Extending Modelica with High-Level Data Structures: Design and Implementation in OpenModelica

Simon Björklén

Reg Nr: LIU-IDA/LITH-EX-A--08/030--SE
Linköping 2008
Master’s Thesis

Extending Modelica with High-Level Data Structures: Design and Implementation in OpenModelica

Simon Björklén

Reg Nr: LIU-IDA/LITH-EX-A--08/030--SE
Linköping 2008

Supervisor: Adrian Pop
IDA, Linköpings universitet

Examiner: Peter Fritzson
IDA, Linköpings universitet

Department of Computer and Information Science
Linköpings universitet
SE-581 93 Linköping, Sweden
# Extending Modelica with High-Level Data Structures: Design and Implementation in OpenModelica

**Författare** Simon Björklén

**Abstract**

Modelica is an equation-based object-oriented language (E OO). PELAB at Linköping University along with the OpenModelica development group, is developing a meta-modeling extension, MetaModelica, to this language along with a compiler called the OpenModelica Compiler (OMC).

The goal of this thesis was to analyze the compiler, extend it with union type support and then write a report about the extension with union types in particular and extension with high level data structures in general, to facilitate further development. The implementation made by this thesis was implemented with the goal of keeping the current structure intact and extending case-clauses where possible.

The main parts of the extension is implemented by this thesis work but some parts concerning the pattern matching algorithms are still to be extended. The main goal of this is to bootstrap the OpenModelica Compiler, making it able to compile itself although this is still a goal for the future.

With this thesis I also introduce some guidelines for implementing a new high-level data structure into the compiler and which modules needs extension.
Abstract

Modelica is an equation-based object-oriented language (Eoo). PELAB at Linköping University along with the OpenModelica development group, is developing a meta-modeling extension, MetaModelica, to this language along with a compiler called the OpenModelica Compiler (OMC).

The goal of this thesis was to analyze the compiler, extend it with union type support and then write a report about the extension with union types in particular and extension with high level data structures in general, to facilitate further development. The implementation made by this thesis was implemented with the goal of keeping the current structure intact and extending case-clauses where possible.

The main parts of the extension is implemented by this thesis work but some parts concerning the pattern matching algorithms are still to be extended. The main goal of this is to bootstrap the OpenModelica Compiler, making it able to compile itself although this is still a goal for the future.

With this thesis I also introduce some guidelines for implementing a new high-level data structure into the compiler and which modules needs extension.
Acknowledgments

I would like to thank Peter Fritzson for making this thesis work possible, Adrian Pop for support and help during the whole process and Kristian Staväker for all his help, helping me getting started, solving all sorts of problems and pointing me in the right direction when I got stuck. Without you this thesis would not had been finished!
Contents

1 Introduction 1
  1.1 Intended Audience .............................................. 1
  1.2 Thesis Contributions ........................................... 1
  1.3 Thesis Outline .................................................. 2
  1.4 Goal ........................................................................ 2
    1.4.1 Learning ......................................................... 2
    1.4.2 Analysis .......................................................... 2
    1.4.3 Design and Implementation .................................... 3

2 Background 5
  2.1 Modelica ............................................................. 5
  2.2 RML ....................................................................... 7
    2.2.1 Where to Get the RML-Compiler? ........................... 8
  2.3 MetaModelica .......................................................... 9
    2.3.1 Union Type ....................................................... 9
    2.3.2 Match-Expressions ............................................. 11
  2.4 OpenModelica Compiler ............................................. 12
    2.4.1 Environment Structure .......................................... 12
    2.4.2 Translation Stages .............................................. 13
    2.4.3 An Overview of the Compiler .................................. 14
    2.4.4 A Simplified Version of the Main Function of the Compiler 14
    2.4.5 Parse .............................................................. 14
    2.4.6 SCode ............................................................. 14
    2.4.7 Inst ............................................................... 14
    2.4.8 Lookup ............................................................ 15
    2.4.9 RTOpts ........................................................... 15
    2.4.10 Codegen ......................................................... 15
    2.4.11 DAELow and SimCodeGen .................................... 15
    2.4.12 C-Runtime ...................................................... 15
    2.4.13 Where to Get the Most Current Version? .................. 16
  2.5 MDT ................................................................. 17
    2.5.1 Where to Get the Modelica Development Tooling? ....... 17
List of Code Examples

2.1 A Trivial Modelica Program ............................................. 6
2.2 A Small Expression Evaluator ........................................... 8
2.3 The Eval Relation .......................................................... 8
2.4 A Factorial Program Written in RML ................................ 9
2.5 A Union Type Declaration ............................................... 9
2.6 A Union Type Definition Along With Its Matchcontinue Elaboration Function ......................................................... 10
2.7 A Simplified Version of the Main Function of the Compiler ... 14
2.8 Some of the Available Data Structure Constructors in the C-Runtime ................................................................. 16
3.1 Introduction of the Absyn.R_METARECORD ......................... 22
3.2 Code Added to SCode ....................................................... 24
3.3 AST Union Type Example .................................................. 25
3.4 Code Extensions to Inst ................................................... 27
3.5 Code Extensions to Lookup ............................................... 28
3.6 Example of METARECORDCALL Creation ............................ 29
3.7 Code Extensions to Static ............................................... 30
3.8 Extensions to Types ....................................................... 31
3.9 Extensions to the Prefix Module ....................................... 32
3.10 Code Extensions to the DAE Module .................................. 33
3.11 Code Extensions to the Exp Module ................................... 34
3.16 The typedef Added to modelica.h .................................... 35
3.12 Generating the Variable Type String .................................. 36
3.13 generateSimpleType ..................................................... 37
3.14 generateExpression ..................................................... 38
3.15 Example of Generated Code ............................................ 39

List of Figures

2.1 The dependencies between RML, MetaModelica, Modelica and the OpenModelica Compiler .......................... 5
2.2 plot(x) on the HelloWorld Example ................................... 6
2.4 An Overview of the OpenModelica Environment Structure .... 12
2.5 The Different Translation Stages From Modelica Code to Simulation ......................................................... 13
2.6 MDT in Action ............................................................... 18
3.1 An Overview of the Union Type Implementation .................. 20
3.2 The New AST for Code 3.3 ............................................... 25
A.1 An Overview of the Compiler ........................................... 56
B.1 Extending the Compiler With a New High-Level Datastructure . 58
Chapter 1

Introduction

In this chapter I will give a brief introduction of this thesis, its prerequisites, outline and goals.

1.1 Intended Audience

To get the most out of this report about extending Modelica with high level data structures, the reader should preferably have some knowledge about the Modelica programming language and the basics about constructing a compiler. Some C/C++ terminology is also used in some parts of the thesis so some experience of C/C++ programming is also preferred.

1.2 Thesis Contributions

Current equation-based object-oriented languages (EOO) lack support for high-level data structures. Without such support the EOO languages need to rely on external tool support or advanced integrated graphical environments (which are only available commercially) to manipulate models or model data. We believe that extending EOO with high-level data structures and functional meta-programming constructs will highly increase the power of these languages and decrease their tight coupling and reliance on external tools and graphical environments. To validate this idea we have extended the Modelica language with support for high-level data structures. This thesis presents the design and the practical implementation of the proposed extension.
1.3 Thesis Outline

The following is a short outline of the thesis:

- **Chapter 2**, Background - starts off with some background information about Modelica, RML, MetaModelica, the OpenModelica Compiler and the Modelica Development Tooling.

- **Chapter 3**, Design and Implementation - contains a thorough description of the design and implementation of my OpenModelica extension for union types.

- **Chapter 4**, Guidelines for Implementing a New High-Level Datastructure - contains some guidelines for implementing a new High-Level Datastructure into the OpenModelica Compiler.

- **Chapter 5**, Discussion and Related Work - has a discussion about the implementation, how it was done, what could have been done differently and what is left for the future.

- **Chapter 6**, Conclusions - contains some conclusions about the thesis work and what has been accomplished.

1.4 Goal

The main goal of this thesis work was to extend the OpenModelica Compiler to be able to handle union types. However, to reach this goal some subgoals were necessary.

1.4.1 Learning

When starting the work of this thesis, the first thing necessary was to learn the MetaModelica programming language which was something completely new to me. I did a lot of programming exercises until I felt comfortable with the language. The *OpenModelica Users Guide* [11] along with *Modelica Meta-Programming and Symbolic Transformations* [9] gave me the essential information here. The next step was to obtain knowledge about the OpenModelica Compiler, how it was built, which modules were used where and to understand the flow of the compiler. The *OpenModelica System Documentation* [10] helped me a lot with this. Next some knowledge about the connection between OpenModelica, MetaModelica, Modelica, RML and the C-Runtime was acquired by analyzing the code using the debugger in MDT (see section 2.5 on page 17 for more information about MDT).

1.4.2 Analysis

The next step was to analyze the features that were going to be implemented. This was to understand where to make the extensions, and how to do it. This was actually quite a challenging part since the compiler consists of quite a few modules (almost 40) and a huge amount of code, making it hard to get “the whole picture”.

1.4 Goal

1.4.3 Design and Implementation

Choosing a design for the implementation was pretty much straight forward after the analysis step. The idea was to keep the structure of the compiler as much as possible and make the extension in form of case-clause additions. This worked out perfectly in most cases and the resulting implementation is separated in almost every module.
Chapter 2

Background

This chapter provides a short overview of the Modelica-programming language, the MetaModelica-programming language, the RML-programming language and the OpenModelica Compiler. To give a short overview of the dependency relation between RML, MetaModelica, Modelica and the OpenModelica Compiler (fig. 2.1.): Modelica is an EOO language, MetaModelica a meta extension to Modelica with its compiler written in RML and the OpenModelica Compiler is an open source compiler for Modelica (and a subset of MetaModelica) written in MetaModelica.

2.1 Modelica

Modelica [3] is a modeling language used for simulation of complex systems. What makes Modelica so powerful is the way you model systems. The entire system is modeled using equations, e.g. to model a physical model, just pass the mathematical equations, along with its start/stop values and then the simulation tools takes
Background

care of the rest. This also leads to some other nice features:

- Since the behavior of a model in Modelica is specified by acausal equations
  the models can be used in different contexts and it is the job of the compiler
  to deduce the control flow from the use of the model. This leads to increased
  reuse of the models.

**Code 2.1 A Trivial Modelica Program**

```plaintext
class HelloWorld "A simple equation"
  Real x(start=1);
end HelloWorld;
```

A trivial Modelica program: Code 2.1 creates a class named `HelloWorld` with
a variable `x`, which have the initial condition `x(0)=1`. The equation section then
lists the equations which have to be fulfilled. In this case, the derivative of `x`
has to be `-x`. To simulate this model in OpenModelica you just have to run:
`simulate(HelloWorld,stopTime=5)`, and to get a nice graph of how the value of
`x` varies over time simply type: `plot(x)`

The resulting graph is shown in fig. 2.2.

![Plot by OpenModelica](image)

**Figure 2.2.** `plot(x)` on the HelloWorld Example

However, the real power of Modelica is not shown until you use it to model
a complex system like the one in fig. 2.3, where each element in the schema
corresponds to one Modelica-class.

![Figure 2.3. A Schema of a GTX Gas Turbine Power Cutoff Mechanism Modeled in MathModelica by MathCore [2]](image)

### 2.2 RML

Natural Semantics\(^1\) are quite popular among language designers when specifying new languages. The problem however, is that the translation from Natural Semantics form into executable code is quite exhausting, demanding, and has to be made manually. This is what Mikael Pettersson realized back in 1995 and he specified RML \([14, 13]\), *Relational Meta-Language*, to allow easy translation from Natural Semantics into efficient code.

Some features of RML:

- RML uses features from languages like SML (Standard ML) to allow strong typing and datatype constructors.
- The RML compiler generates efficient C-Code which is compiled by an ordinary C compiler, e.g. generating an executable Modelica translator.

---

\(^1\)A popularized style of specification developed by Gilles Kahn’s group at INRIA [14]
Code 2.2 shows a small expression evaluator, which 'converts' a mathematical expression into the value of the expression.

### Code 2.2 A Small Expression Evaluator

```plaintext
datatype Exp = NUMBER of real
| ADD of Exp * Exp
| MUL of Exp * Exp
```

E.g. the expression

```
3*(4+5)
```

gets translated into the following tree:

```
MUL(NUMBER(3), ADD(NUMBER(4), NUMBER(5)))
```

After this, we describe the eval relation of the expression. This relation relates expressions to their evaluated values (see Code 2.3)

### Code 2.3 The Eval Relation

```plaintext
relation eval =

axiom  eval(NUMBER(x)) => x

rule  eval(x) => x' & eval(y) => y' &
      real_add(x',y') => sum

rule  eval(ADD(x,y)) => sum

rule  eval(x) => x' & eval(y) => y' &
      real_mul(x',y') => prod

end
```

Even though RML is mainly meant to be used as a specification language for other languages, it is also possible to use it as a functional language which is shown below. Code 2.4 shows a factorial program written in RML.

For more information about RML and its syntax, take a look at *Developing efficient language implementations from structural and natural semantics* [8] by Peter Fritzson and *Compiling Natural Semantics* [14] by Mikael Pettersson.

#### 2.2.1 Where to Get the RML-Compiler?

The most current version of the source code is always available from the RML homepage:

http://www.ida.liu.se/~pelab/rml/
2.3 MetaModelica

To be able to express the whole Modelica-language in Modelica (e.g. writing a Modelica compiler in Modelica) some extension of the language was required. PELAB made this extension with definitions written in RML and the result is known as MetaModelica. Some of the necessary extensions were lists, union types and match-expressions. [15]

2.3.1 Union Type

Declaring a variable as a union type is like declaring a holder variable [6]. The type of the holder variable is a specification of a datastructure. This specification consists of a description of a set of values from which only one is active during execution.

Another way to see this is to think of the union type variable as a base class and all the types it can hold as classes extended from this base class.

Union types are widely used in the MetaModelica language.

Code 2.5 shows a union type declaration and Code 2.6 a union type in use.

Code 2.5 A Union Type Declaration

1. `uniontype UT
2. record REC1
3.    Integer int1;
4.    Real re1;
5. end REC1;
6. record REC2
7.    Integer int2;
8.    Integer int3;
9.    Integer int4;
10. end REC2;
11. end UT;`
Code 2.6 A Union Type Definition Along With Its Matchcontinue Elaboration Function

```plaintext
uniontype EXP
    record NUMBER
        Integer n;
    end NUMBER
    record ADD
        EXP e1;
        EXP e2;
    end ADD;
    record MUL
        EXP e1;
        EXP e2;
    end MUL;
end EXP;

function elabExp;
    input EXP e;
    output Integer out;
algorithm
    out := matchcontinue(e)
    local
        Integer n1, n2;
        EXP e1, e2;
    case (NUMBER(n1)) then n1;
    case (ADD(e1, e2))
        equation
            n1 = elabExp(e1);
            n2 = elabExp(e2);
        then n1 + n2;
    case (MULT(e1, e2))
        equation
            n1 = elabExp(e1);
            n2 = elabExp(e2);
        then n1 * n2;
    end matchcontinue
end elabExp;
```
2.3.2 Match-Expressions

Another important part of the MetaModelica extension is the support of match-continue expressions. A matchcontinue expression is similar to switch-statements in C/C++ with some essential differences. A statement in Modelica can either return true, false or it may fail where a failure means the statement has failed.

A matchcontinue expression works as follows. First it switches over the case-clauses until it finds the first matching case-expression. Then it executes the statements within that case-clause until it reaches the then-statement. The interesting part happens when a statement within a case fails. What happens in such a case is that variables previously bounded to values within cases that failed gets roll-backed (reversed) and the search for another matching case-clause proceeds. Code 2.6 shows an example of a matchcontinue expression when used to elaborate an expression.
2.4 OpenModelica Compiler

The OpenModelica Compiler [10] (OMC) is a compiler for Modelica developed at PELAB (IDA, LiU). Most of the compiler is written in MetaModelica except external functions which are written in C-code. Most of the standard implementation of Modelica is already handled by the compiler and the compiler is currently being extended to handle the MetaModelica extensions as well. That is, adding support for lists, union types and match-expressions.

In this section a brief explanation of the different parts of the compiler will be given.

2.4.1 Environment Structure

The OpenModelica environment consists of several connected subsystems, shown in fig. 2.4.

![Figure 2.4. An Overview of the OpenModelica Environment Structure](image)

A brief explanation of some of the subsystems:

- **Eclipse Plugin Editor/Browser** - Used as an editor and debugger for Modelica/MetaModelica development.
- **Emacs Editor/Browser** - Used as an editor for Modelica/MetaModelica development. - No longer supported.
- **Interactive session handler** - Handles the loading, simulation and plotting of a Modelica model.
- **Modelica Compiler** - Handles the compilation of Modelica-code to C-code.
- **Modelica Debugger** - Handles the debugging of Modelica/MetaModelica code.
2.4 OpenModelica Compiler

- **Graphical Model Editor/Browser** - A graphical model editor, not really a part of OpenModelica but it is integrated into the system and is available from MathCore Engineering AB [2] without cost for academic use.

The subsystems also includes tools for building simulation executables linked with a given equation-solver, e.g. numerical ODE (Ordinary Differential Equation) or DAE (Differential Algebraic Equation) solvers. The default solver is currently DASSL, a differential algebraic equation solver developed by Linda Petzhold in 1991 [5].

### 2.4.2 Translation Stages

As seen in fig. 2.5 the flow from Modelica until simulation is as follow. First the Modelica source code-file is fed to the compiler and is, in the first stage, translated into a flat model. This stage includes type-checking, import statements, performing object-oriented operations (such as inheritance, modifications etc.) and lookup statements. The flat model includes a set of equations with its declarations and functions. This stage is a partial instantiation of the model, called the elaboration or instantiation stage. The next stages, the Analyzer-stage and the Optimizer-stage, are necessary for analyzing and optimizing models with equations. The Code Generator stage generates the C-code for the model which is then compiled with an ordinary C-compiler to produce executable code.

![Figure 2.5. The Different Translation Stages From Modelica Code to Simulation](image-url)
2.4.3 An Overview of the Compiler

The OpenModelica Compiler consists of almost 40 different modules. To understand the flow between the modules, take a look at fig. A.1. Note that the flow is not 100% accurate since some functions are used from many modules (e.g. the functions in the Dump module), and to show the flow of them all would just be confusing.

2.4.4 A Simplified Version of the Main Function of the Compiler

To get an idea of how the flow is controlled from the code, Code 2.7 shows a simplified version of the main function of the compiler.

Code 2.7 A Simplified Version of the Main Function of the Compiler

```plaintext
function main
  input String f; // file name
algorithm
  ast := Parser.parse(f);
  scode1 := SCode.elaborate(ast);
  scode2 := Inst.elaborate(scode1);
  dae1 := Inst.instantiate(scode2);
  dae2 := optimizeDae(dae1, scode2);
  simcodegen(dae2, scode2);
end main;
```

2.4.5 Parse

This module is actually written in C and uses ANTLR [1] (ANother Tool for Language Recognition) to parse the modelica code and to construct an AST (Abstract Syntax Tree) with definitions from the Absyn module. This module is not really a part of the OpenModelica developed compiler (external function call) but is an important part which makes out the base for all the next compilation stages.

2.4.6 SCode

This module converts the Absyn AST into SCode which is an intermediate form. This form is a little more simplified than the AST and groups together important information into nodes.

2.4.7 Inst

This is the most complex module. It takes care of instantiation of the code, type checking, looking up names in the environment and builds new environments. As can be seen in fig. A.1 it calls many different modules and is an important part of
the front-end. The output from this module is another intermediate form called DAE. The purpose of introducing this form is to have a form which holds all the necessary information which is needed for the code generation.

2.4.8 Lookup
This module is responsible for the lookup mechanism. That is looking up classes, types, variables etc. in the environment by following the lookup rules.

2.4.9 RTOpts
This module handles different options passed to the compiler. The most frequently used function is the acceptMetaModelicaGrammar() which checks if the MetaModelica extension is allowed during compile time. Turning this flag off turns off most of the Meta extension case-clauses.

2.4.10 Codegen
This module takes ordinary Modelica functions (transformed into DAE form), and generates C-code.

2.4.11 DAELow and SimCodeGen
Generates code for solving equations and makes it possible to run a simulation on everything. This part is a very important part of the compiler since it makes the equation-based language possible and this is what makes the Modelica language so powerful.

2.4.12 C-Runtime
In order to compile the generated C-Code some external C-modules exists to specify the representation of the Modelica datatypes, reading/writing to files and to handle the simulation environment. The simulation environment also contains an automatic garbage-collector which constantly frees unused memory during simulation. Something worth mentioning are the special data struct definitions which are necessary to make the garbage-collector handle the memory correctly (at least this is the goal in the future. At the moment there are some memory leaks when using \texttt{mk\_box} functions.). Code 2.8 shows some of the available data structure constructors.

For more information about the \texttt{mk\_box} constructors take a look at section 5.3 on page 48 where the generated code along with the \texttt{mk\_box} constructor is discussed a little further.
Code 2.8 Some of the Available Data Structure Constructors in the C-Runtime

1. `extern void *mmc_mk_nil(void); //An empty data structure`
2. `extern void *mmc_mk_cons(void *car, void *cdr); //Cons 2 data structures`
3. `extern void *mmc_mk_box2(unsigned ctor, void *x0, void *x1); //Holder for 1 index + 2 data structures`
4. `extern void *mmc_mk_icon(int i); //Holder for Integer`
5. `extern void *mmc_mk_rcon(double d); //Holder for a Real`
6. `extern void *mmc_mk_scon(char *s); //Holder for a String`

2.4.13 Where to Get the Most Current Version?

The most current version of the source code is always available from the OpenModelica homepage:

http://www.openmodelica.org
2.5 MDT

MDT, or Modelica Development Tooling is an Eclipse-plugin developed by two master students a couple of years ago [12] . The goal of their project was to integrate the Modelica development environment into Eclipse to allow visual views of Modelica models and to create a less complex environment for debugging. After the end of their thesis work, MDT has been extensively extended by Adrian Pop at PELAB (IDA, LiU).

Some features of MDT are:

- Code completion
- Parenthesis matching
- Code coloring
- Automatic code indentation
- Visual browsing for Modelica projects, packages and classes
- Syntax checking

2.5.1 Where to Get the Modelica Development Tooling?

This tool is highly recommended for OpenModelica development and is available without cost at:

http://www.ida.liu.se/~pelab/modelica/OpenModelica/MDT/

Fig. 2.6 shows MDT in action.
Figure 2.6. MDT in Action
One of the most challenging parts about this thesis work was the analysis of the existing code, to understand how to modify the code and where to do it. Since I did not want to modify the existing code, unless I really had to, I tried to extend it by adding more matchcontinue case-clauses where possible. By doing it this way, the code for the union type implementation was separated as much as possible from the existing code which will make it easier to extend the compiler further later on.

In this chapter I will explain the changes I made to all the different modules prior to making the compiler generate the necessary code for handling the union type variables in a simulation environment. For comments about implementation of other data structures take a look at the next chapter.
3.1 A Short Overview of the Extended Modules

Fig. 3.1 shows an overview of the union type implementation and below is a list of the extended modules with a brief explanation:

- **Parse** - extend the parser to handle the union type keyword (external using ANTLR).
- **Absyn** - add the necessary tree nodes for `R_UNIONTYPE` and `R_METARECORD`.
- **SCode** - add the necessary intermediate nodes for `R_UNIONTYPE` and `R_METARECORD` and extend the AST for union types to contain `METARECORDs` as well.
- **Inst** - handle variable declarations of union types.
- **Lookup** - handle variable lookups of union types.
- **Static** - handle `METARECORDs` as function calls.
- **Types** - code for typechecking union types.
- **Prefix** - add prefixes to expressions containing `METARECORDCALLs`.
- **Patternm** - handle union types as input to matchcontinue expressions.
3.1 A Short Overview of the Extended Modules

- **DAE** - add the necessary DAE nodes for `UNIONTYPEs` and `METARECORDs`.
- **Exp** - add the necessary expression node to handle `METARECORDCALLs`.
- **Codegen** - generate C-code.
- **MetaUtil** - helper functions explicit for Modelica extensions.
3.2 The Front-End

The following modules are a part of the front-end of the compiler [7], that is the analysis part of the compiler which creates the intermediate code to be used as a base for the back-end.

3.2.1 Parser

The parsing process is done with ANTLR [1] so I will not go into any detail about this. For more information about this process take a look at the source code in

\<omc-root\>/Compiler/absyn_builder/
\<omc-root\>/modelica_parser

and the ANTLR documentation.

http://www.antlr2.org/doc/index.html

3.2.2 Absyn

To be able to reuse the existing code for looking up names etc. I Introduced a new restriction type called R_METARECORD (see Code 3.1). This restriction is a totally internal one and is defined as a record which holds the name of the union type as a path and an integer which holds the index of the record in the union type. This index number is crucial for pattern matching and fetching data out of a union type later on.

Note: From now on, when I talk about METARECORDs, I refer to the records within a union type.

Code 3.1 Introduction of the Absyn.R_METARECORD

```plaintext
1 uniontype Restriction
2 // ... Lots of restriction definitions ...
3
4 record R_UNIONTYPE
5 end R_UNIONTYPE;
6
7 record R_METARECORD
8 Path name; //Name of the uniontype
9 Integer index; //Index in the uniontype
10 end R_METARECORD;

1 // ... more restriction definitions
2 end Restriction;
```
3.2 The Front-End

3.2.3 SCode

The most important change to this module is the introduction of `R_METARECORD` and `R_UNIONTYPE`, and the changes made to the function `elaborate`. As can be seen in Code 3.2, `elabClass` now has a case clause which elaborates the class and executes the function `MetaUtil.createMetaClasses` on it before continuing. Some minor changes were also made to `elabRestriction` to allow the restrictions `R_METARECORD` and `R_UNIONTYPE` to exist.
Code 3.2 Code Added to SCode

```plaintext
uniontype Restriction
// ... Lots of restriction definitions ...

record R_METARECORD
    Absyn.Path name;  //Name of the union type
    Integer index;    //Index in the union type
end R_METARECORD;
record R_UNIONTYPE end R_UNIONTYPE;

// ... more restriction definitions ...
end Restriction;

public function elaborate *function: elaborate
This function takes an 'Absyn.Program' and constructs a 'Program' from it.
input Absyn.Program inProgram;
output Program outProgram;
algorithm
    outProgram:=
      matchcontinue (inProgram)
        local
          Class c_1;
          Program cs_1;
          Absyn.Class c;
          list <Absyn.Class> cs, cs2;
          Absyn.Within w;
          Absyn.Program p;
          case (Absyn.PROGRAM(classes = {})) then {}
          case (Absyn.PROGRAM(classes = (c :: cs), within_ = w))
            equation
              c_1 = elabClass (c);
            end
          end
          cs2 = MetaUtil.createMetaClasses(c); //Handle union types
          cs = listAppend(cs2, cs);
          cs_1 = elaborate(Absyn.PROGRAM(cs, w));
          then
            (c_1 :: cs_1);
          case (p)
            equation
              Debug.fprint("failtrace ", "elaborate failed\n");
              fail();
          end
        end matchcontinue;
end elaborate;
```
3.2 The Front-End

MetaUtil
MetaUtil.createMetaClasses

This function takes the AST tree and translates it to contain the METARECORDs. The code in Code 3.3 inserts the marked area into the resulting AST in fig. 3.2. This translation is done by traversing the AST tree, searching for union types and when finding one, copying the whole node to the same level as the union type and changing its restriction to R_METARECORD.

Code 3.3 AST Union Type Example

```plaintext
uniontype UT
  record REC1
    Integer y;
  end REC1;

  record REC2
    Integer z;
  end REC2;
end UT;
```

Figure 3.2. The New AST for Code 3.3
3.2.4 Inst

In this module most of the changes are case-extensions to the instElement function to handle R_UNIONTYPE elements. Since most of the added code are reused code for variable instantiation, and environment updates, I won't go into much detail of this.

The idea of this extension is as follows. First create a case clause to catch the declaration. Then make any necessary checks to make sure it matches the correct declarations. After that look up modifications, update the environment and instantiate the variable into the environment. Finally create the new type, add it to a new DAE and update the bindings and we are done. Code 3.4 shows the code for this extension.

Note that R_METARECORDs are not handled here since declaring a variable of a record in a UNIONTYPE is not allowed.
3.2 The Front-End

Code 3.4 Code Extensions to Inst

```java
public function instElement
    input Env.Cache inCache;
    input Env inEnv1;
    input Mod inMod2;
    input Prefix inPrefix3;
    input Connect.Sets inSets4;
    input ClassInf.State inState5;
    input tuple<SCode.Element, Mod> inTplSCodeElementMod6;
    input InstDims inInstDims7;
    input Boolean inBoolean8;
    output Env.Cache outCache;
    output list <DAE.Element> outDAEElementLst;
    output Env outEnv;
    output Connect.Sets outSets;
    output ClassInf.State outState;
    output list <Types.Var> outTypesVarLst;
algorithm
    (outCache, outDAEElementLst, outEnv, outSets, outState, outTypesVarLst):=
        matchcontinue /* All input variables */
        // ... Lots of case clauses to handle all types of declarations ...
        case (cache, env, mods, pre, csets, ci_state, comp)
            /* NOTE: Case shortened due to space limitations */
            equation
                true = RTOpts.acceptMetaModelicaGrammar();
                (cache, env_1) = getDerivedEnv(cache, env, bc);
                (cache, cl, cenv) = Lookup.lookupClass(cache, env_1, t, true);

                // Test if it has the UNIONTYPE restriction
                true = MetaUtil.classHasRestriction(cl, SCode.R_UNIONTYPE);
                allreadyDeclared = checkMultiplyDeclared( ... );

                // ... Code for looking up modifications, updating environment etc. ...
                // Instantiate the component
                (cache, compenv, dae, csets_1, ty) =
                    instVar( ... );
                dims_1 = instDimExpLst(dims, impl);
                cr = Prefix.prefixCref(pre, Exp.CREF_IDENT(n,{}));

                ty = MetaUtil.createUnionType(cl);
                dae = daeDeclare(cr, ClassInf.UNIONTYPE(n), ty, ... );

                // The environment is extended (updated) with the new variable binding.
                (cache, binding) = makeBinding(cache, env2_1, attr, mod_1, ty);

                // ... code for adding bindings to the dae ...
                then
                    (cache, dae, env_1, csets_1, ci_state, vars);

                // ... more case clauses ...
        end matchcontinue;
    end instElement;
```
### 3.2.5 Lookup

In this module, the only change necessary was adding a case-clause to the `lookupType` routine (see Code 3.5). This was to enable type lookups of METARECORDs.

#### Code 3.5 Code Extensions to Lookup

```plaintext
function lookupType
input Env.Cache inCache;
input Env.Env inEnv "environment to search in";
input Absyn.Path inPath "type to look for";
input Boolean inBoolean
output Env.Cache outCache;
output Types.Type outType "the found type";
output Env.Env outEnv "The environment the type was found in";
algorithm
(outCache ,outType ,outEnv):=
matchcontinue (inCache ,inEnv ,inPath ,inBoolean)
//Case−clauses

case (cache ,env ,path ,msg)
local String s ,ident;
Absyn.Path name;
Integer index;
equation
true = RTOpts.acceptMetaModelicaGrammar();
(cache ,( c as SCode.CLASS(id ,_, encflag ,
SCode.R_METARECORD(name ,index ,_)),env_1)
= lookupClass2(cache ,env ,path ,false);
ident = Absyn.pathLastIdent(path);
t = (Types.T_METARECORD(ident ,name ,index ),SOME(path ));
then
(cache ,t ,env);
end matchcontinue;
end lookupType;
```

### 3.2.6 Static

To be able to use a METARECORD as an assignment to a UNIONTYPE where the METARECORD is initialized with its containing variables (as a function call, see Code 3.6), an extension to `elabCallArgs` was necessary. Since I wanted to eas-
3.2 The Front-End

ily distinguish this call from an ordinary function call, I introduced a special \texttt{METARECORDCALL} for this (see Code 3.7).

To aid this process a helper function, \texttt{MetaUtil\_createFunctionArgsList}, was introduced. This function takes a \texttt{R\_METARECORD} class, parses it and returns a list of types of its containing types. That is the types of the variables in the \texttt{METARECORD}. For the complete source of this function take a look in the \texttt{MetaUtil} module.

\textbf{Code 3.6} Example of \texttt{METARECORDCALL} Creation

```
uniontype UT
record REC1
  Integer int1;
end REC1;
end UT;
model testModel
  Integer out;
  function testfunc
    input Integer i;
    output Integer o;
    UT univar;
  algorithm
    univar = REC1(32); /* Here is the, so called, METARECORDCALL created */
    o = i;
end testfunc;
equation
  out = testfunc(10);
end testModel;
```

3.2.7 Types

As can be seen in Code 3.8 the necessary changes are the definition of the new types, \texttt{T\_UNIONTYPE} and \texttt{T\_METARECORD}, and after that the type checking phase which is done in the \texttt{subtype} function. This function takes two types and checks if one of them is a subtype of the other. In the union type case there are two different possibilities here. The first case is when assigning a \texttt{METARECORD} to a union type variable. In this case check if the \texttt{METARECORD} really is a record of the union type. The second case is assigning a union type to another union type. In this case check all containing records and make sure they contain the same ones.
protected function elabCallArgs

input Env.Cache inCache;
input Env.Env inEnv;
input Absyn.Path inPath;
input list<Absyn.Exp> inAbsynExpLst;
input list<Absyn.NamedArg> inAbsynNamedArgLst;
input Boolean inBoolean;
input Option<...> inInteractiveInteractiveSymbolTableOption;
output Env.Cache outCache;
output Exp.Exp outExp;
output Types.Properties outProperties;

algorithm

(outputCache, outputExp, outputProperties) :=
matchcontinue /* input variables */
// ... Lots of case clauses

case (cache, env, fn, args, nargs, impl, st)
local
SCode.Class c;
SCode.Restriction re;
SCode.Ident id;
SCode.Path name;
integer index;
equation
true = RTOpts.acceptMetaModelicaGrammar();
(cache, t as (Types.T_METARECORD(id, name, index),_), env_1) =
Lookup.lookupType(cache, env, fn, false);
(cache, c, env) = Lookup.lookupClass2(cache, env, fn, false);
fargs = MetaUtil.createFunctionArgsList(c, cache, env);
slots = makeEmptySlots(fargs);
(cache, args_1, newslots, constlist) =
elabInputArgs(cache, env, args, nargs,
slots, true /*checkTypes*/ ,impl);
const = Util.listReduce(constlist, Types.constAnd);
tyconst = elabConsts(t, const);
prop = getProperties(t, tyconst);
(cache, newslots2) =
fillDefaultSlots(cache, newslots, c, env, impl);
args2 = expListFromSlots(newslots);
then
(cache, Exp.METARECORDCALL(fn, args2, name, index), prop);

/* ... more case-clauses ... */
case (cache, env, fn, args, nargs, impl, st)
equation
Debug.fprint("failtrace", "= elabCallArgs failed
n");
then
fail();
end matchcontinue;
end elabCallArgs;
3.2 The Front-End

Code 3.8 Extensions to Types

```java
public uniontype TType
   //... lots of type records ...

record T_UNIONTYPE
  list <String> records; //A list with record names.
end T_UNIONTYPE;

record T_METARECORD
  Absyn.Ident ident; //The name of the record
  Absyn.Path name; //The name of the uniontype
  Integer index; //The index in the uniontype
end T_METARECORD;

   //... more type records ...
end TType;

public function subtype
   input Type inType1;
   input Type inType2;
   output Boolean outBoolean;
algorithm
   outBoolean:=
   matchcontinue (inType1, inType2)
   //... lots of type checks ...
   case ((T_METARECORD(ident, name, index),_), (T_UNIONTYPE(slst),_))
      local
      list <String> slst;
      Absyn.Ident ident;
      Absyn.Path name;
      Integer index;
      equation
      true = MetaUtil.stringMember(ident, slst);
      then true;
   // <uniontype> = <metarecord>
   case ((T_METARECORD(ident, name, index),_), (T_UNIONTYPE(slst),_))
      local
      list <String> slst1, slst2;
      equation
      true = MetaUtil.equalLists(slst1, slst2);
      then true;
   //Types doesn’t match
   case (t1, t2) then false;
   end matchcontinue;
end subtype;
```
3.2.8 Prefix

In order to find the necessary prefix for the new Exp-type (a prefix is the name extension to e.g. a variable which makes the name fully qualified and will allow direct lookups), some small extensions were needed here. As can be seen in Code 3.9, this extension was made to the `prefixExp` function and just prefixes its own Exp list and returns the new Exp with all its Exps prefixed.

3.2.9 Pattern

The changes to this module are still to be made. See 5.4.1 for more information about this.

3.2.10 DAE

As seen in Code 3.10 the only extensions necessary in this module was the introduction of the new DAE types (UNIONTYPE and METARECORD).
3.2 The Front-End

Code 3.10 Code Extensions to the DAE Module

```plaintext
uniontype Type

    //... lots of type declarations ...

record UNIONTYPE end UNIONTYPE;
record METARECORD end METARECORD;

    //... more type declarations ...

end Type;
```

3.2.11 Exp

As seen in Code 3.11 another type declaration had to be made for the UNIONTYPE and a declaration for the new expression METARECORDCALL was made here.

3.2.12 ClassInf

This module basically creates a state machine to handle instantiation of all possible variable declarations, classes etc. This state-machine is not really used for MetaModelica extensions but states for handling METARECORDs and UNIONTYPES have been added for future development.
Code 3.11 Code Extensions to the Exp Module

```plaintext
public uniontype Type
//... Lots of type declarations...

record T_UNIONTYPE end T_UNIONTYPE;

//...more type declarations...
end Type;

public uniontype Exp
//... lots of expression declarations ...

/*
Holds a metarecord call
<metarecord>(<args>)
*/
record METARECORDCALL
Absyn.Path path;
list<Exp> args;
SCode.Path name; //Name of the union type
Integer index; //Index in the union type
end METARECORDCALL;

//... more expression declarations ...
end Exp;
```
3.3 The Back-End

The following modules makes up the back-end of the compiler [7], that is the final code generation part. Here the intermediate code from the front-end is used to create an output of some compilable code which on the other hand will be compiled into executable code (together with the C-Runtime).

3.3.1 Codegen

This is the module where all the Modelica code, which now are in the form of DAE-elements, gets translated into C-Code. Since most of the work is already done in the front-end modules, this translation is pretty much straight forward. Some translations from DAE into Exp elements are done in Code 3.12 and the most important translation, that is from METARECORDCALL into C are done in generateExpression (see code 3.14). The last function changed here is the generateSimpleType function which generates the type string for a simple type, such as a union type as a function input/output variable (see code 3.13).

Code 3.15 shows an example of generated code for some expressions.

3.3.2 C-Runtime

To be able to use the simulation environment with the newly implemented union type a type definition of the new type was needed to modelica.h in the C-Runtime. The name conventions of the modelica types are on the form modelica_<typename> and since the used mk_box<#> are handled as void pointers the typedef added to modelica.h can be seen in Code 2.8.

Code 3.16 The typedef Added to modelica.h

```c
typedef void* modelica_uniontype;
```
Code 3.12 Generating the Variable Type String

```plaintext
protected function daeExpType
    input DAE.Type inType;
    output Exp.Type outType;
algorithm
    outType:=
        matchcontinue (inType)
        case DAE.INT() then Exp.INT();
        case DAE.REAL() then Exp.REAL();
        case DAE.STRING() then Exp.STRING();
        case DAE.BOOL() then Exp.BOOL();
        case DAE.ENUM() then Exp.ENUM();
        case DAE.LIST() then Exp.T_LIST(Exp.OTHER());
        case DAE.UNIONTYPE() then Exp.T_UNIONTYPE();
        case _ then Exp.OTHER();
    end matchcontinue;
end daeExpType;

protected function expTypeStr
    input Exp.Type inType;
    input Boolean inBoolean;
    output String outString;
algorithm
    outString:=
        matchcontinue (inType , inBoolean)
        local
            Lib tstr , str ;
            Exp.Type t ;
        case (Exp.T_LIST(_),_)
            equation
                str = 'void *';
            then
                str ;
            // ... Lots of case clauses ...
        case (Exp.T_UNIONTYPE() ,_)
            equation
                str = 'modelica_uniontype *';
            then
                str ;
        end matchcontinue;
end expTypeStr;
```

protected function generateSimpleType
  input Types.Type inType;
  output String outString;
algorithm
  outString :=
  matchcontinue (inType)
  local
    Lib n_1,n_2,n,t_str;
    tuple<Types.TType, Option<Absyn.Path>> t_1,t,ty;
  case ((Types.T_INTEGER(varLstInt = _),_)) then "modelica_integer";
  case ((Types.T_REAL(varLstReal = _),_)) then "modelica_real";
  case ((Types.T_STRING(varLstString = _),_)) then "modelica_string";
  case ((Types.T_BOOL(varLstBool = _),_)) then "modelica_boolean";
  case ((Types.T_LIST(_),_)) then "void"; // MetaModelica list
    /* More code for handling complex types, arrays etc */
  case ((Types.T_UNIONTYPE(_,_))) then "modelica_uniontype";
end matchcontinue;
end generateSimpleType;
public function generateExpression

/* Function header and match−continue case−clauses */

case (Exp.METARECORDCALL(fn , elist , name , index ) , tnr , context)
  local
    Absyn.Path fn , name ;
    Integer index ;
  equation
    true = RTOpts.acceptMetaModelicaGrammar () ;
    (cfn1 , vars1 , tnr1 ) = generateExpressions (elist , tnr , context ) ;
    (decl , tvar , tnr1_1 ) = generateTempDecl ("modelica_uniontype ", tnr1 ) ;
    s = MetaUtil.listToBoxes (vars1 , elist , index ) ;
    cfn1_1 = cAddVariables (cfn1 , {decl}) ;
    ret_type = generateReturnType (fn) ;
    fn_name = generateFunctionName (fn) ;
    args_str = Util.stringDelimitList (vars1 , " , " ) ;
    stmt = Util.stringAppendList ( {tvar , " = " , s , " ; " } ) ;
    cfn = cAddStatements (cfn1_1 , {stmt}) ;
  then
    (cfn , tvar , tnr1_1 ) ;

/* More case−clauses */
end generateExpression ;
uniontype UT
  record REC1
    Integer i1;
  end REC1;
  record REC2
    Integer i2;
  end REC2;
end UT;

model Uniontype2
  Integer k;
  UT re;
  function test2
    input UT r;
    // function body
    end test2;
  function test
    input Integer s;
    output Integer k;
    algorithm
      re := REC1(66);
      re := REC2(100);
      k := test2(re);
    end test;
  equation
    k = test(1);
end Uniontype2;

Uniontype2_test_rettype _Uniontype2_test(modelica_integer s)
{
  Uniontype2_test_rettype tmp1;
  state tmp2;
  modelica_integer k;

  //UT re;
  modelica_uniontype re;

  modelica_uniontype tmp3;
  modelica_uniontype tmp4;
  Uniontype2_test2_rettype tmp5;
  tmp2 = get_memory_state();

  // re := RECT(66);
  tmp3 = mk_box1(0, mmc_mk_icon(66));
  re = tmp3;
  // re := REC2(100);
  tmp4 = mk_box1(1, mmc_mk_icon(100));
  re = tmp4;
  // k := test2(re);
  tmp5 = _Uniontype2_test2(re);
  k = tmp5._Uniontype2_test2_rettype_1;

  __return:
  tmp1.targ1 = k;
  restore_memory_state(tmp2);
  return tmp1;
}

Code 3.15 Example of Generated Code
Chapter 4

Guidelines for Implementing a New High-Level Data Structure

The implementation of the MetaModelica extension in the OpenModelica Compiler involves the addition of several new, high-level, data structures: union types, lists, tuple types and option types. In this chapter I will describe a general course of action when adding a new data structure to the compiler.

Generally, a new data structure type must be added to the compiler type system. If you wish to add a new simply type to a compiler (such as an integer type) this is a relatively straightforward process: the new type is added to the type system module and rules for matching two expressions of this new type are added as well. In the back-end the new type should be matched against a corresponding type in the target language. Minor changes in a few other modules are needed as well. However in this case we are dealing with high-level and, in some cases, parameterized data types, which leads to more work. As of now the array parameterized data type, for instance, is treated in a separate manner in the OpenModelica Compiler. One must also consider the fact that this MetaModelica extension is being added to the compiler in an ad-hoc manner. That is, it is separated with a compiler flag and we have altered the original flow of the compiler as little as possible, this also increases the complexity and difficulty.

New data structures may come with new syntax (other than the type keyword). For instance the list data structure comes with syntax for declaring lists as well as list constructors? syntax for building lists. See the table below. The main work for the extension with the four high-level data structures mentioned above (union types, lists, tuple types and option types) involves:

- Addition to the parser and the abstract syntax module. Note that lists, tuples and option types variables are parsed as variables of new complex types and union type variables are parsed as variables of a new restricted class.
• The new type should be added to the type system. This includes adding type matching rules, etc.

• New expressions associated with the new data structure must be handled. For instance the cons-constructor expression - `cons` or `::` - in connection with the list type or the union type record constructor call - `MyRecord(1,2,3,4)`.

• A union type restricted class declaration must be treated in a special manner, as has been described in this thesis. Lists, tuples, and option types do not involve class declarations (tuples and option types can be said to involve class declarations explicitly).

• These new types should be handled as input and output to functions and in matchcontinue expressions.

• A declaration of a variable of this new type might have to be treated separately in the instantiation phase.
<table>
<thead>
<tr>
<th>Union type</th>
<th>Tuple type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>New class restriction</strong></td>
<td><strong>Variable declaration</strong></td>
</tr>
<tr>
<td>uniontype UT</td>
<td>tuple&lt;...&gt; myVar;</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>record REC1 ... end REC1;</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>end UT;</td>
<td></td>
</tr>
<tr>
<td><strong>Variable declaration</strong></td>
<td></td>
</tr>
<tr>
<td>UT myVar;</td>
<td></td>
</tr>
<tr>
<td><strong>New expressions</strong></td>
<td></td>
</tr>
<tr>
<td>REC1(...) – Record constructor call</td>
<td></td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td></td>
</tr>
<tr>
<td>See previous chapters.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option type</th>
<th>List type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variable declaration</strong></td>
<td><strong>Variable declaration</strong></td>
</tr>
<tr>
<td>Option&lt;...&gt; myVar;</td>
<td>list&lt;...&gt; myVar;</td>
</tr>
<tr>
<td><strong>New expressions</strong></td>
<td><strong>New expressions</strong></td>
</tr>
<tr>
<td>NONE(...), SOME(...) – Constructor calls</td>
<td>;, cons – Cons operators</td>
</tr>
<tr>
<td>...</td>
<td>{ ... } – List constructor</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td><strong>Example</strong></td>
</tr>
<tr>
<td>function myFunction</td>
<td>function myFunction</td>
</tr>
</tbody>
</table>
| ... | ...
| Option<Integer> myVariable; | list<Integer> myVariable; |
| **algorithm** | **algorithm** |
| ... | ...
| myVariable := SOME(9999); // Assignment | myVariable := 11 :: (22 :: {}); // Assignment |
| myVariable := NONE(); | |
| ... | |
| end myFunction; | end myFunction; |
A description of which modules that has to be modified follows.

**Parser:** ANTLR is used as an external OpenModelica parser. From a formal grammar, ANTLR generates a program that determines whether sentences conform to that language. In other words, it is a program that writes other programs. The Modelica grammar is written in EBNF. EBNF means Extended Backus-Naur Form, the grammar for describing parser grammars. Typically a parser consists of a lexer and a parser. In the ANTLR case there is also a walker. The walker maps the abstract syntax from the parser into the Abstract Syntax of the OMC, given by the constructs in Absyn.h (generated from Absyn.mo). The three following files need to be altered in the parser:

- New keywords in `OpenModelica\modelica_parser\src\modelica_lexer.g`
- New grammar rules in `OpenModelica\modelica_parser\src\modelica_parser.g`
- New rules in `OpenModelica\absyn_builder\walker.g`

Other modules that should or might have to be altered follows alphabetically.

**Absyn.mo:** New abstract syntax for the new data structures are added to this module.

**Ceval.mo:** If the new data structure involved a new abstract syntax expression, such as the list-cons expression then in the function `Ceval.ceval` new rules could be added for constant evaluation (1::2, 3::4, etc.)

**ClassInf.mo:** If the new data structure involves a new restricted class, such as the union type, this module should perhaps be altered. For instance in the `ClassInf.trans` function new rules should be added so that we for instance make sure that equations are never declared inside a union type class. However, since the union type extension is handled differently and is part of the MetaModelica extension, it is not unified totally with normal Modelica. This is more of an ad-hoc implementation.

**Codegen.mo:** The correct type in the source language should be generated. Thus the functions `daeExpType, expTypeStr, generateType, generateTypeExternal, etc.` should be altered with new rules. In `generateExpression` we might have to add rules for generating code (from the Exp-expressions). For instance code to `Exp.CONS`, corresponding to the cons-operator expression. Note that in the MetaModelica case, `SimCodegen.mo` etc. does not need to be altered since we are dealing with time-invariant contexts.

**Exp.mo:** This module contains expressions after the instantiation phase, that is, expression with type information and that have been, perhaps, constant evaluated. So in the list type case we have a cons-constructor expression `Exp.CONS`

**Inst.mo:** This is one of the most complex modules of the compiler.

In the function `instElement` a new case-branch has been added/will be added that takes care of variable declaration of the new complex, MetaModelica types.

One must also consider the fact that a type may be derived. That is, we may have for instance:

```java
1 type MyListType1 = list<Integer>;
2 type MyListType2 = MyListType1;
3 MyListType1 var1;
```
This could be dealt with in the function `instVar` (which is called from `instElement`) where a type component is instantiated. Another issue is arrays, we may have arrays of lists, arrays of tuples, etc.

**Prefix.mo:** New rules for prefixing expressions must be added in function `prefixExp`.

**Scode.mo:** Abstract syntax may have to be added.

**Static.mo:** New rules for elaborating the new expressions should be added in the function `elabExp` in this module.

**Types.mo:** In the union type `TType` a new type record should be added. The most important function to alter is the `subtype` function: new rules for type matching should be added to this function.

Fig. B.1 contains a summary of which modules have to be extended in order to implement a new type into the compiler.
Chapter 5

Discussion and Related Work

In this chapter I will discuss the implemented solution, how it was done and positive/negative things of the implementation and the process of implementing it. Finally I will write a little about future development, what is under development right now and what to expect in the future.

5.1 Test Cases

Writing testcases is an important part of an implementation. This is where the final solution gets verified to make sure it fulfills the specification. For the union types I have made three different test-cases. The first one just declares a union type and assigns a record to it. The second uses a union type as an input to a function. The third one uses a union type both for input and output from a function. All the above cases passes just fine.

The test suite is available in:

<omc-root>/testsuite/

5.2 Performance Measurements

When extending a compiler it is always interesting to make some performance measurements before and after the extension. This was unfortunately not possible to do for this implementation since one part of the union type is still missing. That is getting the data out from the union type, and without this it is impossible to make any real use of them. The only assumption I can make about this is that the compiler itself will be not noticeable slowed down due to the extension since all extensions were made by adding case-clauses in different modules.
5.3 Generated Code

The generated code from this implementation consists of a

\[ \text{mk\_box\#}(\text{index}, \text{data\_data}) \]

for each assignment to the union type (Where \# is the number of variables in the record, \text{data\_data} the list of variables and index an index number). The idea is to use the index to be able to easily make a switch on the box to find out which of the records it consists of at the moment. The only problem with this construction is that the garbage collector is not able to handle these boxes correctly at the moment and thus will cause a small memory leak every time it is used. Fixing this is an important thing for the future.

5.4 Future Work

The compiler technology area is an area under heavy development and new theories about generating code is presented almost every other day. Generating code is also a complex thing making it hard to cover all possible code permutations and therefore leaving some bugs. The OpenModelica Compiler is of course no exception from this. The changes made are bug fixes and implementation of new features with the goal of making the OpenModelica Compiler able to handle the full MetaModelica extension. With the implementation of union types this goal is now rather close and will hopefully be reached within the end of this year. However, below are some of the subgoals needed before reaching this milestone.

5.4.1 Getting Union Types and Lists to Work in Matchcontinue Expressions

Making it possible to use lists and union types in matchcontinue expressions is an important goal. This is under development by the OpenModelica development team and will probably only require some minor changes to the pattern matching algorithm.

5.4.2 Getting Rid of the RML-Compiler

After making the compiler able to handle the full MetaModelica extension, the next step is making the OpenModelica Compiler being able to compile itself. The reason for this is to eliminate the use of the RML compiler, making the OpenModelica Compiler standalone and independent. This compiler compilation will be possible through the process of bootstrapping [4] within a near future (as soon as the OpenModelica Compiler is able to handle the full MetaModelica language).
5.4 Future Work

5.4.3 Extending the Comments of the OpenModelica Compiler

Even though most of the functions in the OpenModelica Compiler already have some sort of brief comment about its function, a little more extensive comment about how it was implemented and which other functions are used in order to achieve this, was often something that could had been a real timesaver. To facilitate further extension of the compiler it is therefore probably a good idea to take some time to extend these comments.
Chapter 6

Conclusions

6.1 Accomplishments

The goal for this thesis work was to fully implement support for union types into the OpenModelica Compiler. I believe this goal has mostly been achieved but some minor parts are still needed (see section 5.4.1). The union type implementation as it is now allows you to:

- Declare a Union Type Holder Variable.
- Assign Records of the Union Type to its Holder Variable.
- Reassign the Holder Variable With Another Record.
- Use Union Types as Input to a Function.
- Use Union Types as Output From a Function.
Bibliography


Appendix A

An Overview of the Compiler
Figure A.1. An Overview of the Compiler
Figure B.1. Extending the Compiler With a New High-Level Datastructure
Index

Abstract Syntax Tree, see AST
Absyn, 20, 22
acceptMetaModelicaGrammar, 15
analysis, 19
Analyzer, 13
ANother Tool for Language Recogni-
tion, see ANTLR
ANTLR, 14
AST, 14, 25
back-end, 35
bootstrapping, 48
boxes, 48
C-Runtime, 15, 35
ClassInf, 33
code generation, 13, 15
Codegen, 15, 21
comments, 49
compiler
code generation, 19
flow, 14
overview, 20
translation stages, 13
createFunctionArgsList, 29
createMetaClasses, 25
DAE, 13, 15, 21, 32, 35
DAELow, 15
DASSL, 13
debugging, 17
dependencies, 5
Dump, 14
Eclipse, 12, 17
elabCallArgs, 28
elabClass, 23
elaborate, 23
class, 23
elaboration, 13
elabRestriction, 23
environment, 14
environment updates, 26
EOO, 1
Equation-based Object-Oriented, see
EOO
eval relation, 8
executable code, 13
Exp, 21, 33, 35
extended modules, 20
front-end, 15, 35
functional language, 8
garbage collector, 15, 48
generateExpression, 35
generateSimpleType, 35
goals, 1
graph, 6
import statements, 13
inheritance, 13
Inst, 14, 20, 26
instantiation, 13, 14
instElement, 26
intermediate form, 14
language specification, 7
Lookup, 15, 20, 28
lookup statements, 13
lookupType, 28
main function, 14
match-expressions, 9
matchcontinue, 11, 19, 48
MathCore, 13
MDT, 17
memory leak, 48
MetaModelica, 9, 12, 48
  matchcontinue, 11
METARECORD, 22, 25, 28, 29, 32, 33
METARECORDCALL, 29, 33, 35
MetaUtil, 21, 25
Mikael Pettersson, 8
mmc_mk_box2, 16
mmc_mk_cons, 16
mmc_mk_icon, 16
mmc_mk_nil, 16
mmc_mk_rcon, 16
mmc_mk_scon, 16
Modelica, 6, 13
  compiler, 12
  debugger, 12
  Eclipse, 12
  Emacs, 12
  environment structure, 12
  failure, 11
  graphical model editor, 13
  interactive session handler, 12
OpenModelica, 12
  return, 11
modifications, 13
name lookups, 14
Natural Semantics, 7
ODE, 13
OpenModelica, 6
OpenModelica Compiler, 12, 49
OpenModelica Tooling, 17
optimization, 13
Optimizer, 13
outline, 1, 2
overview, 14
Parse, 14, 20
parsing, 14
pattern matching, 32, 48
Patternm, 20, 32
PELAB, 9, 12
Peter Fritzson, 8
plot, 6
Prefix, 20, 32
prefix, 32
prefixes, see prefix
prefixExp, 32
prerequisites, 1
Programing Environment Laboratory, see PELAB
R_METARECORD, 20, 22, 23, 26
R_UNIONTYPE, 20, 23, 26
record, 29
Relational Meta-Language, see RML
RML, 7–9, 48
RML-Compiler, 8
RTOpts, 15
SCode, 14, 20, 23
SimCodeGen, 15
simple type, 35
simulate, 6
simulation environment, 15
SML, 7
Standard ML, see SML
Static, 20, 28
subtype, 29
subtypes, 29
testcases, 47
type checking, 13, 14, 29
type lookups, 28
Types, 20, 29
union type, 9, 20, 29, 35, 48
  datastructure specification, 9
  holder variable, 9
  index, 22
  pattern matching, 22
union types, 9
UNIONTYPE, 32, 33
variable instantiation, 26
Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/

© Simon Björklén