Towards a Text Generation Template Language for Modelica

Peter Fritzson[*], Pavol Privitzer[+], Martin Sjölund[*], Adrian Pop[*]
[+]Institute of Pathological Physiology, First Faculty of Medicine, University in Prague
[*]PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, SE-581 83 Linköping, Sweden
pavol.privitzer@if1.cuni.cz, {petfr, marsj,adrpo}@ida.liu.se

Abstract

The uses, needs, and requirements of a text generation template language for Modelica are discussed. A template language may allow more concise and readable programming of the generation of textual models, program code, or documents, from a structured model representation such as abstract syntax trees (AST). Applications can be found in generating simulation code in other programming languages from models, generation of specialized models for various applications, generation of documentation, web pages, etc. We present several template language designs and some usage examples, both C code generation and Modelica model generation. Implementation is done in the OpenModelica environment. Two designs are currently operational.

Keywords: template language, unparsing, pretty printing, code generation, Modelica.

1 Introduction

Traditionally, models in a modeling language such as Modelica are primarily used for simulation. However, the modeling community needs not only tools for simulation but also languages and tools to create, query, manipulate, and compose equation-based models. Examples are parallelization of models, optimization of models, checking and configuration of models, generation of program code, documentation and web pages from models.

If all this functionality is added to the model compiler, it tends to become large and complex.

An alternative idea that already to some extent has been explored in MetaModelica [9][21] is to add extensibility features to the modeling language. For example, a model package could contain model analysis and translation features that therefore are not needed in the model compiler. An example is a PDEs discretization scheme that could be expressed in the modeling language itself as part of a PDE package instead of being added internally to the model compiler.

Such transformation and analysis operations typically operate on abstract syntax tree (AST) representations of the model. Therefore the model needs to be converted to tree form by *parsing* before transformation, and later be converted back into text by the process of *unparsing*, also called *pretty printing*.

The MetaModelica work is primarily focused on mechanisms for mapping/transforming models as structured data (AST) into structured data (AST), which is needed in advanced symbolic transformations and compilers.

However, there is an important *subclass* of problems mapping structured data (AST) representations of models into text. Unparsing is one example. Generation of simulation code in C or some other language from a flattened model representation is another example. Yet another use case is model or document generation based on text templates where only (small) parts of the target text needs to be replaced.

We believe that providing a template language for Modelica may fulfill a need for an easier-to-use approach to a class of applications in model transformation based on conversion of structure into text. Particularly, we want to develop an operational template language that enables to retarget OpenModelica compiler simply by specifying a package of templates for the new target language.

1.1 Structure of the Paper

Section 2 tries to define the notion of template language, whereas Section 3 gives more detailed language design requirements, uses, motivation, and design principles. Section 4 shows an example of a very concise template language, its uses, and lessons learned. Section 5 presents model-view-controller separation which has important implications for the design. Section 6 presents a small interpreted template language prototype.

Section 8 briefly discusses applications in code generation from the OpenModelica compiler, whereas Sec-

DOI: 10.3384/ecp09430124

tion 9 presents related work, followed by conclusions in Section 10.

2 What is a Template Language?

In this section we try to be more precise regarding what is meant by the notion of template language.

2.1 Template Language

Definition 1. Template Language. A template language is a language for specifying the transformation of structured data into a textual target data representation, by the use of a parameterized object "the template" and constructs for specifying the template and the passing of actual parameters into the template.

One could generalize the notion of template language to cover target language representations that are not textual. However, in the following we only concern ourselves with textual template languages.

Definition 2. Template. A template is a function from a set of attributes/parameters to a textual data structure.

A template can also be viewed as a text string with holes in it. The holes are filled by evaluating expressions that are converted to text when evaluating the template body. More formally, we can use the definition from [17] (slightly adapted):

A template is a function that maps a set of attributes to a textual data structure. It can be specified via an alternating list of text strings, t_i, and expressions, e_i, that are functions of attributes a_i:

$$F(a_1, a_2, ..., a_m) ::= t_0\, e_0 ... t_i\, e_i\, t_{i+1} ... t_n\, e_n\, t_{n+1}$$

where t_i may be the empty string and e_i is restricted computationally and syntactically to enforce strict model-view separation, see Section 5 and [18]. The e_i are distinguished from the surrounding text strings by bracket symbols. Some design alternatives are angle brackets <...>, dollar sign $...$, combined <$...$>. Evaluating a template involves traversing and concatenating all t_i and e_i expression results.

Definition 3. Textual Data Structure. A textual data structure has text data such as strings of characters as leaf elements. Examples of textual data are: a string, a list (or nested list structure) of strings, an array of strings, or a text file containing a single (large) string. A textual data structure should efficiently be able to convert (flattened) into a string or text file.

2.2 Unparser Specification Language

Definition 4. Unparser Specification Language. A special case of template language which is tailored to specifying unparsers, i.e., programs that transform an abstract syntax (AST) program/model representation into nicely indented program/model text.

Example: The unparser specification language in the DICE system [3] was used to specify unparsers for the Pascal and Ada programming languages. The unparser specification was integrated with the abstract syntax tree specification, to which it referred. See also the example in Section 4.

3 Requirements and Motivation

What are our requirements on a template language for Modelica? Why don't use an existing template language, e.g. one of those mentioned in Section 9. In fact, do we need a template language extension at all? Why not just program this presumable rather "simple" task of converting structure into text by hand in an ordinary programming language? In the following we briefly discuss these issues.

- *Need for a template language*? Conversion of structure into text has of course been programmed many times by hand in a multitude of programming languages. For example, the unparser and the C code generator in the current OpenModelica compiler are hand implemented in MetaModelica. An advantage is usually good performance.

  However, the disadvantages include the lack of extensibility and modeling capability mentioned in Section 1. Another problem is that the code easily gets cluttered by a mix of (conditional) print statements and program logic. A third problem is reuse. For example, when generating target code in similar languages C, C#, or Java, large parts of the output is almost the same. It would be nice to re-use the common core of the code, instead of (as now) need to develop three versions with slight differences
- *Performance needs*. There are different performance needs depending on application. A template language that is mainly used for generation of html pages may need more flexibility in the order of text generation (lazy evaluation), whereas a language used to specify a code generation from AST needs higher performance. Compilation should not take too long even when you compile a hundred thousand lines of models represented as a million AST nodes.
- *Intended users*. Are the intended users just a few compiler specialists, or a larger group including modeling language users who wants easy-to-use tool extensibility?
- *Re-implement/re-use an existing template language*? Why not re-implement (or re-use) an existing tem-

plate language such as for example ST [17] for StringTemplate? This choice depends on the character of the existing language and its implementation, efficiency, and complexity of tool integration.

3.1 Language Design Principles

The following are language design principles [12]:

- *Conceptual clarity*. The language concepts are well defined.
- *Orthogonality*. The language constructs are "independent" and can be combined without restrictions.
- *Readability*. Programs in the language are "easy" to read for most developers.
- *Conciseness*. The resulting program is very short.
- *Expressive Powe*r. The language has powerful programming constructs.
- *Simplicity*. Few and easily understood constructs.
- *Generality*. Few general constructs instead of many special purpose constructs.

Some of these principles are in conflict. Conciseness makes it quick to write but often harder to read, not as easy to use, sometimes less general. Expressive power often conflicts with simplicity.

3.2 Language Embedding or Domain Specific Language?

Should the template language be a completely new language or should it be embedded into an existing language as a small extension to that language?

A language that addresses a specific problem domain is called *domain specific language* (DSL). DSLs can be categorized as *internal* or *external* [4][5].

Internal DSLs are particular ways of using a host language in a domain-specific way. This approach is used, e.g., for the pretty printer library in Haskell where document layouts are described using a set of operators/functions in a language-like way [23].

External DSLs have their own custom syntax and a separate parser is needed to process them. As an example, StringTemplate [18][17] is an external DSL and is provided for three different host languages: Java, C# and Python.

If you only need the template language for simple tasks, or tasks that do not require high performance and tight communication with the host language, a separate language might be the right choice. A small language may be quicker learn and focused on a specific task.

On the other hand, embedding into the host language makes it possible to re-use many facilities such as: efficient compilation, inheritance and specialization of templates, reuse of common programming constructs, existing development environment, etc., which otherwise need to be (partly) re-developed. A disadvantage is that the host language grows if the extension cannot be well separated from the host language.

Proliferation of DSLs might also be a problem. For example, consider a large application with extensive usage of, say, twenty different DSLs that may have incompatible and different semantics for language constructs with similar syntax. This might lead to a maintenance nightmare.

Also, what is exactly domain specific in a text template language? The answer is probably only the handling of the template text string with holes in it, switching between text mode and attribute expressions, and implicit concatenation of elements. All the rest, e.g., expression evaluation, function call, function definition, control structures, etc., can be essentially the same as in a general purpose language.

The design trade-offs in this matter are not easy and the authors of this paper do not (yet) completely agree on all choices. Therefore, in this paper we partly explore several design choices for a template language for Modelica.

4 A Concise Template Language

To make the basic ideas of a template language more concrete, we first present a very concise template language [4] which is primarily an unparser specification language. It has been used to specify unparsers for Pascal, Ada, and Modelica. Specifications are very compact. Implementation is simple and efficient.

We will use the following simple Modelica code example to illustrate this template language:

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

This code needs the abstract syntax tree nodes for its internal representation, specified as follows including small template language unparsing strings.

There are two statements nodes types: `ASSIGN` and `WHILE`. `ASSIGN` has two children,. `lhs` of type `PVAR` and `rhs` of type `EXPR`.

A typical assignment looks like `"variable := expression"`. The unparsing specification `"@1 := @2"` means: `@` signals a command that the next character has special interpretation. `@1` means: unparse the first child node. The following characters in the string `" := "` are just output as they are. The next command: `@2` means: unparse the second child of the `ASSIGN` node.

```
// Statement nodes STM
ASSIGN : (lhs: PVAR;
          rhs: EXPR) : "@1 := @2";
```

```
WHILE  : (condition: EXPR;
          statements: STM_LIST) :  "while
@1 loop @+@n @2;@n@q@-@nend while;@n"
```

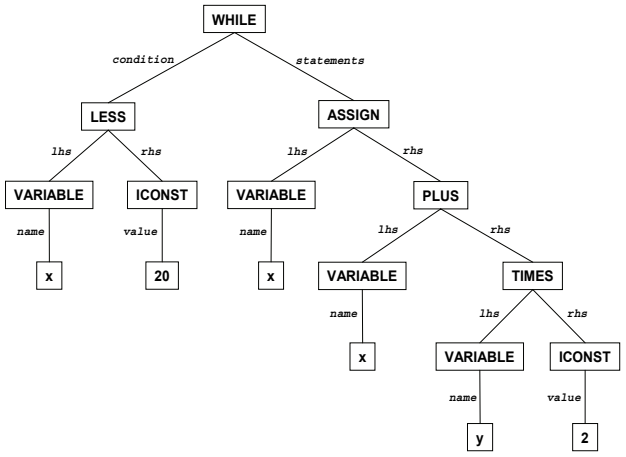

Figure 1. Abstract syntax tree of the while loop.

The template string for `while` has `statements` as a statement list. The semicolon `;` and new line `@n` between `@2` and `@q` (for quit) are emitted between each list item. `@+` and `@-` increase/decrease indentation level.

```
// Expression nodes EXPR
PLUS   : (lhs:EXPR; rhs: EXPR) :
                    "@1+@2" LPRIO 4;
TIMES  : (lhs:EXPR; rhs: EXPR) :
                    "@1*@2" LPRIO 5;
LESS   : (lhs:EXPR; rhs: EXPR) :
                    "@1<@2" BPRIO 3;
VARIABLE : (name: STRING)  : "@1";
ICONST   : (value: INTEGER): "@1";
```

The expression nodes also specify associativity and priority. The latter controls whether parentheses should be emitted. `LPRIO 4` means left associative, priority 4.

4.1 Usage Experience

The full abstract syntax and unparsing specification for Pascal is only 4 pages, and not that hard to write. The full Ada specification is 9 pages, still quite reasonable for a big language. Fifteen years later, such a specification was also developed for Modelica 1.2.

This became more complicated than the one for Ada. Also, maintenance became an issue, especially for other people than the original specification developer. People found the extremely concise unparsing template strings very hard to read and debug. Eventually we decided to rewrite the unparser into normal programming language code (mix of print statements and standard code). Not as elegant, but easier to maintain. Thus, conciseness made specifications short to write, but too hard to read and use/maintain. Another option could have been to redesign the language, e.g. introducing names instead of positions, but there was no time.

5 Model View Controller Separation

A strong design principle argued to especially relevant for template languages is model-view-controller separation [16]. First we define these terms in the context of a template language:

- Model – the data structure, e.g. an AST, to be converted to text according to the view.
- Controller – the piece of software that controls the application of the view to the model, e.g. a tree traversal algorithm applying the templates to the tree nodes.
- View – the mapping from attributes to text, i.e., the actual templates in a template language.

The value of this principle is strongly argued in [16], according to experience with the ST functional template language [17] in the StringTemplate system. Such separation gives more flexibility (multiple views), easier maintainability, better reuse, more ease-of-use, etc.

It is argued that the template language should be kept simple, program computation logic should not be too much intertwined with emitting text. If complex computation needs to be done, it should instead be done on the model (in our case the AST).

Our template language design has been strongly influenced by this principle.

6 A First Template Language for Modelica

A template language maps model items to text attributes (sometimes through intermediate stages). The attributes are referred to by named references in the templates. During template evaluation, the named references are replaced by the text values of these attributes. Thus, a template usually contains two items: a text with named placeholders, and a mapping from attribute names to text values, i.e. a dictionary.

In an advanced implementation (Section 7) the dictionary part can be left out if the template compiler is able to automatically map variable names to string values without an intermediary dictionary data structure.

In the rest of this section we present a first design of a simple template language based on the language embedding idea, together with some examples.

6.1 Text Output with a String Function

As previously mentioned in Section 2.1, a template is a function from structured data, e.g. record structures or abstract syntax trees, to a textual data structure, where the text can be returned as a string or output to a file.

Starting with a small code example:


```
while x < 20 loop ... end while;
```

This can be represented as an abstract syntax tree according to Section 7.3 Section 6.4, from which we have extracted two definitions:

```
uniontype Statement  "Algorithmic stmts"
 record WHILE  "While statement"
   Exp condition;
   list<Statement> statements;
 end WHILE;
end Statement;

uniontype Exp  "Expressions"
 record BINARY  "Binary operator"
   Exp lhs;
   Operator op;
   Exp rhs;
 end BINARY;
end Exp;

type AST = Statement; "Current AST type"
```

We would like to produce the following output from the example abstract syntax tree (AST):

```
The expression loops while x < 20.
```

Below we show three variants of Modelica functions producing this output, where the third one is based on the Modelica template language. Here we assume that an intermediary dictionary is not needed.

6.1.1 Function Returning a String

This function converts the AST example into a string by concatenating string pieces and using the built-in Modelica 3.1 `String` function to convert any record to a string. A locally defined `String` function can be defined within each record type definition (not shown here)

```
function mkString
  input AST whileStm;
  output String out :=
   "The expression loops while " + String(
     whileStm.condition.lhs.name) +
    " < " +  String(
    whileStm.condition.rhs.value) + ".";
end mkString;
```

6.1.2 Function with File Output

If we instead would like to output to a file without first concatenating strings, it might appear as follows:

```
function emitString
  input AST whileStm;
  input FILE file;
algorithm
  print(file,
    "The expression loops while ");
  print(file, String(
    whileStm.condition.lhs.name));
  print(file, " < ");
  print(file,
    String(whileStm.condition.rhs.value));
  print(file, ".");
end emitString;
```

6.1.3 Function Based on a Template

The following function uses the Modelica template language syntax defined in Section 6.3. The idea is to automatically generate the string function in Section 6.1.1 or the file output function in Section 6.1.2.

The escape-code << on a single line signals the start of the template section, and >> on a single line ends it. Text (excluding the first and last single lines) is just used verbatim. Pieces of text are automatically concatenated or output to a file. The escape-code <$ signals the beginning of some piece of Modelica code that should be automatically converted to a string, and $> ends it.

```
function templString
  input AST whileStm;
<<
The expression loops while
<$whileStm.condition.lhs.name$> <
<$whileStm.condition.rhs.value$>.
>>
end templString;
```

One can also let all template functions inherit common characteristics from a common base function, e.g.:

```
function templString
  extends TemplateFunction;
<<
...
>>
end templString;
```

6.1.4 Benefits of Template Functions

The main benefit of the text template approach is that the string conversion, concatenation, and file output code can be generated automatically instead of hand implemented, which increases readability and model-view-controller separation.

Another benefit supported by some template engines (e.g., StringTemplate [17]) is lazy evaluation – all the data structure pieces need not be evaluated in the order they are referred to in the template; instead evaluation is automatically delayed if needed, until the final result is output.

6.2 The Simple Template Language Dictionary

The simple template language dictionary used for lookup in the following small examples is defined below via the `DictItemList` constant, with a simple mapping from key to object. The number of datatypes that the dictionary can hold is very limited compared to more advanced template engines. The idea is that eve-

rything in the model is a Boolean, a string, a collection of strings, or a nested dictionary (to allow recursive datatypes). First we define the dictionary data types needed:

```
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 DictItemList = list<DictItem>;
```

Then we define a sample dictionary to be used in some of our examples:

```
constant DictItemList sampleDict = {
  DictItem("EnableText", ENABLED() ),

  DictItem("People", DICTIONARY_LIST( {
    DICTIONARY( {
     DictItem("Name", STRING("Adam")),
     DictItem("Fruits", STRING_LIST(
       {"Orange "} )
     } ),
    DICTIONARY( {
     DictItem("Name", STRING("Bertil")),
     DictItem("Fruits", STRING_LIST(
       {"Apple", "Banana", "Orange "} )
     } )
   }),

  DictItem("WHILE", ENABLED()),

  DictItem("condition",
   DICTIONARY( {
    DictItem("lhs", DICTIONARY({
     DictItem("VARIABLE", ENABLED()),
     DictItem("name",STRING("x"))
    })),
    DictItem("rhs", DICTIONARY({
     DictItem("ICONST", ENABLED()),
     DictItem("value",STRING("20"))
    }))
   }) )
};
```

6.3 Template Syntax

Below are the constructs used in the simple template language. Each construct contains the identifier used in the compiled template, as well as the character sequence used to construct it.

Note: This is a preliminary, rather cryptic syntax that was quick to implement by an interpreter. Below are also some examples of more readable Modelica syntax are shown for certain constructs.

A key is a string that does not contain any characters using $, or ", and does not start with #,!,=,^, or _. It is used for lookup of attributes from the dictionary environment. The dictionary environment is a simple linked environment where the current scope has the highest priority.

In the Modelica-syntax variant, `<$ $>` are used to contain Modelica code and/or attribute names.

`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 accessed recursively, but the last element can be of any type (e.g. `DICT1.DICT2.DICT3`.key).

6.3.1 Lookup of a Key Value

If `lookup(dict,key)` returns a string, this becomes the output.

Template syntax:

```
$key$
```

Modelica-like template syntax:

```
<$key$>
```

or a variant with explicit Modelica lookup syntax that can be used inside Modelica code context:

```
keyValue(dict,"key")
```

Example template:

```
The expression loops while
$condition.lhs.name$ <
$condition.rhs.value$.
```

Modelica-like example template:

```
The expression loops while
<$condition.lhs.name$> <
<$condition.rhs.value$>.
```

Example output:

```
The expression loops while x < 20.
```

6.3.2 Checking non-empty Attribute Values

If `lookup(dict,key)` returns any non-empty value (empty strings and lists are empty values), run body. The general syntax also includes elseif and else clauses.

Template syntax::

```
$=key$body$/=
```

Modelica-like template syntax (where `[]` means 0 or 1 times, { } means 0 or >= 1 times):

```
<$if key then$>body{<$elseif$>body}
[<$else$>body] <$end if$>
```

Abstract syntax:

```
COND(cond_bodies={(key,true,body)},else_bo
dy={})
```

Example template:

```
$=WHILE$This is a while expression.$/=
```

Modelica-like example template:

```
<$if WHILE then$>This is a while
expression.<$end if$>
```

Example output:

```
This is a while expression.
```

6.3.3 Checking for Empty Attribute Value

Checking for empty attribute values. The opposite of checking nonempty values.

Template syntax::

```
$!key$body$/!
```

Modelica-like template syntax (where `[]` means 0 or 1 times, { } means 0 or >= 1 times):

```
<$if not key$>
body {<$elseif$> body} [<$else$> body]
```

Abstract syntax:

```
COND(cond_bodies={(key,false,body)},else_b
ody={})
```

Example template:

```
$!ASSIGN$This is not an assignment.$/!
```

Modelica-like example template:

```
<$if not ASSIGN then$>This is not an
assignment.<$end if$>
```

Example output:

```
This is not an assignment.
```

6.3.4 For Each Iteration

Use `lookup(dict,key)` to fetch a `STRING_LIST`, `DICTIONARY` or `DICTIONARY_LIST` value, then iterate 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 result of each iteration.

In the Modelica syntax case, an ordinary array iterator { } is used to collect the results of the iterations, and the `insertSep` function to insert separator strings between the items.

Template syntax:

```
$#key[#sep]$body$/#
```

Modelica-like template syntax without separators:

```
<${$>body<$for this in <$key$>}$>
```

Modelica-like template syntax with separators:

```
<$insertSep( {$>body<$ for this in
<$key$>}, sep="...")$>
```

Abstract syntax:

```
FOR_EACH(...)
```

There is an example in the next section.

6.3.5 Current Item Value in Iterations

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

Template syntax:

```
$this$
```

Modelica-like template syntax:

```
<$this$>
```

Abstract syntax:

```
CURRENT_VALUE(...)
```

Example template with nested `for each` (first key is `People`, retrieving a dictionary list where each person dictionary has a key `Name` with string value and another key `Fruits` with string list value:

```
$#People$$Name$ has the following
fruits:\n
$#Fruits#, $$this$$/#\n
$/#
```

Modelica-like example template:

```
<${$><$Name$> has the following fruits:\n
<$insertSep($><$Fruits$><$, sep=", ")$>
<$for person in People}$>
```

Modelica-like example template with explicit `keyValue` calls:

```
<${keyValue(person,"Name")$> has the
following fruits:\n
<$ insertSep(keyValue(person,"Fruits"),
sep=", ") for person in People}$>
```

Output:

```
Adam has the following fruits:
Orange
Bertil has the following fruits:
Apple, Banana, Orange
```

6.3.6 Recursion

Use `lookup(dict,key)` to fetch a `DICTIONARY` or `DICTIONARY LIST` value. 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. The current auto-indentation depth is concatenated to the indent.

Note: the special construct for recursion on the current template is unnecessary in the Modelica syntax case, since you can just call the template with the same name. Calling templates is shown in Section 6.3.8.

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

Modelica-like template syntax, where each subtemplate to be called would need to be explicitly named:
```
<$subtemplate()$>
```

Abstract syntax:
```
RECURSION(...)
```

6.3.7 Increasing Indentation

Opens up a new scope and adds indent to the indentation level.

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

Abstract syntax:
```
ADD_INDENTATION(...)
```

Example template, where we use * instead of space to be more visible as indentation whitespace:
```
$_***$$=EnableText$\n
Listing all the people:\n
$^People#......$
$/=
$!EnableText$$Name$\n
$/!$/_
```

Output:
```
***Listing all the people:
***Adam
***......Bertil
***......
```

6.3.8 Calling a Pre-Compiled Template

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

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

Modelica-like template syntax:
```
<$subtemplate()$>
```

Abstract syntax:
```
INCLUDE(...)
```

Example template:
```
$:AddIndentationExample$$:CurrentValueExam
ple$
```

Modelica like example template:
```
<$AddIndentationExample()$>
<$CurrentValueExample()$>
```

Output:
```
    Listing all the people:
    Adam
    ......Bertil
    ......Adam has the following fruits:
Orange
Bertil has the following fruits:
Apple, Banana, Orange
```

6.4 Generating C Code from a While Loop

We return to the while loop example shown previously in Section 4, to be represented as an AST:

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

The abstract syntax types can be found in Section 7.3.

6.4.1 Small Template Language Example

Templates for emitting C code from the AST of a while loop:

```
$=WHILE$\n
while ($#condition$$:Exp$$/#) {
$^statements#  $\n
}
$/=
$=ASSIGN$
\n$lhs.name$ = $#rhs$$:Exp$$/#;
$/=

$=BINARY$
($^lhs$ $#op$$:op$$/# $^rhs$)
$/=
$=ICONST$    $=PLUS$  $=TIMES$  $=LESS$
$value$      +        *         <
$/=          $/=      $/=       $/=
$=VARIABLE$
$name$
$/=
```

7 Susan – A Compiled Template Language for Modelica

The template language shown in Section 6 (the concise cryptic syntax variant) was implemented as an interpreted external DSL that has both advantages and disadvantages. First the advantages:

- Strictly adheres to the model-view-controller separation as in [16].
- The language is small, and does not perform computation on the model, as advocated in [17].
- Simple to implement and modular.

There are also disadvantages:

- The non-Modelica syntax is cryptic, hard to read.
- Interpretation does not give enough performance.

As the next step we have developed an improved template language design and implementation called *Susan*, with the following main advantages:

- Presumable increased readability
- Compiled to gain maximum performance
- MVC separation is enforced in a more suitable way in context of MetaModelica as the host language

- The language is mature enough to provide a complete vehicle for target code generator specifications in the OpenModelica compiler (OMC) environment.
- The syntax and semantics complies with the MetaModelica type system for textual templates

To summarize, this is a functional, strongly typed, expression oriented template language.

7.1 MVC and Control

Susan's design is strongly influenced by the StringTemplate's (ST) [17] language, briefly described in Section 9.3, and below.

ST's control logic, i.e., conditional inclusion of template parts, is restricted to querying attributes only for their presence/absence or true/false values. This is designed to strictly prevent entanglement of Model and View (MVC). It is primarily obeying the rules *"the view cannot make data type assumptions"* and *"the view cannot compare dependent data values"* [16].

Before an ST template can be rendered to text the attribute values must be transferred to it completely. It is then the work of the Controller to bridge the gap from the Model to the template, e.g. extract data from a database, call some business logic on the Model or walk over an AST, and then transfer the proper values as template attributes.

Susan also transfers data from the Model to the template View, but integrates more control into the View in terms the match construct (Section 7.9).

7.2 Strongly Typed Templates

MetaModelica extends the Modelica type system with union types to facilitate construction of tree-like data structures, in particular Abstract Syntax Trees (ASTs) for efficient modeling of languages.

In our early interpreted template language design we have been using a simple template dictionary (Section 6.2) as an analogy to ST's object model. While general and simple the creation and dynamic lookup implies a certain performance loss.

In order to increase efficiency, we need to avoid the dictionary. As a consequence, templates should be able to directly access MetaModelica data structures. This lead us to strongly typed templates with read-only semantics, with some more control included.

Making templates strongly typed has advantages like generating more efficient code, and avoiding errors that otherwise might occur in applications if only dynamic typing would be used.

7.3 Template Package Type Views

Templates in the Susan language are grouped in *packages*. Each template package can import one or more *type views*, i.e., sets of AST type definitions. Each type view uses MetaModelica syntax and resides in a separate file. Here we will use a type view that can model the while loop example from Section 4:

```
package OriginalPackageName

uniontype Statement  "Algorithmic stmts"
  record ASSIGN  "An assignment stmt"
    Exp lhs; Exp rhs;
  end ASSIGN;

  record WHILE  "A while statement"
    Exp condition;
    list<Statement> statements;
  end WHILE;
end Statement;

uniontype Exp  "Expression nodes"
  record ICONST  "Integer constant value"
    Integer value;
  end ICONST;

  record VARIABLE "Variable reference"
    String name;
  end VARIABLE;

  record BINARY  "Binary ops"
    Exp lhs; Operator op;  Exp rhs;
  end BINARY;
end Exp;

uniontype Operator
  record PLUS end PLUS;
  record TIMES end TIMES;
  record LESS end LESS;
end Operator;

end OriginalPackageName;
```

The `OriginalPackageName` is the name of the original MetaModelica package where types included in the type view are fully defined. A type view can use types from several packages. It usually specifies a subset of the original types defined in several packages and from these types suitable parts can be selected. For example, there can be additional union tags in the `Statement` type, but only those two specified can be used by templates that use this view. Similarly, more record fields can be originally defined in the `ASSIGN` record but only `lhs` and `rhs` can be read inside the template package with the view imported.

AST type view files can be shared across different target languages as a kind of type interface to the compiler generated output ASTs (e.g., simulation code ASTs). It is also an essential feature to support scenarios where users are not allowed to see all original types (e.g., a commercial Modelica compiler) but still can see

and use the intended subset to extend the code generator.

In addition to type views, templates automatically understand all MetaModelica built-in types: `String`, `Boolean`, `Integer`, `Real`, `list`, `Option`, `tuple`, and `Array` types.

7.4 Template Definition

A *template definition* in Susan has a C-like function signature with a name and formal typed arguments, instead of a Modelica-like signature as in the design of Section 6 The body is a single template expression without explicit delimiters:

```
templ-name(Type1 n1, Type2 n2, ...) ::=
                        template-expression
```

A template's textual output is the result of the template expression evaluated with the actual parameter values in its scope. All parameters are input and read-only; in general, all values bound to names are read-only inside template expressions.

Unlike ST, which uses *dynamic scoping* of *attributes*, this language uses *lexical scoping*. After the symbol `::=`, a new lexical scope is created for template parameters that are only accessible by their names inside the scope. Nested lexical scopes can also be created by other constructs, e.g. in map expressions.

ST uses the concept of an implicitly available *default attribute*, named `it`, to decrease the verbosity in some common expression forms. This concept has been adapted for Susan as an implicitly available variable.

In the following sections we provide short descriptions of the five kinds of Susan's expressions:

Textual template expressions, named value references, template calls, match and *conditional expressions, and map expressions.*

7.5 Textual Template Expressions

A fundamental concept used for *textual template expressions* is a "text with holes in it". An example is

```
'Dear Mr. <name>.'
```

When the expression is rendered to text, the value of the `name` parameter is filled into `<...>` angle-bracketed marked hole and the brackets are discarded.

We have chosen single quotes, unlike ST, because we wanted double quotes to be reserved for string constants, thus

```
"Dear Mrs. <nice>"
```

is a constant textual template expression precisely following Modelica string syntax without any holes, and it respects ordinary escape characters like `"\n"` or `"\t"` for new line and tab characters.

To support readability (or verbatimness) of templates to the maximum extent, the `<<...>>` delimiting pair can be also used for longer templates with holes as follows, where there is a rule that a new line right after the opening delimiter and a new line right before the closing delimiter are ignored:

```
<<
Hi '<name>',
today is <dayName>.
>>
```

There is an equivalent to `<<...>>` for longer constant texts, the `%X...X%` verbatim string delimiting pair, where the `X` can be an arbitrary character where pairs of `()` `[]` `{}` are respected like

```
%(
\\ (Really) '<verbatim>' "text\n"
)%
```

or like

```
%*Some shining <*> is over there!*%
```

Everything inside the `%X...X%` is taken verbatim with complete lack of escapes.

We have provided the basis for the text part of the language, e.g. used in this complete template example:

```
hello(String person) ::= <<
Hello <person>!
>>
```

7.6 Named Value References

In the previous section, *named value references* were already used in the examples. A value can be referred by name when it is in the scope of the expression.

Automatic to-string conversion applies for all primitive MetaModelica types (`String`, `Integer`, `Real`, `Boolean`) and for all generic types of primitive types except of `tuple` types, i.e., `list`, `Option` and `Array`. Examples of automatic to-string conversion:

```
templ1(Integer i, Real r, Boolean b)::=
  'Is <b> that <i> = <r>?'

templ2(list<String> names,
       Option<Integer> optId) ::=
  'allNames<optId> = "<names>";'

templ3(String hello) ::= hello
```

`Option` typed values are output conditionally when they hold a value (the value of `SOME`). List types are output in sequence, i.e., effectively the concatenation of the string equivalents of their elements. These to-string conversion rules are elaborated recursively, that is, also a value of type `list<Option<Integer>>` is automatically to-string convertible.

For `list` and `Array` typed values a separator option can be specified right after the value name, like:

```
nameList(list<String> names) ::=
               'Names are: <names ", ">.'
```

There are more possible options for multi-valued expressions tailored for structuring the output text properly, see Section 7.11.

7.7 Template Calls

Templates can be *called* from other templates. Recursive calling of templates is allowed, too. The syntax is:

templ-name(*arg*$_1$, *arg*$_2$, …, *arg*$_n$)

where *templ-name* is the name of the called template, *arg*$_i$ are actual parameter expressions, and *n* can be 0 or more. Parameters are strongly typed with automatic to-string conversion when applicable. Usually actual parameters are named value references or other template expressions, but literal constants of `Integer`, `Real` and `Boolean` types can also be used (it is a sort of restriction to be able to create only non-structured constant values). Some examples:

```
sayN(String msg, Integer n) ::=
  'Say "<msg>" <n> times.'

say3(String msg) ::= sayN(msg,3)

whatToSay(String word) ::= <<
What to say?
<say3('Susan is <word>!')>
>>
```

7.8 Iterative Map Template Expressions

The *map template expression* is used to iterate over lists (or a scalar). It is conceptually similar to map functions heavily used in functional languages instead of imperative constructs like for-loops.

There are several possible design choices of syntax for this construct. The current choice (inspired by ST) is to use the colon (`:`) as a map operator:

value-expr **of** *elem-pattern* : *templ-expr*

However, `:` can be a bit cryptic and hard to see embedded in code. Other possibilities could be:

```
map(templ-expr, value-expr, elem-pattern)
```

or the Modelica iterative expression (without pattern):

```
templ-expr(x) for x in value-expr
```

The above means: "Map element(s) of the *value-expr* that matches *elem-pattern* using *templ-expr;* Concatenate results if they are multiple."

The redesigned part compared to ST is the **`of`** keyword that is a shortcut of meaning close to "*consists **of** element(s) like*". The colon "`:`" then creates a new nested scope for template invocation in an element-wise manner. If the *value-expr* is a scalar value it is treated as a single element.

Value-expr is usually a named value reference, but can be an external or intrinsic function call (see Section **Error! Reference source not found.**).

Elem-pattern is most often a single name value binding or a tuple pattern matching expression, but the same syntax and semantics applies here as for the pattern matching case rules in match-expressions. This, it can work as a filter for elements to be mapped, see the next section for more about patterns.

Templ-expr can be any valid template expression. For example,

```
gentlemen(list<String> names) ::= <<
Hello<names of name: ', Mr. <name>'>!
>>

pairList(
  list<tuple<String,Integer>> pairs
) ::= <<
Pairs:<pairs of (s,i):'(<i>,<s>)'", ">.
>>
```

where `name` binds each element value of `names` list to be used in the provided textual template after the "**`:`**" and the `pairList` template binds the two values of the `pairs` input parameter to map them with the textual template. The `", "` is the optional separator string that is used as a delimiter when concatenating the mapping results.

Map expressions can be used also for scalar typed values, most useful for `tuple` types, like

```
firstSI(tuple<String,Integer> pair) ::=
  pair of (s,_) : s
```

The implicit variable `it` is always implied after the "**`:`**", semantically as the "**`of`** `...`" clause is always rewritten to "**`of`** `it` **`as`** `elem-pattern`". The "**`of`** `...`" clause is then optional with the meaning "**`of`** `it`". Combining this with implicit referencing of `it` when omitting the parameter on a single parameter template call, the intention of the map expression is most succinct, for example:

```
intDecl(String varName) ::=
  'int <varName>;'

intDecls(list<String> varNames) ::= <<
/* integer local variables */
<varNames : intDef() \n>
>>
```

However, when the mapping template has more parameters, all of them must be explicit; while the implicit value can still be referred by the name '**`it`**'.

And again, we have specified an optional separator to new line in the form of unquoted escaped string `\n`. There are more options that are useful in various for-

matting scenarios, see Section 7.11 for their special syntax and semantics.

7.9 Match-Expressions

For example, consider the union type `Statement` from the type definition in Section 7.3. To read record values for an input value of the type in MetaModelica we might use a match-expression with positional pattern matching case rules like these (only fragments):

```
function statement
  input Statement inStatement;
...
match inStatement
  local
   Exp lhs, rhs;
   list<Statement> stmts;
case ASSIGN(lhs,rhs)
//lhs and rhs bound to respective values
   then ...;
case WHILE(stmts) equation
//stmts has value of statements here
...
```

Templates are supposed only to have *read* access to data structure (e.g. AST) attributes, making the usual local variable definitions unnecessary

The *match-expression* in the Susan language has the syntax:

```
[match value-expr]opt
  case pattern-expr  then template-expr
  case pattern-expr2 then template-expr2
  ...
```

Value-expr is usually a named value reference, but can also be an external or intrinsic function call.

The **match**... clause is optional, assumed to have the form **match** `it` when omitted. Each **case** opens a scope after **then**, with the record field names of the matched record node visible, e.g. `lhs` and `rhs` in the `ASSIGN` node. The `statement` function as a template:

```
statement(Statement stmt) ::=
 match stmt
  case ASSIGN  then
//lhs and rhs visible in the immediate scope
  …
  case w as WHILE then
  //w.statements visible while w not hidden
...
```

7.10 Conditional Expressions

Conditional expressions (or *if-expressions*) can be considered as syntactic variants of match-expressions. The general syntax is:

```
if cond-expr then template-expr
[else template-expr2]opt
```

where **if** *cond-expr* can be only have two forms:

```
if [not]opt value-expr …
if value-expr is [not]opt pattern-expr …
```

The first form is intended to query values for their zero-like values, enumerated by type:

```
Boolean             false/true
Integer and Real    0/non-0,
String, list and Array    empty/non-empty
Option              NONE/SOME.
```

The second form uses pattern matching and is, for the case without **not**, semantically equivalent to:

```
match value-expr
 case pattern-expr  then template-expr
 case _ /*the rest*/then template-expr2
```

For the case with **not**, the expressions after **then** are switched (unlike the patterns).

For all forms, when the **else** branch is not specified it is assumed to be the empty string.

7.11 Automatic Indentation and Options

Well indented documents and code are much easier to read than non-indented. Indentation levels are automatically and recursively tracked. For example,

```
lines2(list<String> lines) ::= <<
  <lines \n>
>>

lines4(list<String> lines) ::= <<
  <lines2(lines)>
>>
```

Giving a list of strings to the `lines2` template, all the strings are concatenated using new line as delimiter and indented by 2 spaces. Giving the same list to `lines4` template, the indentation becomes 4 spaces.

There is a set of (*template) expression options* that can be specified with following syntax:

```
<templ-expr sep; opt1=val1; opt2; ...>
```

We have already used the *separator* option in its short form. A separator option is applicable for all multi-result expressions (e.g., map expressions). It has also a *named option* equivalent (a fragment):

```
<lines; separator=\n>
```

Expression options can be specified only in the direct lexical context of **<...>** or (…). The latter is intended for expressions that occur in the top-most or a nested lexical context (e.g., after the **then** keyword), for example (fragments),

```
... ::= (lines \n)
... then (exps : exp(); separator=";\n")
```

In the above examples, the indentation is also applied after any new line embedded in the strings. Sometimes such behavior is not desirable.

There are four indentation controlling options: *anchor*, *absIndent*, *relIndent* and *indent*. They set integer values defaulting to 0 when unspecified. While active, their semantics says: ”apply my behavior when outputting the first non-space character *after* a new line”. Specifically, *anchor* means ”indent relative to where I started“, *absIndent* means ”indent absolutely“, *relIndent* means ”indent relative to actual indent“ and *indent* means ”break the rule, put my indent immediately and behave like relIndent”.

There are even more options, in addition to *separator* , where the most notable are *wrap* and *align*.

Combining indentation controlling options with wrapping/aligning options, most formatting scenarios can be addressed.

7.12 The While Example Using Susan

We have now prepared the ground for the complete while-loop example. Given these templates

```
statement(Statement stmt) ::=
 match stmt
  case ASSIGN then <<
<exp(lhs)> = <exp(rhs)>;
  >>
  case WHILE  then <<
while(<exp(condition)>) {
  <statements : statement() \n>
}
  >>

exp(Exp e1) ::=
 match e1
  case ICONST   then value
  case VARIABLE then name
  case BINARY   then
   '(<exp(lhs)> <oper(op)> <exp(rhs)>)'

oper(Operator) ::=
  case PLUS then "+"
  case TIMES then "*"
  case LESS then "<"
```

The `oper()` template uses the short form of the match. Being fed this ASTvalue of type `Statement`:

```
WHILE(
 BINARY( VARIABLE("x"),LESS(),ICONST(20)),
{ASSIGN( VARIABLE("x"),
   BINARY( VARIABLE("x"),
   PLUS(), BINARY( VARIABLE("y"),
           TIMES(),ICONST(2))))})
```

the `statement()` template will generate this text

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

7.13 The Susan Compiler

The Susan compiler translates source code in the Susan language into the MetaModelica language. The first prototype of the compiler was fully implemented in MetaModelica. Then, its own code generator was re-implemented using the Susan language.

8 Applications in Code Generation

The current code generation in OpenModelica 1.4.5 is hand implemented and transforms the DAELow AST into a list of strings which later is concatenated into the generated code. The only target language is C.

The new template-based code generation brings several advantages:

- Separation of concerns – developing a new code generator is much simpler.
- New target languages (e.g., generating Java code) can be added more easily to the code generator.
- Also end-users (modelers) can develop code generators, specified by template-based models, that can be dynamically linked into the compiler.

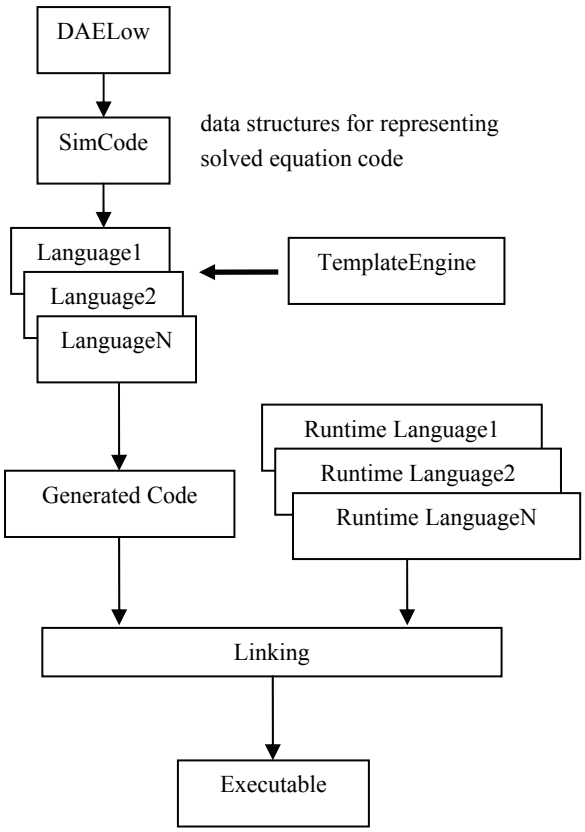

Figure 2. Usage of template-based code generators for producing target code in different languages.

9 Related Work

Template engines and languages can be used to generate code, documentation or web pages. Most of them claim to use a Model-View-Controller concept (MVC),

even though many violate some of the MVC principles. Many tools are based on Java and thus need to be fed XML data or Java classes.

9.1 Ctemplate from Google

Ctemplate [11] is a C++-based template engine that is less complex than most of the Java-based alternatives. The input is a basic dictionary structure. An example of a ctemplate template:

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

The code to use this template is rather complex [22].

9.2 Apache Velocity

Velocity [2] is a Java-based tool that generates output using templates. It is mainly used to serve webpages, SQL and PostScript but can also be used for code generation].

The data consists of Java classes that are fed to the engine. Velocity applies the classes to the template using directives like if-else, foreach (for iterable classes like lists) and can set/get its own variables inside the template. An example Velocity template:

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

9.3 StringTemplate

StringTemplate [18] with the ST language [17] is a template engine tightly integrated with ANTLR [1], including language bindings for Java, C++, and Python. It has been designed [16] to strictly enforce the MVC concept, and is mostly used for generation of web pages.

According to the main author, Terrence Parr [17] only four basic template constructs are needed:

- Attribute reference, `$name$ or <name>`.
- Conditional template inclusion based on presence/absence of an attribute, $if(flag)$text$endif$.
- Recursive template references.
- Template application to a multi-valued attribute (e.g. `names`) similar to lambda functions and LISPs map operator, `$names: templToApply()$`.

The template language, called ST, is actually a functional language. A template example follows:

```
("Hello, $name$\n" +
   "While you were gone $names;
      separator=\", \"$
      called you.",
    DefaultTemplateLexer.class);
```

Use of the template:

```
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() );
} }
```

Output:

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

9.4 Structured Representation Approaches

Invasive software composition [3] is somewhat related to template languages. Programs are decorated with hooks that can be replaced during composition. Operations are typically on abstract syntax instead of strings.

10 Conclusions

The uses, needs, and requirements of text generation template language for Modelica have been discussed.

Several template language designs and some usage examples and experience have been presented, both C code generation and Modelica model generation. There are difficult tradeoffs between different language design options regarding properties like generality, conciseness, consistency, efficiency, etc.

Three Modelica-related designs have been created. The first presented design is embedded in MetaModelica has not yet been implemented due to lack of resources. The second is a simple interpreted template language (as an external DSL) which was implemented and tried early on. The third (Susan) is a recently implemented compiled template language. It is efficient since it is compiled to MetaModelica. The language has several nice features and has already been used for its compilation to MetaModelica. However, some design remains and there is still discussion among the authors regarding the right syntax and semantics in some cases. The language looks very promising as a powerful tool for specifying code generation and similar tasks.

10.1 Future Work

The next mile-stone is to re-implement the code generator of the OpenModelica compiler using the Susan language, for at least two target languages (C/C++, C# and perhaps Java). This will further refine the design and implementation. Moreover, good tooling is important also for template languages. As a start, keyword coloring will soon be available in the OpenModelica MDT (Modelica Development Tooling) environment.

11 Acknowledgements

This work has been supported by Vinnova in the ITEA2 OPENPROD project, by the Swedish Research Council (VR), by the Czech National Research Programme, project No.2C06031, "e-Golem", and by Creative Connections s.r.o., Czech Republic. The Open Source Modelica Consortium supports the OpenModelica work. Peter Aronsson from MathCore Engineering AB gave useful feedback during the design.

References

[1] ANTLR. http://www.antlr.org. Access Nov 2007.

[2] Apache Software Foundation. Velocity Users Guide, 2008.: http://velocity.apache.org/engine/releases/velocity-1.6.1/user-guide.html. Jan 2009.

[3] Uwe Assmann. *Invasive Software Composition*. ISBN 3540443851, 9783540443858, 334 pages. Springer Verlag, 2003.

[4] Martin Fowler: *Domain Specific Language* http://www.martinfowler.com/bliki/ DomainSpecificLanguage.html.

[5] Martin Fowler. *Domain Specific Languages* http://martinfowler.com/dslwip/

[6] Peter Fritzson. *Towards a Distributed Programming Environment based on Incremental Compilation*. PhD thesis no 109, Linköping University, April 13, 1984.

[7] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. *Simulation News Europe*, 44/45, Dec 2005. http://www.openmodelica.org

[8] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 940 pages, Wiley-IEEE Press, 2004.

[9] Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In *Proc. of the 4th International Modelica Conference*, Hamburg, Germany, March 7-8, 2005.

[10] Peter Fritzson, Adrian Pop, Kristoffer Norling, and Mikael Blom. Comment- and Indentation Preserving Refactoring and Unparsing for Modelica. In *Proc. 6th Int. Modelica Conf. (Modelica'2008)*, Bielefeld, Germany, March.3-4, 2008.

[11] Google. ctemplate, 2008. http://code.google.com /p/google-ctemplate/. Accessed 2009.

[12] Kenneth C. Louden. *Programming Languages, Principles and Practice*. ISBN 0-534-95341-7, Thomson Brooks/Cole, 2003.

[13] Modelica Association. *The Modelica Language Specification Version 3.0*, September 2007. http://www.modelica.org.

[14] Martin Mikelsons. Prettyprinting in an interactive programming environment. In *Proc. of ACM SIGPLAN SIGOA symposium on Text manipulation*. Portland, Oregon, 1981.

[15] Eclipse website. http://www.eclipse.org. Referenced Nov 2007.

[16] Terence Parr. Enforcing Strict Model-View Separation in Template Engines. http://www. stringtemplate .org,. May 2004. Accessed May 2009.

[17] Terence Parr. [DRAFT] A Functional Language For Generating Structured Text. http://www.stringtemplate.org. May 2006. Accessed May 2009.

[18] Terence Parr. StringTemplate documentation. http://www.stringtemplate.org. Access May 2009.

[19] Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In *Proceedings of the 4th International Modelica Conference*, Hamburg, , March 7-8, 2005.

[20] Adrian Pop, Peter Fritzson, Andreas Remar, Elmir Jagudin, and David Akhvlediani. OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging. In *Proc 5th International Modelica Conf. (Modelica'2006)*, Vienna, Austria, Sept. 4-5, 2006.

[21] Adrian Pop. *Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages*. www.ep.liu.se. PhD Thesis No. 1183, June 5, 2008.

[22] Martin Sjölund. *Bidirectional External Function Interface Between Modelica/MetaModelica and Java*. Master Thesis. Linköping Univ, Aug. 2009.

[23] Philip Wadler. A Prettier Printer. *Journal of Functional Programming*,1998, pp 223-244. Draft version :homepages.inf.ed.ac.uk/wadler/papers /prettier/prettier.pdf Implementation PPrint by Daan Leijen

