Final Thesis

Translating Natural Semantics to Meta-Modelica

by

Emil Carlsson

LITH-IDA-Ex--05/073--SE

2005-10-17

Linköpings universitet
Department of Computer and Information Science

Final Thesis

Translating Natural Semantics to Meta-Modelica

by

Emil Carlsson

LITH-IDA-EX--05/073--SE

2005-10-17

Supervisor: Adrian Pop

Examiner: Peter Fritzson

Abstract

The report describes the work associated with the analysis, design and implementation of the translator for natural semantics in Relational Meta-Language (RML) to the new meta-programming language Meta-Modelica. It also contains an introduction to the work associated with the development of the Modelica language and also an introduction to meta-programming in Meta-Modelica and RML.

Preface

This final thesis is part of the work associated with PELAB at the University of Linköping.

The translator, which is the result of this thesis will be used mainly to translate the Modelica compiler from an implementation in RML to one in Meta-Modelica.

I will also like to thank Peter Fritzson, my examiner and Adrian Pop, my supervisor for all support and feedback.

Table of Contents

Chapter 1 Introduction 1

1.1 Background and Purpose 1

1.2 Reading instructions 1

Chapter 2 Background 3

2.1 Modelica 3

2.1.1 The Modelica Language 3

2.1.2 Development 3

2.1.3 Extension of Modelica 3

2.2 Compiler/Parser concepts 4

2.2.1 Lexical Analysis 5

2.2.2 Syntax Analysis 5

Chapter 3 Problem statement 7

3.1 RML / meta-programming 7

3.2 RML-parser 7

3.3 Extended Modelica 7

3.4 Implementation 7

Chapter 4 Meta-programming 9

4.1 Meta-programming concept 9

4.2 RML 9

4.2.1 Modules 9

4.2.2 Uniontypes 9

4.2.3 Types. 10

4.2.4 Lists. 10

4.2.5 Relations 11

4.2.6 Rules. 11

4.2.7 Patterns 11

4.2.8 Other constructs 12

4.3 Meta-Modelica 12

4.3.1 Packages. 12

4.3.2 Types. 12

4.3.3 Lists. 12

4.3.4 Uniontype and Recordtypes 13

4.3.5 Access Restrictions 14

4.3.6 Algorithm section using matchcontinue. 14

4.3.7 Case statement 14

4.3.8 Functions as parameters 15

4.3.9 Replaceable types 15
4.3.10 Simple Functions 15

Chapter 5 Implementation. 17

5.1 Overview of the translator structure 17
5.2 The RML-parser 18
5.3 RML-unparser 18
5.3.1 Elements in lists 18
5.3.2 Parenthesis 19
5.3.3 String handling. 19
5.4 Comment handling 19
5.4.1 Solution 20
5.5 RML to Modelica translator 21
5.5.1 Translation of modules to packages 21
5.5.2 Translation of datatypes to uniontypes 22
5.5.3 Translation of relations to functions 22
5.5.4 Program database 24
5.5.5 Identifiers 25
5.5.6 Special generated types. 25
5.5.7 Generation of input and output variables 26
5.5.8 Generation of local variables 26
5.5.9 Translation of rules to case-statements 27
5.5.10 Functions as parameters 27
5.5.11 Replaceable types. 28
5.5.12 Simplified alghorithm section 28
5.5.13 Translation options 28
5.5.14 Comments 28
5.5.15 External modules (functions) 29
5.6 Meta-Modelica Unparser 29

Chapter 6 Conclusions 31

6.1 Meta-programming Knowledge 31
6.2 Testing 31
6.3 Future work 31

Appendix A 35

Chapter 1 Introduction

This chapter gives an short introduction and clarification to the final thesis and the report.

1.1 Background and Purpose

This report describes the background and work associated with implementation of a translator from the meta-programming language RML to the new meta-programming language Meta-Modelica. The translator is a contribution to the work associated with the research and development of the programming language Modelica and Meta-Modelica that is developed and mantained by researchers at PELAB (program enivironment lab) at ida, liu.

The translator will mainly be used to translate source code (RML) for the Modelica compiler.

To clarify some things for the reader it shall be mentioned that the RML-language which is used for the implementation of the translator is the same language that is parsed and translated to Modelica. The RML-language is used to modify the RML-language itself.

1.2 Reading instructions

To get an introduction to the background and the problem statement of the final thesis it is recommended to read chapter 2 and 3. If the reader wants to get information about what Modelica is all about chapter 2, contains short background information to the language.

For a more detailed and theroetical insight of the work of the final thesis chapter 4 and 5 are recommended for reading. Conlsusion of the final thesis can be found in the chapter 7.

Chapter 2 Background

This chapter contains background information related to the final thesis.

The final thesis is based and related to the work and development of the programming language Modelica and therefore this chapter contains background and information related to the Modelica language.

The programming language that is mainly used in the final thesis and also used in the implementation of the Modelica compiler is the meta-programming language RML. For more information and an introduction to the language of RML see chapter 4 about meta-programming.

Since compilers and parsers are strongly related to the work of the final thesis a theoretical background on how and when they are used is given in this chapter.

2.1 Modelica

This section gives some background information on the development and maintenance of the Modelica language. It also gives an short introduction to the language itself and ongoing extension with meta-programming concepts.

2.1.1 The Modelica Language

Modelica is a language used for modeling and simulation. The language is designed for modeling of complex physical systems like electrical, mechanical, hydraulical, thermodynamical model components. Examples of a typical physical systems that can be modeled in Modelica can be an engine, a cell phone, a planar pendulum, etc.

Modelica is object-oriented and equation-based language. The language has support for Component based design is ideally suited as an architectural description language for complex physical systems. The language is strongly typed has no side effects.

2.1.2 Development

The development, promotion and application of the Modelica language is held by a non-profit organization called *Modelica Association* with has its seat in the University of Linköping, but is an international effort.

There exists an open source software for development with Modelica called OpenModelica. Which is developed at PELAB.

There are also commercial software for modeling with Modelica like Dymola and MathModelica.

2.1.3 Extension of Modelica

An extension of the Modelica Language is/was under development and research during the year 2004/2005. The extension consist of several meta-programming concepts added to the Modelica language. The extension of the language is called *Meta-Modelica*.

Meta-programs are programs that have other programs as input and output. Typical applications that make use of meta-programming are program generators, interpreters, compilers, static analyzers and type checkers. It's very common that the compiler for a meta-programming language is implemented using the language itself. This is also the goal for Modelica-language.

The meta-programming concept is more investigated in Chapter 4.

2.2 Compiler/Parser concepts

A compiler is simply explained a program that reads a program written in one language and translates it to an equivalent program represented with another language. The program that is read is called source program and the output language is called target program.

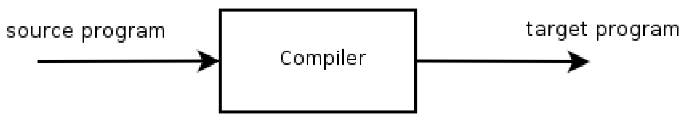


Figure 1: A compiler.

The compiler works in one or more phases. Common phases for a compiler are lexical analysis, syntactic analysis and semantic analysis. Other phases are intermediate code generation, code optimization and code generation. Interacting with all phases is error handling, which checks that the program are syntactic and semantic correct.

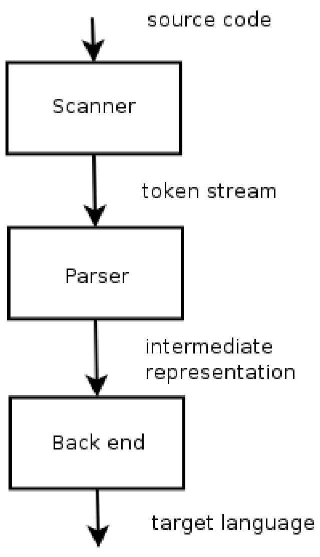


Figure 2: Three pass compiler.

For the this final thesis a compiler with three phases is used. This is a simple model of a compiler which is fully adequate for the final thesis. The compiler is shown in figure 2. The first two phases, lexical and syntax analysis are described in the next subsection to get the reader a short introduction to which type of compiler is used in this final thesis and how such a compiler works. This type of compiler is often referred to as a translator, and is referred as that in the rest of the report.

The interested reader can read more about advanced and traditional compilers in an appropriate book about compilers[3].

2.2.1 Lexical Analysis

In the lexical analysis, also called scanning, each character of the program is identified and and sevreal characters are grouped together. Each group is called a token. Identifiers, keywords and operators of the language are examples of tokens. Each token can be described with a pattern expressed with a regular expression:

```
{digit} [0-9]*
```

Lexical analysis is often done with tools such as lex, flex and jflex.

2.2.2 Syntax Analysis

The syntax analysis or more commonly called parsing involves grouping the tokens read by the scanner into grammatical phrases. These phrases is represented by a parse tree. An example of a parse tree is shown in figure 3 below.

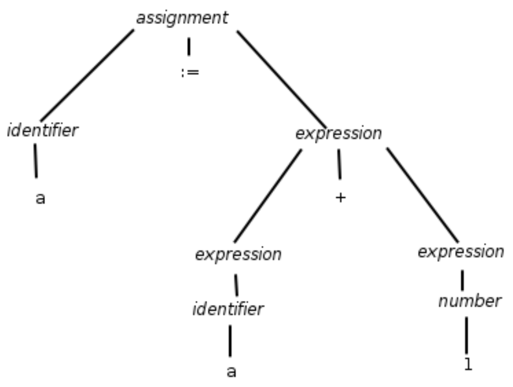


Figure 3: Parse tree for `a := a + 1`.

The phrases are grouped together with certain rules. Rules can be non-recursive or recursive. An expression for example may be a number, an identifier or another expression.

Syntax analysis is often specified using grammars which are interpreted with tools such as bison and yacc in order to automatically generate parsers.

Chapter 3 Problem statement

This chapter describes the problem statement of the thesis. The problem consists of several parts and they are all described here.

3.1 RML / meta-programming

One part of the work of the final thesis was to learn the language of RML and the concepts of meta-programming. To perform operations on program-code its preffered to use some kind of meta-programming language. RML is a meta-programming language and was recommended for me to use in the thesis. Thus, since RML is both the implementaion language and also the language to translate from, it was of great importance to learn the whole meta-programming language of RML. Ofcourse other meta-programming languages like Haskell, OCAML and SML could be learned and used when processing the code.

3.2 RML-parser

Another part of the thesis work is to construct a good parser for the RML-language. To do modifications on RML-code an internal representation of the code, that is easy to modify, is needed. This is most often achieved by using a scanner and a parser (such as lex and bison) that creates an abstract syntax tree (ast) to represent the original code. The ast should then be handled and processed with a meta-language like RML.

Fortunatly there was already a parser, that was implemented by Leif Stensson at ida, available at the time I started the thesis.

For me the work was to understand the implementation of the RML-parser, make it work and adjust it for the needs of the thesis. One important task was to ensure that the parser actually worked as expected.

3.3 Extended Modelica

Another part of the thesis work was to learn the Modelica language and its' newly introduced extension with meta-programming concepts, since that was the target-language of the translator.

The design and syntax of the Meta-Modelica was developed and changed during the work of the final thesis. The work and implementation had to interact and follow along with the development of the language.

3.4 Implementation

The main work of the final thesis was the implementation of the RML to Meta-Modelica translator and the implementation of the RML-unparser. The unparser was implemented mainly to ensure that the parser was parsing and behaving as expected. If you want to do refactorings on the RML code, the unparser can be used to present the refactored code in RML.

The implementation of the translator consist of several more or less difficult subproblems. Example of such problems are generating variables, type declarations, name conflics etc.

More of the implementation and the problems solutions are all described in chapter 5.

Chapter 4 Meta-programming

This chapter describes the concepts of meta-programming and gives a brief overview of the meta-programming languages RML and Meta-Modelica, which are used and processed within the final thesis.

4.1 Meta-programming concept

Meta-programming is mostly used to design and manipulate other languages. The program you write takes another program, a so called object-program, and performs changes on it. For example it could traverse the programs internal structure (the abstract syntax tree) and return a modified structure of the program. It is often used with a scanner and a parser that analysis the program and builds up an internal structure of it.

A meta-programming language has a way to store a tree representation of a program (ast). The tree may be recursive. The meta-programming language has functionality that provides an easy way to manipulate those trees.

4.2 RML

RML is a meta-language and generator tool for Natural Semantics and has been designed and developed by Mikael Petterson as his Ph.D. work.

The RML-language uses natural semantics to perform operations on data. The language declares relations with rules. The rules consists of premises and a conclusion, thus a form of natural semantics. The program tree (ast) or similar is defined in structures called datatypes. The relations in RML is mainly used to take datatypes as input, change or evaluate them, and return appropriate output. The main constructs and functionality of RML is introduced in the following sections. For more detailed information and a complete guide of the RML-language see the rml-guide[2].

4.2.1 Modules

RML uses a simple module system for information hiding and modularization. Relations and type definitions with similar properties are grouped in the same module. The constructs that may be contained in a module are datatypes, relations, type declarations and value definitions.

The module consists of one interface section and one implementation section, which can be optional. All construct declared in the interface section are made available for other modules, thus they are public.

4.2.2 Uniontypes

The structure and design of the ast(most often) is declared in RML with the keyword *datatype* - which is a uniontype. The keyword is followed by a name for the datatype. Each uniontype has one or more constructors consisting of a constructor name and zero or more fields containing different types. Each variant of the uniontype can be viewed as a recordtype and are separated with the symbol |. If the recordtype contains more then one type they are separated with the symbol *. It's possible to refer to the datatype itself inside that recordtype, i.e recursion is supported. This is best shown with a code example like the one below:

```
datatype Exp     =  INT of int
                  |  BINARY   of  Exp * BinOp * Exp
                  |  UNARY    of  UnOp * Exp
                  |  ASSIGN  of  Ident * Exp
                  |  IDENT   of  Ident

  datatype BinOp   =  ADD | SUB | MUL | DIV

  datatype UnOp    =  NEG
```

The datatype declaration in the example above was inspired from the used by standard ML (SML).

4.2.3 Types

The primitive types in RML are:

- *char* - representing a single character.
- *string* - representing a text string.
- *int* - a integer number.
- *real* - a real number.
- *bool* - a boolean value.

There are several primitive operations connected to each type. Like bool_or for booleans. The primitive types can also be compared with each other with the generic operator =.

Each type can have alternate names (aliases) introduced with type declarations:

```
type ident = string.
```

The basic data structures in RML are lists, vectors and option. They are so called parameterized types, which may take another type as parameter. They are declared by writing the keyword *list*, *vector* or *option* after a type, e.g. `int list`, which declares a list of integers.

4.2.4 Lists

A list can be matched and used in different ways and RML uses special language symbols and syntax to refer to a list. This is how list can be used in a pattern or expression:

- `[e1,e2]` - a list with two elements. `list(e1,e2)` has the same meaning.
- `[]` - empty list, keyword *nil* has the same meaning.
- `element::lst` or `cons(element,lst)` - concatenation, makes *element* the first element in the list *lst*.

4.2.5 Relations

A relation in RML is used to match certain types or datatypes. The keyword *relation* is followed by an identifier name. This is followed by an optional type signature of the inputs and outputs of the relation, e.g `(int => Exp)`. The relation consist of one or more rules with same name as the relation.

There are several built-in relations in RML like *int_add*, *int_sub*, etc.

4.2.6 Rules

Each rule has a match expression that matches specific types or datatypes.

The rule also consist of several premises that has to be fulfilled to apply the result. If a rule has no premises the rule can be defined as an axiom. Premises are often other relations or predefined relations with a pattern result. The patterns are used to bind variables to the result.

In the following example the call to the relation `int_neg` must succed in order for the relation `apply_unop` to succed. If it does the result is bound to variable *v2*.

```
relation apply_unop: (UnOp,int) => int =

 rule   int_neg(v) => v2
        ----------------------
        apply_unop(NEG,v) => v2
end
```

If the match-expression in the first rule matches, and all the premises in that rule are fulfilled, the relation will succeed. The result of the rule will then be passed on as the output for that relation. In the example above the value in the variable v2 will be returned as result of the relation. If some premise fail, the next rule will be checked for matching. If no rule succeed in a relation the whole relation will fail. Rules can also be enforced to fail under certain premises. That is achieved by having the keyword *fail* as output to that rule.

4.2.7 Patterns

As mention above patterns are used to decide which rule will match(be selcted) and to bind variables to a result. Patterns may contain the symbol _, which is treated as a wildcard in the pattern, i.e it may match with anything. A pattern may be a primitive type value, like a integer, a specific constructor from a datatype or a list, vector or an option. Parts of lists can be matched with special constructs and symbols described in section 4.2.4.

In the following example, the first rule in the relation rel1 will match if rel1 is called with an empty list, a list with one integer and the value 23.

```
relation rel1: (int list,int list,int) => int =
 axiom rel1([],e1::[],23) => 0
 ...
end
```

4.2.8 Other constructs

There are also other useful language constructs in RML. They are briefly described here:

- Values - declaration and definition of constant values.
- With statement - imports another rml-file to a module, so it's relations and datatypes may be used in that module.

4.3 Meta-Modelica

Modelica is a large programming language and Meta-Modelica is the meta-programming extension of it. Meta-Modelica has many similarities with RML but also many differences. All variables used in Modelica has to be declared. Each type has to be defined e.g. a list of integers. This section is mainly concentrated on the specific Meta-Modelica constructs and if the reader wants more insight to the complete Modelica language the Modelica Book[1] is recommended.

Patterns in Meta-Modelica are similar with the patterns from RML so they are not described here.

4.3.1 Packages

Meta-Modelica is using packages for modularization and encapsulation of data members. The package may consists of uniontypes, functions, import statements, type declarations and definitions of constant variables.

4.3.2 Types

The predefines types in Meta-Modelica are:

- *Integer*, a integer number.
- *String*, a string of characters.
- *Real*, a real number.
- *Boolean*, a boolean value.

The parameterized types used in Meta-Modelica are arrays, option and lists. The vector from RML is absent in Meta-Modelic and only arrays are used instead.

When a parameterized type is going to be used type declaration for them are needed. An example for how a list of integers is declared and used is given here:

```
type IntegerList = list<Integer>;
IntegerList int_list; /* int_list is a list of Integers */
```

4.3.3 Lists

List can be matched and used in the a similar way as in RML with special language symbols and syntax to refer to a list. The main difference is that Meta-

Modelica uses curly brackets instead of [], as list constructor. This is how list may be used in a pattern or expression:

- `{e1,e2}` - a list with two elements. `list(e1,e2)` has the same meaning.
- `{}` - empty list, keyword *nil* has the same meaning.
- `element::lst` or `cons(element,lst)` - concatenation, make *element* the first element in the list *lst*.

4.3.4 Uniontype and Recordtypes

Declarations with the keyword *uniontype* are corresponding to RML's datatypes. Each parametrized type, like lists, has to be declared with a type declaration together with the uniontype. Each uniontype is composed of recordtypes with each member declared as a variable. The recordtype is introduced with the keyword *record* followed by an identifier. An example of a uniontype is given below. This is the equivalent code with the one given in the RML datatype example:

```
public
uniontype Exp
  record INT
    Integer integer;
  end INT;
  record BINARY
    Exp exp1;
    BinOp binop2;
    Exp exp3;
  end BINARY;
  record UNARY
    UnOp unop;
    Exp exp;
  end UNARY;
  record ASSIGN
    Ident ident;
    Exp exp;
  end ASSIGN;
  record IDENT
    Ident ident;
  end IDENT;
end Exp;

public
uniontype BinOp
  record ADD
  end ADD;
  record SUB
  end SUB;
  record MUL
  end MUL;
  record DIV
  end DIV;
end BinOp;

public
uniontype UnOp
```

```
  record NEG
  end NEG;
end UnOp;
```

4.3.5 Access Restrictions

In Modelica class-members may be **`public`** or **`protected`**. A public member may be accessed by other packages that import that package. The protected may not.

Constant values in Modelica, are declared with the keyword **`const`**.

4.3.6 Algorithm section using matchcontinue

In Meta-Modelica functions is used to perform operations on the data specifid using uniontypes. These correspond to the relations in RML. The functions begins with a declaration part that defines the input and output variables of the function. The local variables used in the function also has to be declared here.

```
protected function apply_unop
  input UnOp in_unop;
  input Integer in_integer;
  output Integer out_integer;
algorithm
  out_integer:=
  matchcontinue (in_unop,in_integer)
    local Value v;
    case (NEG(),v) then -v;
  end matchcontinue;
end apply_unop;
```

Modelica uses different type of algorithms and equations. There are many different algorithms, the ones mainly used for meta-programming and Meta-Modelica are simple algorithm and algorithm with match statements.

The match algorithm statements are similar to the rules in RML. But in Meta-Modelica you may have different type of policies for the behavior of the rules. The keywords *match, matchcontinue* and *matchcondition* introduces different types of match policies.

Matchcontinue is the equivalent to the basic behavior of the rules in RML, that is if the first rule fails, the next is tried. Match is the opposite and will not try the next rule, it will fail immediately if a case fails. There are also other match policies with special conditions, but I this is not described here cause they are of less importance in this final thesis.

The match algorithm statement consists of one or more case-statements which are described in the next section.

4.3.7 Case statement

The case-statement is equivalent to a rule in RML. The case-statement has a match-expression followed by a Modelica style equation with equation statements. The equation statements are the premises of the case-statement, and have to be fulfilled for the case-branch to succeed.

The case statement ends with the keyword **`then`** followed by an expression which is the result of the casebranch. Like in RML a case statement can be forced to fail, but here with a call to a built-in function called *fail()*.

4.3.8 Functions as parameters

In RML you may have a relation as input or output to another relation. This relation may then be called and used in the premises. In RML this is declared using a signature like the one describing input and output of the relation. In Meta-Modelica this is represented with a function type with input/output variable declaration only. This function type name is then used to declare such a function variable as input.

4.3.9 Replaceable types

In RML it is possible to use special types in relations that may be of any type.These types are called polymorphic types. Maybe you want a list with elements of certain type, integers or strings. The functions may perform similar operations on them but instead of declaringtwo different relations you may use the replaceable types, shown in the example below.

This kind of replaceable types are supported in Modelica. The keyword replaceable is used to introduce such a type:

```
replaceable type Type_a;
```

4.3.10 Simple Functions

When a function only has one case-statement that will always match it may be written as a simple algorithm statement without the match-statement like the example below:

```
protected function neg_int
  input Integer v1;
  output Integer v2;
algorithm
  v2 := -v1;
end neg_int;
```

Instead of having the standard match continue-case-statement:

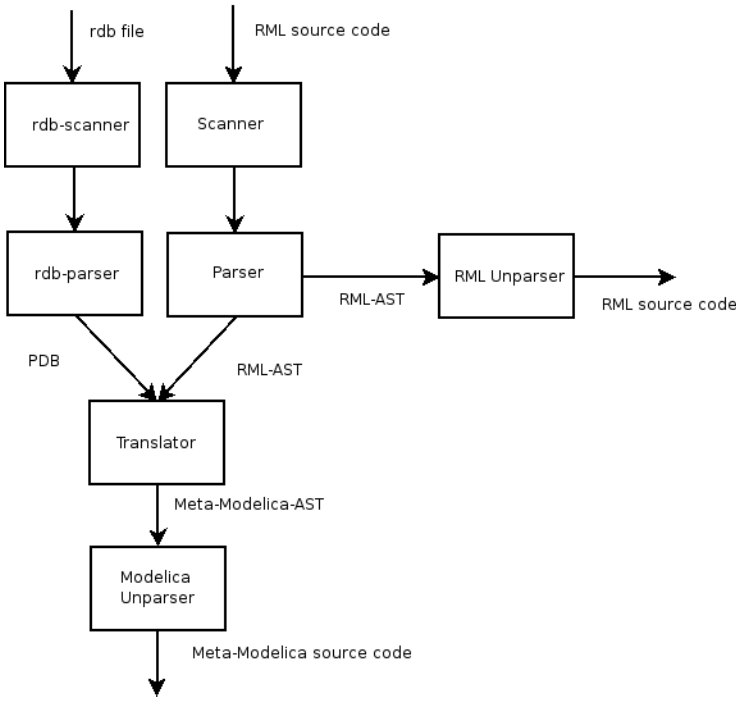
```
protected  function neg_int
  input Integer in_v;
  output Integer out_v;
algorithm
out_v:=
  matchcontinue(in_v)
local v1;
    case(v1) then -v1;
  end matchcontinue;
end neg_int;
```

Chapter 5 Implementation

This chapter contains the description of the implementation. First a general overview of the translator tool is given and then each part is separately described.

5.1 Overview of the translator structure

The program consists of several modules and program blocks. The main structure of the implementations building blocks are shown in figure 4.



Figur 4: Main structure of the RML to Meta-Modelica translator.

There is a scanner and a parser for the RML source code building a parse tree for RML. There is also a RML Unparser in the program that may be used to generate RML source code from the RML-AST.

There is also a scanner and parser for the special rdb-files that contains a program database with information for each identifier in the RML source code. More of how the database is used by the program can be read in section 5.5.4 on page 24.

The result of the two parsers is used as input for the translator which produces a Meta-Modelica-AST that in the last step is unparsed with the Meta-Modelica Unparser in order to generate the Meta-Modelica files.

The RML Unparser, the Translator and Modelica Unparser are all implemented in the RML language. The parsers are using a classical bison implementation and some kind of C-based lexer or clean C implementation for the scanners.

5.2 The RML-parser

When the final thesis started a preliminary a parser was available for RML, but had to be adjusted to be used for interaction with the RML-language. The parser was written by Leif Stensson at ida and an introduction to the parser was given when the final thesis started. Some parts of the parser were incomplete, but with help from Adrian Pop, the supervisor for this final thesis, the parser has been completed.

The scanner used with parser is written in C and the parser was originally written in yacc, but was easily converted to an implementation with bison. The reason to why bison was used is because it is more up to date then yacc and is also more widely used.

The scanner and parser are as mentioned fed with RML-source code and the output is a resulting abstract syntax tree. This tree is actually formed using datatype structures in RML. The other parts of the program can then easy process this datatype. The design of this abstract syntax tree is given in a file called absyn.rml and can be found in appendix A.

During the work with final thesis several parts were added and redesigned in the parser, due to changes in the design of the RML-astes, to further improve the parser and correct errors in it.

5.3 RML-unparser

The RML-unparser was implemented to get a visual view of how the parser was building up the ast. An unparser is simply a module that traverses a parse-tree and printing it to the screen or a text file. This can be done by pretty-printing or not. The RML-unparser is using pretty printing to be easier for a user to compare it with the original source code in RML.

The unparser for the rml-ast was straightforward to implement. An overview of the more complicated and interesting parts of work with the implementation is given in the following subsections.

5.3.1 Elements in lists

When the unparser prints a list to the screen it prints every element to the screen but has to make special considerations with the last element, because it should not have a comma afterwards. In the example below it’s is shown how this behaviors is implemented in the unparser.

```
relation dump_pattern_list =
  rule  print ""
       -----------------------------
       dump_pattern_list([]) => ()

  rule  dump_pattern(last)
       -----------------------
       dump_pattern_list(last :: []) => ()

  rule  dump_pattern(first) &
```

```
        print ", " &
        dump_pattern_list(rest)
        ---------------------------
        dump_pattern_list(first :: rest) => ()

end
```

Many parts of the RML-ast are lists of various types, like pattern lists or expression lists. In the example above it shown how a pattern list is unparsed. The last element in the list is matched with rml cons and the nil which is the empty list. The empty list is also matched to if the list happens to be empty. The last rule is unparsing the first element and a comma and then calling the relation recursively continuing unparsing the rest of the list.

5.3.2 Parenthesis

In RML you can skip the parenthesis on relation calls that have no arguments. The parser treats this the same way as relation calls that have parenthesis. This means that the same abstract syntax is built up in both cases. Thus, the call:

```
rel() => b
```

and the call

```
rel => b
```

gets the same represention in the AST. When this is unparsed the unparser prints the paranthesis for relation calls with no aguments. So all the relation calls without paranthesis are represented with paranthesis in the unparsed RML code. This may be used for automatic addition of paranthesis to relation calls.

5.3.3 String handling

When unparsing strings all the characters in the string is checked to see if they belong to escape/special characters like " \n", " \b", and " \t". Otherwise they are printed with their positional effect directly to the output. In the implementation of the RML-unparser this behavior is realised by converting each character to a char type and then check it’s ascii-value to see if it is one of these escape characters. If so is the case, an extra \ is added to the character to escape the effect. This escape handling is implemented with help of a relation called handle_escape in the RML-unparser. The implementation of the relation is present here:

EXAMPLE

5.4 Comment handling

Normal tokens, like identifiers and keywords, in a program are only allowed in a restricted way. But the comments are allowed anywhere in the program code. The solution to add comments everywere in grammar of the parser is not a good solution as it makes the pareser and the grammar difficult to understand. To allow it in some places is an improper restriction and if it’s not followed, parsing errors will arise.

It is therefor better to implement the comment handlingoutside the traditonal parsing techninque.

The original implementation of the parser did not have any support for the comments. They were just collected in a buffer. As described in the in section 5.1 on page 17 this is a problem that is tricky to handle, because there is no good standard solution to this. The solution used in this final thesis is non-general and only specialized for RML. The next section gives the solution used in this final thesis.

5.4.1 Solution

The scanner is used to gather information about the comments. The comments, their position and some additional information is stored in an array of structures, each containing this information. The structure is shown in the following code-example:

```
struct CommentInfo
{
  int   bound; /* is it a bound? (used to mark next datastructure) */
  int   firstline, firstcol; /* start position of this comment */
  int   lastline, lastcol;   /* end position of this comment */
  char buffer[LEXER_COMMENT_MAXLENGTH+100];
};
```

The array of such structures is then used to place the comments in the right place in the AST while parsing. The structures in the rml-AST have containers for optional comments. Depending on the position of a comment, it is placed out in a suitable structure of the AST.

But there are other problems that arise when using this kind of array. The scanner may not have come far enough when comments shall be placed in the ast. Comments below a certain program element may not have been scanned yet when we want to place them in the corresponding AST-structure. This may result in some changes in the placement of the comments.

Another problem arise when we don't know how far below a program element we want to check for comments. The following comments may belong to the next program element, or even the next after that. The positional information could be used for help in comment analysis but this is hard for compact program elements as datatypes. Therefore there are also variable bound in the comment structure as shown in the code-example above. The scanner also adds comment structures that acts if they are bounds, containing an empty comment, to the array. When a bound is found we know that the following comments are a part of another program element.

Since the comments are placed out in the RML-ast the comments in the translated Meta-Modelica-ast will be based on the ones from in the RML-ast.

5.5 RML to Modelica translator

The main work of the implementation and the objective for the final thesis was the implementation of the translator from RML to Meta-Modelica. There were many subproblems and things to think of when designing this translator.

Due to differences in the languages, there are sveral basic things to consider and the translation is not that straightforward as it first may seem. Some things that has to be dealt with are:

- Declaration of variables with correct types,
- Generation of type declarations. Checks so the type declarations are reused and not declared more than once.
- Name collisions with modelica keywords, generated variable identifiers and type identifiers.
- Placement and handling of comments and strings.

Since the translator is used to translate the Modelica compiler and the code has to be understood by programmers it was desirable to make simple and readable code. Some things to be considered according to these requirements:

- Generate good and smart identifier names,
- Make code as simple as possible and keep the number of lines in the target language as low as possible.
- Make code readable.

These problems are in more detail investigated in the next subsections.

The translator takes a RML-ast as input and gives an Meta-Modelica-ast as output. The translator traverses the whole RML-ast step by step, analyzing every construct and generating a new Meta-Modelica-ast with corresponding construct in Meta-Modelica.

The translator also takes a program database with the information of every identifier. How this is used is described in section 5.5.4. It also takes options (command line paremters) that may be given to the translator, which are detailed in section 5.5.13.

Interesting details about the implementation and the special data structures used in the translator are discussed in the following subsections.

5.5.1 Translation of modules to packages

A module in RML is translated to a package in Meta-Modelica. In RML some members of the module, like relations and so on, are declared in the interface part and some in the definition part, as described in X. In Meta-Modelica the interface vs. definition division is absent, but we need to keep track of which members are in the interface, because those are then consequently going to be **`public`** in Meta-Modelica and those who are not are going to be **`protected`**. The decision for how we handle the public-protected attributes for functions is described in section 5.5.3.

5.5.2 Translation of datatypes to uniontypes

When translating a datatype to a uniontype special types must be generated if needed and each variable in a record must be declared with a suitable generated name. Otherwise the translation is pretty straightforward. The datatype name is used as uniontype name. Constructor names are used as recordnames. The implementation of the special generated types is described in section 5.5.6.

The generation of variables for the members of the recordtypes was realited first by using simple names like x1,x2 etc., but there was a need for smarter names, based on the types and even better, generated from the comments. Since many of the elements in the datatypes have a comment that describes the element with one word this description can successfully be used to generate a suitable variable name. This is implemented in the translator and only used when the comment consists of one word with less than 15 characters. Such strategy was later proven to give good results in the final translation. In the case where the comments are absent, the name of the variable/component is generated from the type names.

The following datatype in RML:

```
(* The basic element type in Modelica *)
  datatype Element = ELEMENT of bool      (* final *)
                            * bool      (* replaceable *)
                            * InnerOuter      (* inner/outer *)
                            * Ident                 (* Element name *)
                            * ElementSpec           (* Actual element
specification*)
                           * string              (* Source code file
*)
                            * int                   (* Line number *)
                            * ConstrainClass option (* only valid for
classdef and component*)
```

is translated to the following uniontype in Meta-Modelica:

```
uniontype Element " - Elements
  The basic element type in Modelica "
  record ELEMENT
    Boolean final "final ";
    Boolean replaceable "replaceable ";
    InnerOuter innerouter "inner/outer ";
    Ident ident "Element name ";
    ElementSpec elementspec "Actual element specification";
    String string "Source code file ";
    Integer integer "Line number ";
   ConstrainClassOption constrainclassoption "only valid for classdef
and component";
  end ELEMENT;
end Element;
```

5.5.3 Translation of relations to functions

When a RML relation is translated to a Meta-Modelica function we perform several steps. Every RML structure is translated to the corresponding structure in Meta-Modelica. Input and output variables are generated from the relation signature and they are assigned suitable variable names. All local variables in each rule have to be collected and declared with the correct type. Each rule is then translated to corresponding Meta-Modelica code. A basic example of a translated relation is presented below.

The following relation in RML:

```
relation eval: Exp => real  =

  axiom eval( RCONST(ival) ) => ival   (* eval of an integer node *)
                                         (* is the integer itself *)

  (* Evaluation of an addition node PLUSop is v3, if v3 is the result
of
   * adding the evaluated results of its children e1 and e2
   * Subtraction, multiplication, division operators have similar
specs.
   *)

  rule  eval(e1) => v1  &  eval(e2) => v2  &  real_add(v1,v2) => v3
        ----------------------------------------------------------
        eval( ADDop(e1,e2) ) => v3

  rule  eval(e1) => v1  &  eval(e2) => v2  &  real_sub(v1,v2) => v3
        ----------------------------------------------------------
        eval( SUBop(e1,e2) ) => v3

  rule  eval(e1) => v1  &  eval(e2) => v2  &  real_mul(v1,v2) => v3
        ----------------------------------------------------------
        eval( MULop(e1,e2) ) => v3

  rule  eval(e1) => v1  &  eval(e2) => v2  &  real_div(v1,v2) => v3
        ----------------------------------------------------------
        eval( DIVop(e1,e2) ) => v3

  rule  eval(e) => v1  &  real_neg(v1) => v2 (*aa*)
        -----------------------------------
        eval( NEGop(e) ) => v2 (*ss*)

end
```

is translated to the following function in Meta-Modelica:

```
public function eval " Abstract syntax of the language Exp1
   Evaluation semantics  of Exp1 "
  input Exp in_exp;
  output Integer out_integer;
algorithm
  out_integer:=
  matchcontinue (in_exp)
    local
      Integer ival,v1,v2;
      Exp e1,e2,e;
    case (INTconst(ival)) then ival;  " eval of an integer node   is
the integer itself "
    case (ADDop(e1,e2)) " Evaluation of an addition node PLUSop is v3,
if v3 is the result of
   * adding the evaluated results of its children e1 and e2
   * Subtraction, multiplication, division operators have similar
specs.
   "
      equation
        v1 = eval(e1);
        v2 = eval(e2); then v1 + v2;
    case (SUBop(e1,e2))
      equation
        v1 = eval(e1);
        v2 = eval(e2); then v1 - v2;
    case (MULop(e1,e2))
      equation
        v1 = eval(e1);
        v2 = eval(e2); then v1*v2;
    case (DIVop(e1,e2))
      equation
        v1 = eval(e1);
        v2 = eval(e2); then v1/v2;
    case (NEGop(e))
      equation
        v1 = eval(e); then -v1;
  end matchcontinue;
end eval;
```

Another aspect that must be decided for each function is if it is **`public`** or **`protected`**. All relations that are declared in the interface in RML will be public in Meta-Modelica. If a function is not present in the interface in RML it will become protected in Meta-Modelica. Such selection has the consequence that it's needed to know which relations are declared in the interface when translating the implementation of the relations. When translating the interface a list with identifiers of the relations that are present is built up. The list is then checked when translating the implementation of each relation to decide wether the functions should be public or protected.

5.5.4 Program database

The RML-compiler has special functionality for generating rdb-files, which contains information of every identifier in a RML-file. The supervisor of this final thesis, Adrian Pop, helped with the implementaion of a parser for these rdb-files, in order to build up an internal program database for each rml-file. The program database is used in the implementation of the translator for retrieving the type of a specific variable. The reader is reffered to figure 4 for a better understanding of the coupling between the translator and the RML compiler. The program database has information about all identifiers position and their type. The data structures representing the program database have the following representation in RML:

```
(* start line/column end line/column *)
  datatype RMLDbRange = RMLDB_RANGE of int * int * int * int

  datatype RMLDbElement = RMLDB_VAR of string * (* filename *)
                                 RMLIdent * (* var name *)
                                 RMLDbRange * (* actual position *)
                                 RMLDbRange * (* scope *)
```

```
                                   RMLIdent     * (* relation name *)
                                   RMLType      (* type *)
                        | RMLDB_REL of string * (* filename *)
                                   RMLIdent *     (* relation name *)
                                     RMLDbRange *  (* relation ident 
position *)
                                     RMLType      (* relation type *)
                        | RMLDB_TY  of string * (* filename *)
                                      RMLIdent * (* type name *)
                                    RMLDbRange    (* type position *)
                        | RMLDB_CTOR of string * (* filename *)
                                       RMLIdent *   (* constructor 
name *)
                                       RMLDbRange * (* position *)
                                       RMLType    (* type *)

  datatype RMLDb = RMLDB of RMLDbElement list
```

When looking up an identifier type in the program database the range and correct file is checked to ensure that the right variable information is retrieved.

5.5.5 Identifiers

All identifiers used in RML must be checked before they are used in Meta-Modelica.

One thing that has to be checked is if the identifier is a standard Modelica keyword. A RML-value with a list of all such Modelica identifiers is used when translating. If a Modelica keyword is found a "_" is attached to the end of the identifier to differentiate them from the keyword.

Another problem with the identifiers is to separate variables and constructors of a datatype in expressions and patterns. The problem arise when the constructor do not have any arguments. The parser builds up the same ast element for these two program elements. This is not a problem for the RML unparser, but when a uniontype with no arguments is referred to in Meta-Modelica it should have parentheses. We needed to separate these two patterns with help from the program database. All identifiers which are data constructors or variables are checked against the program database to see if they are a constructor or not. In this way we know if the translator is going to generate a variable or datatype constructor reference.

5.5.6 Special generated types

As discussed in the analysis the parameterized types need separate type declaration for each combination and they can not be declared again in the same scope. A special structure called AlternativeTypeNames is used in the translator to keep track of which type is already declared. We present this datastructures below:

```
datatype AlternativeTypeNames = ATYPES of Absyn.Ident * 
Absyn.Ident * bool
```

A list of AlternativeTypeNames is passed along to the relations in the translator and the list is consulted when it is needed to know if a special type is already declared.

Thus, these types were implicitly created by the translator. But in RML one can also have, explicit type declarations. In some occasions it could be nice to also reuse these type declarations. The second element in AlternativeTypeNames represent an alternative name for a declared type. During the implementation, the design/choice for where to reuse the explicit types has changed.

To handle these new design decisions a special element, the third one in the AlternativeTypeNames, was added to the structure of the declared types. The element is a boolean which decide if the type is declared explicit or implicit. When the list of declared types is checked in different situations it is easy to choose if the explicit variable shall be excluded or not.

5.5.7 Generation of input and output variables

The signatures ofRML-relations declares what types are used as parameters and results for a relation. In a similar way the Meta-Modelica functions declare input and output variables. The type of each such variable can directly be retrieved from the signatures. But in RML the siganture declaration part is optional. When the signature is missing the translator has to look up the signature in the program database. The names for the input and output variables are based on the name of the type. The only difference is that the letters are lowercased and the the input variables are prefixed with *in_* and the output variables with *out_*. When name collision arise a number is added to each variable among the input or output variables. This is shown in the table 1.

RML	Meta-Modelica

`relation apply_binop:`
`(BinOp,int,int) => int =`
`...`
`end`
	`protected function apply_binop`
	`  input BinOp in_binop1;`
	`  input Integer in_integer2;`
	`  input Integer in_integer3;`
	`  output Integer out_integer;`
	`...`
	`end apply_binop;`

Table 1: Translation of input and putput types.

5.5.8 Generation of local variables

The local variables used in a RML relation need to be declared in the translated Modelica function with the correct type. When translating a rule the variables are kept in a special structure called *TypeVarsElement*. The structure has one element representing a RML type and one element with a list of identifiers. The structure is declared as follows:

```
datatype TypeVarsElement = VTELEMENT of Absyn.RMLType *
Absyn.Ident list
```

The identifiers are kept in this structure because the local variables can not be generated until we have translated/investigated the whole relation. Duplicates...

All identifiers from a rule are first collected and then locked up in the PDB. A special relation to update the TypeVarsElement and check the program database is used. The relation checks wether the variable is already in the list of TypeVarsElements and if it is it exits, otherwise it adds it to the correct element in the TypeVarsElements. If an identifier is presented with the same name but with different type, we need to generate that variable as local in that case-statement. Therefore the update relation gives two outputs, one list of TypeVarsElement for the variables to be declared in the function and one for the ones to be declared in the case statement. For each rule, the list for the variables to be declared in the function are appended to a list that is passed along when translating the rules. When all rules are translated we can generate the local variables from the list of TypeVarsElements.

5.5.9 Translation of rules to case-statements

As discussed in section 4.2.6 the rules consists of premises and a conclusion which are in turn built up with expressions and patterns. The expressions and patterns are straightforward to translate to equivalent Meta-Modelica code. The whole rule is translated to a case statement in Meta-Modelica. An example of how this looks can be found in section 4.3.7.

However as mentioned there are also predefined relations in RML which must be translated to equivalent functions in Meta-Modelica. For example RML has relations for modifying vectors and arrays. In Meta-Modelica there are only arrays so the vectors and vector operations must be transformed to array operations.

In Meta-Modelica it is also possible to transform the expressions which may simplify the code. An example of such transformation is skipping the last assignment to a variable, if that variable is going to be returned, and instead return the assignment expression directly after the keyword **`then`**. An example of this can be found in the translation of relation *eval* above, where the variable v3 is skipped. The transformation is implemented in the translator, as a simple check: if the return variable is the same as the last assignment variable is the expression assigned to the variable is set in the return construct.

5.5.10 Functions as parameters

When translating a relation that has another relation as parameter some aspects have to be considered. The corresponding Meta-Modelica way to achieve such behavior is to define a function type with only input and output variables declared. Such function needs a good name. In an early implementation the translator generated a function name based on input and output but this generated very long function names in some cases. The final implementation uses names like FuncTypeX were X is a number to separate the functions if there are more than one. The name of the input and output variables of these functions are generated in the same way as input and output variables in normal functions described in X.

5.5.11 Replaceable types

The replaceable types in RML are translated to the corresponding code in Meta-Modelica. The declaration and use for Meta-Modelica are described in X. In RML the replaceable types are declared with a quote in the beginning. When translating this from RML, Type_A is used as name if the name in RML was a. When this is combined with parameterized types the translator puts V in front of the generated special type's name. This is because the name Type is used in a Meta-Modelica to RML translator(adrian's) to identify a replaceable type.

5.5.12 Simplified alghorithm section

When it is possible, the translator will the simplified algorithm section, described in X, for functions. Such simplification is possible when a relation has

- only identifiers as input or no inputs at all.
- only one case-statement.

The matchcontinue and case statement then skipped as shown in the example in section 4.3.10.

5.5.13 Translation options

In the current implementation it is possible to give some options to the translator. There are only two options that may be given at the moment, but it easy to add new options to the translator. A special datatype is used to store the options and a list of such values is passed along in the translator:

```
datatype TransOptions  = IMP_PREFIX of string list
                       | DUMMY_GENERATION
```

The list can then be queried at various occasions to ask if a certain option is set. The datatype can also easily be extended with more options.

The two options that can be given to the translator at the moment are:

- Modelica prefixing. The packages that are imported can have a default path which should be added to each import statement. Right now the OpenModelica compiler path is used if no other path is set.
- Use generation of dummy variables for functions that have no input/output variables.

5.5.14 Comments

When translating the comments from RML some processing is needed in order to to better fit the Meta-Modelica language. Some of the processing are necessary for the Meta-Modelica files to work and some is just for adjustment to Meta-Modelica. For every comment each character is checked to see if there are any parts that need to be translated. Here are some of the processing filters applied to the comments when they are translated in the current implementation:

- Escaping of doubleqoutes ". This is allowed in RML comments but not inside Meta-Modelica comments so it is necessary to translate this with the escape character \".
- The word *relation* in the comments is translated to word *function*.
- In RML * and spaces are more often used in the comments. These are removed at appropriate places by the translator.
- Description of lists in RML are translated to list in Meta-Modelica, i.e. [a,b,c] is translated to {a,b,c}.

5.5.15 External modules (functions)

In RML you can use external modules/programs implemented in C with an interface declared in RML. For example an implementation in C. Such external modules are denoted by only declaring the interface part in the RML-file. There is a public relation in the translation module that checks if a RML file is external. The relation is simply checking if the list of definitions is empty.

```
relation is_external =

  axiom is_external(Absyn.RML_FILE(_,_,[],_) )=> true

  axiom is_external(_) => false

end
```

This check is done before translating a RML-ast. If we have a external module all relations in the interface will be translated to external function declarations like this example below:

5.6 Meta-Modelica Unparser

When translating the code to a parse tree in Meta-Modelica one needs to unparse the AST to get a readable output. Thus the Meta-Modelica source code. We started froma preliminary Modelica unparser implemented in the Modelica-compiler. This unparser was used, adjusted and extended to pretty print the Meta-Modelica code. The case, matchcontinue, patterns and union-types constructs are some examples of what was added to complete the Meta-Modelica unparser.

Chapter 6 Conclusions

This chapter contains end conclusions and the result and correctness of the generated Meta-Modelica code.

6.1 Meta-programming

The concepts of meta-programming and the two meta-programming languages Meta-Modelica and RML was succesfully learned during the thesis.

6.2 Testing

The testing of the correctness and the quality of the generated Meta-Modelica code has been done by translating examples in RML. These examples are used in the RML book[2] and are now translated and used in Meta-Modelica version of that book.

Also the whole Modelica-compiler was translated and the result has been evaluated by Ph.D students at the University and by Peter Fritzson. From this translation many errors and new design choices for the translator came up. Now it is possible to compile the translated Meta-Modelica code and the code is readable, very concise and easy to understand.

6.3 Future work

What has not been tested or evaluated is the performance of the translator. Some improvment may be possible to speed up the translation. Now it take aprox. 10 minutes to translate the whole Modelica compiler(about 45.000 rows of code). This could be done faster. One thing that could speed up the translation time using tree's instead of lists in larger lists like, pdb, atypes.

References

[1] Peter Fritzson: *Principles of Object-Oriented Modeling and Simulation With Modelica 2.1*. 2004, Wiley-IEEE Press. 940 pages, ISBN:0-471-471631, Book home page: http://www.matchcore.com/drmodelica.

[2] Petef Fritzson: *Efficient Language Implementation by Natural Semantics.* 2005.

[3] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: *Compilers Principles, Techniques and Tools.* 1986, Addison Wessley Longman, ISBN: 0-201-10088-6.

Appendix A

```
(*
   Copyright PELAB, Linkoping University

   This file is part of Open Source Modelica (OSM).

    OSM is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    OSM is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with OpenModelica; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

*)

(**
 ** file:                 absyn.rml
 ** module:      Absyn
 ** description: Abstract syntax
 **
 ** RCS: $Id: absyn.rml,v 1.83 2004/12/08 12:26:33 petar Exp $
 **
 ** This file defines the abstract syntax for Modelica in RML.  It mainly
 ** contains datatypes for constructing the abstract syntax tree
 ** (AST), relations for building and altering RML datatypes and a few rela-
tions
 ** for printing the AST.
 **
 ** absyn.rml's constructors are primarily used by the walker
 ** (modeq/absyn_builder/walker.g) which takes an ANTLR internal syntax tree
and
 ** converts it into an RML abstract syntax tree.
 **
 ** When the AST has been built, it is normaly used by explode.rml in order to
 ** build the scode (See explode.rml). It is also possile to send the AST do
 ** the dumper (dump.rml) in order to print it.
 **
 ** For details regarding the abstract syntax tree, check out the grammar in
 ** the Modelica language specification.
 **
 **)

module Absyn:

  datatype Info = INFO of string * (* file *)
                               int *    (* startline *)
                               int *    (* startcolumn *)
```

```
                          int *    (* endline *)
                          int      (* endcolumn *)

  (** An identifier, for example a variable name *)
  type Ident = string

  (** - Programs, the top level construct *)
  (** A program is simply a list of class definitions declared at top
   ** level in the source file, combined with a within statement that
   ** indicates the hieractical position of the program.
   **)
  datatype Program = PROGRAM of Class list (* List of classes *)
                               * Within (* Within statement *)
                              | BEGIN_DEFINITION   of Path  (* For split defi-
nitions*)
                               * Restriction (* Class restriction *)
                               * bool (* Partial *)
                               * bool (* Encapsulated *)
                              | END_DEFINITION of Ident (* For split defini-
tions *)
                              | COMP_DEFINITION of ElementSpec (* For split
definitions*)
                                 * Path option (* insert into.
                                          Default, NONE *)
                              | IMPORT_DEFINITION of ElementSpec(* For split
definitions*)
                                 * Path option (* insert into.
                                          Default, NONE *)
                              | RML_FILE of RMLIdent * RMLInterface list *
RMLDefs list * string list


  (** Within statements *)
  datatype Within = WITHIN of Path | TOP

  (** - Classes *)
  (** A class definition consists of a name, a flag to indicate if this *)
  (** class is declared as `partial', the declared class restriction, *)
  (** and the body of the declaration. *)
  datatype Class = CLASS of Ident     (* Name *)
                              * bool     (* Partial *)
                              * bool     (* Final *)
                              * bool     (* Encapsulated *)
                              * Restriction             (* Restricion *)
                              * ClassDef     (* Body *)


  (** The `ClassDef' type contains the definition part of a class *)
  (** declaration.  The definition is either explicit, with a list of *)
  (** parts (`public', `protected', `equationc' and `algorithm'), or it *)
  (** is a definition derived from another class or an enumeration type. *)
  (** For a derived type, the  type contains the name of the derived class and
an optional *)
  (** array dimension and a list of modifications. An enumeration type contains
a list of *)
  datatype ClassDef = PARTS of ClassPart list
                                   * string option     (* string comment *)
```

```
                                | DERIVED of Path
                                     * ArrayDim option     (* *)
                                     * ElementAttributes
                                     * ElementArg list
                                     * Comment option     (* comment *)
                    | DERIVED_TYPES of Path          (*ADDED*)
                                     * Path list
                                     * Comment option
                                | ENUMERATION of EnumLiteral list
                                 * Comment option     (* comment*)
                                     | OVERLOAD of Path list (* function names
*)
                                 * Comment option

  (** EnumLiteral, which is a name in an enumeration and an optional
   Comment.*)
  datatype EnumLiteral = ENUMLITERAL of Ident     (* Literal *)
                               * Comment option     (* comment *)


  (** A class definition contains several parts.  There are public and *)
  (** protected component declarations, type definitions and `extends' *)
  (** clauses, collectively called elements.  There are also equation *)
  (** sections and algorithm sections. The EXTERNAL part is used only by func-
tions *)
  (** which can be declared as external C or FORTRAN functions. *)

  datatype ClassPart = PUBLIC of ElementItem list
                                 | PROTECTED of ElementItem list
                                 | EQUATIONS of EquationItem list
                                 | INITIALEQUATIONS of EquationItem list
                                 | ALGORITHMS of AlgorithmItem list
                                 | INITIALALGORITHMS of AlgorithmItem list
                                 | EXTERNAL of ExternalDecl * Annotation option

  (** An element item is either an element or an annotation *)
  datatype ElementItem = ELEMENTITEM of Element
                                   | ANNOTATIONITEM of Annotation
  (** An element item is either an element or an annotation *)

  (** - Elements *)
  (* The basic element type in Modelica *)
  datatype Element = ELEMENT of bool      (* final *)
                                    * bool      (* replaceable *)
                                    * InnerOuter      (* inner/outer *)
                                    * Ident                (* Element name *)
                                    * ElementSpec          (* Actual element
specification*)
                                    * string              (* Source code file
*)
                                    * int                  (* Line number *)
                                    * ConstrainClass option (* only valid for
classdef and component*)


  (* Constraining type, must be extendes *)
  type ConstrainClass = ElementSpec
  (** An element is something that occurs in a public or protected
   ** section in a class definition.  There is one constructor in the
   ** `ElementSpec' type for each possible element type.  There are
   ** class definitions (`CLASSDEF'), `extends' clauses (`EXTENDS')
   ** and component declarations (`COMPONENTS').
   **
   ** As an example, if the element `extends TwoPin;' appears
   ** in the source, it is represented in the AST as
   ** `EXTENDS(IDENT("TwoPin"),[])'.
   **)
  datatype ElementSpec = CLASSDEF of bool     (* replaceable *)
                                 * Class
                                | EXTENDS of Path * ElementArg list
                                | IMPORT of Import * Comment option
                                | COMPONENTS of ElementAttributes (*1.1 con-
tains Araydim also.*)
                                   * Path     (* type name *)
                                   * ComponentItem list
  (** One of the keyword inner and outer CAN be given to reference an inner or
      outer component. Thus there are three disjoint possibilities. **)
  datatype InnerOuter = INNER | OUTER | UNSPECIFIED

  (* Import statements, different kinds  *)
  datatype Import = NAMED_IMPORT of Ident * Path
                             | QUAL_IMPORT of Path
                             | UNQUAL_IMPORT of Path

  (* Collection of component and an optional comment *)
  datatype ComponentItem = COMPONENTITEM of Component
                                  * Comment option

  (* Some kind of Modelica entity (object or variable) *)
  datatype Component = COMPONENT of Ident      (* component name *)
                                * ArrayDim        (* Array dimensions, if
any *)
                                * Modification option (* Optional modifica-
tion *)

  (** Several component declarations can be grouped together in one *)
  (** `ElementSpec' by writing them on the same line in the source. *)
  (** This type contains the information specific to one component. *)
  datatype EquationItem = EQUATIONITEM of Equation * Comment option
                             | EQUATIONITEMANN of Annotation

  (** Info specific for an algorithm item. *)
  datatype AlgorithmItem = ALGORITHMITEM  of Algorithm * Comment option
                              |  ALGORITHMITEMANN of Annotation

  (* Information on one (kind) of equation, different constructors for differ-
ent
     kinds of equations *)
  datatype Equation = EQ_IF of Exp                   (* Conditional expression
*)
                                 * EquationItem list     (* true branch *)
                                 * (Exp * EquationItem list) list (* elseif
branches *)
                                 * EquationItem list     (* else branch *)
                                | EQ_EQUALS of Exp * Exp          (* Standard
2-side eqn*)
                                | EQ_CONNECT of ComponentRef * ComponentRef (*
Connect stmt *)
                                | EQ_FOR of Ident * Exp * EquationItem list (*
For-loops *)
                                | EQ_WHEN_E of Exp  (* Condition *)
                                 * EquationItem list (* Loop body *)
                                 * (Exp * EquationItem list) list (* else when
*)
                                | EQ_NORETCALL of Ident * FunctionArgs (*
fcalls without return value *)
                            (* RML goals *)
                                | EQ_LET of Pattern * Exp    (* let pat = exp *)
                                | EQ_STRUCTEQUAL of Ident * Exp    (* ident =
exp *)
                                | EQ_FAILURE of Equation list    (* not goal or
not (g1 & g2 & g3) *)
                                | EQ_CALL of Path *          (* the name of the
function to call, ex: eval Absyn.dump etc *)
                                      FunctionArgs *  (* parameters *)
                                 Pattern         (* result pattern *)

  (** The `Algorithm' type describes one algorithm statement in an *)
  (** algorithm section.  It does not describe a whole algorithm.  The *)
  (** reason this type is named like this is that the name of the *)
  (** grammar rule for algorithm statements is `algorithm'. *)
  datatype Algorithm = ALG_ASSIGN of ComponentRef * Exp
                                 | ALG_TUPLE_ASSIGN of Exp (*tuple*)
                                   * Exp (* value*)
                                 | ALG_IF of Exp
                                     * AlgorithmItem list     (* true branch
*)
                                     * (Exp * AlgorithmItem list) list (*
elseif *)
                                     * AlgorithmItem list     (* else branch
*)
                                 | ALG_FOR of Ident * Exp * AlgorithmItem list
                                 | ALG_WHILE of Exp * AlgorithmItem list
                                 | ALG_WHEN_A of Exp
                                   * AlgorithmItem list
                                   * (Exp * AlgorithmItem list) list (* else-
when *)
                                 | ALG_NORETCALL of ComponentRef * FunctionArgs
(* general fcalls without return value *)
                                       | ALG_MATCH of ComponentRef list * (*
option result := match ... end match *)
                                    Exp * (* match expression of *)
                                    ElementItem list *(* local decls *)
                                    Case list (* case list + else in the end
with pat = [] *)
                      | ALG_SIMPLEMATCH of EquationItem list

  datatype Case = CASE of Pattern list * (* patterns to be matched *)
                                   ElementItem list * (* local decls *)
                                   ClassPart * (* equations [] for no equa-
tions: axioms /change to Equations*)
                                   Exp *(* to result *)
```

```
                                        Comment option (*the comment*)

  (** Modelica+ Patterns **)
  datatype Pattern = MWILDpat  (* from RMLPAT_WILDCARD *)
                             | MLITpat of Exp   (* from RMLPAT_LITERAL of RML-
Literal *)
                             | MCONpat of Path  (* from RMLLONGID of Ident *
Ident *)
                             | MSTRUCTpat of Path option * Pattern list (*
from RMLPAT_STRUCT of RMLIdent option * RMLPattern list *)
                             | MBINDpat of Ident * Pattern (* from RMLPAT_AS
of Ident * RMLPattern *)
                             | MIDENTpat of Ident * Pattern  (* from
RMLPAT_IDENT Ident *)



  (** - Modifications *)
  (** Modifications are described by the `Modification' type.  There *)
  (** are two forms of modifications: redeclarations and component *)
  (** modifications. *)
  datatype Modification = CLASSMOD of ElementArg list * Exp option

  (* Wrapper for things that modify elements, modifications and redeclarations
*)
  datatype ElementArg = MODIFICATION of bool * Each * ComponentRef * Modifica-
tion option * string option
                                  | REDECLARATION of bool * Each * ElementSpec
* ConstrainClass option


  (** - Each attribute *)
  (** The each keyword can be present in both MODIFICATION's and REDECLARA-
TION's. *)
  datatype Each = EACH | NON_EACH

  (** - Component attributes *)
  datatype ElementAttributes = ATTR of bool(* flow *)
                                       * Variability (* parameter, constant
etc. *)
                                       * Direction
                                     * ArrayDim  (*1.1*)

  (* Dete *)
  datatype Variability = VAR | DISCRETE | PARAM | CONST


  datatype Direction = INPUT | OUTPUT | BIDIR
  (** Component attributes are *)
  (** properties of components which are applied by type prefixes. *)
  (** As an example, declaring a component as `input Real x;' will *)
  (** give the attributes `ATTR([],false,VAR,INPUT)'. *)

  (** - Array dimensions *)
  type ArrayDim = Subscript list
  (** Components in Modelica can be scalar or arrays with one or more *)
  (** dimensions. This datatype is used to indicate the dimensionality *)
  (** of a component or a type definition. *)

  (** - Expressions *)

  datatype Exp = INTEGER of int
                               | REAL of real
                               | CREF of ComponentRef
                               | STRING of string
                               | BOOL of bool
                               | BINARY of Exp * Operator * Exp (* Binary
operations, e.g. a*b *)
                               | UNARY of Operator * Exp (* Unary operations,
e.g. -(x) *)
                               | LBINARY of Exp * Operator * Exp (* Logical
binary operations: and, or *)
                               | LUNARY of Operator * Exp (* Logical unary
operations: not *)
                               | RELATION of Exp * Operator * Exp (* Rela-
tions, e.g. a >= 0 *)
                               | IFEXP of Exp * Exp * Exp * (Exp * Exp) list
(* If expressions *)
                               | CALL of ComponentRef * FunctionArgs (* Func-
tion calls *)
                               | ARRAY of Exp list (* ARRAY consists of an
vector of the dimension sizes and an vector with the data.*)
                               | MATRIX of Exp list list
                               | RANGE of Exp * Exp option * Exp (* Range
expressions, e.g. 1:10 or 1:0.5:10 *)
                               | TUPLE of Exp list (*PR.*) (* Tuples used in
function calls returning several values *)
                               | END (* array access operator for last ele-
ment, e.g. a[end]:=1; *)
                               | CODE of Code (* Modelica AST Code construc-
tors *)
                               | RMLCALL of RMLIdent * Exp list
                 | RMLCONS of Exp * Exp
                 | RMLNIL
                 | RMLLIST of Exp list (*addedfor []*)
                 | RMLLIT of RMLLiteral (* FIXED *)
                               | RML_REFERENCE of RMLIdent
                 | MSTRUCTURAL of Path option * Exp list (* returned from match
exps *)
  (** The `Exp' datatype is the container of a Modelica expression. *)

  datatype Code = C_TYPENAME of Path
                           | C_VARIABLENAME of ComponentRef
                           | C_EQUATIONSECTION of bool * EquationItem list
                           | C_ALGORITHMSECTION of bool * AlgorithmItem list
                           | C_ELEMENT of Element
                           | C_EXPRESSION of Exp
                           | C_MODIFICATION of Modification
  (** The 'Code' datatype is used for Meta-programming. It orgiginates from the
Code quotation. *)

  datatype FunctionArgs =  FUNCTIONARGS of Exp list * NamedArg list
                             | FOR_ITER_FARG of Exp * Ident * Exp
  (** The `FunctionArgs' datatype consists of a list of positional arguments *)
  (** followed by a list of named arguments (Modelica v2.0) *)
  datatype NamedArg = NAMEDARG of Ident * Exp
  (** The `NamedArg' datatype consist of an Identifier for the argument and an
expression *)
  (** giving the value of the argument *)

  datatype Operator = ADD   | SUB    | MUL     | DIV       | POW
                                | UPLUS | UMINUS
                    | RADD   | RSUB    | RMUL     | RDIV
                                | RUPLUS | RUMINUS
                                | AND   | OR
                                | NOT
                                | LESS  | LESSEQ | GREATER | GREATEREQ | EQUAL
| NEQUAL
                                | RLESS  | RLESSEQ | RGREATER | RGREATEREQ |
REQUAL | RNEQUAL

  (** - Subscripts *)
  datatype Subscript = NOSUB
                                 | SUBSCRIPT of Exp
  (** The `Subscript' datatype is used both in array declarations and *)
  (** component references.  This might seem strange, but it is *)
  (** inherited from the grammar.  The `NOSUB' constructor means that *)
  (** the dimension size is undefined when used in a declaration, and *)
  (** when it is used in a component reference it means a slice of the *)
  (** whole dimension. *)

  (** - Component references and paths *)
  datatype ComponentRef = CREF_QUAL of Ident * (Subscript list) * ComponentRef
                                | CREF_IDENT of Ident * (Subscript list)

  datatype Path = QUALIFIED of Ident * Path
                                | IDENT of Ident
  (** A component reference is the fully or partially qualified name of *)
  (** a component.  It is represented as a list of *)
  (** identifier--subscript pairs.  The type `Path', on the other hand, *)
  (** is used to store references to class names, or names inside *)
  (** class definitions. *)

  (** - Restrictions *)
  datatype Restriction = R_CLASS
                                   | R_MODEL
                                   | R_RECORD
                                   | R_BLOCK
                                   | R_CONNECTOR
                                   | R_TYPE
                                   | R_PACKAGE
                                   | R_FUNCTION
                                   | R_ENUMERATION
                                   | R_PREDEFINED_INT
                                   | R_PREDEFINED_REAL
                                   | R_PREDEFINED_STRING
                                   | R_PREDEFINED_BOOL
                                   | R_PREDEFINED_ENUM
                        | R_UNIONTYPE
                       (* | R_TYVAR *)
  (** These constructors each correspond to a different kind of class *)
  (** declaration in Modelica, except the last four, which are used *)
  (** for the predefined types.  The parser assigns each class *)
  (** declaration one of the restrictions, and the actual class *)
  (** definition is checked for conformance during translation.  The *)
  (** predefined types are created in the `Builtin' module and are *)
  (** assigned special restrictions. *)

  (** Annotation *)
  datatype Annotation = ANNOTATION of ElementArg list
  (** An Annotation is a class_modification. *)

  (** Comment *)
  datatype Comment = COMMENT of Annotation option
                              * string option

  (* ExternalDecl *)
  datatype ExternalDecl = EXTERNALDECL of
                                Ident option  * (* The name of the external
function *)
                                string option * (* Lanugage of the external
function *)
                                ComponentRef option * (* ouput parameter as
return value*)
                                Exp list (* only positional arguments, i.e.
expression list*)


  (* RML Stuff - work in progress *)
  datatype RMLDatatype = DATATYPE of RMLType list * RMLIdent * DTMember list

  datatype RMLDecl = RELATION_INTERFACE of RMLIdent * RMLType (*changed*)
                             | DATATYPEDECL of RMLDatatype * string list
                             | TYPE of RMLIdent * RMLType * string list
                             | WITH of string * string list
                   | VALINTERFACE of RMLIdent * RMLType * string list
                   | VALDEF of RMLIdent * Exp * string list
                  | RELATION_DEFINITION of RMLIdent * RMLType option * RMLRule
list * string list
                   | RMLDECLCOMMENT of string

  datatype RMLComment = RMLCOMMENT of string (*use instead of string ?*)

  type RMLInterface = RMLDecl
  type RMLDefs      = RMLDecl

  datatype RMLSignature = CALLSIGN of RMLType list * RMLType list (*changed*)

  datatype RMLType = RMLTYPE_INT
                   | RMLTYPE_STRING
                   | RMLTYPE_REAL
                   | RMLTYPE_TYCONS of RMLType list * RMLIdent (* added *)
                   | RMLTYPE_SIGNATURE of RMLSignature (* change*)
                   | RMLTYPE_TUPLE of RMLType list
                   | RMLTYPE_TYVAR of RMLIdent
                   | RMLTYPE_USERDEFINED of RMLIdent (*could use tycons *)

  datatype RMLRule = RMLRULE of RMLIdent *
                              RMLPattern * (* changed *)
```

```
                                RMLGoal option*
                                RMLResult *
                                string list

  datatype RMLResult = RMLNoResult of string list
                     | RMLResultExp of Exp list * string list(*should be exp*)
                     | RMLResultFail of string list

  datatype RMLGoal = RMLGOAL_NOT of RMLGoal
                              | RMLGOAL_AND of RMLGoal * RMLGoal
                   | RMLGOAL_PAT of RMLPattern
                              | RMLGOAL_LET of RMLPattern * Exp * string list
                   | RMLGOAL_EQUAL of RMLIdent * Exp * string list
                 | RMLGOAL_RELATION of RMLIdent * Exp list * RMLPattern option
* string list(*added option*)

  datatype RMLPattern = (*RMLPAT_CALL of RMLIdent * RMLPattern list
                      |*) RMLPAT_WILDCARD
                      | RMLPAT_LITERAL of RMLLiteral
                      | RMLPAT_IDENT of RMLIdent
                      | RMLPAT_AS of RMLIdent * RMLPattern
                      | RMLPAT_CONS of RMLPattern * RMLPattern
                      | RMLPAT_STRUCT of RMLIdent option * RMLPattern list
                      | RMLPAT_NIL
                      | RMLPAT_LIST of RMLPattern list (*added for []-lists *)

  datatype RMLIdent   = RMLSHORTID of Ident * Info
                      | RMLLONGID of Ident * Ident

  datatype RMLLiteral = RMLLIT_INTEGER of int
                      | RMLLIT_STRING of string
                      | RMLLIT_REAL of real
                      | RMLLIT_CHAR of int


  datatype DTMember = DTCONS of RMLIdent * RMLType list * string list list

  (* start line/column end line/column *)
  datatype RMLDbRange = RMLDB_RANGE of int * int * int * int

  datatype RMLDbElement = RMLDB_VAR of string * (* filename *)
                                   RMLIdent * (* var name *)
                              RMLDbRange * (* actual position *)
                                   RMLDbRange * (* scope *)
                                   RMLIdent     * (* relation name *)
                                   RMLType      (* type *)
                        | RMLDB_REL of string * (* filename *)
                                       RMLIdent *      (* relation name *)
                                      RMLDbRange *  (* relation ident position
*)
                                       RMLType      (* relation type *)
                        | RMLDB_TY  of string * (* filename *)
                                       RMLIdent * (* type name *)
                                       RMLDbRange    (* type position *)
                        | RMLDB_CTOR of string * (* filename *)
                                        RMLIdent *   (* constructor name *)
                                        RMLDbRange * (* position *)
                                        RMLType    (* type *)
  datatype RMLDb = RMLDB of RMLDbElement list
end
```