Institutionen för datavetenskap
Department of Computer and Information Science

Master's Thesis

Bidirectional External Function Interface Between Modelica/MetaModelica and Java

by

Martin Sjölund

LIU-IDA/LITH-EX-A--09/041--SE

2009-08-19

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Master's Thesis

Bidirectional External Function Interface Between Modelica/MetaModelica and Java

by

Martin Sjölund

LIU-IDA/LITH-EX-A--09/041--SE

2009-08-19

Supervisor: Adrian Pop

Examiner: Peter Fritzson

Avdelning, Institution
Division, Department

Division for Software and Systems
Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Datum
Date

2009-08-19

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ ________________

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ ________________

ISBN
—

ISRN
LIU-IDA/LITH-EX-A--09/041--SE

Serietitel och serienummer
Title of series, numbering

ISSN
—

URL för elektronisk version

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-20386

Titel
Title

Tvåvägs funktionsgränssnitt mellan Modelica/MetaModelica och Java

Bidirectional External Function Interface Between Modelica/MetaModelica and Java

Författare
Author

Martin Sjölund

Sammanfattning
Abstract

A complete Java interface to OpenModelica has been created, supporting both standard Modelica and the metamodeling extensions in MetaModelica. It is bidirectional, and capable of passing both standard Modelica data types, as well as abstract syntax trees and list structures to and from Java and process them in either Java or the OpenModelica Compiler. It currently uses the existing CORBA interface as well as JNI for standard Modelica. It is also capable of automatically generating the Java classes corresponding to MetaModelica code. This interface opens up increased possibilities for tool integration between OpenModelica and Java-based tools, since for example models or model fragments can be extracted from OpenModelica, processed in a Java tool, and put back into the main model representation in OpenModelica.

A first version text generation template language for MetaModelica is also presented. The goal for such a language is the ability to create a more concise and readable code when translating an abstract syntax tree (AST) to text.

Nyckelord
Keywords

Java, OpenModelica, MetaModelica, external function, abstract syntax, template language

Abstract

A complete Java interface to OpenModelica has been created, supporting both standard Modelica and the metamodeling extensions in MetaModelica. It is bidirectional, and capable of passing both standard Modelica data types, as well as abstract syntax trees and list structures to and from Java and process them in either Java or the OpenModelica Compiler. It currently uses the existing CORBA interface as well as JNI for standard Modelica. It is also capable of automatically generating the Java classes corresponding to MetaModelica code. This interface opens up increased possibilities for tool integration between OpenModelica and Java-based tools, since for example models or model fragments can be extracted from OpenModelica, processed in a Java tool, and put back into the main model representation in OpenModelica.

A first version text generation template language for MetaModelica is also presented. The goal for such a language is the ability to create a more concise and readable code when translating an abstract syntax tree (AST) to text.

Acknowledgments

I would like to thank Peter Fritzson for creating a fun and challening project to work on, Adrian Pop for quickly answering technical questions, Kristian Stavåker for his continuous help during the process and Joanne Ting for her help with the non-technical parts of the text. Without all of you, the result of this thesis would have been quite different.

Contents

List of Figures

List of Tables

Listings

Terms and Definitions

- *AST*: Abstract Syntax Tree - a tree representation of the syntactic structure of source code.
- *Bootstrapping*: Bootstrapping is a term often used in the context of compiler construction. When bootstrapping a compiler, the compiler is written in the target programming language of the compiler (or possibly a subset of the target language to make the process simpler).
- *CORBA*: Common Object Request Broker Architecture - uses an IDL to let programs interoperate over a network.
- *IDL*: Interface Definition Language - describes a language-neutral interface through which software components can communicate.
- *JDK*: Java Development Toolkit - required to develop Java applications.
- *JNI*: Java Native Interface.
- *JRE*: Java Runtime Environment - required to run Java applications.
- *JVM*: Java Virtual Machine - part of the JRE. Knows how to run Java bytecode in a virtual machine.
- *Method*: What Java programmers call a “function”.
- *OMC*: OpenModelica (Interactive) Compiler.
- *RML*: Relational Meta-Language. RML is used to compile MetaModelica code.
- *SWIG*: Simplified Wrapper and Interface Generator.
- *XML*: EXtensible Markup Language.
- *XSLT*: EXtensible Stylesheet Language Transformations - a markup language used to transform XML to other formats.

Chapter 1

Introduction

1.1 Background

Since the amount of information we know is limited and there are several aspects of the world that lay obscure, we feel a need to find out how things work, and thus perform experiments. However, experiments have problems: they can be too expensive, they can take too long or simply be impossible to perform under normal circumstances. That is why we make models and simulations.

The Modelica language is an equation-based object-oriented language specialized for simulations. It also has an extensive standard library covering multiple domains. OpenModelica is an open source project[1] that aims to "create a complete Modelica modeling, compilation and simulation environment".

The Modelica language has support for external functions. There is no need to write your own log or sin function because they already exist in the C standard library. The two languages described in the standard are C and Fortran 77. However, new code is commonly written in higher-level languages, such as Java and C#, which hide things like data pointers from the user.

1.2 Intended Audience

The reader of this document is someone who wants a deeper understanding of the Java interface in OpenModelica. If the reader is an OpenModelica developer he/she probably wants a more in-depth explanation than what is written in the system documentation. The reader is assumed to know Java programming [2] and some knowledge of compiler construction is preferred although a brief introduction to the subject will be given.

[1] OpenModelica uses a strict GPL3 license for the public (members have more rights) and can be compiled using only free software. The full text can be seen on `http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/Documents/LICENSE.txt`.

[2] Some knowledge of C programming is useful because the OpenModelica Compiler translates Modelica code to C code. If you know Java, C syntax is not much of a problem, however.

1.3 Goals

- OpenModelica should be extended to handle external Java functions.
- Since C, Fortran and Java functions all share a common structure, the OpenModelica code generator should use a more general method, such as template-based code generation, when generating code for external function calls.
- It should be possible to analyze the abstract syntax tree of OpenModelica from a Java application to create a Java mapping of the code loaded in OpenModelica.
- It should be possible to use said mapping to call OpenModelica functions (including the MetaModelica extensions) from Java.

1.4 Method

Because the project has a focus on creating a working implementation there was no need to do a lot of prestudies. Similar things have been done before and most of the required knowledge comes from taking courses in Compiler Construction.

What was needed was to read up on Modelica and MetaModelica (the language that the compiler is written in). I read the introductory parts and the parts covering functions in Fritzson's book [8, Chapters 2, 3 and 9]. OpenModelica is shipped with a User Guide, System Documentation, MetaModelica Programming Guide and some ready to run examples in OMNotebook [24]. I mainly studied the MetaProgramming Guide but I used the other documents as reference throughout my work.

Finally, working implementations of the OpenModelica Compiler (OMC) to External Java and Java to OMC modules were created in a test branch and subsequently merged into the OMC bootstrapping branch. The development was done on Ubuntu Linux (8.10 and 9.04) using OpenJDK6 (Java version 1.6.0) and GCJ 4.3.3 (Java version 1.5.0). Some consideration was taken to update the corresponding Windows parts of the code.

1.5 Limitations

The full Modelica AST including the MetaModelica extensions can currently not be compiled by OMC. Since the AST uses MetaModelica constructs like uniontypes, the Java mapping needs to handle these extensions of Modelica. Thus, the part of the project that relies on the MetaModelica extension cannot be fully completed until the compiler has been bootstrapped and can compile itself (which might be after this project ends). What can be done before the bootstrapping is complete is to create a mapping and preliminary testing of these datatypes, as well as creating the external Java interface.

1.6 Thesis Outline

The thesis starts by familiarizing the reader with Modelica and Compiler Construction in general. It then covers an assortment of tools and technologies that could be used to complete the goals of the thesis. After the necessary theory and tools have been presented, the different parts of the implementation are presented. Finally, the project is assessed.

Chapter 2

Background

2.1 Modeling and Simulation

For the purpose of this text, a model is a mathematical model. We can use variables like weight and temperature to decide how they affect a simulation. We do not consider physical models, like building a miniature house to see if it collapses or not. A simulation is defined as an experiment on a model.

2.2 Modelica

The Modelica language is an object-oriented language specialized for modeling and simulation. Rather than declare how to solve a problem, you model the problem using mathematical equations. In an imperative language like C you use statements like `x = 150; y = x/2;` and execute them from bottom and down, left to right. Because Modelica is declarative you can write `x = 2*y; x = 150;` instead and let Modelica solve `x = 150; y = 75;`. Simply solving an equation is not what is needed to simulate a model since we might have differential equations or an explicit time variable.

The usual "hello world" example for programming languages is a program that simply prints the string "hello world". This example does not apply to Modelica because the language is not used to print Strings. Instead, we simulate a simple model (Listing 2.1) and plot the result (Figure 2.1).

Listing 2.1. HelloWorld.mo

```
class HelloWorld "A simple diff. equation"
    Real x(start=1);
    Real y(start=-2);
    Real z;
    parameter Real c1=0.5;
    parameter Real c2=5;
equation
```

```
    der(x) = -c1*x;
    -c2*der(x) = der(y);
    -z = cos(10*x);
end HelloWorld;
```

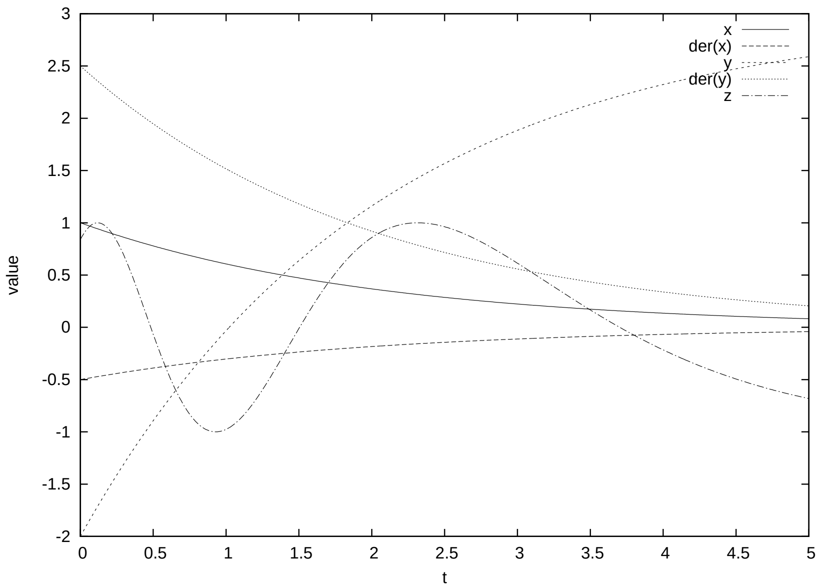

Figure 2.1: Hello World Output

Since Modelica has a concept of connections you can create a connector such as an electrical pin in order to connect components like resistors and inductors to simulate a circuit. See Listing 2.2 for an example of a connector class. Modelica also comes with a large standard library covering multiple domains, e.g. mechanical, electrical, hydraulic and thermal applications.

Listing 2.2. A Modelica Connector

```
connector Pin "Electrical pin"
  Voltage      v "Potential at the pin";
  flow Current i "Current flowing into the component";
end Pin;
```

For larger circuits you can use graphical modeling tools to connect the components. The Modelica standard supports annotation of variables and classes. There exists standardized annotations in several areas; some change how code is generated and others define how "Graphical Objects" are described [2]. OpenModelica recommends using SimForge [15] for graphical modeling. The sample project shipped with SimForge contained the SimpleCircuit model seen in Listing 2.3. The Modelica code models the RC circuit in Figure 2.2 (the figure has been reconstructed using vector graphics to look better in print). When you simulate and plot the project you end up with the graph in Figure 2.3.

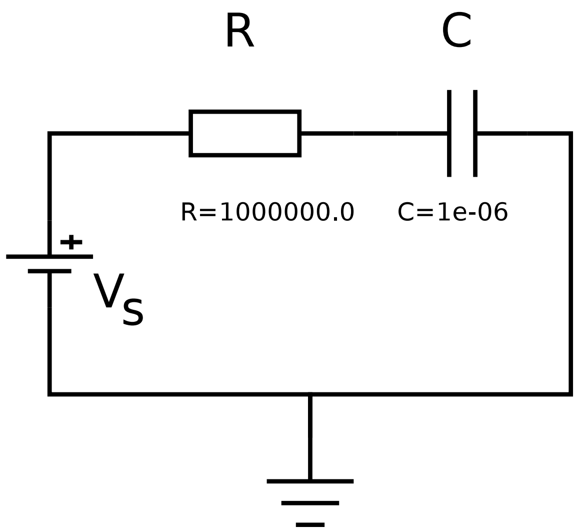

Figure 2.2: Simple Circuit: Graphical Modeling in SimForge

Listing 2.3. Simple Circuit: Modelica Code

```
model SimpleCircuit "Model of a RC circuit"
annotation(Diagram(coordinateSystem(extent = {{ −100,
    −100},{100,100}})));
Modelica.Electrical.Analog.Sources.StepVoltage Vs
    annotation(Placement(transformation(x =  −35.0, y =
    −21.0, scale = 0.21000001), iconTransformation(x =
    −58.0, y =  −22.0, scale = 0.21000001, rotation =
    −90.0)));
Modelica.Electrical.Analog.Basic.Ground ground annotation
    (Placement(transformation(x = 5.0, y =  −79.0, scale
    = 0.21000001), iconTransformation(x = 2.1198158, y =
     −80.15208, scale = 0.21000001)));
Modelica.Electrical.Analog.Basic.Resistor R(R =
    1000000.0) annotation(Placement(transformation(x =
    17.0, y = 7.0, scale = 0.21000001),
    iconTransformation(x =  −23.0, y = 19.0, scale =
    0.21000001)));
Modelica.Electrical.Analog.Basic.Capacitor C(C = 1e−06)
    annotation(Placement(transformation(x = 82.0, y =
    −22.0, scale = 0.21000001), iconTransformation(x =
    36.0, y = 19.0, scale = 0.21000001)));
```

```
equation
  connect(Vs.n,ground.p) annotation(Line(points = {{ -58.0,
      -43.0},{3.0, -43.0},{3.0, -59.0}}));
  connect(C.n,ground.p) annotation(Line(points =
      {{57.0,19.0},{58.0, -44.0},{25.0, -44.0},{3.0,
      -43.0},{3.0, -59.0},{3.0, -59.0}}));
  connect(R.n,C.p) annotation(Line(points = {{
      -2.0,19.0},{16.0,19.0},{16.0,17.0},{15.0,19.0}}));
  connect(R.p,Vs.p) annotation(Line(points = {{
      -44.0,19.0},{ -45.0,20.0},{ -59.0,20.0},{ -58.0,
      -1.0}}));
end SimpleCircuit;
```

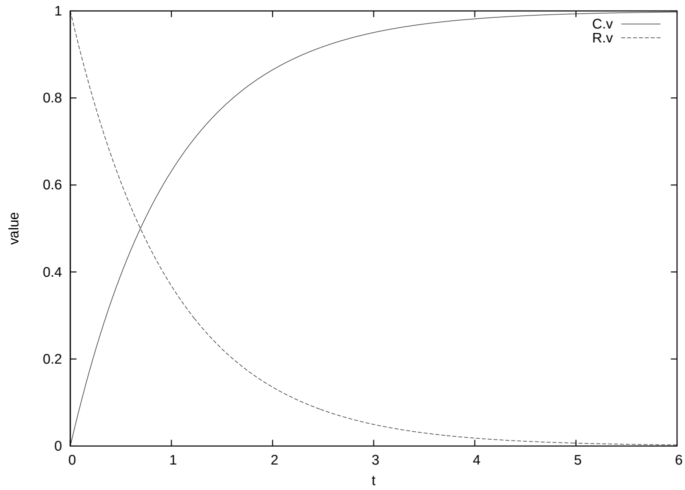

Figure 2.3: Simple Circuit: Simulation Result

2.2.1 Modelica External Functions

Modelica supports multiple output results/parameters in the same function. Because of this, the same function can be mapped to an external function in several different ways. The functions in Listing 2.4 will be called in the same way from Modelica code. Only the first function specifies the language of the external function. The default language is C. The first function specifies that the external function should have a C prototype that looks like `double nameOfFunc( double x, double* y2)`. That is, it returns y1 and passes a pointer to the location where y2 should be stored. By contrast, the second function expects that the C prototype should look like `void nameOfFunc( double x, double* y1, double* y2)`. The third example uses the default mapping, which becomes `void example3( double x, double* y1, double* y2)`. [8]

Listing 2.4. Modelica External Function

```
function example1
  input Real x;
  output Real y1;
  output Real y2;
external "C" y1=nameOfCFunc(x,y2);
end example1;

function example2
  input Real x;
  output Real y1;
  output Real y2;
external nameOfCFunc(x,y1,y2);
end example2;

function example3
  input Real x;
  output Real y1;
  output Real y2;
external;
end example3;
```

2.3 MetaModelica

Because Modelica lacks common language constructs like lists, it cannot be used to implement a Modelica compiler. MetaModelica is an extension of Modelica that is used to write the OpenModelica compiler. The language is defined in [24, Metaprogramming]. It is supposed to be so powerful that it can later compile the MetaModelica compiler itself. The main constructs that were introduced are `matchcontinue`, `uniontype`, `list` and the `option` type.

2.3.1 Uniontype

In Modelica you can declare a `record`, a `class` without `equation block`. It is simply a collection of variables. A `uniontype` contains any number of `record` members. The structure may be recursive, that is the `records` are allowed to contain `uniontype` members. Whenever you pass a variable of the `uniontype`, you actually pass an instance of a member `record`. If you are familiar with Java, you can view a `uniontype` as an `interface` or `abstract base class` with no methods. See Listing 2.5 for an example on how to declare an expression uniontype.

Listing 2.5. Expression Union Type

```
uniontype Exp
```

```
  record ICONST
    Integer integer;
  end ICONST;
  record IDENT
    Ident id;
  end IDENT;
  record ADD
    Exp lhs;
    Exp rhs;
  end ADD;
  record LESS;
    Exp lhs;
    Exp rhs;
  end LESS;
end Exp;
```

2.3.2 Match-Continue Expressions

The `matchcontinue` construct can be seen as what is usually called either `switch` or `case` in C/Java/Pascal or pattern matching in functional languages such as Standard ML or Haskell, although `matchcontinue` can do more. In a matchcontinue expression, you can match more than a single variable in the same statement. You can also match on members of `records`, or what `record` type a `uniontype` actually is. In MetaModelica, function calls and `cases` may `fail()`. If it does `fail()` in a `case`, MetaModelica will undo the variable bindings and try the next `case`. The code in Listing 2.6 is a simple example using `matchcontinue`. The example uses the expression uniontype in Listing 2.5. Also note that the operator for addition of real numbers is `+.` and not `+`, which is reserved for integer addition.

Listing 2.6. Match-Continue Example

```
function ApplyExpression "Calculate the value of an
    expression"
  input Exp exp;
  output Real out;
algorithm
  out := matchexpression (exp)
    local
      Integer i;
      Real r;
      Ident id;
      Exp exp1,exp2;
    case ICONST(i) then intReal(i);
    case RCONST(r) then r;
    case IDENT(id) then LookupConstantValue(id);
    case IDENT(id) then LookupVariableValue(id);
```

```
    case ADD(ICONST(1),exp2) then 1.0 +. ApplyExpression(exp2);
    case ADD(exp1,exp2) then ApplyExpression(exp1) +. ApplyExpression(exp2);
    case SUB(exp1,exp2) then ApplyExpression(exp1) -. ApplyExpression(exp2);
  end matchexpression;
end ApplyExpression;
```

2.3.3 List

If you ever programmed in LISP, you consider the high-level data-structure “list” very powerful. MetaModelica lists are not quite as flexible as LISP lists because they are typed[1]. This means that a list can only have a finite depth (as deep as you declare list of list of list of...), which in turn means that you cannot represent the high-level data-structure “tree” using MetaModelica lists (that is what uniontypes are for). MetaModelica lists are used to iterate over data using recursive functions rather than arrays and for loops.

In MetaModelica, the type `list` of `Integer` is declared as `list<Integer>`. A list is either the empty list {} or a head and a tail (called a cons-cell). The head is a value of the type that the list was declared as having. The tail is another `list` of the same type. In order to construct a `list` you prefix a value to the top of a `list`, starting with the empty `list`. Listing 2.7 contains a function that converts a `list<Integer>` to a `list<Real>`.

Listing 2.7. List Example

```
function listIntToReal "Convert list<Integer> to list<Real>"
  input list<Integer> lInt;
  output list<Real> lReal;
algorithm
  lReal := matchexpression (lInt)
    local
      Integer int;
      Real real;
      list<Integer> restInt;
      list<Real> restReal;
    case {} then {};
    case int::restInt
      equation
        real = intReal(int);
        restReal = listIntToReal(restInt);
      then real::restReal;
  end matchexpression;
end listIntToReal;
```

[1]When the Any type is introduced in MetaModelica, lists can also be used to create tree structures the same way as in LISP.

2.3.4 Option Type

The `Option` type is the MetaModelica answer to `NULL` in C[2]. The `Option` is either `SOME(value)` or `NONE`. Just like with the `list`, you must specify an `Integer Option` using `Option-<Integer>`.

Listing 2.8. Option Example

```
function getExtLanguageName "Fetches the name of the optional external language"
  input Option<String> opt;
  output String language;
algorithm
  language := matchexpression (opt)
    local String language;
    case NONE then "C";
    case SOME(language) then language;
  end matchexpression;
end getExtLanguageName;
```

2.4 Compiler Construction

There are many different books on compiler construction and the steps described are never quite the same. The following sequential steps were taken from a standard textbook[1] and describes “normal” compilers, not a compiler for an equation-based language. To better illustrate what each step does we will show what the following statement looks like after each step:

Listing 2.9. Example Statement

```
a = b + (13.0 - 2*6)
```

2.4.1 Lexical Analysis

Sometimes called scanning. It is the process of converting a stream of characters into a stream of lexemes, or tokens, on the form of `<name, attribute>`. The lexer should also create a symbol table which is a fast way to look up type information of an identifier. There exists programs that create code for lexical analysers based on a description.

Listing 2.10. Statement after Scanning

```
<id,1> <=> <id,2> <+> <(> <13.0> <-> <2> <*> <6> <)>
```

[2]The main difference is that C pointers are very prone to errors if you ever forget to check for `NULL` pointers. MetaModelica forces you to check for `SOME(value)`.

2.4.2 Syntax Analysis

Sometimes called parsing. This process verifies that the stream of tokens produce an abstract syntax tree (AST) according to the grammar. Examples of programs that automatically generate parsers are YACC and ANTLR, which produce code for LALR(k) or LL(k) grammars [3].

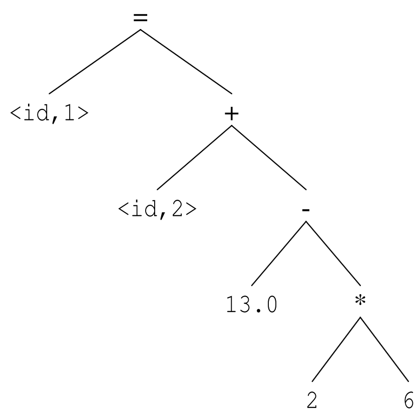

Figure 2.4: Abstract Syntax Tree after Parsing

2.4.3 Semantic Analysis

Semantic analysis is type checking the tree using the symbol table. Expressions like `true+1` may be grammatically correct but it does not necessarily mean anything according to the language semantics. If we assume that the semantics is similar to elementary school math, `1+3.0` does mean something. It is the addition of two numbers (one integer and one float). The semantic analysis should convert the expression to `inttofloat(1)+3.0`, which results in another floating point number.

[3] Read up on "automata theory" to learn how to construct grammars in the context of Compiler Construction.

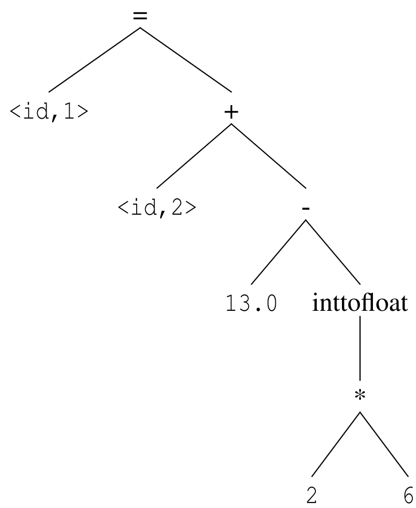

Figure 2.5: Abstract Syntax Tree after Semantic Analysis

2.4.4 Intermediate Code Generation

The intermediate code should be a simple one, closer to the target language. Most intermediate codes represent the AST as a sequences of statements. The *three-address code* is an example of an assembler-like code where every instruction has at most three operands (target,arg1,arg2) and an operator. Because the three-address code is sequential, tree structures in the AST need to be flattened out and the operations carried out in the correct order.

Listing 2.11. Statement as Intermediate Code

```
t1 = 6*2
t2 = inttofloat(t1)
t3 = 13.0-t2
t4 = id2+t3
id1 = t4
```

2.4.5 Code Optimization

Code optimization is usually done in two or more phases for machine-independent and machine-dependent optimizations. The output of the optimization process is usually the same as the previous step so you can easily disable it. An example of a machine-independent optimization is *constant folding*, where you calculate the result of a constant expression like `3.0+inttofloat(2*4)`. Machine-dependent optimizations could use special instructions to perform vector operations.

Listing 2.12. Intermediate Code after Constant Folding

```
id1 = id2+1.0
```

2.4.6 Code Generation

This phase takes the intermediate code as input and translates the code to the target language. It should also perform register allocation if required by the target language (usually only needed for assembler). Code Generation should be a simple step.

2.5 The OpenModelica Environment

In order to complete its goals to be a complete Modelica environment, OpenModelica has several components. For the purpose of this thesis, the OpenModelica (Interactive) Compiler, OMC, is the most important component. It can be used in several modes. Its interactive mode can be used to process scripts (.mos files) or to spawn a daemon which you can communicate with using CORBA or sockets. While the interactive mode is usually used to simulate models, the focus in this thesis is on functions. When the user calls a function in the interactive session, OMC will translate the Modelica function to C-code which it compiles and executes. The compiler used to compile MetaModelica code is called RML and the goal is to replace it once OpenModelica is bootstrapped (i.e. OMC can compile itself). Figure 2.6 shows the main data flow and connections between

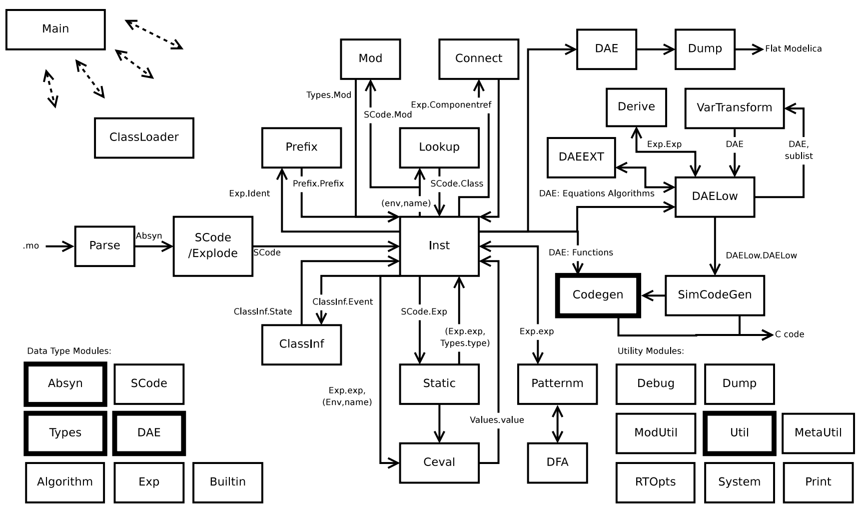

Figure 2.6: OMC: Modules

modules in OMC. The modules most relevant to the external Java parts of the thesis have a strong outline. As you can see, the Codegen module is very far into the translation process from Modelica code to C code.

- The Codegen module is the module that translates Modelica code to C code. This includes external C and Fortran77 functions, and now also external Java functions.
- The Util module contains functions used to iterate over lists and other useful things. It is important to use these in order to reduce duplicated code because MetaModelica is a very verbose language.
- The Absyn and Types modules are used by the Interactive interface. Absyn contains the abstract syntax after parsing. The code is not checked for semantics like inheritance and type names. The interactive module can list the code of a function, but that does not mean that the function is semantically correct.
- The DAE and Types modules are used for example by the Codegen module. From an external function viewpoint, the DAE is the datatype that contains the function declaration (i.e. in/output types, external language, external function name, call structure mapping and annotations).
- The Interactive module implements the API that is used to communicate with OMC. While it does not show in this figure it plays a role in the Java to Modelica communication.

Chapter 3

Existing Technologies

There are two different kinds of technologies in this list. The ones that need to be used in the project and the ones that needs to be reimplemented. Most are mentioned briefly and some have example code listed to better differentiate them.

3.1 Java Native Interface

The Java Native Interface (JNI) is useful if you want your Java code to access native libraries or legacy code. It is also used to link to a JVM from a native application.

JNI has more than a few problems. By using it, you lose the type safety of Java. If your C program passes some bad data to Java, the result is undefined. JNI does not provide any exception handling for the native method. This means that the programmer has to manually check for exceptions after each JNI function call. You can also modify fields declared as `final` without throwing an exception. It is recommended to use JNI only if the Java application needs to access native code in the same process. Alternatives to JNI include TCP/IP, sockets and the Java IDL API (CORBA). [16]

3.1.1 Java Calling C

There are a few ways to call native code from Java. If you just want a wrapper you can use so-called shared stubs. It is a faster way of writing the interface code but you will need to manipulate C pointers in Java if the function returns data that is not one of the basic types. You can also write the interface yourself, by declaring functions as native. The JDK has tools that generates C headers and code for the native functions. Using these and jni.h, it is possible to create and call Java classes from within C.

3.1.2 C Calling Java

Calling a Java program from C is simple. It requires a few lines of code to setup but after that it is the same as Java calling C. The main difference is that your Java code cannot access the native application in the same way that it can open up a shared object and call functions in that.

3.1.3 SWIG

Simplified Wrapper and Interface Generator (SWIG) is a tool that simplifies writing interfaces from C or C++ to 18 different languages. The idea is that you write one single interface file that is similar to a C header file and tell SWIG to generate interfaces for your target languages. [29]

The interesting part is exporting an interface to Java. Because it generates usable code and not stubs, you can use it without manually adding code. The code that is generated is some C code that uses JNI to communicate with a Java proxy class that talks to another Java class. The class that the user will access contains no references to native methods or C pointers. Structs are handled by the proxy class and gives the user a simple interface even though the structures are handled in the C memory space. [29]

3.1.4 GlueGen

GlueGen is used to automatically generate interfaces between Java and C. The generated interface is similar to that of SWIG, except it uses Java buffers instead of CPointer internally. [9]

3.1.5 Java Native Access

Java Native Access (JNA) is a library that allows you to call native C functions without using JNI or native code. Because it works without analyzing the source, it has no knowledge of the C structures and they need to be added manually as below. [14]

Listing 3.1. JNA Structure Glue

```
import com.sun.jna.*;
public class Timeval extends Structure {
    public int tv_sec;
    public int tv_usec;
}
```

Listing 3.2. JNA Example

```
import com.sun.jna.*;
public class Hello {

  public interface CLibrary extends Library {
    CLibrary INSTANCE = (CLibrary)
      Native.loadLibrary((Platform.isWindows() ? "msvcrt" :
          "c"),
                          CLibrary.class);
    void printf(String format, Object... args);
    // C says gettimeofday(timeval* tv, null)
    // JNA passes structures by reference as default
```

```
    int gettimeofday(Timeval tv, Pointer p);
  }

  public static void main(String[] args) {
    Timeval tv1 = new Timeval();
    CLibrary.INSTANCE.gettimeofday(tv1, null);
    CLibrary.INSTANCE.printf("%d.%d\n", tv1.tv_sec, tv1.
        tv_usec);
  }

}
```

3.2 ANTLR3

ANother Tool for Language Recognition v3 is a tool that takes a grammar as input and creates lexers, parsers, translators, interpreters or compilers [27]. Creating a lexer is simple. You enter the rules and ANTLR will create a finite automata that reads a stream of characters and outputs a stream of tokens. Using the ANTLRWorks tool you can see the automata for each token. It is possible to assign elements in the AST and return your own datatype instead of the auto-generated AST. You can also use these actions to interpret the input directly. A simple lexer and parser pair is shown in Listings 3.3 and 3.4. ANTLR will produce a lexer that uses finite automata like the one in Figure 3.1. Given the input `{1, 2, 3}`, it will produce an AST like the one in Figure 3.2.

Listing 3.3. OMCorba Lexer

```
lexer grammar OMCorbaLexer;
BOOL : 'true'|'false';
ID : ('_'|'a'..'z'|'A'..'Z')('_'|'a'..'z'|'A'..'Z
     '|'0'..'9')*;
STRING : '"'('\\"'|~'"')*'"';
REAL : '-'? (('.''0'..'9'+)|('0'..'9'+'.''0'..'9'*))(('e'|'
     E')(('+'|'-')?'0'..'9'+))?
       | '-'? '0'..'9'+('e'|'E')(('+'|'-')?'0'..'9'+);
INT :  '-'?'0'..'9'+ ;
WS  :   ('\r'|'\n'|' '|'\t')+ {skip();} ;
```

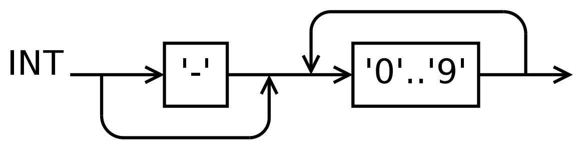

Figure 3.1: DFA for recognizing an integral number

Listing 3.4. OMCorba Parser

```
grammar OMCorbaParser;
options {
  output=AST;
  tokenVocab = OMCorbaLexer;
}
prog: object? EOF;
object: INT | REAL | BOOL | STRING | record | array;
record : 'record' id1=ID (field (',' field )*)? 'end' id2=ID {if (!id1.text.equals(id2.text)) throw new RecognitionException();} ';';
array : '{' (object (',' object {vector.add(memory);})*)? '}';
field : ID '=' object;
```

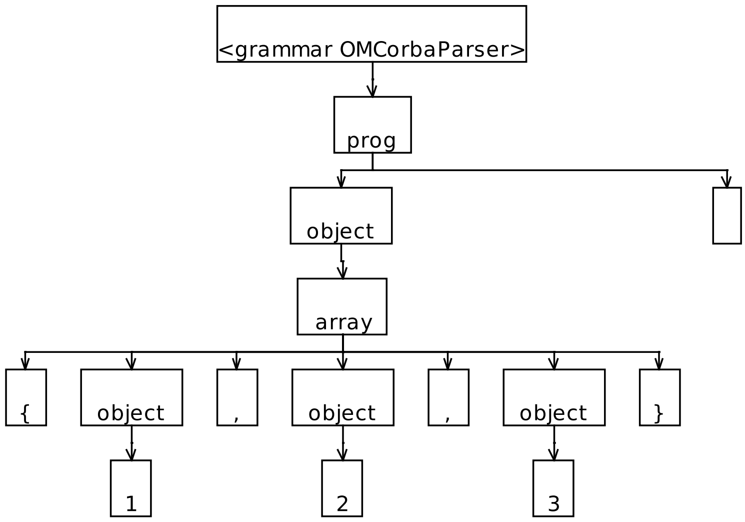

Figure 3.2: Parsing an array of integral numbers

3.3 Template Engines

Template engines can be used to generate code, documentation or web pages. Most of them use a Model-View-Controller concept (MVC), but they are not strict. The theory is that you have some static data, a template and an output based on the first two. Violations of the MVC concept include the possibility to alter the model (`a=91`), do computations of values (`a*7`), do conditional checks (`a<31`) or array indexing (`a[84]`). [25, 26, 28]

It is worth noting that most of these tools are based on Java and need to be fed XML data, Java classes or be converted from some data to Java classes. The reason is that these tools expect to iterate over data structures that implement some sort of iterator, methods or fields.

3.3.1 StringTemplate

StringTemplate is a template engine that has been designed to be strict when it comes to enforcing the MVC concept. The main author, Terrence Parr, says that:

> One only needs four template constructs: attribute reference, conditional template inclusion based upon presence/absence of an attribute, recursive template references, and most importantly, template application to a multi-valued attribute similar to lambda functions and LISPs map operator. [25]

Basically, you only need: insert-value-here, if-exist-include-body, foreach-element-do-body and outputting tree structures of unknown depth.

Listing 3.5. StringTemplate Input

```
import org.antlr.stringtemplate.*;
import org.antlr.stringtemplate.language.*;

class sttest {
public static void main(String[] args) {
  StringTemplate hello = new StringTemplate
     ("Hello, $name$\n" +
"While you were gone $names; separator=\", \"$ called you."
    ,
DefaultTemplateLexer.class);
  hello.setAttribute("name", "General");
  String[] names = {"Alpha","Bravo","Charlie"};
  hello.setAttribute("names", names);
  System.out.println(hello.toString());
}
}
```

Listing 3.6. StringTemplate Output

```
Hello, General
While you were gone Alpha, Bravo, Charlie called you.
```

3.3.2 Google ctemplate

Ctemplate is a C++-based template engine that is less complex than most of the Java-based alternatives. The input is a basic dictionary, which is a data structure that is easy to implement in functional languages. [10]

Listing 3.7. ctemplate Template

```
Hello {{NAME}},
You have just won ${{VALUE}}!
{{#IN_CA}}${{TAXED_VALUE}} after taxes.{{/IN_CA}}
```

Listing 3.8. ctemplate Input

```
#include <stdlib.h>
#include <string>
#include <iostream>
#include <google/template.h>
int main(int argc, char** argv) {
  google::TemplateDictionary dict("example");
  dict.SetValue("NAME", "John Smith");
  int winnings = rand() % 100000;
  dict.SetIntValue("VALUE", winnings);
  dict.SetFormattedValue("TAXED_VALUE",
      "%.2f", winnings * 0.83);
  // For now, assume everyone lives in CA.
  // (Try running the program with a 0 here instead!)
  if (1) {
    dict.ShowSection("IN_CA");
  }
  google::Template* tpl =
      google::Template::GetTemplate(
          "example.tpl",
          google::DO_NOT_STRIP);
  std::string output;
  tpl->Expand(&output, &dict);
  std::cout << output;
  return 0;
}
```

Listing 3.9. ctemplate Output

```
Hello John Smith,
You have just won $89383!
$74187.89 after taxes.
```

3.3.3 Apache Velocity

Velocity is a Java-based tool that generates output using templates. It is mainly used to serve webpages, SQL and PostScript [6] but can also be used for code generation [21].

The data consists of Java classes that are fed to the engine. Velocity applies the classes to the template using directions like if-else, foreach (for classes that implement the `Iterable` interface) and can set/get its own variables inside the template. It can also access methods in the input classes.

Listing 3.10. Velocity Template

```
class Structure {
#foreach( $var in $list )
  public $var.type.name $var.name ;
#end
}
```

Listing 3.11. Velocity Input Data

```
public class VarInfo {
  public Class type; public String name;
  public VarInfo (Class type, String name) {
    this.type = type; this.name = name;
  }
  public Class getType() {return type;}
  public String getName() {return name;}
}

public ArrayList<VarInfo> getInput()
{
  ArrayList<VarInfo> list = new ArrayList<VarInfo>();

  list.add(new VarInfo(Integer.class, "myInt"));
  list.add(new VarInfo(String.class, "myString"));
  list.add(new VarInfo(Double.class, "myDouble"));
  list.add(new VarInfo(Velocity.class, "myVelocity"));

  return list;
}
```

Listing 3.12. Velocity Output

```
class Structure {
  public java.lang.Integer myInt ;
  public java.lang.String myString ;
  public java.lang.Double myDouble ;
  public org.apache.velocity.app.Velocity myVelocity ;
}
```

3.3.4 FreeMarker

FreeMarker is a tool based on Velocity, but has more features and a different syntax. Among them are better loop handling and the ability to access array elements in the template. [7]

Listing 3.13. FreeMarker Template

```
${.node.example.@title}
<#list .node.example.test as x>
${x}<#if x_has_next>,</#if></#list>
```

Listing 3.14. FreeMarker Input Data

```
<?xml version="1.0"?>
<example title="FreeMarker example">
  <test>Output</test>
  <test>from</test>
  <test>FreeMarker</test>
</example>
```

Listing 3.15. FreeMarker Output

```
FreeMarker example
Output,from,FreeMarker
```

3.3.5 XSLT

XSLT can be used for code generation. It is recommended that you do the code generation in several passes because the templates become complex and hard to maintain if you try to do it all at once [13]. The idea of doing the code generation in several passes is good if you have several target languages that are similar in structure. One problem with XSLT is that the input is required to be XML. This means data structures need to be converted to XML before the tool can be used.

3.4 Java Metadata Interface

Java Metadata Interface (JMI) is the Java interface to the Meta Object Facility (MOF) specification from the Object Management Group (OMG) [19, 12, 11]. In the MOF standard you have a set of modeling artifacts (described in UML). You use these artifacts to describe any kind of metamodels. That is, you describe the meaning of methods and attributes in classes rather than just describing how the data is structured (like in XML).

> [...] although any one XML Schema can represent the metadata for a particular application, component, or service, it does not give a general solution to modeling metadata across many different domains. [20]

3.5 Dymola's Java Interface

Dymola is a Modelica environment that also supports Java as an external language. In order to handle records, Dymola uses the `com.dynasim.record` class, which internally represents the record as a `Map`. By using a `Map` as the datatype you can avoid problems if the order of the data changes in the underlying code. Exceptions in Java are mapped to

Table 3.1. Dymola Mapping of Java Datatypes

Modelica	External Java
Real	double
Integer	int
Boolean	boolean
String	java.lang.String
Record	com.dynasim.record
Real[]	double[]
Integer[]	int[]
Boolean[]	boolean[]
String[]	java.lang.String[]
Record[]	com.dynasim.record[]

Modelica assertions and the other way around. The syntax for declaring an external Java function is similar to that of C and Fortran (see Listing 3.16). The mapping of datatypes in Dymola can be seen in Table 3.1. [17]

Listing 3.16. Dymola External Java Function

```
function example1
  input  Real x1;
  input  Real x2;
  output Real y;
external "Java" y='Package.Class.StaticMethod'(x1,x2);
end example1;
```

Chapter 4

Implementation

4.1 Mapping of Datatypes

In order to introduce some compatibility between the Dymola and OpenModelica implementations of external Java functions, it makes sense to declare them in the same way ('Package.Class.StaticMethod'). The mappings between datatypes will not be the same because we will use the same mapping when Java is the calling language as opposed to the Dymola version. By doing it this way you get a consistent interface that can

Table 4.1. OMC Mapping of Java Datatypes

Modelica	**External Java**
Real	ModelicaReal
Integer	ModelicaInteger
Boolean	ModelicaBoolean
String	ModelicaString
Record	ModelicaRecord
Uniontype	IModelicaRecord
List<T>	ModelicaArray<T>
Tuple<T1,T2>	ModelicaTuple
Option<T>	ModelicaOption<T>
T[:]	ModelicaArray<T>

also be naturally extended for MetaModelica types (ModelicaTuple, ModelicaOption). Because the full MetaModelica mapping (Table 4.1) uses Modelica-specific classes for all datatypes, it cannot be used to call e.g. the Java method Integer.parseInt since it uses Java String and int. By annotating your external Java function declaration using annotation(JavaMapping = "simple"), an alternative mapping (Table 4.2) will be used. This mapping only supports the most basic Modelica types and only one output

Table 4.2. OMC Simple Mapping of Java Datatypes

Modelica	External Java
`Real`	`double`
`Integer`	`int`
`Boolean`	`bool`
`String`	`String`

value, but it can be used to call standard Java functions. This is a subset of the functionality that Dymola has, which also supports arrays, records and output variables that are not the return value of an external function call. Compare Tables 4.1 and 4.2 with the Dymola mapping in Table 3.1 to see the differences more clearly.

If the `ModelicaRecord` datatype is represented by a `java.util.Map` from `String` to `ModelicaObject`, it follows that it can contain any datatype we use in OMC[1]. By using a `LinkedHashMap` the field keys are in the same order as they are in Modelica[2]. One advantage of this solution is that the Java mapping of a record does not depend on creating a Java class before the program is executed. The disadvantage is that you need to check that you received the correct record type, and then get the fields using the method `ModelicaObject get(String key)`. This is equivalent to performing type checking during runtime. For those who want functions to perform said typecasting, see Section 4.3.3 for a method that creates Java class definitions from Modelica code.

4.2 Calling Java External Functions from Modelica

4.2.1 Generated Files

When using external C functions, OMC translates a Modelica file (Listing 4.1) to a C file (Listing 4.2). First of all, OpenModelica copies all input variables before the external call is made since arrays (as well as variables for Fortran functions) are passed by reference. Then the external call is performed and the output is copied into the return `struct` (since Modelica supports multiple output values).

Listing 4.1. exampleC.mo

```
function logC
  input Real x;
  output Real y;
external "C" y = log(x);
end logC;
```

[1] The interface `ModelicaObject` includes MetaModelica constructs. Naming it (Meta)ModelicaObject would be more appropriate, but it is not a valid identifier in Java.

[2] The Modelica standard enforces a strict field ordering because it is relevant for example in external C functions.

Listing 4.2. logC.c

```
logC_rettype _logC(modelica_real x)
{
  logC_rettype out;
  double x_ext;
  double y_ext;
  x_ext = (double)x;
  y_ext = log(x_ext);
  out.targ1 = (modelica_real)y_ext;
  return out;
}
```

When using external Java functions, OMC should generate a C file that is similar to the ones generated by external C functions. External Java calls translated the variables to Java objects, and fetch the correct method from the JVM through the Java Native Interface (JNI). The flow of data in Figure 4.1 is explained in detail below. Before the call, each

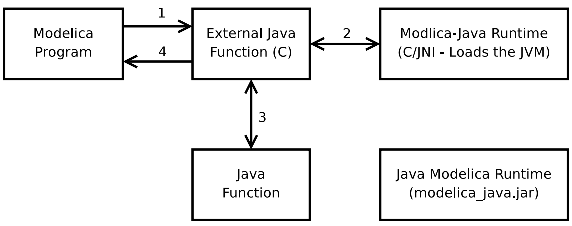

Figure 4.1: External Java Call (Data Flow)

argument is translated to a JNI `jobject` (i.e. a C pointer to a Java class) and then after copying the result back to the respective C variable. This ensures that the code works in the same way as external C (and thus the “correct” Modelica behaviour). Compare the C file for external C (Listing 4.2) to the one for external Java (Listing 4.4, generated by the Modelica code in Listing 4.3). The Java code is essentially the same with the difference being that instead of one line of code for an external call, it is 17 lines of code to set up the Java call properly.

Listing 4.3. exampleJava.mo

```
function logJava
  input Real x;
  output Real y;
external "Java"
  y = 'java.lang.Math.log'(x)
  annotation(
    JavaMapping="simple"
  );
end logJava;
```

Listing 4.4. logJava.c

```
logJava_rettype _logJava(modelica_real x)
{
  logJava_rettype out;
  double x_ext;
  double y_ext;
  JNIEnv* __env = NULL;
  jclass __cls = NULL;
  jmethodID __mid = NULL;
  jdouble x_ext_java;
  jdouble y_ext_java;
  x_ext = (double)x;
  __env = getJavaEnv();
  x_ext_java = x_ext;
  __cls = (*__env)->FindClass(__env, "java/lang/Math");
  CHECK_FOR_JAVA_EXCEPTION(__env);
  __mid = (*__env)->GetStaticMethodID(__env, __cls, "log","(D)D");
  CHECK_FOR_JAVA_EXCEPTION(__env);
  y_ext_java = (*__env)->CallStaticDoubleMethod(__env, __cls, __mid, x_ext_java);
  CHECK_FOR_JAVA_EXCEPTION(__env);
  y_ext = y_ext_java;
  (*__env)->DeleteLocalRef(__env, __cls);
  out.targ1 = (modelica_real)y_ext;
  return out;
}
```

4.2.2 Dynamic versus Static Linking

There are two main ways to add the functionality of an external library to a program. They are static and dynamic linking and both have their advantages and disadvantages.

Because the Java libraries are commonly not on a searchable path, you need to specify absolute paths if you link statically (this is true on for example Ubuntu Linux). This causes problems if the JVM library exists on a different path than the system it was compiled on. The external Java runtime library was initially programmed using static linking because this was easier to debug.

Once OMC worked using a statically linked JVM, the code was modified to use dynamic linking. The path to the JVM library needs to be specified using the environment variable `JAVA_HOME`. If `JAVA_HOME` is not specified, the runtime will also try the Windows registry (on Windows only) and /usr/lib/jvm/default-java (on UNIX platforms). An alternative solution would be to use a flag to the compiler `+jvm=/path/to/libjvm.so`. Using an environment variable is "better" because it is inherited by child processes which means you can use `system()` or external Java calls that spawn new OMC shells that also have external Java enabled. It also means tools do not have to be updated since the user can simply add `JAVA_HOME` to his environment. If the runtime cannot find a JVM, it will print an error and `abort()`.

4.2.3 Java Exceptions and Modelica Assertions

It is possible to use `assert(cond,message)` in Modelica. The obvious way to allow assertions to be used from Java is to map exceptions to assertions. This means that after every external Java call, the functions needs to check if an exception was thrown. If an exception occurred, the OpenModelica run-time should use throw an exception. This only works in C++ mode, which means exceptions are only properly handled for simulations. Function calls made in Interactive sessions will simply print the error message and terminate.

4.3 Calling Modelica Functions from Java

In OpenModelica, Modelica files are translated into C++ Simulations (executable) or object files (.o or .dll depending on platform). MetaModelica files can currently only be compiled by RML. The files are translated to object files that do not use C call conventions. The parameters and results are passed using the RML runtime, which is a separate stack. Because the goal is to make OpenModelica capable of handling MetaModelica syntax, it makes sense to ignore RML and write one single Java interface for the OpenModelica Compiler instead. This does however mean that the Java interface requires OpenModelica to be fully bootstrapped before it can be used fully.

OpenModelica communicates with other tools through sockets or CORBA using its Interactive module. The Java interface can do the same, just as the Eclipse plugin (MDT) does. Figure 4.2 shows the existing Java-OpenModelica communication using CORBA. The OMCProxy class does not only communicate with OMC using CORBA. It also starts OMC in server mode if it cannot find a server to communicate with.

The marked nodes in Figure 4.3 are what have been added on top of OMCProxy. SmartProxy only glues OMCProxy and OMCorbaParser together, so the user does not need to be aware that those classes exist. The CORBA interface is an untyped string-to-string function which means you can send {`1,2.0,3`} even though the Modelica standard disallows mixed types in arrays [2] [23].

Listing 4.5 contains an example of an interactive OpenModelica session. The user tells OMC to add a record definition to the AST, and then calls the record constructor. The result is a record.

Listing 4.5. Interactive OMC Session

```
>> record ABC Integer a;Integer b;Integer c; end ABC;
{ABC}
>> ABC(1,2,3)
record ABC
    a = 1,
```

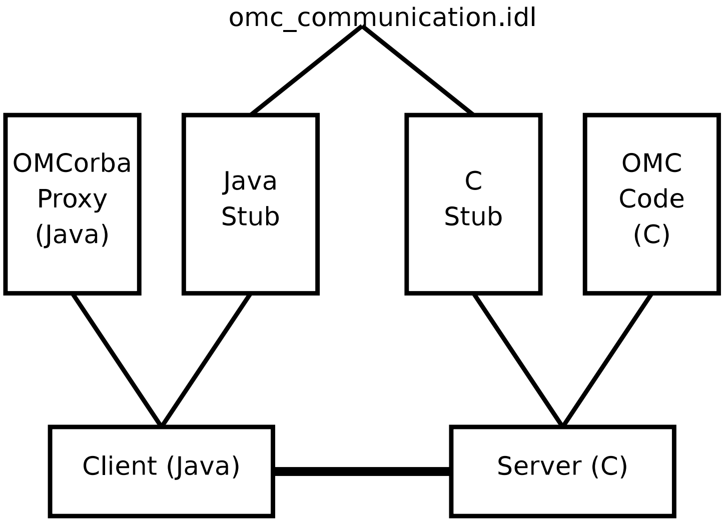

Figure 4.2: CORBA Communication

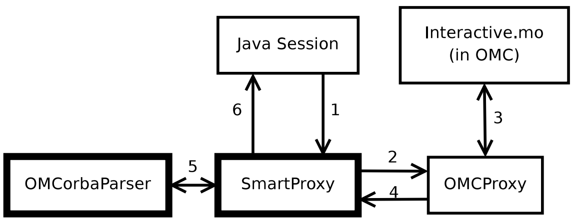

Figure 4.3: Interactive Java Session (data flow)

```
    b = 2,
    c = 3
end ABC;
```

4.3.1 Mapping Textual Representations of MetaModelica Constructs to Java

All Modelica objects implement the dummy Java interface `ModelicaObject`, which helps tagging any Modelica data. Table 4.1 contained the mappings from Modelica types to Java types. The problem with the CORBA interface is that the textual representations are ambiguous. {`1,2,3`} can represent either a MetaModelica list or a Modelica array. `(1,2,3)` can represent either a MetaModelica tuple or multiple function output values. This implementation will treat both cases in the same way. {`1,2,3`} is represented by `ModelicaArray` while `(1,2,3)` is represented by `ModelicaTuple`. Both of these classes extend `java.util.Vector` (which supports both random access and implements the `List` interface).

4.3.2 Parsing CORBA Output

In order to create a reasonably efficient and maintainable parser ANTLRv3 [27] is used to parse the results from the Interactive interface. ANTLRv2 has been used in other parts of OpenModelica with good results, so the choice of parser was quite easy. A shortened version of the grammar used can be found in Listing 4.6. Listings A.3 and A.4 have the full details.

Listing 4.6. OMCorba.g

```
// ANTLRv3 Grammar to parse the corba output from OMC to Java structures
// Java-specific code to construct the datatypes was stripped
grammar OMCorba;

prog: object EOF | EOF;
object: INT | REAL | BOOL | STRING | record | array | tuple | option;

record : 'record' id1=ident
         (field (',' field)*)?
         'end' id2=ident';';
array : '{' '}' | '{' object (',' object)* '}';
tuple : '(' (object (',' object)*)? ')';
option : 'NONE()' | 'SOME(' object ')';
ident : ID | FQID;
field : ID '=' object;

BOOL : 'true'|'false';
FQID : (ID '.')+ ID;
ID : ('_'|'a'..'z'|'A'..'Z')('_'|'a'..'z'|'A'..'Z'|'0'..'9')* |
       '\''(~('\\'|'\'')|'\\\'' | '\\"'| '\\?' | '\\\\' | '\\a' | '\\b' | '\\f' | '\\n' | '\\r' | '\\t' | '\\v')*'\'';
STRING : '"'('\\"'|~'"')*'"';
REAL : '-'? (('.' '0'..'9'+)|('0'..'9'+'.' '0'..'9'*))(('e'|'E')(('+'|'-')?'0'..'9'+))? |
       '-'? '0'..'9'+('e'|'E')(('+'|'-')?'0'..'9'+);
INT :  '-'?'0'..'9'+ ;
WS  :  ('\r'|'\n'|' '|'\t')+ {skip();};
```

What you end up with at this point is an interface that can call Modelica functions, pass Modelica structures and cast the results to the expected type.

This parser translates strings parsed over the OpenModelica interactive interface to the basic Java classes. For example, records are translated to a “generic” record that uses the map interface instead of accessing fields more or less directly). This means you have to write wrapper classes if you want to access these fields without typing lots of code.

When sending an expression from Java to OpenModelica you get back a Java `ModelicaObject`. But if you already know the return type, you do not want to create a lot of code just to cast that object to the expected class. For this reason the Java call `sendModelicaExpression` (and related functions, see Figure 4.4) can take a Java `Class<ModelicaObject>` and after it has parsed the returned data, the function will attempt to cast the object to the expected class. Should the cast fail, it will also try

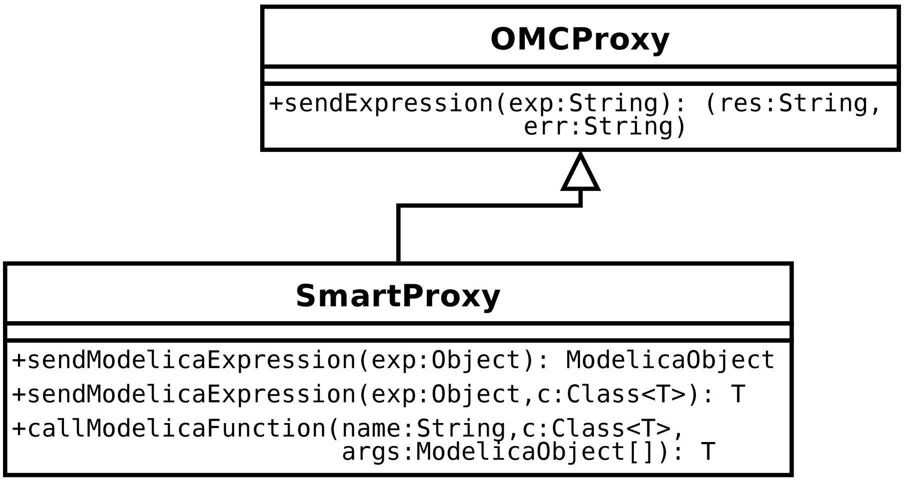

Figure 4.4: CORBA Communication Proxies

to construct a new object of the return type using the object as the argument. Thus, all classes implementing `ModelicaObject` have a constructor taking a single `ModelicaObject` (where it will determine if the object is indeed a supertype of the expected type). This is because any record is parsed as a generic `ModelicaRecord` rather than e.g. `ExpressionRecord`. The `ExpressionRecord` constructor should analyze the `ModelicaObject` and determine if it is indeed a `ModelicaRecord` with the correct record name, field names and data types in the fields. The process of creating this class can be done automatically, see Section 4.3.3 for details on the implementation.

4.3.3 Translating MetaModelica Definitions to Java Classes

Since it would be nice to translate MetaModelica AST definitions to Java AST definitions, in the form of a Java JAR file, a second parser was created. This parser is to be used prior to the application development since it tells OMC to load a number of Modelica files and return an AST containing type definitions, functions, uniontypes and records of the files. Extracting the AST is done by a new API call, `getDefinitions`, in the OpenModelica compiler Interactive module, `Interactive.mo`. The output of the call is a tree in textual prefix notation, similar to LISP syntax. It contains a partial extraction of the syntax tree from the `Absyn` module. Note that the OpenModelica Interactive module uses the `Absyn.Program` AST and not the lowered intermediate tree `SCode.Program` or `DAE` ASTs. Because the AST may contain errors (type checking, syntax, etc), you may get some cryptic error messages in programs containing errors in for example unused functions since RML only compiles referenced functions. The textual extraction format is as follows (Modelica code to textual format to Java code):

Packages

Modelica packages are used to place its parts in its corresponding Java packages.

Modelica: `package myPackage; ... end myPackage;`

Intermediate: `(package myPackage ...)`

Type aliasing

In the example below, all occurrences of `myInt` will eventually be replaced by `ModelicaInteger`. The reason is that Java does not support type aliasing.

Modelica: `type myInt = Integer`

Intermediate: `(type myInt Integer)`

Java: `ModelicaInteger`

Records

Records are transformed into Java classes extending ModelicaRecord. The class has set and get functions for each field in the record. Fields of any extended records are looked up. The Java class will not inherit from a base record class because multiple inheritance is disallowed.

Listing 4.7. Modelica Record

```
record abc
  extends ab;
  Integer c;
end abc;
```

Intermediate: `(record abc (extends ab) (Integer c))`

Java: `class abc extends ModelicaRecord ...`

Replaceable Types

Replaceable types are handled using Java generics.

Modelica: replaceable type T subtypeof Any

Intermediate: (replaceable type T)

Java: <T extends ModelicaObject>

Uniontypes

Uniontypes are tagged using interfaces.

Listing 4.8. MetaModelica Uniontype

```
uniontype ut
  record ab
    Integer a; Integer b;
  end ab;
  record bc
    Integer b; Integer c;
  end bc;
end ut;
```

Intermediate: (uniontype ut) (metarecord ab 0 ut (Integer a) (Integer b)) (metarecord bc 1 ut (Integer b) (Integer c))

Listing 4.9. MetaModelica Uniontype (Java)

```
interface ut extends IModelicaRecord {
}
class ab extends ModelicaRecord implements ut {
    ...
}
class bc extends ModelicaRecord implements ut {
    ...
}
```

Functions

Functions are translated to classes extending ModelicaFunction. The method call performs the actual function call over the CORBA interface. Functions with multiple return values have two call methods, one that returns a ModelicaTuple and one that performs a call-by-reference.

Listing 4.10. Modelica Function

```
function add
  input Integer lhs;
  input Integer rhs;
  output Integer out;
algorithm
  out := lhs+rhs;
end add;
```

Intermediate: `(function abc (input Integer lhs) (input Integer rhs) (output Integer out))`

Listing 4.11. Modelica Function (Java)

```
class  add extends ModelicaFunction {
   ...
   ModelicaInteger call(ModelicaInteger lhs,
      ModelicaInteger rhs) {
     ...
   }
}
```

Partial Functions

Partial functions are undefined function pointers (can also be seen as as types). The Java implementation is essentially an identifier (it discards the in/output).

Listing 4.12. MetaModelica Partial Function

```
partial function addFn
  input Integer lhs;
  input Integer rhs;
  output Integer out;
end addFn;
```

Intermediate: `(partial function addFn)`

Java: `new ModelicaFunctionReference("addFn")`

4.3.4 Translating Two Modelica Functions to Java Classes

The number of steps required to translate a Modelica file into a JAR-file containing all of the definitions is quite large. Figure 4.5 shows the flow of data and the steps are explained through a simple example. The Modelica code in Listing 4.13 will be used as the example for the translation from Modelica code to Java classes.

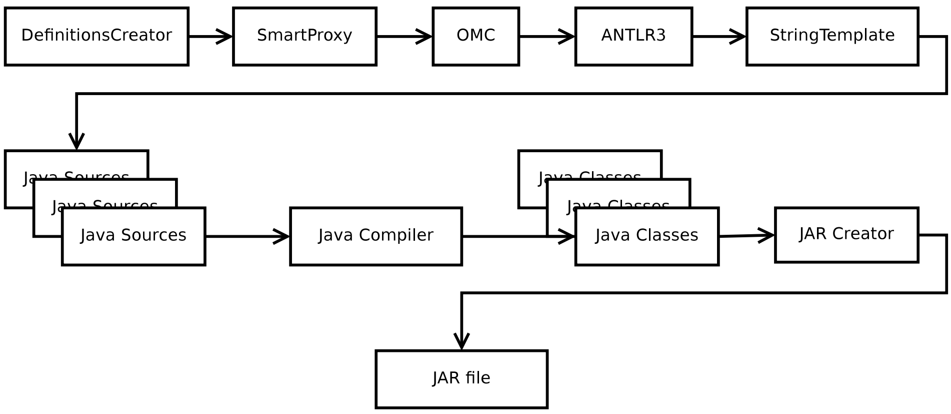

Figure 4.5: DefinitionsCreator data flow

Listing 4.13. Modelica source to be translated to Java

```
package Simple
function AddOne
  input Integer i;
  output Real out;
  Integer one = 1;
algorithm
  out := i+one;
end AddOne;

function AddTwo
  input Integer i;
  output Integer out1;
  output Integer out2;
algorithm
  out1 := i+1;
  out2 := i+2;
end AddTwo;
end Simple;
```

The process starts when you invoke `DefinitionsCreator`. Listing 4.14 shows how to create `~/examples/simple.jar` (with package prefix `org.openmodelica.example`) from `~/examples/Simple.mo`. The inner workings of the class are described below.

Listing 4.14. Invoking DefinitionsCreator

```
$ java -classpath $OPENMODELICAHOME/share/java/antlr-3.1.3:
    $OPENMODELICAHOME/share/java/modelica_java.jar org.
    openmodelica.corba.parser.DefinitionsCreator ~/examples
```

```
/simple.jar org.openmodelica.example ~/examples Simple.
mo
```

The string representation of the definitions in the AST returned by OMC is:

Listing 4.15. getDefinitions String corresponding to the Modelica functions

```
(package Simple
(function AddOne
  (input Integer i)
  (output Real out))
(function AddTwo
  (input Integer i)
  (output Integer out1)
  (output Integer out2))
)
```

By using the OMCorbaDefinitions ANTLRv3 grammar [27] and StringTemplate templates [26], Java source files (Listings 4.16 and 4.17) corresponding to the definitions are created. The primary StringTemplate template used is shown in Listing 4.18.

Listing 4.16. Corresponding Java source for AddOne

```java
public class AddOne extends ModelicaFunction {
  public AddOne (SmartProxy proxy) {
    super("AddOne", proxy);
  }
  public ModelicaReal call (ModelicaInteger i) throws
      ParseException ,ConnectException
  {
    return proxy.callModelicaFunction("Simple.AddOne",
        ModelicaReal.class, i);
  }
}
```

Listing 4.17. Corresponding Java source for AddTwo

```java
public class AddTwo extends ModelicaFunction {
  public AddTwo (SmartProxy proxy) {
    super("AddTwo", proxy);
  }
  public ModelicaTuple call (ModelicaInteger i) throws
      ParseException ,ConnectException
  {
    return proxy.callModelicaFunction("Simple.AddTwo",
        ModelicaTuple.class, i);
  }
```

```
  public void call (ModelicaInteger i, ModelicaInteger out1, ModelicaInteger out2) throws ParseException, ConnectException
  {
    ModelicaTuple __tuple = proxy.callModelicaFunction("Simple.AddTwo", ModelicaTuple.class, i);
    java.util.Iterator<ModelicaObject> __i = __tuple.iterator();
    if (out1 != null) out1.setObject(__i.next()); else __i.next();
    if (out2 != null) out2.setObject(__i.next()); else __i.next();
  }
}
```

Listing 4.18. Template for Modelica functions as Java classes

```
$header()$

@SuppressWarnings("unchecked")
public class $function.name$ extends ModelicaFunction {
  public $function.name$ (SmartProxy proxy) {
    super("$myFQName(var = function.name)$", proxy);
  }

  $if(rest(function.output))$
  public $function.generics$ ModelicaTuple call ($function.input:{$it.TypeName$ input__$it.varName$}; separator = ", "$) throws ParseException,ConnectException
  {
    return super.call(ModelicaTuple.class$if(function.input)$, $endif$$function.input:{input__$it.varName$}; separator=","$);
  }

  public $function.generics$ void call ($function.input:{$it.TypeName$ input__$it.varName$}; separator = ", "$$if(function.input)$, $endif$$function.output:{$it.TypeName$ output__$it.varName$}; separator = ", "$) throws ParseException,ConnectException
  {
    ModelicaTuple __tuple = super.call(ModelicaTuple.class$if(function.input)$, $endif$$function.input:{input__$it.varName$}; separator=","$);
    java.util.Iterator<ModelicaObject> __i = __tuple.iterator();
```

```
    $function.output:{if (output__$it.varName$ != null) output__$it.varName$.setObject(__i.next()); else __i.next();}; separator = "\n"$
  }

$elseif(function.output)$
  public $function.generics$ $first(function.output).TypeName$ call ($function.input:{$it.TypeName$ input__$it.varName$}; separator = ", "$$if(first(function.output).GenericReference)$$if(function.input)$, $endif$Class<$first(function.output).TypeName$> __outClass$endif$) throws ParseException, ConnectException
  {
    return super.call($first(function.output).TypeClass$$if(function.input)$, $endif$$function.input:{input__$it.varName$}; separator=", "$);
  }

$else$
  public $function.generics$ ModelicaVoid call ($function.input:{$it.TypeName$ input__$it.varName$}; separator = ","$) throws ParseException, ConnectException
  {
    return super.call(ModelicaVoid.class$if(function.input)$, $endif$$function.input:{input__$it.varName$}; separator=", "$);
  }

$endif$
}
```

The Java files are compiled using `javac`, the Java Compiler. They are then archived using the `java.util.jar` class. Because StringTemplate is used, the code could potentially be re-targeted in order to create for example C# definitions, but the Java compilation and JAR steps would need to be replaced with functions that could handle C#.

4.3.5 Java Limitations

Java is very restrictive regarding inner classes with regards to reflective programming. While it would be possible to write a class such that a function contains an inner record definition, it is a lot of work and those inner classes will not be put into Java files (which might cause compilation issues for some obscure Modelica files).

Java is quite limited when it comes to generics. Generics in Java is just something that helps the programmer do static type checking. In running code, Java has no concept of generic types and is totally unchecked. This is one of the reasons why `ModelicaTuple` is untyped in Java.

Type definitions/aliases and record extensions are looked up during the translation process. If you define `type abc = Integer` in Modelica, you cannot access the type `abc` in Java. It would be possible to create a class `abc` and extend `ModelicaInteger`, and then create functions that take arguments of the type `abc` as input. The problem is that the Modelica function actually accepts `Integer` as input, while a Java program would only accept one of the types.

Identifiers of arguments in some function calls use a prefix in order to prevent the use of identifiers that are Java keywords.

Chapter 5

Bootstrapping

The bootstrapping branch of the OpenModelica code base was a merge of several branches when work on the external Java implementation started. Some pieces were missing and some had stopped working. Sending ASTs between Java and MetaModelica was one of the goals of the thesis. To verify that the interface is working properly, the `Interactive` module needs to have at least limited support for uniontypes and records.

5.1 Uniontype Implementation

There exists an implementation of the MetaModelica uniontype extension [4]. The problem is that when the code was merged into the bootstrapping branch, it stopped working due to changes in the code. The Interactive.getTotalProgram optimization, which removes unused classes from the abstract syntax before compiling, was disabled because it could not handle MetaModelica datatypes.

Some functions did not have a case for uniontypes. Once cases for uniontype input were added to `ClassInf.start`, `ClassInf.trans`, `Inst.getUsertypeDimensions`, `Inst.daeDeclare4` and `Types.getAllExpsTt`, code generation started working again.

Functions in the C runtime which creates, reads and writes uniontypes were also added. There were three test cases provided, constructing uniontypes, uniontype as function argument and uniontype as function output. They all worked as expected, but the implementation was incomplete and could only handle constant values as arguments to the uniontype constructor.

5.1.1 Improvements to the Uniontype Implementation

`Inst.instElement` had a separate cases for uniontype components and regular components. This was merged a single case handling all components.

`Static.elabCallArgs` had multiple bugs regarding the environment and lookup in the Uniontype constructor case. Fixing these allowed you to use local variables in a uniontype constructor call, as well as calling the constructor from a different scope than the one that the uniontype resides in.

`MetaUtil.createConstantCExp` is supposed to convert constant values to a tagged MetaModelica datatype. It only handled true constant values, but not more complex expression like looking up the value of a variable. It will now construct code like `mmc_mk_icon(i)` instead of `i` (the datatypes are tagged and a pointer is expected).

`MetaUtil.createFunctionArgsList2` did not have a case for uniontype arguments. Fixing this enabled the construction of recursive uniontypes (they expect uniontypes as arguments in the uniontype constructor).

`MetaUtil.createConstantCExp2` Added cases for strings and complex datatypes (records).

`MetaUtil.listToBoxes` takes `Exp.Type` instead of `Exp.Exp` so the same code can be used from `MetaUtil.createConstantCExp2` when using regular records in MetaModelica datatypes.

Furthermore, uniontypes did not work in the Interactive module. In order to ease looking up the name and fields of a uniontype, the boxed uniontypes adds an extra field containing a pointer to a record description. In order to save memory and processing during runtime, this is a pointer to a constant struct in compiled code. The cost is a constant `n` bytes plus 4 bytes for each record created. The gain comes when converting the data to the `Values.Value` structure (which is what the Interactive module uses to store data). If this information was not accessible in the C runtime, you would have to look up field names in the environment. Another advantage is that the debugger (which needs to be rewritten anyway) could access the field names instantly instead of looking up this information.

5.2 Record Arguments and Constructors

In the merge of code bases, record arguments to functions stopped working. Record constructors and records containing other records did not work properly to begin with either.

`Ceval.cevalCallFunction` and `Ceval.cevalFunction` both interpreted the implicit record constructors (and also tried to interpret explicit constructors written by the user). The two functions were disabled because they contained bugs (or incomplete implementations) and since they could not be used within compiled functions. The code that compiles record constructors was improved instead.

`Lookup.buildRecordConstructorClass` adds assignment statements to the implicit constructor so that the result is no longer undefined.

`Static.elabCall` has a case which fixes problems when looking up record constructors of records in packages.

`Codegen.generateFunctions` now returns record definitions already created so they will not be duplicated in `CevalScript.mo`. A similar functionality already existed so the same function prototype would not be declared twice (you get compiler errors if you declare the same type twice in C).

The fields in a record can now contain records and uniontypes.

5.3 Partial Functions

One particular feature of the Modelica language is the partial function. It is a function without any algorithm section. The equivalent in other programming languages (C in particular) would be a type definition of a pointer to a specific function type. While OMC does use partial functions to perform higher-order programming[1] internally, it could not compile such. Thus adding support for partial functions in OMC is part of the bootstrapping process.

The changes made allow (partial) functions to be used as the type of a function variable (argument). It also allows a CALL in the algorithm section to use variables as functions. The code generation was modified to output C code that uses `modelica_fnptr` and `read_modelica_fnptr`. Because the only covered code generation for compiled code (i.e. not the `Interactive` module), the runtime was not modified to add these structures. Simply put, `modelica_fnptr` could be a `void(*)(void)`-style function pointer, but it could perhaps also be a file/function string pair using dynamic linking. The code below assumes it can be cast to a function pointer. A simple model and the generated code can be seen in Listings 5.1 and 5.2. The function ApplyIntOp applies the operation AddInt to the input argument. A more complete work on partial functions will be covered in [5].

Listing 5.1. Partial functions: Model

```
// name:       PartialFn1
// keywords:  PartialFn
// status:    correct
//
// Using function pointers.
//

model M1

function AddInt
  input Integer i;
  output Integer out;
algorithm
  out := i+1;
end AddInt;

function ApplyIntOp

  input FuncIntToInt inFunc;
  input Integer i;
  output Integer outInt;

  partial function FuncIntToInt
```

[1]A higher-order function takes a function as an argument. A common use is to iterate over a list and apply a function to each element in it (listMap).

```
      input Integer in1;
      output Integer out1;
    end FuncIntToInt;

algorithm
    outInt := inFunc(i);
end ApplyIntOp;

Integer i1;
Integer i2;
equation
    i1 = AddInt(1);
    i2 = ApplyIntOp(AddInt,i1);
end M1;

// Result:
// fclass M1
// Integer i1;
// Integer i2;
// equation
//    i1 = 2;
//    i2 = M1.ApplyIntOp(M1.AddInt,i1);
// end M1;
```

Listing 5.2. Partial functions: C code

```
#ifdef __cplusplus
extern "C" {
#endif
/* header part */
#define M1_ApplyIntOp_rettype_1 targ1
typedef struct M1_ApplyIntOp_rettype_s
{
     modelica_integer targ1; /* [] */
} M1_ApplyIntOp_rettype;

M1_ApplyIntOp_rettype _M1_ApplyIntOp(modelica_fnptr inFunc,
     modelica_integer i);

int in_M1_ApplyIntOp(type_description * inArgs,
    type_description * outVar);
/* End of header part */

/* Body */
M1_ApplyIntOp_rettype _M1_ApplyIntOp(modelica_fnptr inFunc,
     modelica_integer i)
```

```
{
  M1_ApplyIntOp_rettype tmp1;
#define inFunc_rettype_1 targ1
  typedef struct inFunc_rettype_s
  {
    modelica_integer targ1;
  } inFunc_rettype;
  inFunc_rettype(*_inFunc)(modelica_integer) = (inFunc_rettype(*)(modelica_integer))inFunc;
  state tmp2;
  modelica_integer outInt;
  inFunc_rettype tmp3;
  tmp2 = get_memory_state();
  tmp3 = _inFunc((modelica_integer)i);
  outInt = tmp3.inFunc_rettype_1;

  _return:
  tmp1.targ1 = outInt;
  restore_memory_state(tmp2);
  return tmp1;
}

int in_M1_ApplyIntOp(type_description * inArgs, type_description * outVar)
{
  modelica_fnptr inFunc;
  modelica_integer i;
  M1_ApplyIntOp_rettype out;
  if(read_modelica_fnptr(&inArgs, &inFunc)) return 1;
  if(read_modelica_integer(&inArgs, &i)) return 1;
  out = _M1_ApplyIntOp(inFunc, i);
  write_modelica_integer(outVar, &out.targ1);
  return 0;
}

/* End Body */

#ifdef __cplusplus
}
#endif
```

Chapter 6

Template-Based Code Generation

6.1 Creating a Template-Based Code Generator

This section covers how the template-based code generator in Section 6.2 was implemented. Template-based code generation would be very benefitial for the OpenModelica Compiler because MetaModelica is very verbose and it is hard to read what strings each function actually outputs. By contrast, a template looks similar to the intended final output. For this thesis, template-based code generation was intended to be used for generating C code for external functions.

In the end, the template-based code generator was not put in use because there were some speed concerns when translating the internal data structure to a more general, string-based, dictionary datatype. There was also a problem with the data structure currently used for code generation. Today, there is a lot of logic embedded in the code generation module. Another template engine, named Susan, is currently being designed. It uses typed data structures instead of the untyped approach below (where all data is eventually expressed as strings).

Since there are no code generators that can receive input straight from MetaModelica you only have two choices. Either you translate your data structure and use some other tool or you write your own Template Engine in MetaModelica. The latter seems simple given that we base the input on a dictionary like [10] [1]. The dictionary datatype is nested, which introduces the possibility to open sub-dictionaries and apply part of the template on them. By allowing this, you can model nested records (e.g. loop over world, then countries in the world and finally cities in the country).

The dictionary lookup supports dotted keys like `$Car.Brand.Name$`. If this was not supported, you would have to write something like `$#Car$ $#Brand$ $Name$ $/# $/#` to "open up" each record and create a new scope for each level of lookup.

[1] This part of the thesis was written before the C data to Java object translation was completed. It would also be possible to translate the structure to Java objects and use StringTemplate as the engine. The performance would have suffered even more, however.

The first approach to the language was purely interpreted, passing and copying lists and strings all around. Although speed was not the primary concern at this point, there is a significant performance boost when compiling the templates to a syntax tree, so a template compiler was written. The input is the textual template and the output is an intermediate form. The templates can be compiled into the intermediate form and stored in Modelica files as constant uniontype trees. This speeds up the template engine significantly if you re-use the template.

The first versions of the language could not use recursion to walk through a dictionary that contained tree structures. This can be worked around for example by flattening the input to a structure with a known depth. For example, the tree in Figure 6.1 can be

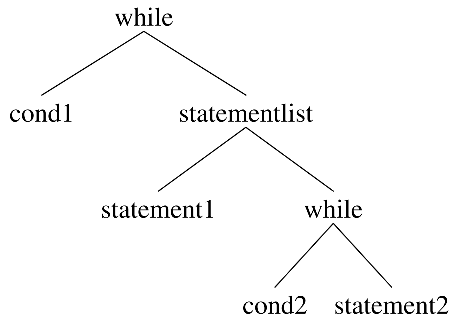

Figure 6.1: Syntax tree for a nested while-statement

flattened to the sequence in Listing 6.1. If it is important to preserve the indentation level, that information can be saved for each line instead of being handles by the function parsing the tree.

Listing 6.1. Flattened syntax tree

```
label tmp1_while_begin
if not cond1 goto tmp1_end_while
statement1
label tmp2_while_begin
if not cond2 goto tmp2_end_while
statement2
goto tmp2_while_begin
goto tmp1_while_begin
label tmp1_end_while
```

Rather than force the user to flatten his syntax tree, the template engine was extended to support a simple kind of recursion. When a scope is opened (for example when iterating over a `FOR_EACH` node), its body is stored. The `RECURSION` node simply uses that body and applies itself over it. This is not as powerful as the constructs available in StringTemplate because you can only use it to recurse over the same structure. For example, the engine works well if you iterate over expression and do something like [2]:

[2] Using StringTemplate-like variable:macro() to apply a macro to the variable, but otherwise using our template engines syntax.

Listing 6.2. Recursion Example 1

```
$=BinOp$ $exp1:exp()$ $op$ $exp2:exp()$ $/=
$=UnOp$ $exp1:exp()$ $/=
```

But it does not work if you iterate over statements and try to do do something like:

Listing 6.3. Recursion Example 2

```
$=Assignment$ $identifier$ = $exp:exp()$ $/=
$=SimpleExpression$ $exp:exp()$ $/=
```

This is because this engine will only have the `statement` scope available (which is actually an anonymous scope). In order to output the expressions the same code would have to be duplicated for each expression.

Instead of duplicating code for statements and expressions you can also use several templates. While generating the model, you send the generator a compiled template that knows how to transform expressions into strings. This approach has more benefits. The templates become smaller and easier to read. They can also be re-used for different target languages (e.g. C, Java and C++ have very similar syntax for integer operations). But you cannot sequentially output the result to file without returning strings and storing them in memory.

One could also use an approach similar to that of StringTemplate. That is, use macro expansion or some method of marking where to "include" another template. By including the templates directly, you have to translate the whole AST into a dictionary without being able to garbage collect it.

The template engine has support for both transforming the code to output in several passes as well as including other templates into a main template.

6.1.1 Modifying MetaModelica to use Template-Based Code Generation

Figure 6.2 is a simplified flowchart of how the new code generation would work. The basic idea is that you convert the AST to a `Dictionary` and fetch the template set that you target uses (for example a template that outputs C code that generates a simulation). Once you have input and template, call the template-based code generator and compile the output. Listings C.4 and D.2 contain templates, AST and output to show the simplicity of such a solution. Note that if MetaModelica had a way of doing reflective programming, such as allowing a function with `Anytype obj` and `String fieldName` input to return the data corresponding to `obj.fieldName`, the AST could be used without having to first translate it.

6.2 Using the Template-Based Code Generator

This section covers how to use the template-based code generator. It covers the format of the dictionary and template input, as well as some short examples. For larger examples,

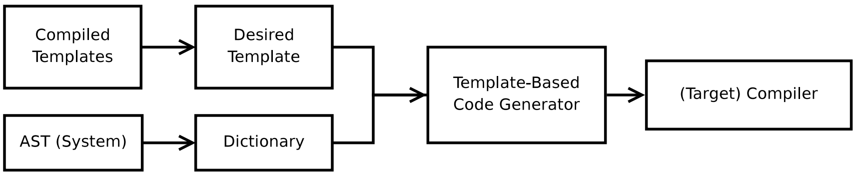

Figure 6.2: Flowchart for the new Code Generation

see Appendix C. Note that the source code in Appendix D has two front-ends, one covering the syntax described below as well as one that is closer to StringTemplate. They use the same interpreter as the back-end.

The focus of the development for this template language was mainly in the back-end, and mostly how to create and do lookup in a syntax tree built by uniontypes. It was also important that the abstract syntax is expressive enough to interpret recursive structures. Because the textual representation is so hard to read, the names of the actual types in the abstract syntax are also provided. The syntax of the language was not important when researching if a template-based code generator was feasible to implement in the present version of OpenModelica.

6.2.1 The Dictionary

The dictionary used for lookup is a `list<DictItem>` (see Listing 6.4). The dictionary is a simple mapping from key to object. The number of datatypes that the dictionary can hold is very limited compared to more advanced engines. The idea is that everything in the model is a boolean, a string, a collection of strings or a nested dictionary (to allow recursive datatypes).

Listing 6.4. Dict.mo

```
uniontype Dict
  record ENABLED
  end ENABLED;

  record STRING_LIST
    list<String> strings;
  end STRING_LIST;

  record STRING
    String string;
  end STRING;

  record DICTIONARY
    list<DictItem> dict;
  end DICTIONARY;

```

```
  record DICTIONARY_LIST
    list<list<DictItem>> dict;
  end DICTIONARY_LIST;
end Dict;

record DictItem
  String key;
  Dict dict;
end DictItem;
```

Listing 6.5. SampleDict.mo

```
constant list<DictItem> whileDict = {
 DictItem("Annotations", STRING_LIST({
  "This is a dictionary containing a while expression.",
  "Why does this example contain annotations?",
  "To try out FOR_EACH of course!"
 })),
 DictItem("ALG_WHILE", ENABLED()),
 DictItem("boolExpr", DICTIONARY({
  DictItem("LBINARY", ENABLED()),
  DictItem("exp1", DICTIONARY({
   DictItem("CREF", ENABLED()),
   DictItem("componentRef", STRING("x"))
  })),
  DictItem("op", DICTIONARY({
   DictItem("LESS", ENABLED())
  })),
  DictItem("exp2", DICTIONARY({
   DictItem("INTEGER", ENABLED()),
   DictItem("value", STRING("20"))
  }))
 })),
 DictItem("whileBody", DICTIONARY_LIST({{
  DictItem("ALG_ASSIGN", ENABLED()),
  DictItem("assignComponent", DICTIONARY({
   DictItem("CREF", ENABLED()),
   DictItem("componentRef", STRING("x"))
  })),
  DictItem("value", DICTIONARY({
   DictItem("BINARY", ENABLED()),
   DictItem("exp1", DICTIONARY({
    DictItem("CREF", ENABLED()),
    DictItem("componentRef", STRING("x"))
   })),
   DictItem("op", DICTIONARY({
```

```
      DictItem("ADD", ENABLED())
    })),
    DictItem("exp2", DICTIONARY({
      DictItem("BINARY", ENABLED()),
      DictItem("exp1", DICTIONARY({
        DictItem("CREF", ENABLED()),
        DictItem("componentRef", STRING("y"))
      })),
      DictItem("op", DICTIONARY({
        DictItem("MUL", ENABLED())
      })),
      DictItem("exp2", DICTIONARY({
        DictItem("INTEGER", ENABLED()),
        DictItem("value", STRING("2"))
      }))
    }))
  }))
 }}))
};
```

6.2.2 Template Syntax

Below are the constructs used in the template language. Each construct contains the identifier used in the compiled template, as well as the character sequence used to construct it. The general idea is that a construct is either built like `$Xkey$body$/X` (where $/X ends X) or `$Xkey$` (where no body is necessary). A fictional Modelica-like template syntax is also introduced. The constructs have a textual description and is followed by a template and output. The dictionary used to apply the templates can be found in Listing 6.5, which corresponds to the expression in Listing 6.6.

Listing 6.6. While Expression

```
while x<20 loop
  x := x+y*2;
end while;
```

A key is an alphanumerical string $((A-Z)|(a-z)|(0-9))^{+}$ [3]. Keys are used for lookup from the dictionary environment. The dictionary environment is simply a set of dictionaries where the current scope has the highest priority. `FOR_EACH` loops and `RECURSION` both change the dictionary environment. If the key contains dots, they are used for nested lookup. Only items of the type `DICTIONARY` can be followed but the last element can be of any type (e.g. `DICT1.DICT2.DICT3`.key).

[3] The implementation does not actually force keys to conform to this regular expression, but restricting yourself to a smaller set of characters is a good idea.

Inserting Text

Text is copied verbatim from template to compiled template with one exception. In order to make the template easier to read, `\n` is required to output a newline character. In order to escape any character, use a single \ before the character in question (or \\ in order to print \).

Abstract syntax:
```
TEXT(text)
```

Example template:
```
This is some text
```

Example output:
```
This is some text
```

Auto-Indentation

The template engine looks for newline characters in the original template and inserts the current indentation level on each new line.

Abstract syntax:
```
INDENT
```

Lookup of a Key Value

If `lookup(dict,key)` returns a string, it is emitted.

Abstract syntax:
```
LOOKUP_KEY(key)
```

Template syntax:
```
$key$
```

Modelica template syntax:
```
$key$
```

Example template:
```
The expression loops while $boolExpr.exp1.componentRef$ <
$boolExpr.exp2.value$.
```

Example output:
```
The expression loops while x < 20.
```

Checking non-empty Attribute Values

If `lookup(dict,key)` returns any non-empty value (empty strings and lists are empty values), run body. Note that the abstract syntax supports more powerful constructs (LISP-style cond expressions) than the ones presented here.

Abstract syntax:
`COND(cond_bodies={(key,true,body)},else_body={})`

Template syntax:
`$=key$body$/=`

Modelica template syntax:
`$if key then body [else body] end if$`

Example template:
`$=ALG_WHILE$This is a while expression.$/=`

Example output:
`This is a while expression.`

Checking non-empty Attribute Values

Check for empty attribute values (simply the opposite of checking for non-empty attribute values).

Abstract syntax:
`COND(cond_bodies={(key,false,body)},else_body={})`

Template syntax:
`$!key$body$/!`

Modelica template syntax:
`$if not key then body [else body] end if$`

Example template:
`$!ALG_ASSIGN$This is not an assignment.$/!`

Example output:
`This is not an assignment.`

For Each Iteration

Use `lookup(dict,key)` to fetch a `STRING_LIST`, `DICTIONARY` or `DICTIONARY_LIST` value, then loop over the elements in the fetched item. Iterating over `DICTIONARY` and `DICTIONARY_LIST` modifies the dictionary environment (it adds the dictionary to the top-most dictionary in use). The (optional) separator is inserted verbatim between the results of each iteration. Iteration is usually used in conjuction with referencing the current value of the iteration, see below.

Abstract syntax:
`FOR_EACH(key,sep,body)`

Template syntax:
`$#key[#sep]$body$/#`

Modelica template syntax:
`$for body in key$` and `$delimit(for this in key, ", ")$`

Current Item Value in Iterations

Only valid when looping over a `STRING_LIST` value. Outputs the current value item string.

Abstract syntax:
`CURRENT_VALUE`

Template syntax:
`$this$`

Modelica template syntax:
`$this$`

Example template:
```
$#Annotations#** $
$this$ \n
$/#
```

Example output:
```
This is a dictionary containing a while expression.
** Why does this example contain annotations?
** To try out FOR_EACH of course!
```

Adding indentation

Opens up a new scope and adds indentation to the current indentation level. This is useful when you want to add indentation only for the first occurance of something you are recursing over. Note that the example uses non-whitespace indentation so it is easier to counter the number of characters in print.

Abstract syntax:
```
ADD_INDENTATION(indent,body)
```

Template syntax:
```
$_indent$body$/_
```

Modelica template syntax:
Do it like StringTemplate (count number of spaces on the line before calling subtemplates and add that to the existing indentation level).

Example template:
```
Some talking points:\n
$_*** $
Why is the MVC concept important for template languages? \n
How do you create an efficient template language? \n
Why is this template language so hard to read? \n
$/_
```

Example output:
```
Some talking points:
*** Why is the MVC concept important for template languages?
*** How do you create an efficient template language?
*** Why is this template language so hard to read?
```

Including a Pre-Compiled Template

When compiling a template you also send the engine a list of keys mapped to pre-compiled templates. Including a template opens up a new scope.

Abstract syntax:
```
ADD_INDENTATION("",body)
```

Template syntax:
```
$:subtemplate$
```

Modelica template syntax:
`$subtemplate()$`

Recursion

Use `lookup(dict,key)` to fetch a `DICTIONARY` or `DICTIONARY_LIST`. It will then use the current scope (from `FOR_EACH` or the global scope) to iterate over the elements from the `DICTIONARY_LIST` as the new top of the dictionary environment (since if it was added on top of the old dictionary you would never break the recursion). The current auto-indentation depth is concatenated with the (optional) indent value.

Abstract syntax:
`RECURSION(key,indent)`

Template syntax:
`$^key[#indent]$body$/^`

Modelica template syntax:
`$subtemplate(this=key)$` *(all subtemplates would need to be named)*

Example template "OP":
```
$=ADD$
+
$/=
$=MUL$
*
$/=
$=LESS$
<
$/=
```

Example template "EXP":
```
$=BINARY$
($^exp1$ $#op$$:OP$$/# $^exp2$)
$/=
$=LBINARY$
($^exp1$ $#op$$:OP$$/# $^exp2$)
$/=
$=INTEGER$
$value$
$/=
$=CREF$
$componentRef$
$/=
```

Example template:

```
$=ALG_WHILE$\n
while ($#boolExpr$$:EXP$$/#) {
$^whileBody# $\n
}
$/=
$=ALG_ASSIGN$
\n$assignComponent.componentRef$ = $#value$$:EXP$$/#;
$/=
```

Example output:

```
while ((x < 20)) {
   x = (x + (y * 2));
}
```

Chapter 7

Discussion and Related Work

7.1 Java External/Interactive Testsuite

The OMC testsuite has been extended with test cases for the external Java interface as well as the interactive Java interface. Because the interactive testsuite tests includes most of the normal external test cases and has the same results, those results are not presented here. The testsuite itself is quite simple. It tests basic Modelica datatypes as in/output, then arrays, records and multiple arguments/output. It then proceeds with the MetaModelica list, option, uniontype and function pointer. The test cases (both Modelica and Java code) can be seen in Appendix A.

The results of the tests can be seen in Listing 7.1. The string “Failed to cast NULL to X” appears a few times. OMC does not always output an error message when a function call fails, but instead returns an empty string. This is the same return value as a function without output (a void function), so the parser thinks the output is valid and tries to cast it to the requested type. For the test cases that do not work (because the OpenModelica cannot handle the constructs in the functions), it is possible to verify that the Java interface is sending the expected strings to OMC.

Listing 7.1. Java Interactive Testsuite Results

```
true
"GetJavaInternalValues"
(2,3.0,"Java function got: Values from OMC")
"RunInteractiveTestsuite
Modelica Constructs:
JavaTest.JavaIntegerToInteger   [OK]
JavaTest.JavaRealToReal         [OK]
JavaTest.JavaBooleanToBoolean   [OK]
JavaTest.JavaStringToString     [OK]
JavaTest.JavaMultipleInOut      [OK]
JavaTest.arrayTestInteger       [OK]
JavaTest.arrayTestReal          [OK]
```

```
JavaTest.arrayTestReal          [OK]
JavaTest.arrayTestBoolean       [OK]
JavaTest.arrayTestString        [OK]
JavaTest.RecordToRecord         [OK]
JavaTest.RecordToString         [OK]
JavaTest.EmptyRecordToString    [OK]
MetaModelica Constructs:
JavaTest.listIntegerIdent       [OK]
JavaTest.someToNone             [OK]
JavaTest.tupleIdent             [OK]
JavaTest.ApplyIntOp             [failed]
Expression JavaTest.ApplyIntOp(JavaTest.
    JavaIntegerToInteger,1) returned an error: \"Error:
    Class inFunc (its type) not found in scope JavaTest.
    ApplyIntOp.
\"
JavaTest.anyToString            [failed]
Expression JavaTest.anyToString(1) returned an error: \"
    Error: No matching function found for JavaTest.
    anyToString(1) of type function(inTypeA:Integer) =>
    String, candidates are function(inTypeA:Type_a type) =>
     String
\"
JavaTest.anyToString            [failed]
Expression JavaTest.anyToString(false) returned an error:
    \"Error: No matching function found for JavaTest.
    anyToString(false) of type function(inTypeA:Boolean) =>
     String, candidates are function(inTypeA:Type_a type)
    => String
\"
JavaTest.uniontypeIdent         [OK]
JavaTest.calcExpressionDummy    [OK]
JavaTest.calcExpressionExtJava  [OK]
JavaTest.calcExpressionMatchcontinue [failed]
Expression JavaTest.calcExpressionMatchcontinue(JavaTest.
    ADD(lhs=JavaTest.ICONST(value=2),rhs=JavaTest.SUB(lhs=
    JavaTest.ICONST(value=5),rhs=JavaTest.ICONST(value=1)))
    )-> Failed to cast NULL to org.openmodelica.
    ModelicaInteger
"
```

7.2 Java Interface Testsuite

The part of the runtime that was written in Java has a jUnit [3] testsuite. It mainly tests that classes can be created and give expected results to certain input. But it also tests that the `DefinitionsCreator` can create jar files from certain parts of the OpenModelica source.

7.3 Performance

Performance was not a big concern in this project, it was mostly about getting things to work. However, it is easy to get poor performance out of either of MetaModelica or Java. This especially true if you are a C programmer since you are not used to immutable datatypes. Appending is an expensive operation because you have to copy all the data for each append operation. I had some performance problems due to using `String` rather than `StringBuffer` in Java (and String concatenation rather than the `Print` module in MetaModelica). This made operations that should takes 10 milliseconds instead take hours (performance in the order of $O(n^2)$ when essentially copying a string is bad when n is $658 * 2^{10}$). In order to verify that no similar scaling problems exist, a simple performance test was performed. Table 7.3 contains some of test cases used in the testsuite. The system used was a single-core 1.5GHz Pentium M running Ubuntu Linux.

Table 7.1. Breakdown Modelica-to-JAR (times in ms)

Defs	OMC	ANTLR	ST	javac	Total	JAR	Speed
0kB	193	47	1393	4851	6967	16kB	2kB/s
32kB	1098	116	1317	18360	20892	210kB	10kB/s
81kB	2248	119	2624	84064	89057	1.3MB	15kB/s
64kB	25555	111	2748	65951	94374	1.2MB	13kB/s
658kB	23475	1594	13414	362809	401586	6.2MB	16kB/s

The size listed is the String returned from OMC, not the size of the Modelica sources. The entries are mostly MetaModelica files of different sizes. The last two entries are the same files (the whole source code of the OpenModelica Compiler), but the smaller one had all functions filtered out in the `Interactive` module. The "speed" measured is $size(jarfile)/time(total)$ and it seems to grow in the order of $O(n)$ for large files. The largest impact on the speed is the Java compiler, and we cannot improve that figure.

7.4 Related Work

Dymola has the capability to call Modelica functions and the Dymola API from external Java functions [17]. Their approach was to use a single entry-point (`com.dynasim.-dymola.interpretMainStatic`). This is probably a bit faster than passing and parsing strings if done correctly in the internals. It would also have been possible to accomplish in OpenModelica, but the solution using CORBA is a bit different in that you do not have to share the same Modelica session as the calling function. The CORBA identifier uses an arbitrary string handle that is unique for each Modelica session. There exists another Dymola external interface that uses Microsoft Dynamic Data Exchange (DDE)

[18]. The DDE interface uses strings for communication with most applications, but it is capable of sending some native datatypes to Matlab [22]. This interface is more akin to the OpenModelica CORBA interface. They are both client-server architectures and they both pass strings. The main difference is that the CORBA interface is cross-platform.

Chapter 8

Conclusions

8.1 Accomplishments

An important overall goal of this thesis work was to be able to send and receive ASTs between Java and OpenModelica. This has been achieved, at least for the datatypes that the OpenModelica Compiler supports.

- OpenModelica should be extended to handle external Java functions.
- Since C, Fortran and Java functions all share a common structure, the OpenModelica code generator should use a more general method, such as template-based code generation, when generating code for external function calls.
- It should be possible to analyze the abstract syntax tree of OpenModelica from a Java application to create a Java mapping of the code loaded in OpenModelica.
- It should be possible to use said mapping to call OpenModelica functions from Java.

Regarding the subgoals, the only one that has not been completely addressed is the one regarding template-based code generation. The MetaModelica implementation used today (RML) is not very suitable to create an efficient template-based code generator. After bootstrapping, the OpenModelica implementation of MetaModelica might be better suited for this purpose. The current input to the `Codegen` module also requires some extra calculations and logic which makes it unsuitable for an MVC-based template-based code generator. Instead of using template-based code generation, the code for external Java functions was written such that it is structurally very similar to external C functions. This will make the transition to a template-based code generator a bit simpler.

The `DefinitionsCreator` class seemed quite fast even when creating Java mappings for huge applications. OMC is around 180,000 lines of Modelica code and the mappings for such an application could be compiled in under 7 minutes. There was no actual performance requirements for any of the work regarding the Java interface, but knowing that it is reasonably efficient is quite important.

The Java interface is quite general and could be used by other Java-based tools to extract data structures more consistently. It also opens up the possibility to complex functions operating on MetaModelica abstract syntax trees in Java (and the other way around for external Java functions).

8.2 Future Work

8.2.1 Bootstrapping

The OpenModelica Compiler is currently being extended to support the datatypes introduced in MetaModelica needed to represent and communicate abstract syntax trees. Another planned extension is to replace the current text-based CORBA interface with a directly linked version, giving higher performance.

As work progresses, support for new datatypes needs to be added in the Interactive module since the CORBA interface depends on this module being updated. Most of the work so far has been limited to compiling code using these datatypes (e.g. the uniontype implementation [4]).

8.2.2 MetaModelica External Java

The current implementation of external Java functions only supports the standard Modelica datatypes, uniontypes and the option type. Once external C functions support the remaining MetaModelica extensions (tuple and list), it should be trivial to implement the same for Java functions. The C and Java runtime libraries can already transform MetaModelica C datatypes to Java objects and the other way around. However, a few cases need to be added in `Codegen.mo` and test cases need to be written. Note that since work is being done on OMC, it is possible that support already exists for the remaining types by the time this document is published.

8.2.3 Uniontype Inheritance Hierarchies

MetaModelica could be extended to allow a uniontypes to form type hierarchies by inheriting existing uniontypes. That is, an Expression could be the union of SimpleExpression and PointerExpression. Today, one would model it as:

Listing 8.1. MetaModelica Hierarchy

```
uniontype Expression
  record SIMPLE_EXPRESSION
    SimpleExpression ex;
  end SIMPLE_EXPRESSION;
  record POINTER_EXPRESSION
    PointerExpression ex;
  end POINTER_EXPRESSION;
end Expression;
```

It would make sense to instead be able to model it as something similar to:

Listing 8.2. New MetaModelica Hierarchy

```
uniontype Expression
  extends SimpleExpression;
```

```
  extends PointerExpression;
end Expression;
```

Because interfaces in Java support multiple inheritance this is simpler than the workaround used to model records in Java (where you had to determine all members and extend ModelicaRecord). In Java you cannot declare an interface as the union of two others, so it needs to be done "backwards". Make the extended uniontypes extend the parent interface as such:

Listing 8.3. Java representation of MetaModelica Hierarchy

```
public interface Expression extends ModelicaObject {}
public interface SimpleExpression extends ModelicaObject,
    Expression {}
public interface ComplexExpression extends ModelicaObject,
    Expression {}
public interface VerySimpleExpression extends
    ModelicaObject,SimpleExpression {}
```

8.2.4 Template Engine

Because of the changes made to the internal representation of uniontypes OMC (they include a pointer to a `record_description`), it would be easier to implement an efficient template engine once OMC supports the full MetaModelica language. Uniontypes now stores a pointer to record and field names. It would be possible to create a generic `lookup(anyType,key)` function in the runtime libraries (external C functions). This would eliminate the need to traverse the whole syntax tree just to convert all the nodes to strings (which takes a lot of time when you have syntax trees that can reach a size of 200MB).

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools.* Pearson, 2007.

[2] Modelica Association. The modelica language specification version 3.0, 2007. Available from World Wide Web: `http://www.modelica.org/`.

[3] Kent Beck, Erich Gamma, and David Saff. junit 4.6, 2009. Available from World Wide Web: `http://junit.org`.

[4] Simon Björklén. Extending modelica with high-level data structures: Design and implementation in openmodelica. Master's thesis, Linköping University, June 2008. Available from World Wide Web: `http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-12148`.

[5] Stefan Brus. Bootstrapping the openmodelica compiler: Implementing functions as arguments. Technical report, Linköping University, Department of Computer and Information Science, 2009. Not yet published.

[6] Apache Software Foundation. Velocity user guide, 2008. Available from World Wide Web: `http://velocity.apache.org/engine/releases/velocity-1.6.1/user-guide.html`.

[7] FreeMarker, 2009. Available from World Wide Web: `http://freemarker.org/`.

[8] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley, 2004.

[9] GlueGen, 2007. Available from World Wide Web: `https://gluegen.dev.java.net/`.

[10] Google. ctemplate, 2008. Available from World Wide Web: `http://code.google.com/p/google-ctemplate/`.

[11] Object Management Group. Metaobject facility, 2009. Available from World Wide Web: `http://www.omg.org/mof/`.

[12] Object Management Group. Object management group, 2009. Available from World Wide Web: `http://www.omg.org/`.

[13] Jack Herrington. Code generation in xslt 2.0, 2005. Available from World Wide Web: `http://www.ibm.com/developerworks/xml/library/x-xslphp1/index.html`.

[14] JNA. Java native access, 2008. Available from World Wide Web: `https://jna.dev.java.net/`.

[15] Alberto Leva, Filippo Donida, Francesco Casella, and more. Simforge, 2009. Available from World Wide Web: `http://trac.elet.polimi.it/simforge/`.

[16] Sheng Liang. *The Java^{TM}Native Interface*. Addison-Wesley, 1999. Available from World Wide Web: http://java.sun.com/docs/books/jni/.

[17] José Díaz López and Hans Olsson. Dymola interface to java - a case study: Distributed simulations. *Proceedings of the 5th International Modelica Conference, Vienna, Austria, September 4–5*, 2:477–483, 2006.

[18] Microsoft. About dynamic data exchange, 2009. Available from World Wide Web: http://msdn.microsoft.com/en-us/library/ms648774.aspx.

[19] Sun Microsystems. Java metadata interface, 2002. Available from World Wide Web: http://java.sun.com/products/jmi/.

[20] Chuck Mosher. A new specification for managing metadata, 2002. Available from World Wide Web: http://java.sun.com/developer/technicalArticles/J2EE/JMI/.

[21] Giuseppe Naccarato. Template-based code generation with apache velocity, 2004. Available from World Wide Web: http://www.onjava.com/pub/a/onjava/2004/05/05/cg-vel1.html.

[22] Hans Olsson. External interface to modelica in dymola. *Proceedings of the 4th International Modelica Conference, Hamburg, Germany, March 7–8*, 2:603–611, 2005.

[23] OpenModelica. Openmodelica system documentation, 2008. Available from World Wide Web: http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/releases/1.4.5/doc/OpenModelicaUsersGuide.pdf.

[24] OpenModelica, 2009. Available from World Wide Web: http://www.ida.liu.se/labs/pelab/modelica/OpenModelica.html.

[25] Terence Parr. Enforcing strict model-view separation in template engines, 2004. Available from World Wide Web: http://www.cs.usfca.edu/~parrt/papers/mvc.templates.pdf.

[26] Terence Parr. About the stringtemplate template engine, 2009. Available from World Wide Web: http://www.stringtemplate.org/about.html.

[27] Terence Parr. Antlr parser generator, 2009. Available from World Wide Web: http://www.antlr.org/.

[28] Terence Parr. Lecture notes: Using stringtemplate to generate web pages, 2009. Available from World Wide Web: http://www.cs.usfca.edu/~parrt/course/601/lectures/stringtemplate.html.

[29] SWIG, 2009. Available from World Wide Web: http://www.swig.org/.

Appendix A

Testsuite Source

Listing A.1. Java External Testsuite (.mo)

```
package JavaTest

record myEmptyRecord
end myEmptyRecord;

record myRecord
  Integer a;
  Real b;
  Boolean c;
  String d;
end myRecord;

record nestedRecord
  myRecord rec;
  String desc;
end nestedRecord;

function NestedRecordExtIdent
  input nestedRecord rec;
  output nestedRecord out;
  external "Java" 'JavaExt.RecordToRecord'(rec,out);
end NestedRecordExtIdent;

function RecordToRecord
  input myRecord inRecord;
  output myRecord out;
  external "Java" 'JavaExt.RecordToRecord'(inRecord,out);
end RecordToRecord;
```

```

function RecordToString
  input myRecord inRecord;
  output String out;
  external "Java" out='JavaExt.RecordToString'(inRecord);
end RecordToString;

function EmptyRecordToString
  input myEmptyRecord inRecord;
  output String out;
  external "Java" out='JavaExt.RecordToString'(inRecord);
end EmptyRecordToString;

function SumArray
  input Integer x[:];
  output Integer y;
  external "Java" y='JavaExt.SumArray'(x);
end SumArray;

function arrayTestInteger
  input Integer x[:];
  output Integer y[size(x,1)];
  external "Java" 'JavaExt.testArrays'(x,y);
end arrayTestInteger;

function arrayTestReal
  input Real x[:,:,:];
  output Real y[size(x,1),size(x,2),size(x,3)];
  external "Java" 'JavaExt.testArrays'(x,y);
end arrayTestReal;

function arrayTestBoolean
  input Boolean x[:];
  output Boolean y[size(x,1)];
  external "Java" 'JavaExt.testArrays'(x,y);
end arrayTestBoolean;

function arrayTestString
  input String xstr[:];
  output String ystr[size(xstr,1)];
  external "Java" 'JavaExt.testArrays'(xstr,ystr);
end arrayTestString;

function JavaIntegerToInteger
  input Integer o;
  output Integer out;
```

```
  external "Java" out='JavaExt.IntegerToInteger'(o);
end JavaIntegerToInteger;

function JavaRealToReal
  input Real o;
  output Real out;
  external "Java" out='JavaExt.RealToReal'(o);
end JavaRealToReal;

function JavaStringToString
  input String o;
  output String out;
  external "Java" out='JavaExt.StringToString'(o);
end JavaStringToString;

function JavaBooleanToBoolean
  input Boolean o;
  output Boolean out;
  external "Java" out='JavaExt.BooleanToBoolean'(o);
end JavaBooleanToBoolean;

function JavaMultipleInOut
  input Real i0;
  input Real i1;
  input Real i2;
  input Real i3;
  output Real o0;
  output Real o1;
  output Real o2;
  output Real o3;
  external "Java" o3='JavaExt.MultipleIO'(
      i0,i1,i2,i3,o0,o1,o2);
end JavaMultipleInOut;

function GetOMCInternalValues
  input Integer in_i;
  input Real in_r;
  output Integer out_i;
  output Real out_r;
  output String out_s;
algorithm
  out_i := in_i+1;
  out_r := in_r+1.5;
  out_s := "Values from OMC";
end GetOMCInternalValues;
```

```
function GetJavaInternalValues
  input Integer in_i;
  input Real in_r;
  output Integer out_i;
  output Real out_r;
  output String out_s;
  external "Java" 'JavaExt.GetValuesFromOMCThroughJava'(
      in_i,in_r,out_i,out_r,out_s);
end GetJavaInternalValues;

function RunInteractiveTestsuite
  output String out;
  external "Java" out='JavaExt.RunInteractiveTestsuite'();
end RunInteractiveTestsuite;

/* MetaModelica / Interactive Tests */

function listIntegerIdent
  input list<Integer> lst;
  output list<Integer> out;
algorithm
  out := lst;
end listIntegerIdent;

function someToNone
  input Option<Integer> opt;
  output Option<Integer> out;
algorithm
  out := NONE();
end someToNone;

function tupleIdent
  input tuple<Integer,Integer> tup;
  output tuple<Integer,Integer> out;
algorithm
  out := tup;
end tupleIdent;

function ApplyIntOp
  input FuncIntToInt inFunc;
  input Integer i;
  output Integer outInt;

  partial function FuncIntToInt
    input Integer in1;
    output Integer out1;
```

```
  end FuncIntToInt;
algorithm
  outInt := inFunc(i);
end ApplyIntOp;

function anyToString
  input Type_a inTypeA;
  output String out;
  replaceable type Type_a subtypeof Any;
algorithm
  out := "OK";
end anyToString;

uniontype fruit
  record APPLE
  end APPLE;
  record PEAR
  end PEAR;
end fruit;

function uniontypeIdent
  input fruit in1;
  output fruit out;
algorithm
  out := in1;
end uniontypeIdent;

uniontype Expression
  record ADD
    Expression lhs;
    Expression rhs;
  end ADD;
  record SUB
    Expression lhs;
    Expression rhs;
  end SUB;
  record ICONST
    Integer value;
  end ICONST;
  record RCONST
    Real value;
  end RCONST;
  record IFEXP
    Boolean cond; // Simple test
    Expression trueExp;
    Expression falseExp;
```

```
  end IFEXP;
  record STRLEN
    String str;
  end STRLEN;
end Expression;

function calcExpressionDummy
  input Expression exp;
  output Integer out;
algorithm
  out := 6; // Because matchcontinue is not working yet
end calcExpressionDummy;

function calcExpressionMatchcontinue
  input Expression exp;
  output Integer out;
algorithm
  out := matchcontinue(exp)
  local
    Expression lhs,rhs;
    Integer lval,rval;
  case ADD(lhs,rhs)
    equation
      lval = calcExpressionMatchcontinue(lhs);
      rval = calcExpressionMatchcontinue(rhs);
    then lval+rval;
  case SUB(lhs,rhs)
    equation
      lval = calcExpressionMatchcontinue(lhs);
      rval = calcExpressionMatchcontinue(rhs);
    then lval-rval;
  case ICONST(rval)
    then rval;
  end matchcontinue;
end calcExpressionMatchcontinue;

function calcExpressionExtJava
  input Expression exp;
  output Real out;
  external "Java" out='JavaExt.calcExpression'(exp);
end calcExpressionExtJava;

function expIdentExtJava
  input Expression in1;
  output Expression out;
  external "Java" out='JavaExt.expIdent'(in1);
```

```
end expIdentExtJava;

function expIdentExtJava2
  input Expression in1;
  output Expression out;
  external "Java" 'JavaExt.expIdent'(in1,out);
end expIdentExtJava2;

function extJavaTestAllMMCTypes
  output Expression out;
  external "Java" out='JavaExt.testAllMMCTypes'();
end extJavaTestAllMMCTypes;

end JavaTest;
```

Listing A.2. Java External Testsuite (.java)

```
import org.openmodelica.*;
import org.openmodelica.corba.SmartProxy;
import static org.openmodelica.corba.parser.OMCStringParser
    .parse;
import org.openmodelica.JavaExtTest.JavaTest.myRecord;
import org.openmodelica.JavaExtTest.JavaTest.myEmptyRecord;
import org.openmodelica.JavaExtTest.JavaTest.APPLE;
import org.openmodelica.JavaExtTest.JavaTest.Expression;
import org.openmodelica.JavaExtTest.JavaTest.ICONST;
import org.openmodelica.JavaExtTest.JavaTest.RCONST;
import org.openmodelica.JavaExtTest.JavaTest.IFEXP;
import org.openmodelica.JavaExtTest.JavaTest.ADD;
import org.openmodelica.JavaExtTest.JavaTest.STRLEN;
import org.openmodelica.JavaExtTest.JavaTest.SUB;
import java.io.*;

public class JavaExt {

public static ModelicaInteger SumArray(ModelicaArray<
    ModelicaInteger> iarr) {
  ModelicaInteger sum = new ModelicaInteger(0);
  for (ModelicaInteger mi : iarr) {
    sum.i += mi.i;
  }
  return sum;
}

public static ModelicaObject ObjectToObject(ModelicaObject
    mo) {
```

```java
  if (mo instanceof ModelicaReal) {
    ModelicaReal mr = (ModelicaReal) mo;
    return RealToReal(mr);
  } else if (mo instanceof ModelicaInteger) {
    ModelicaInteger mi = (ModelicaInteger) mo;
    return IntegerToInteger(mi);
  } else if (mo instanceof ModelicaBoolean) {
    ModelicaBoolean mb = (ModelicaBoolean) mo;
    return BooleanToBoolean(mb);
  } else if (mo instanceof ModelicaString) {
    ModelicaString ms = (ModelicaString) mo;
    return StringToString(ms);
  } else
    throw new RuntimeException("ObjectToObject failed: " + mo);
}

public static ModelicaInteger IntegerToInteger(ModelicaInteger mi) {
  return new ModelicaInteger(mi.i * 2);
}

public static ModelicaReal RealToReal(ModelicaReal mr) {
  return new ModelicaReal(mr.r * 2.5);
}

public static ModelicaBoolean BooleanToBoolean(ModelicaBoolean mb) {
  return new ModelicaBoolean(!mb.b);
}

public static ModelicaString StringToString(ModelicaString ms) {
  return new ModelicaString(ms.s + ":" + ms.s);
}

public static void testArrays(ModelicaArray<?> marr, ModelicaArray<?> marr2) {
  marr2.setObject(marr);
  marr2.unflattenModelicaArray();

  marr2.flattenModelicaArray();
  for (ModelicaObject mo : marr2) {
    mo.setObject(ObjectToObject(mo));
  }

```

```
  marr2.unflattenModelicaArray();
}

public static ModelicaReal MultipleIO(ModelicaReal i0,
    ModelicaReal i1, ModelicaReal i2, ModelicaReal i3,
    ModelicaReal o0, ModelicaReal o1, ModelicaReal o2) {
  o0.r = i0.r*1.5;
  o1.r = i1.r*2.5;
  o2.r = i2.r*3.5;
  return new ModelicaReal(i3.r*4.5);
}

public static ModelicaString RecordToString(ModelicaRecord
    rec) {
  return new ModelicaString(rec.toString());
}

public static void RecordToRecord(ModelicaRecord rec,
    ModelicaRecord out) {
  // System.out.println("rec: "+rec.get_ctor_index()+": "+
      rec);
  // System.out.println("out: "+out.get_ctor_index()+": "+
      out);
  out.setObject(rec);
}

public static ModelicaReal calcExpression(IModelicaRecord
    rec) throws Exception
{
  Expression exp = ModelicaAny.cast(rec, Expression.class);
  if (exp instanceof ADD) {
    ADD add = (ADD) exp;
    return new ModelicaReal(calcExpression(add.get_lhs()).r
        + calcExpression(add.get_rhs()).r);
  }
  if (exp instanceof SUB) {
    SUB sub = (SUB) exp;
    return new ModelicaReal(calcExpression(sub.get_lhs()).r
        - calcExpression(sub.get_rhs()).r);
  }
  if (exp instanceof ICONST) {
    ICONST iconst = (ICONST) exp;
    return new ModelicaReal(iconst.get_value().i);
  }
  if (exp instanceof RCONST) {
    RCONST rconst = (RCONST) exp;
```

```
      return rconst.get_value();
    }
    if (exp instanceof IFEXP) {
      IFEXP ifexp = (IFEXP) exp;
      if (ifexp.get_cond().b)
        return calcExpression(ifexp.get_trueExp());
      else
        return calcExpression(ifexp.get_falseExp());
    }
    if (exp instanceof STRLEN) {
      STRLEN strlen = (STRLEN) exp;
      return new ModelicaReal(strlen.get_str().s.length());
    }
    throw new Exception("Unknown Modelica Expression : " +
        exp);
  }

  public static IModelicaRecord expIdent(IModelicaRecord rec)
      throws Exception
  {
    Expression exp = ModelicaAny.cast(rec, Expression.class);
    return exp;
  }

  public static void expIdent(IModelicaRecord rec,
      IModelicaRecord out) throws Exception
  {
    Expression exp = ModelicaAny.cast(rec, Expression.class);
    out.setObject(exp);
  }

  /** Extend the ModelicaRecord so we can return whatever we
      want without warnings :)
   */
  static class DummyRecordDoNotUse extends ModelicaRecord {
    public DummyRecordDoNotUse(ModelicaOption<?> none,
        ModelicaOption<?> some, ModelicaArray<?> arr) throws
        ModelicaRecordException {
      super("dummy", new String[]{"none", "some", "arr"},
          none, some, arr);
    }
    @Override
    public int get_ctor_index() {
      return 3000;
    }
  }
```

```java

public static IModelicaRecord testAllMMCTypes() throws Exception
{
  ModelicaInteger mi = new ModelicaInteger(1);
  ModelicaReal mr = new ModelicaReal(2.5);
  ModelicaBoolean mb = new ModelicaBoolean(false);
  ModelicaString ms = new ModelicaString("OpenModelica Test");
  ModelicaTuple tup = new ModelicaTuple(mi,mr,mb,ms);
  ModelicaOption none = new ModelicaOption(null);
  ModelicaOption some = new ModelicaOption(tup);
  ModelicaArray<ModelicaObject> arr = new ModelicaArray<ModelicaObject>(mi,mr,mb,ms);
  return new DummyRecordDoNotUse(none, some, arr);
}

public static void DummyTest(ModelicaObject obj)
{
  System.out.println(obj.getClass().getName() + ": " + obj.toString());
}

private static org.openmodelica.JavaExtTest.JavaTest.GetOMCInternalValues GetOMCInternalValues;
private static SmartProxy proxy;

// Do this by a separate call because the file is used by several test cases
private static void setProxy() throws Exception {
  if (GetOMCInternalValues != null)
    return;

  proxy = new SmartProxy("JavaExtTest", "MetaModelica", true, false);
  // The spawned OMC shell can be in somewhat random locations...
  proxy.sendExpression("cd(\""+System.getProperty("user.dir")+"\")");
  proxy.sendExpression("loadFile(\"JavaExt.mo\")");
  GetOMCInternalValues = new org.openmodelica.JavaExtTest.JavaTest.GetOMCInternalValues(proxy);
}

public static void GetValuesFromOMCThroughJava(ModelicaInteger in_i, ModelicaReal in_r,
```

```
    ModelicaInteger out_i, ModelicaReal out_r,
    ModelicaString out_s) throws Exception {
  PrintStream out = System.out;
  ByteArrayOutputStream baos = new ByteArrayOutputStream();
  System.setOut(new PrintStream(baos, false));

  setProxy();
  GetOMCInternalValues.call(in_i, in_r, out_i, out_r, out_s);
  out_s.s = "Java function got: " + out_s.s;

  System.setOut(out);
}

private static String TestFunction(String fnname, Class<?
    extends ModelicaObject> c, ModelicaObject expected,
    ModelicaObject... args) {
  try {
  ModelicaObject res = proxy.callModelicaFunction(fnname, c, args);

  if (expected.equals(res))
    return String.format("%-30s [OK]\n", fnname);
  return String.format("%-30s [failed]\nWrong result: %s !=
     %s\n", fnname, expected.toString(), res.toString());
  } catch (Throwable t) {
    return String.format("%-30s [failed]\n%s\n", fnname, t.
       getMessage()); // ModelicaHelper.getStackTrace(t));
  }
}

public static ModelicaString RunInteractiveTestsuite()
    throws Exception {
  PrintStream out = System.out;
  ByteArrayOutputStream baos = new ByteArrayOutputStream();
  System.setOut(new PrintStream(baos, false));
  String res = "RunInteractiveTestsuite\n";

  try {
  setProxy();
  res += "Modelica Constructs:\n";

  res += TestFunction(
    "JavaTest.JavaIntegerToInteger", ModelicaInteger.class,
    new ModelicaInteger(2),
    new ModelicaInteger(1));
```

```

  res += TestFunction(
    "JavaTest.JavaRealToReal", ModelicaReal.class,
    new ModelicaReal(2.5),
    new ModelicaReal(1));

  res += TestFunction(
    "JavaTest.JavaBooleanToBoolean", ModelicaBoolean.class,
    new ModelicaBoolean(true),
    new ModelicaBoolean(false));

  res += TestFunction(
    "JavaTest.JavaStringToString", ModelicaString.class,
    new ModelicaString("Test:Test"),
    new ModelicaString("Test"));

  res += TestFunction(
    "JavaTest.JavaMultipleInOut", ModelicaTuple.class,
    new ModelicaTuple(new ModelicaReal(1.5), new
        ModelicaReal(5.0), new ModelicaReal(10.5), new
        ModelicaReal(18.0)),
    new ModelicaReal(1.0), new ModelicaReal(2.0), new
        ModelicaReal(3.0), new ModelicaReal(4.0));

  res += TestFunction(
    "JavaTest.arrayTestInteger", ModelicaArray.class,
    new ModelicaArray<ModelicaInteger>(new ModelicaInteger
        []{
      new ModelicaInteger(2),
      new ModelicaInteger(4),
      new ModelicaInteger(6)
    }),
    new ModelicaArray<ModelicaInteger>(new ModelicaInteger
        []{
      new ModelicaInteger(1),
      new ModelicaInteger(2),
      new ModelicaInteger(3)
    }));

  res += TestFunction(
     "JavaTest.arrayTestReal", ModelicaArray.class,
     ModelicaArray.createMultiDimArray(new ModelicaReal[]{
       new ModelicaReal(2.5),
       new ModelicaReal(5),
       new ModelicaReal(7.5)
     },1,1,3),
```

```
      ModelicaArray.createMultiDimArray(new ModelicaReal[]{
        new ModelicaReal(1),
        new ModelicaReal(2),
        new ModelicaReal(3)
      },1,1,3)));

  res += TestFunction(
    "JavaTest.arrayTestReal", ModelicaArray.class,
    ModelicaArray.createMultiDimArray(new ModelicaReal[]{
      new ModelicaReal(2.5),
      new ModelicaReal(5),
      new ModelicaReal(7.5)
    },1,1,3),
    ModelicaArray.createMultiDimArray(new ModelicaReal[]{
      new ModelicaReal(1),
      new ModelicaReal(2),
      new ModelicaReal(3)
    },1,1,3));

  res += TestFunction(
    "JavaTest.arrayTestBoolean", ModelicaArray.class,
    ModelicaArray.createMultiDimArray(new ModelicaBoolean[]{
      new ModelicaBoolean(true),
      new ModelicaBoolean(false),
      new ModelicaBoolean(true)
    },3),
    ModelicaArray.createMultiDimArray(new ModelicaBoolean[]{
      new ModelicaBoolean(false),
      new ModelicaBoolean(true),
      new ModelicaBoolean(false)
    },3));

  res += TestFunction(
    "JavaTest.arrayTestString", ModelicaArray.class,
    ModelicaArray.createMultiDimArray(new ModelicaString[]{
      new ModelicaString("1:1"),
      new ModelicaString("2:2"),
      new ModelicaString("3:3")
    },3),
    ModelicaArray.createMultiDimArray(new ModelicaString[]{
      new ModelicaString("1"),
      new ModelicaString("2"),
      new ModelicaString("3")
    },3));
```

```java

    res += TestFunction(
      "JavaTest.RecordToRecord", myRecord.class,
      new myRecord(new ModelicaInteger(1), new ModelicaReal(1.5), new ModelicaBoolean(true), new ModelicaString("test")),
      new myRecord(new ModelicaInteger(1), new ModelicaReal(1.5), new ModelicaBoolean(true), new ModelicaString("test")));

    res += TestFunction(
      "JavaTest.RecordToString", ModelicaString.class,
      new ModelicaString("JavaTest.myRecord(a=1,b=1.5,c=true,d=\"test\")"),
      new myRecord(new ModelicaInteger(1), new ModelicaReal(1.5), new ModelicaBoolean(true), new ModelicaString("test")));

    res += TestFunction(
      "JavaTest.EmptyRecordToString", ModelicaString.class,
      new ModelicaString("JavaTest.myEmptyRecord()"),
      new myEmptyRecord());

    res += "MetaModelica Constructs:\n";

    res += TestFunction( // lists are the same as arrays for the Java implementation
      "JavaTest.listIntegerIdent", ModelicaArray.class,
      new ModelicaArray<ModelicaInteger>(new ModelicaInteger[]{new ModelicaInteger(1), new ModelicaInteger(2)}),
      new ModelicaArray<ModelicaInteger>(new ModelicaInteger[]{new ModelicaInteger(1), new ModelicaInteger(2)}));

    res += TestFunction(
      "JavaTest.someToNone", ModelicaOption.class,
      new ModelicaOption<ModelicaInteger>(null),
      new ModelicaOption<ModelicaInteger>(new ModelicaInteger(1)));

    res += TestFunction(
      "JavaTest.tupleIdent", ModelicaTuple.class,
      new ModelicaTuple(new ModelicaInteger[]{new ModelicaInteger(1), new ModelicaInteger(2)}),
```

```
    new ModelicaTuple(new ModelicaInteger[]{new
        ModelicaInteger(1), new ModelicaInteger(2)}));

  res += TestFunction(
    "JavaTest.ApplyIntOp", ModelicaArray.class,
    new ModelicaInteger(2),
    new org.openmodelica.JavaExtTest.JavaTest.
        JavaIntegerToInteger(proxy).getReference(),
    new ModelicaInteger(1));

  res += TestFunction(
    "JavaTest.anyToString", ModelicaString.class,
    new ModelicaString("OK"),
    new ModelicaInteger(1));

  res += TestFunction(
    "JavaTest.anyToString", ModelicaString.class,
    new ModelicaString("OK"),
    new ModelicaBoolean(false));

  res += TestFunction(
    "JavaTest.uniontypeIdent", APPLE.class,
    new APPLE(),
    new APPLE());

  Expression exp = parse("record JavaTest.ADD\n" +
"  lhs = record JavaTest.ICONST\n" +
"  value = 2\n" +
"end JavaTest.ICONST;,\n" +
"  rhs = record JavaTest.SUB\n" +
"  lhs = record JavaTest.ICONST\n" +
"  value = 5\n" +
"end JavaTest.ICONST;,\n" +
"  rhs = record JavaTest.ICONST\n" +
"  value = 1\n" +
"end JavaTest.ICONST;\n" +
"end JavaTest.SUB;\n" +
"end JavaTest.ADD;\n", Expression.class);

  res += TestFunction(
    "JavaTest.calcExpressionDummy", ModelicaInteger.class,
    new ModelicaInteger(6),
    exp);

  res += TestFunction(
    "JavaTest.calcExpressionExtJava", ModelicaReal.class,
```

```
      new ModelicaReal(6.0),
      exp);

    res += TestFunction(
      "JavaTest.calcExpressionMatchcontinue", ModelicaInteger
          .class,
      new ModelicaInteger(6),
      exp);

    proxy.stopServer();

    } catch (Exception ex) {
      res += "Exception:\n" + ModelicaHelper.getStackTrace(ex
          );
    } finally {
      System.setOut(out);
      baos.flush();
      FileOutputStream fout = new FileOutputStream("
          JavaExtInteractiveTrace.txt");
      baos.writeTo(fout);
      fout.close();
    }

    return new ModelicaString(res);
}

}
```

A.1 OMCorba Parser

Listing A.3. OMCorba.g

```
// ANTLRv3 Grammar to parse the corba output from OMC to
    Java structures
grammar OMCorba;

options {
output=none;
k=1;
}

@header {package org.openmodelica.corba.parser;
import java.util.LinkedHashMap;
import java.util.Vector;
import org.openmodelica.*;}
```

```
@lexer::header {package org.openmodelica.corba.parser;
import org.openmodelica.*;}

@members {
protected ModelicaObject memory;
private String key;
}


prog: object EOF
    | EOF {memory = new ModelicaVoid();};

object: INT {memory = new ModelicaInteger($INT.int);}
      | REAL {memory = new ModelicaReal(new Double($REAL.
          text));}
      | BOOL {memory = new ModelicaBoolean(new Boolean(
          $BOOL.text));}
      | STRING {memory = new ModelicaString($STRING.text.
          substring(1,$STRING.text.length()-1), true);}
      // | STRING {memory = new ModelicaString($STRING.text
          .substring(1,$STRING.text.length()-1), false);}
      | record
      | array
      | tuple
      | option;

record : 'record' {LinkedHashMap<String,ModelicaObject> map
    = new LinkedHashMap<String,ModelicaObject>();}
         id1=ident
         (field {map.put(key, memory);} (',' field {map.put
             (key, memory);})*)?
         'end' id2=ident {if (!$id1.text.equals($id2.text))
              throw new RecognitionException();} ';'
         {memory = new ModelicaRecord($id1.text, map);};

array : '{' {Vector<ModelicaObject> vector = new Vector<
    ModelicaObject>();}
         (object {vector.add(memory);}
         (',' object {vector.add(memory);})*)?
        '}' {try{memory = ModelicaArray.createModelicaArray
            (vector);} catch (ModelicaObjectException ex) {
            throw new RecognitionException();}};

tuple : '(' {ModelicaTuple tuple = new ModelicaTuple();}
        (object {tuple.add(memory);}
        (',' object {tuple.add(memory);})*)?
```

```
        ')' {memory = tuple;};

option : 'NONE()' {memory = new ModelicaOption(null);}
       | 'SOME(' object ')' {memory = new ModelicaOption(memory);};

ident : ID | FQID;

field : ID '=' object {key = new String($ID.text);};

BOOL : 'true'|'false';
FQID : (ID '.')+ ID;
ID : ('_'|'a'..'z'|'A'..'Z')('_'|'a'..'z'|'A'..'Z'|'0'..'9')* |
     '\''(~('\\'|'\'')|'\\\'' | '\\"'| '\\?' | '\\\\' | '\\a' | '\\b' | '\\f' | '\\n' | '\\r' | '\\t' | '\\v')*'\'';
STRING : '"'('\\"'|~'"')*'"';
REAL : '-'? (('.' '0'..'9'+)|('0'..'9'+ '.' '0'..'9'*))(('e'|'E')(('+'|'-')?'0'..'9'+))? |
       '-'? '0'..'9'+('e'|'E')(('+'|'-')?'0'..'9'+);
INT :  '-'?'0'..'9'+ ;
WS  :  ('\r'|'\n'|' '|'\t')+ {skip();} ;
```

Listing A.4. OMCStringParser.java

```
package org.openmodelica.corba.parser;

import org.antlr.runtime.*;
import org.openmodelica.ModelicaAny;
import org.openmodelica.ModelicaObject;

public class OMCStringParser {
  public static ModelicaObject parse(String s) throws ParseException {return parse(s,ModelicaObject.class);}
  public static <T extends ModelicaObject> T parse(String s, Class<T> c) throws ParseException {
    ANTLRStringStream input = new ANTLRStringStream(s);
    OMCorbaLexer lexer = new OMCorbaLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    OMCorbaParser parser = new OMCorbaParser(tokens);
    try {
      parser.prog();
    } catch (RecognitionException e) {
```

```
      new ParseException("OMCStringParser: Failed to parse:
          " + s);
    } catch (ClassCastException e) {
      new ParseException("OMCStringParser: Failed to parse:
          " + s);
    }
    if (parser.getNumberOfSyntaxErrors() != 0)
      throw new ParseException("OMCStringParser: "+parser.
          getNumberOfSyntaxErrors()+" syntax errors, failed
           to parse:\n" + s);

    ModelicaObject o = parser.memory;
    try {
      return ModelicaAny.cast(o, c);
    } catch (Exception ex) {
      throw new ParseException(String.format("Failed to
          cast %s to %s", o.toString(), c.getName()), ex);
    }
  }
}
```

Appendix B

OMCorbaDefinitions Parser

Listing B.1. OMCorbaDefinitions.g

```
grammar OMCorbaDefinitions;

options {
        language = Java;
        output = none;
        k = 2;
}

@header {package org.openmodelica.corba.parser;import java.util.Vector;}
@lexer::header {package org.openmodelica.corba.parser;}

@members {
public Vector<PackageDefinition> defs = new Vector<PackageDefinition>();
public SymbolTable st = new SymbolTable();
private Object memory;
private String curPackage;
protected Object recoverFromMismatchedToken(IntStream input, int ttype, BitSet follow) throws RecognitionException {
  MismatchedTokenException ex = new MismatchedTokenException(ttype, input);
  throw ex;
}
}
```

```
definitions : {this.curPackage = null; PackageDefinition pack = new PackageDefinition(null);}
  '(' (object {pack.add(memory);})* ')' EOF {defs.add(pack); memory = null; st.add(pack, null);};

object : package_ | record | function | uniontype | typedef | replaceable_type;

package_ : '(' 'package' ID {String oldPackage = curPackage; curPackage = (curPackage != null ? curPackage + "." + $ID.text : $ID.text); PackageDefinition pack = new PackageDefinition(curPackage);}
         (object {pack.add(memory);})* ')' {defs.add(pack); memory = null; st.add(pack, null); curPackage = oldPackage;};
record : '(' 'record' ID1=ID {String oldPackage = curPackage; curPackage = (curPackage != null ? curPackage + "." : "") + $ID1.text ; RecordDefinition rec = new RecordDefinition($ID1.text, curPackage); PackageDefinition pack = new PackageDefinition(curPackage + ".inner");}
        ((('(' varDef ')')|extends_){rec.fields.add(memory);}
         | object {pack.add(memory);}
         )* ')' {memory = rec; curPackage = oldPackage; st.add(rec, curPackage);}
        |'(' 'metarecord' ID1=ID {String recID = $ID1.text; String oldPackage = curPackage; curPackage = (curPackage != null ? curPackage + "." : "") + $ID1.text ; RecordDefinition rec; PackageDefinition pack = new PackageDefinition(curPackage + ".inner");}
          INT {int index = $INT.int;}
          UT=ID {String uniontype = $UT.text;}
          {rec = new RecordDefinition(recID, uniontype, index, curPackage);}
          ((('(' varDef ')')|extends_){rec.fields.add(memory);}
           | object {pack.add(memory);}
           )* ')' {memory = rec; curPackage = oldPackage; st.add(rec, curPackage);};
extends_ : '(' 'extends' fqid ')';
function : '(' 'function' ID {FunctionDefinition fun = new FunctionDefinition($ID.text); String oldPackage = curPackage; curPackage = (curPackage != null ? curPackage + "." : "") + $ID.text; PackageDefinition
```

```
    pack = new PackageDefinition(curPackage + ".inner");}
            ( input{fun.input.add((VariableDefinition)
                memory);}
            | output{fun.output.add((VariableDefinition)
                memory);}
            | object {pack.add(memory);}
            )*
            ')' {curPackage = oldPackage; memory = fun; st.
                add(fun, curPackage);};
uniontype : '(' 'uniontype' ID ')' {UniontypeDefinition
    union = new UniontypeDefinition($ID.text); memory =
    union; st.add(union, curPackage);};
typedef : '(' 'partial' 'function' ID ')' {memory = new
    VariableDefinition(new ComplexTypeDefinition(
    ComplexTypeDefinition.ComplexType.FUNCTION_REFERENCE),
    $ID.text, curPackage);st.add((VariableDefinition)memory
    , curPackage);}
        | '(' 'type' ID type ')' {memory = new
            VariableDefinition((ComplexTypeDefinition)
            memory, $ID.text, curPackage); st.add((
            VariableDefinition)memory, curPackage);};

replaceable_type : '(' 'replaceable' 'type' ID ')' {memory
    = new VariableDefinition(new ComplexTypeDefinition(
    ComplexTypeDefinition.ComplexType.GENERIC_TYPE, "
    ModelicaObject"), $ID.text, curPackage); st.add((
    VariableDefinition)memory, curPackage);};

type : basetype
     | complextype
     | '[' INT type {memory = new ComplexTypeDefinition(
          ComplexTypeDefinition.ComplexType.ARRAY, (
          ComplexTypeDefinition) memory, $INT.int);}
     | fqid {memory = new ComplexTypeDefinition(
          ComplexTypeDefinition.ComplexType.DEFINED_TYPE, (
          String) memory);};
varDef : type ID {memory = new VariableDefinition((
    ComplexTypeDefinition)memory, $ID.text, curPackage);};
input  : '(' 'input' varDef ')';
output : '(' 'output' varDef ')';
basetype : 'Integer' {memory = new ComplexTypeDefinition(
    ComplexTypeDefinition.ComplexType.BUILT_IN, "
    ModelicaInteger");}
          |'Real' {memory = new ComplexTypeDefinition(
              ComplexTypeDefinition.ComplexType.BUILT_IN, "
              ModelicaReal");}
```

```
            |'Boolean' {memory = new ComplexTypeDefinition(ComplexTypeDefinition.ComplexType.BUILT_IN, "ModelicaBoolean");}
            |'String' {memory = new ComplexTypeDefinition(ComplexTypeDefinition.ComplexType.BUILT_IN, "ModelicaString");};
complextype : /* MetaModelica */
        ('list') {ComplexTypeDefinition def = new ComplexTypeDefinition(ComplexTypeDefinition.ComplexType.LIST_TYPE);}
        '<' type {def.add((ComplexTypeDefinition)memory);} '>' {memory = def;}
      | ('tuple') {ComplexTypeDefinition def = new ComplexTypeDefinition(ComplexTypeDefinition.ComplexType.TUPLE_TYPE);}
        '<' type (',' type)* '>' {memory = def;}
      | ('Option') {ComplexTypeDefinition def = new ComplexTypeDefinition(ComplexTypeDefinition.ComplexType.OPTION_TYPE);}
        '<' type {def.add((ComplexTypeDefinition)memory);} '>' {memory = def;};
fqid : ID {memory = $ID.text;}
     | QID {memory = $QID.text;};

QID : (ID '.')+ ID;
ID : ('_'|'a'..'z'|'A'..'Z')('_'|'a'..'z'|'A'..'Z'|'0'..'9')* |
       '\''(~('\\'|'\'')|'\\\''  | '\\"'| '\\?' | '\\\\' | '\\a' | '\\b' | '\\f' | '\\n' | '\\r' | '\\t' | '\\v')*'\'';
INT :  '-'?'0'..'9'+ ;
WS :   ('\r'|'\n'|' '|'\t')+ {skip();} ;
```

Listing B.2. OMCorbaDefinitions.st

```
/* function.st */

$header()$

@SuppressWarnings("unchecked")
public class $function.name$ extends ModelicaFunction {
  public $function.name$ (SmartProxy proxy) {
    super("$myFQName(var = function.name)$", proxy);
  }

  $if(rest(function.output))$
```

```
  public $function.generics$ ModelicaTuple call ($function.input:{$it.TypeName$ input__$it.varName$}; separator = ", "$) throws ParseException,ConnectException
  {
    return super.call(ModelicaTuple.class$if(function.input)$, $endif$$function.input:{input__$it.varName$}; separator=","$);
  }

  public $function.generics$ void call ($function.input:{$it.TypeName$ input__$it.varName$}; separator = ", "$$if(function.input)$, $endif$$function.output:{$it.TypeName$ output__$it.varName$}; separator = ", "$) throws ParseException,ConnectException
  {
    ModelicaTuple __tuple = super.call(ModelicaTuple.class$if(function.input)$, $endif$$function.input:{input__$it.varName$}; separator=","$);
    java.util.Iterator<ModelicaObject> __i = __tuple.iterator();
    $function.output:{if (output__$it.varName$ != null) output__$it.varName$.setObject(__i.next()); else __i.next();}; separator = "\n"$
  }

$elseif(function.output)$
  public $function.generics$ $first(function.output).TypeName$ call ($function.input:{$it.TypeName$ input__$it.varName$}; separator = ", "$$if(first(function.output).GenericReference)$$if(function.input)$, $endif$Class<$first(function.output).TypeName$> __outClass$endif$) throws ParseException, ConnectException
  {
    return super.call($first(function.output).TypeClass$$if(function.input)$, $endif$$function.input:{input__$it.varName$}; separator=", "$);
  }

$else$
  public $function.generics$ ModelicaVoid call ($function.input:{$it.TypeName$ input__$it.varName$}; separator = ","$) throws ParseException,ConnectException
  {
    return super.call(ModelicaVoid.class$if(function.input)$, $endif$$function.input:{input__$it.varName$};
```

```
        separator=", "$);
  }

$endif$
}
/* header.st */

$if(skipHeader)$$else$
/*
 * This file was auto-generated by the Modelica/
     MetaModelica to Java/JAR translator
 * See http://openmodelica.org/ or read the documentation
     for more information
 */
$if(package.name)$
package $basepackage$.$package.name$;
import org.openmodelica.*;
$else$
package $basepackage$;
import org.openmodelica.*;
$endif$
import org.openmodelica.corba.SmartProxy;
import org.openmodelica.corba.parser.ParseException;
import org.openmodelica.corba.ConnectException;
$endif$
/* myFQName.st */

$if(package.name)$$package.name$.$endif$$var$
/* record.st */

$header()$

/* Record $record.name$ */
@SuppressWarnings({ "unchecked", "serial" })
public class $record.name$ $record.generics$ extends
    ModelicaRecord $record.uniontype:{implements $it$}$ {

  public $record.generics$ $record.name$($record.fields:{
      $it.TypeName$ __$it.varName$}; separator = ","$) {
    super(new ModelicaRecord("$myFQName(var = record.name)$
        ", new String[]{$record.fields:{"$it.varName$"};
        separator=","$}));
    $record.fields:{put("$it.varName$", __$it.varName$);};
        separator = "\n"$
  }

```

```
  public $record.name$(ModelicaObject o) {
    super("$myFQName(var = record.name)$",
          new String[]{$record.fields:{"$it.varName$"}; separator=","$},
          new java.lang.Class[] {$record.fields:{$it.TypeClass$}; separator=","$}, (ModelicaRecord) o);
  }

  $record.fields:{public $record.generics$ $it.TypeName$ get_$it.varName$() {return get("$it.varName$", $it.TypeClass$);}
public $record.generics$ void set_$it.varName$($it.TypeName$ new__$it.varName$) {put("$it.varName$", new__$it.varName$);}

}$

  @Override
  public int get_ctor_index() {
    return $record.ctor_index$;
  }
}

/* uniontype.st */

$header()$
/* Uniontype $uniontype$ */
public interface $uniontype$ extends IModelicaRecord {
}
```

Appendix C

Template-Based Code Generator Examples

Listing C.1. expression.tpl

```
$=IsBinaryOp$
($^Exp1$$Op$$^Exp2$)
$/=
$=IsIntConst$
 $IntLiteral$
$/=
$=IsCRef$
$CRef$
$/=
```

Listing C.2. statementsC.tpl

```
$=IsIf$\n
if ($#Cond$$:Exp$$/#) {
$^IfPart#  $\n
} else {
$^ElsePart#  $\n
}
$/=
$=IsWhile$\n
while ($#Cond$$:Exp$$/#) {
$^AST#  $\n
}
$/=
$=IsExpression$\n
$#Exp$$:Exp$$/#;
```

```
$/=
$=IsASTList$
$^List$
$/=
$=IsDefine$\n
$CRef$ = $#Exp$$:Exp$$/#;
$/=
```

Listing C.3. statementsPython.tpl

```
$=IsIf$\n
if $#Cond$$:Exp$$/#:
$^IfPart#  $\n
else:
$^ElsePart#  $
$/=
$=IsWhile$\n
while $#Cond$$:Exp$$/#:
$^AST#  $
$/=
$=IsExpression\n
$#Exp$$:Exp$$/#
$/=
$=IsASTList$
$^List$$!List$\n
pass$/!
$/=
$=IsDefine$\n
$CRef$ = $#Exp$$:Exp$$/#
$/=
```

Listing C.4. Example Output (Statement templates)

```
AST Dict:
IsASTList: ENABLED
List: DICTIONARY_LIST
  IsIf: ENABLED
  Cond: DICTIONARY
    IsBinaryOp: ENABLED
    Op: STRING
    Exp1: DICTIONARY
      IsIntConst: ENABLED
      IntLiteral: STRING
    Exp2: DICTIONARY
      IsIntConst: ENABLED
```

```
      IntLiteral: STRING
  IfPart: DICTIONARY
    IsASTList: ENABLED
    List: DICTIONARY_LIST
      IsWhile: ENABLED
      Cond: DICTIONARY
        IsBinaryOp: ENABLED
        Op: STRING
        Exp1: DICTIONARY
          IsCRef: ENABLED
          CRef: STRING
        Exp2: DICTIONARY
          IsIntConst: ENABLED
          IntLiteral: STRING
      AST: DICTIONARY
        IsDefine: ENABLED
        CRef: STRING
        Exp: DICTIONARY
          IsBinaryOp: ENABLED
          Op: STRING
          Exp1: DICTIONARY
            IsCRef: ENABLED
            CRef: STRING
          Exp2: DICTIONARY
            IsIntConst: ENABLED
            IntLiteral: STRING

      IsDefine: ENABLED
      CRef: STRING
      Exp: DICTIONARY
        IsBinaryOp: ENABLED
        Op: STRING
        Exp1: DICTIONARY
          IsCRef: ENABLED
          CRef: STRING
        Exp2: DICTIONARY
          IsBinaryOp: ENABLED
          Op: STRING
          Exp1: DICTIONARY
            IsIntConst: ENABLED
            IntLiteral: STRING
          Exp2: DICTIONARY
            IsIntConst: ENABLED
            IntLiteral: STRING

  ElsePart: DICTIONARY
```

```
      IsASTList: ENABLED
      List: DICTIONARY_LIST

    IsDefine: ENABLED
    CRef: STRING
    Exp: DICTIONARY
      IsBinaryOp: ENABLED
      Op: STRING
      Exp1: DICTIONARY
        IsCRef: ENABLED
        CRef: STRING
      Exp2: DICTIONARY
        IsIntConst: ENABLED
        IntLiteral: STRING


AST transformations:
C:
if ((1<2)) {
  while ((a<2)) {
    a = (a-1);
  }
  a = (a*(4-1));
} else {
}
a = (a/3);
Python:
if (1<2):
  while (a<2):
    a = (a-1)
  a = (a*(4-1))
else:
  pass
a = (a/3)
```

Listing C.5. BigTemplate.tpl (alternative syntax)

```
<include "BigTemplateHeader.tpl">\n

/* Body */\n
<Functions:{
<Func>_rettype <Func>(<VarsIn:{<Type> <Name>}", ">) \{\n
  <Func>_rettype out;\n
  <VarsIn:{<Type> <Name>_ext;\n}>
  <VarOut.Type> <VarOut.Name>_ext;\n
  <VarsIn:{<Name>_ext = (<Type>) <Name>;\n}>
```

```
  \n
<cond
case Java:{
  JNIEnv* __env; jclass __cls; jmethodID __mid;\n
  __env = getJavaEnv();\n
  getJavaMethod("'<ExternalClass>.<ExternalName>'","<ExternalSignature>",__env,&__cls,&__mid);\n
}>
  <VarOut.Name>_ext =
<cond
case Java:{(*__env)-\>CallStaticDoubleMethod(__env, __cls, __mid, <VarsIn:{<Name>_ext}", ">);\n}
case !Java:{
<ExternalName>(<VarsIn:{<Name>_ext}",">);\n
  CHECK_FOR_JAVA_EXCEPTION(__env);
  (*__env)-\>DeleteLocalRef(__env, __cls);
}
>\n
  out.<VarOut.StructField> = (<VarOut.ModelicaType>) <VarOut.Name>_ext;\n
  return out;\n
\}\n
}> <! Loop over functions !>\n
/* End Body */\n
```

Listing C.6. BigTemplateHeader.tpl (alternative syntax)

```
/* Header */\n
<Functions:{\n
typedef struct <Func>_rettype_s\n
\{\n
    <VarOut.ModelicaType> <VarOut.StructField>;\n
\} <Func>_rettype;\n
}>\n

<Functions:{<cond <! LISP-style cond !>
case Java then {/* External Java doesn't declare external C functions... */\n}
case !Java then {external <VarOut.Type> <ExternalName>(
<VarsIn:{<Type> <Name>}", ">)
;\n
}
else {/* Alternative else */\n}
>}>

<Functions:{
```

```
<Func>_rettype <Func>(<VarsIn:{<Type>}",">);
}"\n">
\n
/* End Header */\n
```

Listing C.7. Example Output (BigTemplate)

```
Dictionary:
Functions: DICTIONARY_LIST
  WithNames: ENABLED
  VarsIn: DICTIONARY_LIST
    Type: STRING
    ModelicaType: STRING
    Name: STRING

  VarOut: DICTIONARY
    Type: STRING
    ModelicaType: STRING
    Name: STRING
    StructField: STRING
  ExternalSignature: STRING
  ExternalClass: STRING
  ExternalName: STRING
  Java: ENABLED
  Func: STRING

  VarsIn: DICTIONARY_LIST
    Type: STRING
    ModelicaType: STRING
    Name: STRING

  VarOut: DICTIONARY
    Type: STRING
    ModelicaType: STRING
    Name: STRING
    StructField: STRING
  ExternalName: STRING
  C: ENABLED
  Func: STRING

Applied Compiled Template (C and Java):
/* Header */

typedef struct _logJava_rettype_s
{
    modelica_real targ1;
```

```
} _logJava_rettype;

typedef struct _logC_rettype_s
{
    modelica_real targ1;
} _logC_rettype;

/* External Java doesn't declare external C functions... */
external jdouble log(jdouble x);
_logJava_rettype _logJava(jdouble);
_logC_rettype _logC(jdouble);
/* End Header */

/* Body */
_logJava_rettype _logJava(jdouble x) {
  _logJava_rettype out;
  jdouble x_ext;
  jdouble y_ext;
  x_ext = (jdouble) x;

  JNIEnv* __env; jclass __cls; jmethodID __mid;
  __env = getJavaEnv();
  getJavaMethod("'java.lang.Math.log'","(D)D",__env,&__cls,&__mid);
  y_ext = (*__env)->CallStaticDoubleMethod(__env, __cls, __mid, x_ext);

  out.targ1 = (modelica_real) y_ext;
  return out;
}
_logC_rettype _logC(jdouble x) {
  _logC_rettype out;
  jdouble x_ext;
  jdouble y_ext;
  x_ext = (jdouble) x;

  y_ext = log(x_ext);
  CHECK_FOR_JAVA_EXCEPTION(__env); (*__env)->DeleteLocalRef(__env, __cls);
  out.targ1 = (modelica_real) y_ext;
  return out;
}

/* End Body */
```

Appendix D

Template-Based Code Generator Code Listings

Listing D.1. TemplCG.mo

```
package TemplCG

import Print;
import Error;
import Util;
import System;

public type TemplateTreeSequence = list<TemplateTree>;

uniontype TemplateTree
  record TEMPLATE_COND
    list<KeyBody> cond_bodies;
    TemplateTreeSequence else_body;
  end TEMPLATE_COND;

  record TEMPLATE_FOR_EACH
    String key;
    String separator;
    TemplateTreeSequence body;
  end TEMPLATE_FOR_EACH;

  record TEMPLATE_RECURSION
    String key;
    String indent;
  end TEMPLATE_RECURSION;

```

```
  record TEMPLATE_ADD_INDENTATION
    String indent;
    TemplateTreeSequence body;
  end TEMPLATE_ADD_INDENTATION;

  record TEMPLATE_LOOKUP_KEY
    String key;
  end TEMPLATE_LOOKUP_KEY;

  record TEMPLATE_CURRENT_VALUE
  end TEMPLATE_CURRENT_VALUE;

  record TEMPLATE_TEXT
    String text;
  end TEMPLATE_TEXT;

  record TEMPLATE_INDENT
  end TEMPLATE_INDENT;

end TemplateTree;

uniontype Environment

  record ENV_STRING_LIST
    list<String> strings;
  end ENV_STRING_LIST;

  record ENV_DICT_LIST
    list<DictItemList> dicts;
  end ENV_DICT_LIST;

  record ENV_NULL
  end ENV_NULL;

end Environment;

uniontype Dict
  record ENABLED
  end ENABLED;

  record STRING_LIST
    list<String> strings;
  end STRING_LIST;

  record STRING
    String string;
```

```
  end STRING;

  record DICTIONARY
    DictItemList dict;
  end DICTIONARY;

  record DICTIONARY_LIST
    list<DictItemList> dict;
  end DICTIONARY_LIST;
end Dict;

record DictItem
  String key;
  Dict dict;
end DictItem;
type TemplDict = list<DictItemList>;
type DictItemList = list<DictItem>;

record TemplateInclude
  String key;
  TemplateTreeSequence body;
end TemplateInclude;

record KeyBody
  String key;
  Boolean negateValue;
  TemplateTreeSequence body;
end KeyBody;

public function Unescape
  input String str;
  input String indent;
  output String out;
algorithm
  out := Unescape2(stringListStringChar(str), indent);
end Unescape;

protected function Unescape2
  input list<String> str;
  input String indent;
  output String out;
algorithm
  out := matchcontinue(str, indent)
    local
      String char;
      list<String> rest;
```

```
    case ({}, _) then "";
    case ("\\"::"n"::rest, indent) then "\n" +& Unescape2(rest, indent);
    case (char::rest, indent) then char+&Unescape2(rest, indent);
  end matchcontinue;
end Unescape2;

protected function GetTemplateInclude
  input list<TemplateInclude> includes;
  input String key;
  output TemplateTreeSequence body;
algorithm
  body := matchcontinue (includes, key)
  local
    list<TemplateInclude> rest;
    String thisKey;
    TemplateTreeSequence res;
    case ({}, _) then fail();
    case (TemplateInclude(thisKey, res)::rest, key) equation
      true = thisKey ==& key;
    then res;
    case (_::rest, key) then GetTemplateInclude(rest, key);
  end matchcontinue;
end GetTemplateInclude;

public function PrintDict
  input DictItemList dict;
  input String indent;
algorithm
  _ := matchcontinue(dict, indent)
    local
      Dict item;
      DictItemList rest;
      String key;
    case ({}, _) then ();
    case (DictItem(key, item)::rest, indent) equation
      print(indent);
      print(key +& ": ");
      PrintDictItem(item, indent);
      PrintDict(rest, indent);
    then ();
  end matchcontinue;
end PrintDict;

```

```
public function PrintDictList
  input list<DictItemList> dict;
  input String indent;
algorithm
  _ := matchcontinue(dict, indent)
    local
      DictItemList item;
      list<DictItemList> rest;
      String key;
    case ({}, _) then ();
    case (item::rest, indent) equation
      PrintDict(item, indent);
      print("\n");
      PrintDictList(rest, indent);
    then ();
  end matchcontinue;
end PrintDictList;

protected function PrintDictItem
  input Dict dict;
  input String indent;
algorithm
  _ := matchcontinue(dict, indent)
    local
      DictItemList d;
      list<DictItemList> dl;
    case (ENABLED(), indent) equation
      print("ENABLED\n");
    then ();
    case (STRING(_), indent) equation
      print("STRING\n");
    then ();
    case (STRING_LIST(_), indent) equation
      print("STRING_LIST\n");
    then ();
    case (DICTIONARY(d), indent) equation
      print("DICTIONARY\n");
      PrintDict(d, indent+&"  ");
    then ();
    case (DICTIONARY_LIST(dl), indent) equation
      print("DICTIONARY_LIST\n");
      PrintDictList(dl, indent+&"  ");
    then ();
  end matchcontinue;
end PrintDictItem;
```

```
public function PrintTemplateTreeSequence
  input TemplateTreeSequence tree;
algorithm
  print("{\n");
  PrintTemplateTreeSequence_(tree, "  ");
  print("\n}");
end PrintTemplateTreeSequence;

protected function PrintTemplateTreeSequence_
  input TemplateTreeSequence tree;
  input String indentLevel;
algorithm
  _ := matchcontinue(tree, indentLevel)
    local
      TemplateTree element;
      TemplateTreeSequence rest;
    case ({}, _) then ();
    case (element :: {}, indentLevel) equation
      PrintTemplateTree(element, indentLevel);
    then ();
    case (element :: rest, indentLevel) equation
      PrintTemplateTree(element, indentLevel);
      print(", \n");
      PrintTemplateTreeSequence_(rest, indentLevel);
    then ();
  end matchcontinue;
end PrintTemplateTreeSequence_;

protected function PrintTemplateCond
  input list<KeyBody> bodies;
  input String indentLevel;
algorithm
  _ := matchcontinue(bodies, indentLevel)
    local
      String key;
      TemplateTreeSequence body;
      list<KeyBody> rest;
      Boolean negateValue;
    case ({}, _) then ();
    case (KeyBody(key, negateValue, body)::rest, indentLevel) equation
      key = Util.stringReplaceChar(key, "\"", "\\\"");
      print(indentLevel +& "TemplCG.KeyBody(\"" +& key +& "\", ");
      print(Util.if_(negateValue, "true", "false"));
      print(", {\n");
```

```
      PrintTemplateTreeSequence_(body, "  " +& indentLevel);
      print("\n" +& indentLevel +& "})");
      print(Util.if_(listLength(rest) == 1, ", \n", "\n"));
    then ();
  end matchcontinue;
end PrintTemplateCond;

protected function PrintTemplateTree
  input TemplateTree element;
  input String indentLevel;
algorithm
  _ := matchcontinue(element, indentLevel)
    local
      String key, sep, text;
      TemplateTreeSequence body, if_body, else_body;
      list<KeyBody> condBodies;
    case (TEMPLATE_COND(condBodies, else_body = else_body), indentLevel) equation
      print(indentLevel +& "TemplCG.TEMPLATE_COND({\n");
      PrintTemplateCond(condBodies, indentLevel);
      print(indentLevel +& "}, /* else */ {\n");
      PrintTemplateTreeSequence_(else_body, "  " +& indentLevel);
      print("\n" +& indentLevel +& "} \n" +& indentLevel +& ")");
    then ();
    case (TEMPLATE_FOR_EACH(key = key, separator = sep, body = body), indentLevel) equation
      key = Util.stringReplaceChar(key, "\"", "\\\"");
      sep = Util.stringReplaceChar(sep, "\"", "\\\"");
      print(indentLevel +& "TemplCG.TEMPLATE_FOR_EACH(\"" +& key +& "\", \"" +& sep +& "\", {\n");
      PrintTemplateTreeSequence_(body, "  " +& indentLevel);
      print("\n" +& indentLevel +& "})");
    then ();
    case (TEMPLATE_RECURSION(key = key, indent = sep), indentLevel) equation
      key = Util.stringReplaceChar(key, "\"", "\\\"");
      sep = Util.stringReplaceChar(sep, "\"", "\\\"");
      print(indentLevel +& "TemplCG.TEMPLATE_RECURSION(\"" +& key +& "\", \"" +& sep +& "\")");
    then ();
    case (TEMPLATE_ADD_INDENTATION(indent = sep, body=body), indentLevel) equation
```

```
        sep = Util.stringReplaceChar(sep, "\"", "\\\"");
        print(indentLevel +& "TemplCG.TEMPLATE_ADD_INDENTATION(\"" +& sep +& "\", {\n");
        PrintTemplateTreeSequence_(body, "  " +& indentLevel);
        print("\n" +& indentLevel +& "})");
      then ();
      case (TEMPLATE_LOOKUP_KEY(key = key), indentLevel) equation
        key = Util.stringReplaceChar(key, "\"", "\\\"");
        print(indentLevel +& "TemplCG.TEMPLATE_LOOKUP_KEY(\"" +& key +& "\")");
      then ();
      case (TEMPLATE_CURRENT_VALUE(), indentLevel) equation
        print(indentLevel +& "TemplCG.TEMPLATE_CURRENT_VALUE()");
      then ();
      case (TEMPLATE_INDENT(), indentLevel) equation
        print(indentLevel +& "TemplCG.TEMPLATE_INDENT()");
      then ();
      case (TEMPLATE_TEXT(text = text), indentLevel) equation
        text = Util.stringReplaceChar(text, "\\", "\\\\");
        text = Util.stringReplaceChar(text, "\"", "\\\"");
        text = Util.stringReplaceChar(text, "\n", "\\n");
        print(indentLevel +& "TemplCG.TEMPLATE_TEXT(\"" +& text +& "\")");
      then ();
    end matchcontinue;
end PrintTemplateTree;

public function CompileTemplateFromFile
  input String templateFileName;
  input list<TemplateInclude> includes;
  output TemplateTreeSequence out;
algorithm
  out := matchcontinue (templateFileName, includes)
  local
    String template, error;
    list<String> templateNoComments;
    TemplateTreeSequence out;
  case (templateFileName, includes) equation
    template = System.readFile(templateFileName);
    (templateNoComments, error) = RemoveComments(stringListStringChar(template), 0);
```

```
    error = Util.if_(error ==& "", "", "\nError: " +& error);
    print(error);
    true = error ==& "";

    (out, error) = CompileTemplate_Angles(templateNoComments, includes);
    error = Util.if_(error ==& "", "", "\nError: " +& error);
    print(error);
    true = error ==& "";
  then out;

  case (templateFileName, _) equation
    print("Parsing template " +& templateFileName +& " failed");
  then fail();
  end matchcontinue;
end CompileTemplateFromFile;

public function CompileTemplateFromFile_Old
  input String templateFileName;
  input list<TemplateInclude> includes;
  output TemplateTreeSequence out;
algorithm
  out := matchcontinue (templateFileName, includes)
  local
    String template, error;
    list<String> templateNoComments;
    TemplateTreeSequence out;
  case (templateFileName, includes) equation
    template = System.readFile(templateFileName);
    out = CompileTemplate_Old(stringListStringChar(template), includes);
  then out;

  case (templateFileName, _) equation
    print("Parsing template " +& templateFileName +& " failed");
  then fail();
  end matchcontinue;
end CompileTemplateFromFile_Old;


public function CompileTemplate
  input String template;
```

```
  input list<TemplateInclude> includes;
  output TemplateTreeSequence out;
algorithm
  out := CompileTemplate_Old(stringListStringChar(template), includes);
end CompileTemplate;

protected function RemoveComments
  input list<String> template;
  input Integer numNested;
  output list<String> out;
  output String error;
algorithm
  (out, error) := matchcontinue (template, numNested)
    local
      String char, error;
      list<String> out, rest;
    case ({}, _) then ({}, "");
    case (rest as "!"::">"::_, 0) equation
      error = flattenStringList(rest);
      error = "Unbalanced comment tag: " +& error;
    then ({}, error);
    case ("<"::"!"::rest, numNested) equation
      (out, error) = RemoveComments(rest, numNested+1);
    then (out, error);
    case ("!"::">"::rest, numNested) equation
      (out, error) = RemoveComments(rest, numNested-1);
    then (out, error);
    case (char::rest, numNested as 0) equation
       (out, error) = RemoveComments(rest, numNested);
    then (char::out, error);
    case (char::rest, numNested) equation
       (out, error) = RemoveComments(rest, numNested);
    then (out, error);

  end matchcontinue;
end RemoveComments;

protected function FindAngleBody
  input list<String> template;
  input Integer numNested;
  input String opener;
  input String closer;
  output list<String> body;
  output list<String> afterBody;
algorithm
```

```
  (body, afterBody) := matchcontinue(template, numNested,
      opener, closer)
    local
      String char;
      list<String> rest, afterBody, out;
    case ({}, _, _, _) then fail();

    case ("\\"::char::rest, numNested, opener, closer)
        equation
      (out, afterBody) = FindAngleBody(rest, numNested,
          opener, closer);
    then ("\\"::char::out, afterBody);

    case (char::rest, 0, _, closer) equation
      true = char ==& closer;
      //print("\nFound closer: " +& char);
    then ({}, rest);

    case (char::rest, numNested, opener, closer) equation
      false = char ==& "\\";
      //print("\n" +& intString(numNested));
      numNested = Util.if_(char ==& opener, numNested+1,
          numNested);
      numNested = Util.if_(char ==& closer, numNested-1,
          numNested);
      //print(" " +& intString(numNested) +& ": " +& char);
      (out, afterBody) = FindAngleBody(rest, numNested,
          opener, closer);
    then (char::out, afterBody);
  end matchcontinue;
end FindAngleBody;

protected function FindAngleBodyKey
  input list<String> template;
  output list<String> key;
  output list<String> afterKey;
algorithm
  (body, afterKey) := matchcontinue(template)
    local
      String char;
      list<String> rest, afterKey, out;
    case ({}) then ({}, {});
    case (":"::rest) then ({}, rest);
    case ("t"::"h"::"e"::"n"::rest) then ({}, rest);

    case (" "::rest) equation
```

```
      ({}, afterKey) = FindAngleBodyKey(rest);
    then ({}, afterKey);
    case ("\n"::rest) equation
      ({}, afterKey) = FindAngleBodyKey(rest);
    then ({}, afterKey);

    case (char::rest) equation
      false = char ==& "<"; false = char ==& ">";
      false = char ==& "{"; false = char ==& "}";
      false = char ==& " "; false = char ==& "\n";
      (out, afterKey) = FindAngleBodyKey(rest);
    then (char::out, afterKey);
  end matchcontinue;
end FindAngleBodyKey;

protected function SkipCommentBody
  input list<String> template;
  output list<String> out;
algorithm
  (out) := matchcontinue(template)
    local
      String char;
      list<String> rest, afterKey, out;
    case ({}) equation
      Error.addMessage(Error.TEMPLCG_INVALID_TEMPLATE, {"
          Failed to end comment"});
    then fail();
    case ("!"::">"::rest) then rest;
    case (char::rest) then SkipCommentBody(rest);
  end matchcontinue;
end SkipCommentBody;

protected function SkipWhitespace
  input list<String> template;
  output list<String> out;
algorithm
  out := matchcontinue(template)
  local
    list<String> rest;
    case {} then {};
    case " "::rest then SkipWhitespace(rest);
    case "\n"::rest then SkipWhitespace(rest);
    case rest then rest;
  end matchcontinue;
end SkipWhitespace;
```

```
protected function FindAngleSep
  input list<String> afterBody;
  output String sep;
algorithm
  sep := matchcontinue (afterBody)
  local
    String error;
    case (afterBody) then FindAngleSep2(afterBody, 0);
    case (afterBody) equation
      error = flattenStringList(afterBody);
      error = "FindAngleSep failed: " +& error;
      Error.addMessage(Error.TEMPLCG_INVALID_TEMPLATE, {
          error}); then fail();
  end matchcontinue;
end FindAngleSep;

protected function FindAngleSep2
  input list<String> afterBody;
  input Integer state;
  output String sep;
algorithm
  sep := matchcontinue (afterBody, state)
  local
    String char, sep;
    list<String> rest;
    case ({}, 0) then "";
    case ({}, 2) then "";
    case ("\n"::rest, state) then FindAngleSep2(rest, state
        ); // Ignore \n in a separator, use \\n to enter
        newline
    case ("\""::rest, 0) then FindAngleSep2(rest, 1);
    case ("\""::rest, 1) then FindAngleSep2(rest, 2);
    case (char::rest, 1) then char+&FindAngleSep2(rest, 1);
    case (" "::rest, state) then FindAngleSep2(rest, state)
        ;
  end matchcontinue;
end FindAngleSep2;

protected function CompileTemplate_Angles_CondBody
  input list<String> template;
  input list<TemplateInclude> includes;
  output TemplateTree out;
  output String error;
algorithm
  out := matchcontinue(template, includes)
    local
```

```
      String key, error1, error2, error3, body, firstChar;
      list<String> keyList1, keyList2, rest, template,
          body, afterBody, caseBody, shouldBeWhitespace;
      list<TemplateInclude> includes;
      TemplateTreeSequence elseBody, bodySeq;
      list<KeyBody> condBodies;
      Boolean negateValue;
    case ({}, includes) then (TEMPLATE_COND({}, {}), "");
    case ("\n" :: rest, includes) equation
      (out, error) = CompileTemplate_Angles_CondBody(rest,
          includes);
    then (out, error);
    case (" " :: rest, includes) equation
      (out, error) = CompileTemplate_Angles_CondBody(rest,
          includes);
    then (out, error);

    case ("c"::"a"::"s"::"e"::rest, includes) equation
      (keyList1 as firstChar::keyList2, caseBody) =
          FindAngleBodyKey(SkipWhitespace(rest));
      negateValue = firstChar ==& "!";
      keyList1 = Util.if_(negateValue, keyList2, keyList1);
      key = flattenStringList(keyList1);
      false = key ==& "_";
      "{"::caseBody = SkipWhitespace(caseBody);
      (caseBody, afterBody) = FindAngleBody(caseBody, 0, "{
          ", "}");
      (bodySeq, error1) = CompileTemplate_Angles(caseBody,
          includes);
      (TEMPLATE_COND(condBodies, elseBody), error2) =
          CompileTemplate_Angles_CondBody(afterBody,
          includes);
      error1 = Util.if_(error1 ==& "", error2, error1);
    then (TEMPLATE_COND(KeyBody(key, negateValue, bodySeq)
        ::condBodies, elseBody), error1);

    case ("c"::"a"::"s"::"e"::rest, includes) equation
      ({"_"}, caseBody) = FindAngleBodyKey(SkipWhitespace(
          rest));
      "{"::caseBody = SkipWhitespace(caseBody);
      (caseBody, afterBody) = FindAngleBody(caseBody, 0, "{
          ", "}");
      (bodySeq, error1) = CompileTemplate_Angles(caseBody,
          includes);
      (TEMPLATE_COND({}, {}), error2) =
          CompileTemplate_Angles_CondBody(afterBody,
```

```
includes);
    error1 = Util.if_(error1 ==& "", error2, error1);
  then (TEMPLATE_COND({}, bodySeq), error1);

  case ("e"::"l"::"s"::"e"::rest, includes) equation
    "{"::caseBody = SkipWhitespace(rest);
    (caseBody, afterBody) = FindAngleBody(caseBody, 0, "{", "}");
    (bodySeq, error1) = CompileTemplate_Angles(caseBody, includes);
    (TEMPLATE_COND({}, {}), error2) = CompileTemplate_Angles_CondBody(afterBody, includes);
    error1 = Util.if_(error1 ==& "", error2, error1);
  then (TEMPLATE_COND({}, bodySeq), error1);

  case (rest as "c"::"a"::"s"::"e"::_, _)
  then (TEMPLATE_COND({}, {}), flattenStringList(rest));
  case (rest as "e"::"l"::"s"::"e"::_, _)
  then (TEMPLATE_COND({}, {}), flattenStringList(rest));

  case (rest, _)
  then (TEMPLATE_COND({}, {}), flattenStringList(rest));
 end matchcontinue;
end CompileTemplate_Angles_CondBody;

protected function CompileTemplate_Angles_Body
 input String key;
 input list<String> template;
 input list<TemplateInclude> includes;
 output TemplateTreeSequence out;
 output String error;
algorithm
 out := matchcontinue(key, template, includes)
  local
    String key, sep, error;
    list<String> rest, template, body, afterBody;
    list<TemplateInclude> includes;
    TemplateTreeSequence bodySeq;
    TemplateTreeSequence out;
  case (key, {}, includes) equation
    //print("\nFound simple FOR_EACH, no template to apply");
  then ({TEMPLATE_FOR_EACH(key, "", {TEMPLATE_LOOKUP_KEY("it")})}, "");
  case (key, "{" :: rest, includes) equation
```

```
      (body, afterBody) = FindAngleBody(rest, 0, "{", "}");
      //print("\nbody=");
      //print(flattenStringList(body));
      sep = FindAngleSep(afterBody);
      (bodySeq, error) = CompileTemplate_Angles(body, includes);
    then ({TEMPLATE_FOR_EACH(key, sep, bodySeq)}, error);
    case (key, (rest as "{" :: _), includes)
    then ({}, flattenStringList(rest));
    case (key, " " :: rest, includes) equation
      (out, error) = CompileTemplate_Angles_Body(key, rest, includes);
    then (out, error);
    case (key, "\n" :: rest, includes) equation
      (out, error) = CompileTemplate_Angles_Body(key, rest, includes);
    then (out, error);
    case (_, rest, _) then ({}, flattenStringList(rest));
  end matchcontinue;
end CompileTemplate_Angles_Body;

protected function CompileTemplate_Angles
  input list<String> template;
  input list<TemplateInclude> includes;
  output TemplateTreeSequence out;
  output String error;
algorithm
  (out, error) := matchcontinue(template, includes)
    local
      String error, error2, key, char, sep, keyAndSep, textBody, newTextBody;
      list<String> keyList, rest, body, afterBody;
      TemplateTreeSequence out, out2, nextBody;
      TemplateTree condBody, newBody;
      list<TemplateInclude> includes;
    case ({}, _) then ({}, "");

    case ("<"::"c"::"o"::"n"::"d"::rest, includes) equation
      (body, afterBody) = FindAngleBody(rest, 0, "<", ">");
      (condBody, error) = CompileTemplate_Angles_CondBody(body, includes);
      (out, error2) = CompileTemplate_Angles(afterBody, includes);
      error = Util.if_(error ==& "", error2, error);
    then (condBody::out, error);
```

```
    case (rest as "<"::"c"::"o"::"n"::"d"::_, includes) equation
    then ({}, flattenStringList(rest));

    case ("<"::"i"::"n"::"c"::"l"::"u"::"d"::"e"::rest, includes) equation
      (body, afterBody) = FindAngleBody(rest, 0, "<", ">");
      key = FindAngleSep(body);
      out = CompileTemplateFromFile(key, includes);
      (out2, error) = CompileTemplate_Angles(afterBody, includes);
    then (listAppend(out, out2), error);
    case (rest as "<"::"i"::"n"::"c"::"l"::"u"::"d"::"e"::_, includes) equation
    then ({}, flattenStringList(rest));

    case ("<"::rest, includes) equation
      (body, afterBody) = FindAngleBody(rest, 0, "<", ">");
      //print("\nForEach full body=");
      //print(flattenStringList(body));
      (keyList, rest) = FindAngleBodyKey(body);
      key = flattenStringList(keyList);
      //print("\nForEach body=");
      //print(flattenStringList(rest));
      //print("\nForEach afterBody=");
      //print(flattenStringList(afterBody));
      (nextBody, error) = CompileTemplate_Angles_Body(key, rest, includes);
      (out, error2) = CompileTemplate_Angles(afterBody, includes);
      error = Util.if_(error ==& "", error2, error);
      out = listAppend(nextBody, out);
    then (out, error);
    case ("<":: rest, includes) equation
    then ({}, flattenStringList(rest));

    case ("\\" :: "n" :: rest, includes) equation
      (out, error) = CompileTemplate_Angles(rest, includes);
    then (TEMPLATE_TEXT("\n") :: TEMPLATE_INDENT() :: out, error);
    case ("\n" :: rest, includes) equation
    (out, error) = CompileTemplate_Angles(rest, includes);
    then (out, error);

    case ("\\" :: char :: rest, includes) equation
```

```
        (TEMPLATE_TEXT(text = textBody) :: nextBody, error) =
            CompileTemplate_Angles(rest, includes);
        newTextBody = char +& textBody;
      then
        (TEMPLATE_TEXT(newTextBody) :: nextBody, error);
      case ("\\" :: char :: rest, includes) equation
      (out, error) = CompileTemplate_Angles(rest, includes);
      then (TEMPLATE_TEXT(char)::out, error);

      case (char :: rest, includes) equation
        false = char ==& "<"; false = char ==& ">";
        false = char ==& "{"; false = char ==& "}";
        (TEMPLATE_TEXT(text = textBody) :: nextBody, error) =
            CompileTemplate_Angles(rest, includes);
        newTextBody = char +& textBody;
      then
        (TEMPLATE_TEXT(newTextBody) :: nextBody, error);
      case (char :: rest, includes) equation
        false = char ==& "<"; false = char ==& ">";
        false = char ==& "{"; false = char ==& "}";
        (nextBody, error) = CompileTemplate_Angles(rest,
            includes);
      then
        (TEMPLATE_TEXT(char) :: nextBody, error);

      case (rest, _) equation
        error = flattenStringList(rest);
      then ({}, error);
    end matchcontinue;
end CompileTemplate_Angles;

protected function CompileTemplate_Old
  input list<String> template;
  input list<TemplateInclude> includes;
  output TemplateTreeSequence out;
algorithm
  out := matchcontinue(template, includes)
    local
      String char, key, sep, keyAndSep, textBody,
          newTextBody;
      list<String> rest, afterBody;
      TemplateTreeSequence body, newBody;
      list<TemplateInclude> includes;
    case ({}, _) then {};
    case ("$" :: "=" :: rest, includes) equation
```

```
      (key, body, newBody) = FindKeyAndBody(rest, "=", includes);
    then
      TEMPLATE_COND({KeyBody(key, false, body)}, {}) :: newBody;
    case ("$" :: "!" :: rest, includes) equation
      (key, body, newBody) = FindKeyAndBody(rest, "!", includes);
    then
      TEMPLATE_COND({KeyBody(key, true, body)}, {}) :: newBody;
    case ("$" :: "#" :: rest, includes) equation
      (keyAndSep, body, newBody) = FindKeyAndBody(rest, "#", includes);
      (sep, key) = FindSepAndVar(stringListStringChar(keyAndSep), "", "");
    then
      TEMPLATE_FOR_EACH(key, sep, body) :: newBody;
    case ("$" :: "t" :: "h" :: "i" :: "s" :: "$" :: rest, includes) equation
      newBody = CompileTemplate_Old(rest, includes);
    then
      TEMPLATE_CURRENT_VALUE() :: newBody;
    case ("$" :: "_" :: rest, includes) equation
      (sep, body, newBody) = FindKeyAndBody(rest, "_", includes);
    then
      TEMPLATE_ADD_INDENTATION(sep, body) :: newBody;
    case ("$" :: "^" :: rest, includes) equation
      (keyAndSep, afterBody) = FindKey(rest, "");
      (sep, key) = FindSepAndVar(stringListStringChar(keyAndSep), "", "");
      newBody = CompileTemplate_Old(afterBody, includes);
    then
      TEMPLATE_RECURSION(key, sep) :: newBody;
    case ("$" :: ":" :: rest, includes) equation
      (key, afterBody) = FindKey(rest, "");
      body = GetTemplateInclude(includes, key);
      newBody = CompileTemplate_Old(afterBody, includes);
    then
      // Including a body opens a new scope; as does adding a 0-deep indentation level
      TEMPLATE_ADD_INDENTATION("", body) :: newBody;
    case ("$" :: char :: rest, includes) equation
      false = (char ==& "^");
      false = (char ==& "_");
```

```
      false = (char ==& "=");
      false = (char ==& "!");
      false = (char ==& "#");
      false = (char ==& ":");
      (key, afterBody) = FindKey(char :: rest, "");
      newBody = CompileTemplate_Old(afterBody, includes);
    then
      TEMPLATE_LOOKUP_KEY(key) :: newBody;

    case ((rest as "$" :: _), includes) equation
      textBody = flattenStringList(rest);
      textBody = "Couldn't match $: " +& textBody;
      Error.addMessage(Error.TEMPLCG_INVALID_TEMPLATE, {
          textBody});
    then fail();

    case ("\\" :: "n" :: rest, includes) equation
      newBody = CompileTemplate_Old(rest, includes);
    then
      TEMPLATE_TEXT("\n") :: TEMPLATE_INDENT() :: newBody;
    case ("\n" :: rest, includes) then
      CompileTemplate_Old(rest, includes);
    case (char :: rest, includes) equation
      false = char ==& "$";
      TEMPLATE_TEXT(text = textBody) :: body =
          CompileTemplate_Old(rest, includes);
      newTextBody = char +& textBody;
    then
      TEMPLATE_TEXT(newTextBody) :: body;
    case (char :: rest, includes) equation
      false = char ==& "$";
      newBody = CompileTemplate_Old(rest, includes);
    then
      TEMPLATE_TEXT(char) :: newBody;
  end matchcontinue;
end CompileTemplate_Old;

protected function FindKey
  input list<String> template;
  input String keyAcc;
  output String key;
  output list<String> afterKey;
algorithm
  (key, afterKey) := matchcontinue(template, keyAcc)
    local
      String char, out;
```

```
      list<String> rest, afterKey;
    case ({}, keyAcc) then fail();
    case ("$" :: rest, keyAcc) then (keyAcc, rest);
    case ("\n" :: rest, keyAcc) equation
      (out, afterKey) = FindKey(rest, keyAcc);
    then
      (out, afterKey);
    case (char :: rest, keyAcc) equation
      (out, afterKey) = FindKey(rest, keyAcc+&char);
    then
      (out, afterKey);
  end matchcontinue;
end FindKey;

protected function FindKeyAndBody
  input list<String> template;
  input String scopeEndChar; // "!", "=", "#"
  input list<TemplateInclude> includes;
  output String key;
  output TemplateTreeSequence body;
  output TemplateTreeSequence afterBody;
algorithm
  (body, afterKey) := matchcontinue(template, scopeEndChar,
      includes)
    local
      String key;
      list<String> afterKey, afterBody, bodyAcc;
      list<TemplateInclude> includes;
    case(template, scopeEndChar, includes) equation
      (key, afterKey) = FindKey(template, "");
      (bodyAcc, afterBody) = FindBody(afterKey,
          scopeEndChar, 0);
    then (key, CompileTemplate_Old(bodyAcc, includes),
        CompileTemplate_Old(afterBody, includes));
  end matchcontinue;
end FindKeyAndBody;

protected function FindBody
  input list<String> template;
  input String scopeEndChar; // "!", "=", "#"
  input Integer numNested;
  output list<String> body;
  output list<String> afterKey;
algorithm
  (body, afterKey) := matchcontinue(template, scopeEndChar,
      numNested)
```

```
    local
      String char;
      list<String> rest, afterKey, out;
    case ({}, _, _) then fail();
    case ("$" :: char :: rest, scopeEndChar, numNested)
        equation
      true = scopeEndChar ==& char;
      (out, afterKey) = FindBodySkipToEnd(rest,
          scopeEndChar, numNested+1);
    then ("$"::char::out, afterKey);
    case ("$" :: "/" :: char :: rest, scopeEndChar, 0)
        equation
      true = scopeEndChar ==& char;
    then ({}, rest);
    case ("$" :: "/" :: char :: rest, scopeEndChar,
        numNested) equation
      true = scopeEndChar ==& char;
      (out, afterKey) = FindBody(rest, scopeEndChar,
          numNested-1);
    then ("$"::"/"::char::out, afterKey);
    case (char :: rest, scopeEndChar, numNested) equation
      (out, afterKey) = FindBody(rest, scopeEndChar,
          numNested);
    then
      (char::out, afterKey);
  end matchcontinue;
end FindBody;

protected function FindBodySkipToEnd
  input list<String> template;
  input String scopeEndChar; // "!", "=", "#"
  input Integer numNested;
  output list<String> body;
  output list<String> afterKey;
algorithm
  (body, afterKey) := matchcontinue(template, scopeEndChar,
      numNested)
    local
      String char;
      list<String> rest, afterKey, out;
    case ("$" :: rest, scopeEndChar, numNested) equation
      (out, afterKey) = FindBody(rest, scopeEndChar,
          numNested);
    then ("$"::out, afterKey);
    case (char :: rest, scopeEndChar, numNested) equation
```

```
      (out, afterKey) = FindBodySkipToEnd(rest,
          scopeEndChar, numNested);
    then (char::out, afterKey);
  end matchcontinue;
end FindBodySkipToEnd;

protected function Lookup
  input TemplDict dict;
  input String key;
  input Environment curEnv;
  output Dict value;
protected
  list<String> split;
algorithm
  split := Util.stringSplitAtChar(key, ".");
  value := LookupCheckForIt(dict, split, curEnv);
end Lookup;

protected function LookupCheckForIt
  input TemplDict dict;
  input list<String> keys;
  input Environment curEnv;
  output Dict value;
algorithm
  value := matchcontinue (dict, keys, curEnv)
    local
      String key, string;
      list<String> rest;
      DictItemList newDict;
      list<DictItemList> dicts;
      TemplDict dict;
    case ({}, _, _) then
      fail();
    case (_, "it" :: {}, curEnv) equation
      ENV_STRING_LIST(strings = string :: _) = curEnv;
    then STRING(string);
    case (_, "it" :: rest, curEnv) equation
      ENV_DICT_LIST(dicts = dicts) = curEnv;
      newDict = Util.listFirst(dicts);
      dict = {{DictItem("", DICTIONARY(newDict))}};
    then Lookup2(dict, rest);
    case (dict, rest, _) then Lookup2(dict, rest);
  end matchcontinue;
end LookupCheckForIt;

protected function Lookup2
```

```
  input TemplDict dict;
  input list<String> keys;
  output Dict value;
algorithm
  value := matchcontinue (dict, keys)
    local
      String key;
      list<String> rest;
      DictItemList newDict;
    case ({}, _) then
      fail();
    case (dict, key :: {}) then
      GetDictItem_(dict, key);
    case (dict, key :: rest) equation
      DICTIONARY(dict = newDict) = GetDictItem_(dict, key);
    then
      Lookup2({newDict}, rest);
  end matchcontinue;
end Lookup2;

protected function GetDictItem_
  input TemplDict dict;
  input String key;
  output Dict value;
algorithm
  value := matchcontinue (dict, key)
    local
      DictItemList curDict;
      TemplDict rest;
    case ({}, _)
      equation
      // Error.addMessage(Error.DICT_NO_SUCH_KEY, {key});
      then
        fail();
    case (curDict :: rest, key)
      then GetDictItem2(curDict, key);
    case (curDict :: rest, key)
      then GetDictItem_(rest, key);
  end matchcontinue;
end GetDictItem_;

protected function GetDictItem2
  input DictItemList dict;
  input String key;
  output Dict value;
algorithm
```

```
  value := matchcontinue (dict, key)
    local
      Dict res;
      DictItemList rest;
      String lkey;
    case ({}, _) then fail();
    case (DictItem(key = lkey, dict = res) :: rest, key)
      equation
        true = (key ==& lkey);
      then res;
    case (_ :: rest, key)
      then GetDictItem2(rest, key);
  end matchcontinue;
end GetDictItem2;

public function ApplyCompiledTemplate
  input DictItemList dict;
  input TemplateTreeSequence tree;
algorithm
  ApplyCompiledTemplate_({dict}, tree, ENV_NULL(), tree, "", "");
end ApplyCompiledTemplate;

protected function IsEmpty "(Successfully looked up) values that are empty:
  empty DICTIONARY_LIST {}
  empty STRING_LIST {}
  empty STRING \"\"
  "
  input Dict value;
  output Boolean out;
algorithm
  out := matchcontinue(value)
  case STRING("") then true;
  case STRING_LIST({}) then true;
  case DICTIONARY_LIST({}) then true;
  case _ then false;
  end matchcontinue;
end IsEmpty;

protected function ApplyCompiledTemplate_Cond
  input TemplDict dict;
  input list<KeyBody> restBodies;
  input TemplateTreeSequence elseBody;
  input Environment curEnv;
  input TemplateTreeSequence treeCopy;
```

```
  input String sep;
  input String indent;
algorithm
  _ := matchcontinue (dict, restBodies, elseBody, curEnv,
      treeCopy, sep, indent)
    local
      Dict value;
      String key;
      TemplateTreeSequence elseBody, body;
    case (dict, {}, elseBody, curEnv, treeCopy, sep, indent
        ) equation
      ApplyCompiledTemplate_(dict, elseBody, ENV_NULL(),
          treeCopy, sep, indent);
    then ();

    /* IF_EXIST */
    case (dict, KeyBody(key, false, body)::restBodies,
        elseBody, curEnv, treeCopy, sep, indent) equation
      value = Lookup(dict, key, curEnv);
      false = IsEmpty(value);
      ApplyCompiledTemplate_(dict, body, ENV_NULL(),
          treeCopy, sep, indent);
    then ();

    /* IF_NOT_EXIST */
    case (dict, KeyBody(key, true, body)::restBodies,
        elseBody, curEnv, treeCopy, sep, indent) equation
      failure(_ = Lookup(dict, key, curEnv));
      ApplyCompiledTemplate_(dict, body, ENV_NULL(),
          treeCopy, sep, indent);
    then ();
    case (dict, KeyBody(key, true, body)::restBodies,
        elseBody, curEnv, treeCopy, sep, indent) equation
      value = Lookup(dict, key, curEnv);
      true = IsEmpty(value);
      ApplyCompiledTemplate_(dict, body, ENV_NULL(),
          treeCopy, sep, indent);
    then ();

    case (dict, _::restBodies, elseBody, curEnv, treeCopy,
        sep, indent) equation
      ApplyCompiledTemplate_Cond(dict, restBodies,
          elseBody, curEnv, treeCopy, sep, indent);
    then ();

  end matchcontinue;
```

```
end ApplyCompiledTemplate_Cond;

protected function ApplyCompiledTemplate_ForEach
  input Dict value;
  input TemplDict dict;
  input TemplateTreeSequence body;
  input Environment curEnv;
  input String sep;
  input String indent;
algorithm
  _ := matchcontinue (value, dict, body, curEnv, sep, indent)
    local
      list<String> strings;
      String string;
      DictItemList dictionary;
      Dict value;
      TemplDict dicts;
      Boolean isEmpty;
    case (STRING(string = string), dict, body, curEnv, sep, indent) equation
      ApplyCompiledTemplate_(dict, body, ENV_STRING_LIST({string}), body, sep, indent);
    then ();
    case (STRING_LIST(strings = strings), dict, body, curEnv, sep, indent) equation
      ApplyCompiledTemplate_(dict, body, ENV_STRING_LIST(strings), body, sep, indent);
    then ();
    case (DICTIONARY(dict = dictionary), dict, body, curEnv, sep, indent) equation
      ApplyCompiledTemplate_(dictionary :: dict, body, ENV_DICT_LIST({dictionary}), body, sep, indent);
    then ();
    case (DICTIONARY_LIST(dict = dicts), dict, body, curEnv, sep, indent) equation
      dictionary = Util.listFirst(dicts);
      ApplyCompiledTemplate_(dictionary :: dict, body, ENV_DICT_LIST(dicts), body, sep, indent);
    then ();
    case (DICTIONARY_LIST(dict = {}), dict, body, curEnv, sep, indent) then ();
  end matchcontinue;
end ApplyCompiledTemplate_ForEach;

protected function ApplyCompiledTemplate_
```

```
  input TemplDict dict;
  input TemplateTreeSequence tree;
  input Environment curEnv;
  input TemplateTreeSequence treeCopy;
  input String sep;
  input String indent;
algorithm
  _ := matchcontinue(dict, tree, curEnv, treeCopy, sep,
     indent)
    local
      TemplateTree first;
      TemplateTreeSequence rest, body, else_body;
      list<String> strings, envRest;
      String string, var, key, separator;
      DictItemList dictionary, dictEnv, dictEnvNext,
         topCurDict;
      Dict value;
      TemplDict dicts, dictEnvRest, restCurDict;
      list<KeyBody> restBodies;

      Boolean isEmpty;
      // Looping over ENV_STRING_LIST
    case (_, _, ENV_STRING_LIST( {} ), _, _, _) then ();
    case (_, {}, ENV_STRING_LIST( var :: {} ), _, _, _)
       then ();
    case (dict, {}, ENV_STRING_LIST(var :: envRest),
       treeCopy, sep, indent) equation
      Print.printBuf(sep);
      ApplyCompiledTemplate_(dict, treeCopy,
         ENV_STRING_LIST(envRest), treeCopy, sep, indent);
    then ();

      // Looping over ENV_DICT_LIST
    case (_, {}, ENV_DICT_LIST( {} ), _, _, _) then ();
    case (_, {}, ENV_DICT_LIST( dictEnv :: {} ), _, _, _)
       then ();
    case (topCurDict :: restCurDict, {}, ENV_DICT_LIST(
       dictEnv :: dictEnvNext :: dictEnvRest), treeCopy,
       sep, indent) equation
      Print.printBuf(sep);
      ApplyCompiledTemplate_(dictEnvNext :: restCurDict,
         treeCopy, ENV_DICT_LIST(dictEnvNext ::
         dictEnvRest), treeCopy, sep, indent);
    then ();

      // Looping over ENV_NULL
```

```
    case (_, {}, ENV_NULL(), _, _, _) then ();

      // Input for current iteration
    case (dict, TEMPLATE_TEXT(text = string) :: rest,
        curEnv, treeCopy, sep, indent) equation
      Print.printBuf(string);
      ApplyCompiledTemplate_(dict, rest, curEnv, treeCopy,
          sep, indent);
    then ();
    case (dict, TEMPLATE_ADD_INDENTATION(indent = string,
        body = body) :: rest, curEnv, treeCopy, sep, indent
        ) equation
      ApplyCompiledTemplate_(dict, body, ENV_NULL(), body,
          sep, string+&indent);
      ApplyCompiledTemplate_(dict, rest, curEnv, treeCopy,
          sep, indent);
    then ();
    case (dict, TEMPLATE_LOOKUP_KEY(key = key) :: rest,
        curEnv, treeCopy, sep, indent) equation
      STRING(string = string) = Lookup(dict, key, curEnv);
      Print.printBuf(string);
      ApplyCompiledTemplate_(dict, rest, curEnv, treeCopy,
          sep, indent);
    then ();
    case (dict, TEMPLATE_CURRENT_VALUE() :: rest, curEnv,
        treeCopy, sep, indent) equation
      ENV_STRING_LIST(strings = strings) = curEnv;
      string = Util.listFirst(strings);
      Print.printBuf(string);
      ApplyCompiledTemplate_(dict, rest, curEnv, treeCopy,
          sep, indent);
    then ();

    case (dict, TEMPLATE_INDENT() :: rest, curEnv,
        treeCopy, sep, indent) equation
      Print.printBuf(indent);
      ApplyCompiledTemplate_(dict, rest, curEnv, treeCopy,
          sep, indent);
    then ();

    case (dict, TEMPLATE_COND(cond_bodies = restBodies,
        else_body = else_body) :: rest, curEnv, treeCopy,
        sep, indent) equation
      ApplyCompiledTemplate_Cond(dict, restBodies,
          else_body, curEnv, treeCopy, sep, indent);
```

```
        ApplyCompiledTemplate_(dict, rest, curEnv, treeCopy,
            sep, indent);
      then ();

      case (dict, TEMPLATE_FOR_EACH(key = key, separator =
          separator, body = body) :: rest, curEnv, treeCopy,
          sep, indent) equation
        value = Lookup(dict, key, curEnv);
        separator = Unescape(separator, indent);
        ApplyCompiledTemplate_ForEach(value, dict, body,
            curEnv, separator, indent);
        ApplyCompiledTemplate_(dict, rest, curEnv, treeCopy,
            sep, indent);
      then ();

      case (topCurDict :: dict, TEMPLATE_RECURSION(key = key,
          indent = string) :: rest, curEnv, treeCopy, sep,
          indent) equation
        DICTIONARY(dict = dictionary) = Lookup(topCurDict::
            dict, key, curEnv);
        ApplyCompiledTemplate_(dictionary :: dict, treeCopy,
            ENV_NULL(), treeCopy, sep, indent+&string);
        ApplyCompiledTemplate_(topCurDict :: dict, rest,
            curEnv, treeCopy, sep, indent);
      then ();
      case (dict, TEMPLATE_RECURSION(key = key, indent =
          string) :: rest, curEnv, treeCopy, sep, indent)
          equation
        DICTIONARY_LIST(dict = {}) = Lookup(dict, key, curEnv
            );
        ApplyCompiledTemplate_(dict, rest, curEnv, treeCopy,
            sep, indent);
      then ();
      case (topCurDict :: dict, TEMPLATE_RECURSION(key = key,
          indent = string) :: rest, curEnv, treeCopy, sep,
          indent) equation
        DICTIONARY_LIST(dict = dicts) = Lookup(topCurDict::
            dict, key, curEnv);
        dictionary = Util.listFirst(dicts);
        ApplyCompiledTemplate_(dictionary :: dict, treeCopy,
            ENV_DICT_LIST(dicts), treeCopy, sep, indent+&
            string);
        ApplyCompiledTemplate_(topCurDict :: dict, rest,
            curEnv, treeCopy, sep, indent);
      then ();

```

```
    // Failures
    case (_, TEMPLATE_LOOKUP_KEY(key = key) :: rest, _, _, _, _) equation
      string = "LOOKUP_KEY(" +& key +& ")\n";
      Error.addMessage(Error.TEMPLCG_FAILED_TO_APPLY_TEMPLATE, {string});
    then fail();
    case (_, TEMPLATE_CURRENT_VALUE() :: rest, _, _, _, _) equation
      string = "CURRENT_VALUE()\n";
      Error.addMessage(Error.TEMPLCG_FAILED_TO_APPLY_TEMPLATE, {string});
    then fail();
    case (_, TEMPLATE_ADD_INDENTATION(indent = string) :: rest, _, _, _, _) equation
      string = "ADD_INDENTATION()\n";
      Error.addMessage(Error.TEMPLCG_FAILED_TO_APPLY_TEMPLATE, {string});
    then fail();
    case (_, TEMPLATE_COND(_, _) :: rest, _, _, _, _) equation
      string = "COND()\n";
      Error.addMessage(Error.TEMPLCG_FAILED_TO_APPLY_TEMPLATE, {string});
    then fail();
    case (_, TEMPLATE_FOR_EACH(key = key) :: rest, _, _, _, _) equation
      string = "FOR_EACH(" +& key +& ")\n";
      Error.addMessage(Error.TEMPLCG_FAILED_TO_APPLY_TEMPLATE, {string});
    then fail();
    case (_, TEMPLATE_RECURSION(key = key) :: rest, _, _, _, _) equation
      string = "RECURSION(" +& key +& ")\n";
      Error.addMessage(Error.TEMPLCG_FAILED_TO_APPLY_TEMPLATE, {string});
    then fail();

    /*case (dict, tree, _, _, _, _) equation
      print("Failed to apply compiled template: \n");
```

```
      PrintTemplateTreeSequence(tree);
      print("Dictionaries used: \n");
      PrintDictList(dict, "");
    then fail();*/
  end matchcontinue;
end ApplyCompiledTemplate_;

// Generic Utility functions
protected function FindSepAndVar
  input list<String> inStr;
  input String sepAcc;
  input String varAcc;
  output String sep;
  output String var;
algorithm
  (sep, var) := matchcontinue (inStr, sepAcc, varAcc)
    local
      list<String> rest;
      String char;
    case ({}, sepAcc, varAcc) then (sepAcc, varAcc);
    case ("#" :: char :: rest, "", varAcc) equation
      (sep, var) = FindSepAndVar(rest, char, varAcc);
    then
      (sep, var);
    case (char :: rest, "", varAcc) equation
      (sep, var) = FindSepAndVar(rest, "", varAcc +& char);
    then
      (sep, var);
    case (char :: rest, sepAcc, varAcc) equation
      (sep, var) = FindSepAndVar(rest, sepAcc +& char,
          varAcc);
    then
      (sep, var);
  end matchcontinue;
end FindSepAndVar;

protected function flattenStringList
  input list<String> lst;
  output String out;
algorithm
  out := matchcontinue lst
    local
      String char;
      list<String> rest;
    case {} then "";
    case char :: rest then char +& flattenStringList(rest);
```

```
  end matchcontinue;
end flattenStringList;


end TemplCG;
```

The code in Listing D.2 was used to produce the output in some of the examples in Appendix C. It was meant to be a proof-of-concept for traversing a recursive data structure and generating code for it.

Listing D.2. AST.mo

```
package AST

import TemplCG;
import Util;

// Note: Not all of the expressions will be handled by the
    example code

public
type Operator = String;

public
uniontype Exp
  record ICONST
    Integer integer "Integer constants" ;
  end ICONST;

  record RCONST
    Real real "Real constants" ;
  end RCONST;

  record SCONST
    String string "String constants" ;
  end SCONST;

  record BCONST
    Boolean bool "Bool constants" ;
  end BCONST;

  record CREF "component references, e.g. a.b{2}.c{1}"
    String componentRef; // Changed to string to simplify
  end CREF;

  record BINARY "Binary operations, e.g. a+4"
    Exp exp1;
```

```
    Operator operator;
    Exp exp2;
  end BINARY;

  record UNARY "Unary operations, -(4x)"
    Operator operator;
    Exp exp;
  end UNARY;

  record CAST "Cast operator"
    Exp exp;
  end CAST;

end Exp;

uniontype AST
  record AST_WHILE
    Exp cond;
    AST ast;
  end AST_WHILE;

  record AST_IF
    Exp cond;
    AST ast_if;
    AST ast_else;
  end AST_IF;

  record AST_EXP
    Exp exp;
  end AST_EXP;

  record AST_LIST
    list<AST> astlist;
  end AST_LIST;

  record AST_DEFINE
    Exp cref;
    Exp exp;
  end AST_DEFINE;
end AST;

constant AST constantTestTree =
  AST_LIST({
    AST_IF(BINARY(ICONST(1),"<",ICONST(2)),
      AST_LIST({
        AST_WHILE(BINARY(CREF("a"),"<",ICONST(2)),
```

```
            AST_DEFINE(CREF("a"),BINARY(CREF("a"),"-",ICONST(1)))),
          AST_DEFINE(CREF("a"),
            BINARY(CREF("a"),"*",
              BINARY(ICONST(4),"-",ICONST(1))))}),
        AST_LIST({})),
      AST_DEFINE(CREF("a"),
                BINARY(CREF("a"),"/",ICONST(3)))});

public function Exp_To_Dict
  input Exp exp;
  output TemplCG.DictItemList out;
algorithm
  out := matchcontinue(exp)
  local
    Integer integer;
    Exp exp1, exp2;
    TemplCG.DictItemList dict1,dict2;
    Operator op;
    String string;
    case (ICONST(integer)) equation
      string = intString(integer);
      then
      TemplCG.DictItem("IsIntConst", TemplCG.ENABLED()) ::
      TemplCG.DictItem("IntLiteral", TemplCG.STRING(string)) ::
      {};
    case (BINARY(exp1,op,exp2)) equation
      dict1 = Exp_To_Dict(exp1);
      dict2 = Exp_To_Dict(exp2);
      then
      TemplCG.DictItem("IsBinaryOp", TemplCG.ENABLED()) ::
      TemplCG.DictItem("Op", TemplCG.STRING(op)) ::
      TemplCG.DictItem("Exp1", TemplCG.DICTIONARY(dict1)) ::
      TemplCG.DictItem("Exp2", TemplCG.DICTIONARY(dict2)) ::
      {};
    case (CREF(string)) then
      TemplCG.DictItem("IsCRef", TemplCG.ENABLED()) ::
      TemplCG.DictItem("CRef", TemplCG.STRING(string)) ::
      {};
  end matchcontinue;
end Exp_To_Dict;

public function AST_To_Dict
```

```
  input AST ast;
  output TemplCG.DictItemList out;
algorithm
  out := matchcontinue(ast)
  local
    list<AST> astlist;
    Exp exp;
    AST ast,astelse;
    TemplCG.DictItemList dict1,dict2,expDict;
    TemplCG.TemplDict dictlist;
    String ref;
    case AST_DEFINE(CREF(ref),exp) equation
      expDict = Exp_To_Dict(exp);
      then
      TemplCG.DictItem("IsDefine", TemplCG.ENABLED()) ::
      TemplCG.DictItem("CRef", TemplCG.STRING(ref)) ::
      TemplCG.DictItem("Exp", TemplCG.DICTIONARY(expDict)) ::
      {};
    case AST_IF(exp,ast,astelse) equation
      dict1 = AST_To_Dict(ast);
      dict2 = AST_To_Dict(astelse);
      expDict = Exp_To_Dict(exp);
      then
      TemplCG.DictItem("IsIf", TemplCG.ENABLED()) ::
      TemplCG.DictItem("Cond", TemplCG.DICTIONARY(expDict)) ::
      TemplCG.DictItem("IfPart", TemplCG.DICTIONARY(dict1)) ::
      TemplCG.DictItem("ElsePart", TemplCG.DICTIONARY(dict2)) ::
      {};
    case AST_WHILE(exp,ast) equation
      dict1 = AST_To_Dict(ast);
      expDict = Exp_To_Dict(exp);
      then
      TemplCG.DictItem("IsWhile", TemplCG.ENABLED()) ::
      TemplCG.DictItem("Cond", TemplCG.DICTIONARY(expDict)) ::
      TemplCG.DictItem("AST", TemplCG.DICTIONARY(dict1)) ::
      {};
    case AST_EXP(exp) equation
      expDict = Exp_To_Dict(exp);
      then
      TemplCG.DictItem("IsExpression", TemplCG.ENABLED()) ::
```

```
      TemplCG.DictItem("Exp", TemplCG.DICTIONARY(expDict))
          ::
      {};
    case AST_LIST(astlist) equation
      dictlist = ASTList_To_DictList(astlist);
      then
      TemplCG.DictItem("IsASTList", TemplCG.ENABLED()) ::
      TemplCG.DictItem("List", TemplCG.DICTIONARY_LIST(
          dictlist)) ::
      {};
  end matchcontinue;
end AST_To_Dict;

public function ASTList_To_DictList
  input list<AST> astlist;
  output list<TemplCG.DictItemList> out;
algorithm
  out := Util.listMap(astlist, AST_To_Dict);
end ASTList_To_DictList;

end AST;
```