is found for the current error token. In these cases a generic error message is displayed to the user without any further description of the error than the location of the token.

In this situation the developer needs to fix the error token without any hint [9] [11].

3.2.7 Custom Error Message

While designing a grammar it is sometimes necessary to communicate to the developer that a certain transformation rule must not be used. A more clear language than presented by the error messages above should be used. For this reason the possibility of a custom error message has been introduced. Such an error message is added in the grammar rules. The use of this custom message will activate the error flag in the parser and will start the simple error recovery technique.

4 MetaModelica

MetaModelica is an extension of the Modelica language which can be used to model the semantics of languages, specify symbolic transformations, etc. It is based on the operational semantics language specification formalism; a rule in operational semantics becomes a case within match-expressions [6] [7].

4.1 Uniontype

A `uniontype` in MetaModelica is a collection of one or more record types. It can be recursive and can include other uniontypes. It is mainly used for the construction of Abstract Syntax Trees (AST) [6] [7].

Example:

```
uniontype Exp
  record INT
    Integer integer;
  end INT;
  record BINARY
    Exp exp1;
    BinOp binOp;
    Exp exp2;
  end BINARY;
end Exp;
```

4.2 Match Expressions

The `match` expression is similar to switch statements in C but even closer to match in functional programming languages with some extra features.

The `match` expression can return more than one value and supports pattern matching. The match construct contains case blocks. Each `case` can contain an equation block. The program flow tries to execute one instruction after the other in a specific local equation block. If an instruction is not executed or failed the next case is tried until a match is found. The wildcard '_' (underscore) can be used to match all the cases [6] [7].

For example

```
function eval
  input Exp inExp;
  output Integer outInteger;
algorithm
  outInteger := match(inExp)
    local
      Integer ival,v1,v2,v3;
      Exp e1,e2,e;
  case INT(ival)  then ival;
  case BINARY(e1,binop,e2)
   equation
    v1 = eval(e1);
    v2 = eval(e2);
    v3 = applyBinop(binop, v1, v2);
   then
    v3;
  case UNARY(unop,e)
   then v2=applyUnop(unop, eval(e));
end match;
end eval;
```

In the above example we can see that the function `eval` contains one input formal parameter of type `Exp` and one output formal parameter of type `Integer` followed by the match-expression which tries to match any of the three cases according to the user input.

The first case results in the evaluation of `INT` record constructor node applied to an integer. The second case results in evaluation of a binary operator node binary to `v3`, if `v3` is the result of successfully applying the binary operator to `v1` and `v2`, which is the evaluated result of its children `e1` and `e2`. The third case results in the evaluation of unary operator node `UNARY` to `v2`.

4.3 List

The `List` constructor is used to create linked list structures. Lists are used in modeling of flexible variable-length collections of symbolic elements. The operand `::` is used to add an element at the front of a list (or, depending on the context, retrieve by matching) an element from the list [6] [7].

Example

```
list<Integer> a= {1,2,3};
i::a=a;
a=i::a;
```

The first line creates a list `a` of integers, the second line is used for the retrieve operation; it takes the top element 1 from the list and stores it in the variable `i` and stores the remaining list in variable `a`. The third line performs the add operation which adds an item `i` into the list `a`.

DOI 10.3384/ECP14096567

5 OMCCp DESIGN

The design architecture of OMCCp with the lexer and parser components is depicted in Figure 2.

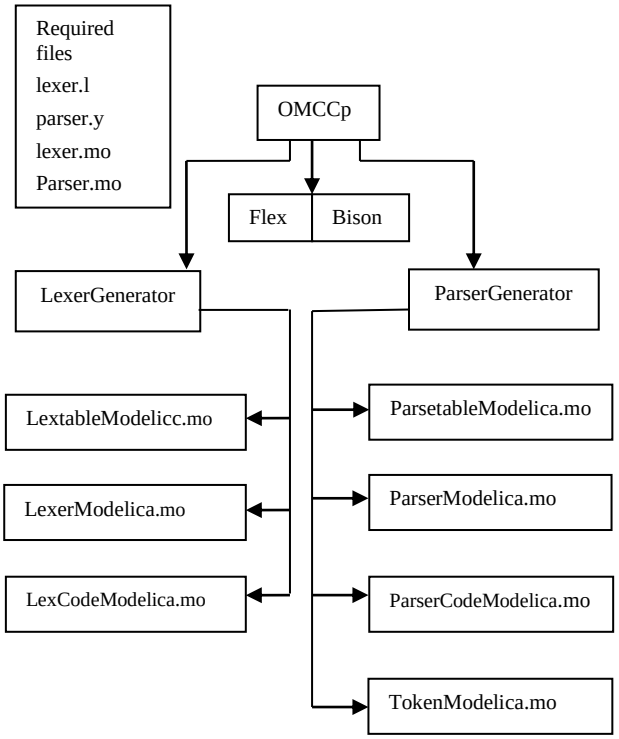

Figure 2. OMCCp (OpenModelica Compiler-Compiler) Lexer and Parser Generator

The lexer and parser are generated based on C files generated by the tools Flex and Bison from the grammar files lexer.l and parser.y.

The generated C file contains three main parts which constitutes the lexer and parser. We can identify: a section with the transition arrays, a section with code that runs the algorithm for the lexer or parser and finally an action resolution section which contains the returns tokens actions for the lexer and reduce operation which constructs the AST.

Based on the identification of the main parts, the OMCCp design is divided into two parts namely the LexerGenerator and the ParserGenerator which are responsible for generating the necessary Lexer and Parser components in MetaModelica.

5.1 LexerGenerator

The `LexerGenerator` is the main package of the lexer generator which generates the three necessary lexer packages in MetaModelica, namely LexerModelica.mo, LextableModelica.mo and LexcodeModelica.mo.

5.1.1 Lexer.mo

Lexer.mo is the main file which contains the calls to other functions in Lextable.mo and LexCode.mo that constitutes the lexer. The main function of this package is to load the source code file and recognize all the tokens described by the grammar.

To recognize the 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 performs 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
```

5.1.2 LexTable.mo

LexTable.mo is the source file for Lexer.mo which contains the transitions arrays for performing transitions to new states and finding the tokens from the input stream.

5.1.3 LexCode.mo

LexCode.mo contains all the specific actions that a lexer performs when a token is recognized. There are three types of action a lexer can perform: *ignore* token, *return* specific token, and *switch* to another DFA.

The first action, ignore token, is performed by the lexer when a space, line feed, or block of comment has been found in the input stream. The tokens ignored by the lexer simplify the task of the parser as these tokens will not be used in the construction of the grammar.

The second action returns a specific token when a token is recognized by the lexer, the code of the action determines the tokens to be returned.

The third action is to switch from one DFA to another. This operation is performed in a few situations for example when the DFA finds a starting comment block '/*', and all the subsequent tokens are required to be ignored or categorized as a different token, e.g. in case of determining a string. For this action a new startup state is set in the machine and new characters are processed by a different DFA than the original.

After the end of tokens is reached, i.e., '*/' then the start state returns to the original one.

5.2 Parser Generator

The parser generator is the main package of the parser that performs the syntax analysis of the compiler. This package generates four packages comprising the parser in MetaModelica namely TokenModelica.mo, ParserTable.mo, ParserCodeModelica.mo and ParserModelica.mo.

5.2.1 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 the 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;
```

5.2.2 ParseTable.mo

ParseTable.mo contains the arrays that allow the Parser package to run the Push down Automata (PDA) and perform the shift-reduce action which constructs the AST.

5.2.3 ParseCode.mo

The package ParseCode.mo contains the specific Reduce operations that each grammar performs when a certain rule matches the input tokens. The main function of this file is to handle the MultiTypedStack that is used by the parser to construct the AST. The MultiTypedStack handles the reduce operations requested by the LALR parsing algorithm. The MultiTyped stack contains one stack for each type found in the grammar specification and is defined in the MetaModelica language. An example interface is presented.

```
uniontype AstStack
    record ASTSTACK
      list<Absyn.Exp>  stackExp;
      list<String>    stackString;
      list<Integer>  stackInteger;
    end ASTSTACK;
end AstStack;
```

When the parser finds a Shift operation it calls the `function push' on this file which will push a String value into the stackString. During the reduce operation the parser needs to know which stack to use for each of the constructions of the AST. The way this MultiTypedStack works can be explained in the following example that presents the reduce operation, the build of the AST operation and finally a push back into the stack which builds the AST.

```
case (82,_) // #line 413
"parserModelica.y"
      equation
        // reduce
        (info, skToken) =
getInfo(skToken,mm_r2[act]);
        v2Comment::skComment = skComment;
        v1Algorithm::skAlgorithm =
skAlgorithm;
        // build
  vAlgorithmItem
=Absyn.ALGORITHMITEM((v1Algorithm),SOME((v
2Comment)),info);
        // push Result
  skAlgorithmItem=
vAlgorithmItem::skAlgorithmItem;
   then ();
```

In the above case the reduction takes three items from the three different stacks and constructs an `Absyn.ALGORITHMITEM` object. After this it pushes the result back into the stack for the `Absyn.Algorithm` type called `skAlgorithmItem`. Another feature that can be useful for the reductions is the use of the `info` keyword. In the example, we can see that the instruction uses a stack called `skToken` to retrieve the token information.

The token information returns an info token of type `Absyn.Info` which contains the combined information of the first and the last token in the stack that are used for this reduction. This makes it possible for the developer to insert information about the location that can be used later in the other phases of the compiler. The package also contains another important function `getAst` which returns the result of AST tree to the parser. The interface of the function is given below.

```
function getAST "returns the AST built by
    the parser"
 input AstStackastStk aststack
    "MultiTypedStack used by the parser";
 output AstTree ast "returns the AST in
       the final type of the tree";
end getAST;
```


DOI 10.3384/ECP14096567

5.2.4 Token.mo

The package contains the complete list of tokens used by the grammar with their respective codes. This file is the link between lexer and parser; it is used by both the lexer and parser to identify the tokens in the same way.

When the parser receives the token code from the lexer it performs a translation into local codes that are only used by the parser to simplify the addressing in predictive arrays. Each token is defined as an Integer-type constant. The interface of this package is presented below.

```
package TokenModelica
 constant Integer T_ALGORITHM = 258;
 constant Integer T_AND = 259;
 constant Integer BLOCK = 261;
 constant Integer CLASS = 262;
 constant Integer CONNECT = 263;
end TokenModelica
```

6 Testing

We performed testing using Modelica files from the OpenModelica testsuite. We also ensured that the tests covered full Modelica and MetaModelica grammar proving the correctness of the parser.

In the later stages of testing we also covered the ParModelica and Optimization grammars. During the testing we did not face any memory overhead problems and also measured the performance of the parser including AST building. The performance times are reasonable when compared to ANTLR. An example of sample test case of correct and an erroneous model with the respective output message are presented.

A Sample of correct input

```
model Circle
  Real x_out;
  Real y_out;
  Real x(start=0.1);
  Real y(start=0.1);
equation
  der(x) = -y;
  der(y) = x;
  x_out = x;
  y_out = y;
end Circle;
```

This input will generate the following success message

```
Parsing Modelica with file
../omcc_test/Test1.mo
// SUCCEED
// args:../omcc_test/Test1.mo
```

This sample of erroneous input will on the other hand generate a parse error.

```
class error_test
int x,y,z,w;
algorithm
while x <> 99
  x := (x+111) - (y/3);
   if x == 10 then
    y := 234;
  end if;
end while;
end error_test;
```

This input will cause the following error message to be emitted by an OMCCp generated parser

```
Parsing Modelica with file
../omcc_test/error3.mo
[../../testsuite/omcc_test/error3.mo:9:3-
9:4:writable] Error: Syntax error near:
'while x <> 99', INSERT token 'LOOP'
args:../../testsuite/omcc_test/error3.mo
```

The same input will cause the following error message to be emitted by an ANTLR generated parser

```
Loaded all files without error
"true
"
""
{fail()}
```

Take a look at a second erroneous sample input, an erroneous piece of MetaModelica code.

```
function add
  input Integer ininteger;
  input Integer ininteger1;
  output Integer outinteger;
  algorithm
    outinteger:=
      match(ininteger,ininteger1)
       local
        Integer a,b,c;
       case(a,b)
        equation
          c=a+b;
          then
            c;
   end matchcontinue;
 end add;
```

This will cause the following error message to be emitted using an OMCCp generated parser

```
Parsing Modelica with file
../omcc_test/error2.mo
[../../testsuite/omcc_test/error2.mo:14:13
-14:30:writable] Error: Syntax error near:
'end matchcontinue', REPLACE token with
'ENDMATCH'
args:../../testsuite/omcc_test/error3.mo
```

It will cause the following error message to be emitted using an ANTLR generated parser.

```
Loaded all files without error
 "true
 "
 ""
 {fail()}
```

Regard a third erroneous sample input

```
class error_test
 int x,y,z,w;
 algorithm
 while x <> 99 loop
   x := (x+111) - (y/3);
    if x = = 10 then
     y := 234;
   end if;
 end while;
 end error_test;
```

This will cause the following error message from an OMCCp generated parser.

```
Parsing Modelica with file
../omcc_test/error6.mo
[../../testsuite/omcc_test/error4.mo:6:9-
6:10:writable]
Error: Syntax error near:'if x = = 10
then', MERGE tokens '=' and '='
```

and this error message using an ANTLR generated parser.

```
Loaded all files without error
"true
"
""
{fail()}
```

6.1 Time Performance

The performance measurements for the test cases have been done using the OpenModelica test server. The models are taken from the Modelica Standard Library (MSL 3.2.1) as well as from the OpenModelica test suite. A selection of measurements is listed in the following Table 1.

Table 1. Time measurement of OMCCp and ANTLR generated parsers on a set of test models.

Model	Size	No of Tokens	OMCCp (time ms)	ANTLR (time ms)
Dcmotor	2kB	172	3.1	0.45
Influenza	4kB	642	10.4	1.10
Test1	26kB	8443	84.3	6.90
Icon	42kB	9234	179.7	8.68
SIunit	94kB	14330	303.8	18.57
Electrical/ Digital	366kB	71117	1576	93
Electrical/ Machine	763kB	143505	3202	158
Test2	1MB	214617	4681.8	254
Total	11MB	2078841	51362	2704

We can see that the ANTLR generated parser is about 5 to 16 times faster than OMCCp including AST building. The OpenModelica implementation of an ANTLR generated parser is extremely fast. The OMCCp parser performance is comparable to typical non-optimized parser generators.

The OMCCp generated parser is roughly 16 times slower than ANTLR for the very large Modelica models like `Total.mo` which contains the whole Modelica standard library 3.2.1 of 11 MB, but in general for smaller test cases the parser is only 6 times slower.

From the table we can also see that the OMCCp parser does not scale as well in the case of large models compared to ANTLR. Even though the ANLTR parser is faster, the OMCCp tool has several advantages compared to ANTLR. One should also note that the current implementation of OMCCp was not optimized and its performance can be further improved.

During testing we found that the OMCCp tool did not have any memory overhead problems when parsing a large number of test cases in a single attempt whereas the ANTLR parser had higher memory overhead when many test cases were parsed in a single attempt. Moreover, the OMCCp tool has better error handling features compared to the ANTLR tool, which makes it easier to use.

6.1.1 Performance Graph

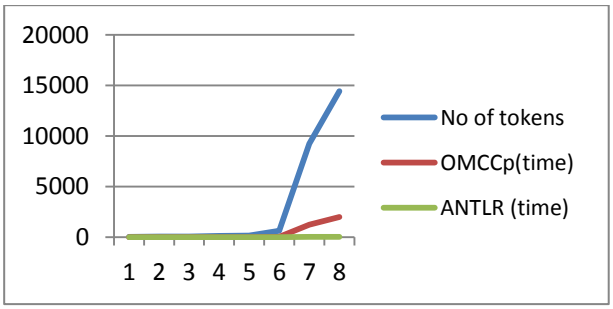

Figure 3: Graph representation of time performance of OMCCp with ANTLR

From the above graph we can see that the OMCCp parser keeps its timing performance close to that of ANTLR for a certain number of tokens recognized by the lexer, but when the number of tokens increases the performance gap increases between the tools.

Still the OMCCp tool has good performance when parsing grammars of size less than 100kB, with times of under a second. One of the reasons that the OMCCp is slower is that garbage collection is used in Meta-Modelica whereas in ANTLR the memory allocation/de-allocation is manually programmed.

DOI 10.3384/ECP14096567

7 Conclusion

In this paper we have presented a fully implemented OMCCp lexer and parser generator integrated with MetaModelica as a semantic specification language in the new bootstrapped OpenModelica compiler.

We have tested this tool on number of small languages as well as on the large Modelica grammar. The generated parsers offer good error handling and comprehensive error messages to the user. OMCCp can also be used as a parser generator for any language for which an LALR(1) grammar is available. The associated language semantics can be specified using MetaModelica.

The generated parsers are still a bit slow for production usage on large programs but we expect to improve the performance by further tuning.

8 Acknowledgments

This work has been partially supported by the Swedish Governmental Agency for Innovation Systems (Vinnova) within the ITEA2 MODRIO project, and by the Swedish Research Council (VR).

References

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

[2] Rober Bilos. *Syntactic Error Diagnosis and Recovery*. Master Thesis, Linköping University, Department of Computer and Information Science. 1983.

[3] Michael Burke and G.A. Fisher Jr. A Practical Method for Syntactic Diagnosis and Recovery. In *Proceedings of the 1982 SIGPLAN symposium on Compiler constructions*, 1982.

[4] Michael G. Burke and Gerald A. Fisher. A practical method for LR and LL Syntactic Error Diagnosis and Recovery. *ACM Transactions on Programming Languages and Systems*, March 1987.

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

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

[7] 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.

[8] Peter Fritzson et al. Compiler Construction laboratory assignments. Compendium, Bokakademin, Linköping University, Department of Computer and Information Science, 2011.

[9] Edgar Alonso Lopez-Rojas. *OMCCp: A MetaModelica Based Parser Generator Applied to Modelica*. Master Thesis, Linköping University, Department of Computer and Information Science, PELAB- Programming Environment Laboratory, ISRN:LIU-IDA/LITH-EX-A--11/019--SE, May 2011.

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

[11] 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.

[12] Terence Parr and R W Quong. *ANTLR: A Predicated- LL(k) Parser Generator*. Software Practice Experience, 25(7):789, 1995. ISSN 00380644. URLhttp://portal.acm.org/citation.cfm?id=213593.213603.

[13] Vern Paxson. Flex Manual, 2002. URL http://flex.sourceforge.net/manual/. [Accessed May 2011.

[14] Martin Sjölund, Peter Fritzson, and Adrian Pop. Bootstrapping a Modelica Compiler aiming at Modelica 4. In *Proceedings of the 8th International Modelica Conference* (Modelica'2011), Dresden, Germany, September 2011.