A Natural Semantics specification for the equation-based modeling language Modelica

David Kågedal
LiTH-IDA-Ex-98/48

6th October 1998

Abstract

Modelica is a new modeling language. It is an object-oriented, equation-based languages that is designed to incorporate features from several previous modeling languages and to form a common ground for modeling and simulation. As a part of the design process, the exact semantics of the language needs to be defined.

Formal descriptions of the syntax of programming languages has long been accepted as a natural way of describing the syntactical form of a language. Today everybody expects a BNF-like grammar for a new language. But that is not the case when it comes to describing the semantics of languages. Formalisms for the specifications of semantics have been available for many years, but are not often used by others than researchers.

One advantage of using a formal description of the semantics is of course that the specification becomes more strict and unambiguous. However, there is also the possibility of using the semantic specification to generate a language translator, or compiler, in an automatic way.

This report describes a partial formal semantics for Modelica. The notation used enables it to be passed to a compiler generator that generates a Modelica translator, which translates a Modelica model into a simple set of equations.

The language used for the specification is called RML, and is based on Natural semantics, an operational semantic specification formalism.

Contents

Chapter 1

Background

1.1 Introduction

This chapter tries to give some background information outlining why this work was done, and what the basic concepts used are.

1.2 Overview of formal semantics

Any new programming language needs to be described in one way or another. The syntax of the language is conveniently described by a grammar in a common format, such as BNF[3]. However, when it comes to the semantics of a programming language, things are not so straightforward.

The semantics of the language describe what any particular program written in the language "means", i.e. what should happen when the program is executed, or evaluated in some form.

Most programming language semantic definitions are given by a standard text written in English which tries to give a complete and unambiguous specification of the language. Unfortunately, this is often not enough. Natural language is inherently not very exact, and it is hard to make sure that all special cases are covered. One of the most important requirements of a language specification is that two different language implementors should be able to read the specification and make implementations that interpret the specification in the same way. This implies that the specification must be exact and unambiguous.

For this reason formalisms for writing formal specifications of languages have been invented[9]. These formalisms allow for a mathematical semantic description, which as a consequence is exact, unambiguous and easy to verify.

Several approaches has been explored as to how the semantics should be described formally. See [12] for a good introduction to the various approaches.

1.3 Natural semantics

The formalism that we use is called ***Natural Semantics***. The "natural" in Natural semantics is intended to indicate its similarities to natural deduction, which is a way of constructing proofs in formal logic systems. A specification in Natural semantics is usually given a proof-theoretic interpretation, meaning that it is used to construct proofs.

There are many different presentations of Natural semantics in the literature, and I choose to present the simple form that RML uses.

Natural semantics uses ***inference rules*** to describe the semantics. These rules consist of a ***consequence*** and a number of ***premises***.

$$\frac{p_1 \Rightarrow q_1 \quad \dots \quad p_n \Rightarrow q_1}{p \Rightarrow q}$$

The number of premises may be zero, in which case we say that the rule is an ***axiom***.

$$p \Rightarrow q$$

1.3.1 A simple expression evaluator

The proof construction mechanism is described by a small example, describing a very basic expression language, with numbers and addition and multiplication operators. First, we define the abstract syntax of the expression language:

$$n \in \text{Int} \tag{1.1}$$
$$e \in \text{Exp} ::= n \mid e + e \mid e \cdot e \tag{1.2}$$

Then we give the rules for the evaluation of expressions to integer values.

$$n => n \tag{1.3}$$
$$\frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad n = n_1 + n_2}{e_1 + e_2 \Rightarrow n} \tag{1.4}$$
$$\frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad n = n_1 \cdot n_2}{e_1 \cdot e_2 \Rightarrow n} \tag{1.5}$$

If we want to prove, using these rules, that the expression $3 \cdot (4+5)$ has a value, we do this by finding a rule, whose consequence can be instantiated to match this expression. Such an instance is called a ***sequent***.

The only rule that matches is rule 1.5, leading to the following rule instance, where 3 is substituted for e_1 and $(4+5)$ is substituted for e_2:

$$\frac{3 \Rightarrow n_1 \quad (4+5) \Rightarrow n_2 \quad n = n_1 \cdot n_2}{3 \cdot (4+5) \Rightarrow n}$$

This is not a complete proof. We need to prove the sequent $3 \Rightarrow n_1$, and the only rule matching that is rule 1.3, leading to the following proof tree where 3 is substituted for n_1:

$$\frac{3 \Rightarrow 3 \quad (4+5) \Rightarrow n_2 \quad n = 3 \cdot n_2}{3 \cdot (4+5) \Rightarrow n}$$

The sequent labeled with a 3 also needs to be proved, and this can be done by instantiating rule 1.4.

$$\dfrac{3 \Rightarrow 3 \qquad \dfrac{4 \Rightarrow n_3 \quad 5 \Rightarrow n_4 \quad n_2 = n_3 + n_4}{4 + 5 \Rightarrow n_2} \qquad n = 3 \cdot n_2}{3 \cdot (4 + 5) \Rightarrow n}$$

Instantiating rule 1.3 twice leads to the following proof tree:

$$\dfrac{3 \Rightarrow 3 \qquad \dfrac{4 \Rightarrow 4 \quad 5 \Rightarrow 5 \quad n_2 = 4 + 5}{4 + 5 \Rightarrow n_2} \qquad n = 3 \cdot n_2}{3 \cdot (4 + 5) \Rightarrow n}$$

Now we can substitute 9 for n_2, and consequently 27 for n, leading to the complete proof tree.

$$\dfrac{3 \Rightarrow 3 \qquad \dfrac{4 \Rightarrow 4 \quad 5 \Rightarrow 5 \quad 9 = 4 + 5}{4 + 5 \Rightarrow 9} \qquad 27 = 3 \cdot 9}{3 \cdot (4 + 5) \Rightarrow 27}$$

Not only does this prove that the initial expression $3 \cdot (4 + 5)$ has a value, it also computes the value 27.

1.4 RML

One of the nice properties of Natural semantics is that it lends itself to efficient implementation in a computer. The semantic specification can be regarded as an operational description of how a language is interpreted or translated.

This is what RML[11] is all about. The RML language provides a simple text syntax for specifying rules and data types. This specification can then be compiled into an executable that can be used to interpret or translate any file in the described language.

1.4.1 RML syntax

The RML syntax borrows elements from languages like SML[7] to introduce a strict type system and a module system. The syntax for rules looks like the normal Natural semantics rule layout, adjusted for ASCII text.

The rules are grouped in ***relations***. A relation is a set of rules that have the same ***signature***, meaning that they perform the same operation, but are applicable at different occasions.

The type system includes basic types, such as `int`, `real` and `bool`. User-defined algebraic types may be declared in a SML-like syntax, as in the following example that declares the expression type used in the previous section.

```
datatype Exp = INT of int | ADD of Exp * Exp | MUL of Exp * Exp
```

With this type declaration, the expression $1 * (2 + 3)$ is represented as the RML value `MUL(INT(1),ADD(INT(2),INT(3)))`.

The signature of a relation can also be described as an RML type, using the => type constructor. A relation has a name, and each rule has a number of input arguments on the left side of the arrow, and a number of output arguments on the right side. The types of these arguments form the signature. The rules in the example above can be grouped in a relation that we call `eval`, with the following signature:

```
relation eval : Exp => int
```

The rules are placed inside the relation definition. It uses the built-in relations `int_add` and `int_mul` that perform integer addition an multiplication, respectively.

```
relation eval : Exp => int =

  axiom eval INT(i) => i

  rule  eval e1 => n1  &  eval e2 => n2  &  int_add(n1,n2) => n
        -----------------------------------------------------
        eval ADD(e1,e2) => n

  rule  eval e1 => n1  &  eval e2 => n2  &  int_mul(n1,n2) => n
        -----------------------------------------------------
        eval MUL(e1,e2) => n

end
```

1.5 Modelica

Modelica[8] is an object-oriented language for modeling of physical systems for the purpose of efficient simulation. The language unifies and generalizes previous object-oriented modeling languages designed by different companies and research institutions.

Compared with the widespread simulation languages previously available, this language offers three important advances:

- Non-causal modeling based on differential and algebraic equations.
- Multi-domain modeling capability, i.e. it is possible to combine electrical, mechanical, thermodynamic, hydraulic etc. model components within the same application model
- A general type system that unifies object-orientation, multiple inheritance, and templates within a single class construct.

A Modelica model is defined in terms of classes containing equations and definitions. The semantics of such a model is defined via translation of classes, instances, connections and functions into a flat set of constants, variables and equations. Equations are sorted and converted to assignment statements when possible. Strongly connected sets of equations are solved by using a symbolic or numeric solver.

Most tools that use Modelica will also let users define models and describe their relations using a graphical tool, with icons representing the models, and lines representing connections between them.

This section is not intended to teach Modelica, but to give an introduction to the basic ideas behind the language. See [4] for a more comprehensive introduction to the language.

1.5.1 Object-orientation

Modelica is an object-oriented modeling language, which means that the building blocks of a Modelica model are objects which correspond more or less to real-world object. When modeling an electrical circuit with a voltage source and a resistor, it will be built up by one object representing the voltage source and one representing the resistor. There will most likely also be other objects, such as connectors used for connecting the "pins" of the components.

This is a principle that has become very popular in the programming community, but it is still not very common in the modeling and simulation world, partly because the mathematical description using ***hybrid differential algebraic equations*** (***hybrid DAE***s) is not very familiar to the engineers who do the modeling.

1.5.2 The class concept

In Modelica, the basic means of structuring models is the ***class*** concept. Everything in Modelica is declared as a class. There are several keywords in the language that is used for declaring classes, depending on what kind of class is being defined. A common kind is ***model***, which is used for describing the main objects in a model definition.

Example 1.1 (A model) *The following piece of Modelica code defines a model named* `Resistor`.

```
model Resistor
  parameter Real R(unit = "Ohm");
  Pin n, p;
  Real v, i;
equation
  v = p.v - n.v;
  i = n.i;
  i = p.i;
  R * i = v;
end Resistor;
```

The definition contains two variables `i`*, and* `v`*, a parameter* `R`*, and two connector components* `p`*, and* `n`*, each containing* `i` *and* `v` *variables.*

The `Pin` in the example above is not defined as as `model`, but rather as a `connector`, which is a simpler variant of `model`, which is used to define connect models to each other. The connectors define interfaces (or ports), which connect to each other using the Modelica `connect` clause. Example 1.2 shows how the model definition `Resistor` can be used to build more complex models, in this case a silly circuit with a voltage source (`ACSource`) and two resistors connected in a loop.

Example 1.2 (A silly circuit)

```
model SillyCircuit
  Resistor r1(R = 1.0), r2(R = 2.0);
  ACSource ac(V = 10.0);
equation
  connect(r1.n, r2.p);
  connect(ac.n, r1.p);
  connect(r2.n, ac.p);
end SillyCircuit;
```

1.5.3 Hierarchical modeling

The basic principle when building models in Modelica is that of model reuse, and hierarchical modeling. This means that complex models are created by taking previously defined models (and connectors etc) and combining them into a new model definition that can be used just like any other model.

Example 1.1 show an example of this, where the connector `Pin` is reused twice.

1.6 The Modelica design group

The work on designing a new modeling language began in 1996 when a number of experts formed a group to design a new multi-paradigm modeling language for hybrid systems. Since the start, the group has expanded to include many new members from around Europe.

A first version of the Modelica language definition was finalized in September 1997, and there was much rejoicing. But no complete implementation existed yet, and there was actually a lot more work needed to make the specification comprehensive enough to be able to build a implementation for it.

The work on the RML specification of Modelica began in November 1997.

1.7 PELAB

The Programming Environment Laboratory, PELAB, (under the leadership of professor Peter Fritzson) is concerned with research in software engineering, i.e. tools and methods for the specification, development and maintenance of computer programs. Some examples are: programming languages, debuggers, incremental programming environments and compilers, compiler generators, tools for debugging and maintenance of distributed and real-time systems, compilers and programming environments for parallel computers, high-level environments and mathematical modeling languages and systems for scientific computing, program transformation systems, etc. The groups view of programming environment research is rather pragmatic, and the primary interest is in developing and investigating new methods and tools that have potential for practical applications, e.g. in support systems for software specialists. Developing such tools is very important, since most of the rising cost of computer systems is due to development, debugging and maintenance of software.

The Modelica language and development tools for Modelica are concrete examples of research done at PELAB.

Chapter 2

Motivation and scope of the thesis

This chapter describes the goals and the scope of this thesis.

2.1 Goal

The initial assignment was very simple in its formulation: "Write a formal semantic specification for Modelica using RML." There were at least three important reasons for doing exactly this.

2.1.1 A Modelica specification is needed

The Modelica design process was, and still is, in dire need of a formalized description of the language. The language specification available at the time (version 1.0 from September 1997)[4] did not contain much about the semantics of the language. This meant that nobody knew exactly how the language worked, and it was not certain that everybody in the design group agreed on the semantical details, since they were not clearly described anywhere.

Peter Fritzson is a member of the Modelica design group and there are several projects at PELAB that concerns Modelica. This project would be an important contribution to the Modelica design effort.

2.1.2 Gaining experience with RML

The RML language and compiler were developed at PELAB by Mikael Pettersson. They were new, and only a few language specifications had yet been written using it. More experience with various language specifications was desired to gain experience with its advantages and shortcomings.

2.1.3 Natural semantics and equation-based languages

Modelica is not a programming language, such as C or LISP. Instead it is a modeling language, with a different distinction between static and dynamic semantics. For this reason it was interesting to see how a formalism like Natural semantics, and more specifically RML, which is usually used to specify programming languages, could be applied to it.

2.2 Scope of the thesis

The work that forms the basis for this thesis only concerns the formal specification of the Modelica semantics. The initial goal was to write a complete semantic specification for Modelica 1.0. Unfortunately, Modelica 1.0 was not very well-defined, which means that considerable effort has been needed to figure out what the semantics were. This has been done by taking part of the Modelica design effort, including participation in design meetings and electronic discussions.

These discussions lay outside the scope of the thesis, and were not directly part of the work. However, it was needed to be able to write the formal semantics. Unfortunately, this lead to a chicken-and-egg problem, because until I had tried to write the specification I didn't know what the problems with the Modelica semantics were. These problems could then be discussed with the design group, leading to new ideas about how the semantics should have been written in the first place.

Another goal of the formal specification was to build a usable Modelica translator, which could be used for real-world applications. Since this partly conflicts with the desire to produce a clear semantic description, which not necessarily gives translation efficiency high priority, a decision needed to be made on what to focus on. I chose to concentrate on the semantic description, which shows in things like describing the semantics of equations between two matrices as a number of simple equations between the individual elements of the two matrices.

2.3 Limitations of the semantic specification

For the reasons outlined above, the semantic specification is far from complete. Even the language Modelica itself is still incomplete, but much effort is currently being put into making a new and complete specification of the language.

Most of the work has been put into describing the most important aspects of instantiation and expression analysis, while other things, like functions and other algorithmic aspects need more attention. Also more work is needed to add the polish to fix all details preventing things like instantiation of partial classes. But this is something that should not be done before the overall structure is fixed.

Below is a list of issues that are currently not covered by the semantic specification, or were implemented before it was decided how it should have been done. The limitations are listed in no particular order.

1. Class names can only denote classes, which means that it is not possible to define a "type synonym" for an array type. As an example, it is not possible to write `type Point = Real[3]`.

This might seem like a very basic feature to support, but it depends on properties of the type system which are not completely clear yet.

A type is a property of a piece of data. Instantiating a class gives an object of a certain class, and by examining the class definition it is possible to determine what the type will be *if the class is instantiated without modifications*. But the class may be modified giving the instance another, slightly different, type. For this reason it is not possible to say that a class name denotes a type, but instead it simply denotes a class definition. So the above definition of `Point` would not be a definition of a new type (in the type system sense, the Modelica keyword `type` is used for classifying class definitions). But what is it? It is some sort of "array of class definitions," but the semantic problems of this has not fully been brought to the attention of the design group.

2. Modifying whole array dimensions does not work, as in the example `AModel a(x[:,2] = [1.0,2.0,3.0])`. This is simply a limitation in the implementation. When the modifications are examined, it is unknown what the size of the component being modified is.

3. Array slices are not expanded in array equations.

4. The built-in pseudo-functions (size etc.) are not implemented. These are not functions in the normal sense, but are constructions which must be evaluated at instantiation-time, since they are often used to define the type (array sizes) of objects.

5. Actually no evaluation of function calls are performed during translation. This would be needed when the value of a constant (or parameter) is defined in terms of a function call.

6. Functions are implemented in a too simplistic way. Simple type-checking of arguments and a single return value is performed, but it only allows complete types, meaning that the sizes of array dimensions in input or output arguments must by fully specified, and can not depend on the actual arguments given in each function call.

   The function semantics is still uncertain. There have been discussions about a proposal where function calls are defined in terms of "implicit instantiation", where each function definition would be instantiated once for each function call in the model. This would make it possible to determine the full type of the arguments for each function call.

   However, this is currently hard to specify in this semantic specification due to RML limitations. The RML module which performs the instantiation is called `Inst` (see section A.8). It needs to do static analysis of expressions, and for that purpose it uses the relations in the module `Static`. But if the `Static` module encounters a function call it would need to do an implicit instantiation, which would require the use of the `Inst` module. Unfortunately, circular dependencies between modules are not allowed in RML. This can be worked around in several ways, but all of them would make the specification less structured or harder to understand.

7. The checking of component attributes is lacking severely. For instance, it is currently possible to use a variable in an outer scope. This should only be possible with constants, if at all.

8. Modifying class names is done even for non-local class names.

9. Unused modifications are simply discarded. They should probably produce an error, although this is not yet clearly defined in the language specification. This is also true for out-of-bounds array modifications.

10. Redeclarations are not implemented. This is simply because of time restrictions. It could probably be added without too much trouble.

11. The builtin type attributes are not completely implemented. This includes things like `final`, `partial`.

12. Assertions are lost during translation.

13. When instantiating an `extends` element, the extended class definition is instantiated in the scope where the `extends` element appears. This means that the extended class may refer to elements defined in the extending class, as long as it appears above the `extends` element.

14. There are some remaining bugs with the parts that are specified. And some of the most recent design decisions and clarifications of the Modelica semantics, such as array constructors, are not incorporated into the design.

Chapter 3

Development environment

3.1 Compiling RML

The RML compiler is currently at version 2.0. When the work on the Modelica semantics began, the RML version was at 1.5. The new compiler provided much improved error messages and freedom in writing source files.

Compilation of an RML file is done using the RML compiler driver `rmlc`. The driver first translates the RML specification to a C program using the `rml2c` compiler. Then it compiles the generated C file using a C compiler. We have used the GNU C compiler (GCC), but most modern C compilers should be usable.

3.2 Parser

The Modelica parser was generated by the PCCTS[10] compiler generation system. It generates a parser in C, which is linked with the RML object files. The glue between the languages is a special RML relation `parse`, which is implemented in C, rather than RML.

The parser only builds an abstract syntax tree (AST), and leaves the rest of the transformation logic to the RML code. Therefore, the parser itself is not further described in this report.

The parser generator in PCCTS is powerful, but it has a few problems, and is currently not maintained very actively. Because of problems with error handlers, the translator does not deal very well with syntax errors. If an error occurs while parsing, an error message will be printed, but the translator will try to translate the result of parsing anyway. Since the resulting AST will be severely broken, this will fail.

3.3 The report

This report was written using the LaTeX[6] text formatting system.

To produce the annotated semantics in appendix A a small program was written to convert the RML source files with comments to LaTeX source with the comments converted to ordinary text. This program is a quick'n'dirty Pike[1] hack, and is provided in appendix B

Chapter 4

Some important notions in the Modelica semantics

4.1 Introduction

The Modelica 1f specification[5] contains an informal description of the intended semantics of the language. Unfortunately, the description lacks considerably in detail, which means that this work needed to formalize things not covered by the specification. In some cases, the semantics was easy to guess, but in other cases, discussions with the authors and reading of design meeting minutes was needed to find out what was really intended.

This chapter complements the Modelica specification by filling in some of the larger holes in the informal language semantics. It is intended to help the understanding of the formal semantics in appendix A.

4.2 Terminology

There has been some confusion about the terminology regarding Modelica language elements, and currently the language definition is not very strict on this point. This will hopefully improve, but for now it is necessary to include a short list of terms here to specify what they mean.

Definition 4.1 (Element) *The term **element** corresponds closely to the non-terminal* `element` *in the grammar in the Modelica specification, which is a part of a class declaration. It can be either a class definition, a component declaration or an extends clause, that appears inside a class definition.*

Example 4.1 *The following class definition contains four elements. One* `extends` *statement, one class definition, and two component declarations. Note that the definitions of* `width` *and* `height` *are separate elements, but the grammar only needs one* `element` *rule to match them.*

```
class C
  extends OtherClass;
  type Length = Real(unit = "m");
  Length width, height;
end C;
```

The word ***instantiate*** is used in many language descriptions. In this specification it is used in two related, but slightly different, meanings.

Definition 4.2 (Instantiation) *Instantiating a class means creating an object (or class instance) from the description contained in the class definition, possibly modified by any given modifiers. This instance is not a run-time object that needs an allocated piece of memory, as in many dynamic object-oriented languages, but the principle is the same.*

For every component declaration in the class definition (possibly inherited from another class definitions using an `extends` *element), the class of the component declaration is also instantiated.*

When instantiating a class all of the elements in the definition of that class are also said to be instantiated. Instantiating an element means different things depending of what kind of element is instantiated. Basically it means to add the element to the partially instantiated parent (see the Modelica specification for an explanation of this).

- *Instantiating a class definition simply means adding the class definition to the partially instantiated parent.*
- *Instantiating a component declaration means instantiating the class of the component and adding that object to the partially instantiated parent.*
- *Instantiating an extends element means instantiating all the elements in the extended class.*

Example 4.2 *The following Modelica example contains three class definitions. If the last one (*`C`*) is instantiated, it means that the class* `B` *needs to be instantiated twice to create the subcomponents* `b1` *and* `b2`*.*

```
class B
  Real x;
end B;

class C
  B b1,b2;
end C;
```

Definition 4.3 (Component) *The term **Component** refers to a class instance. Many components are composed of several other components.*

Definition 4.4 (Immediate Subcomponent) *If component* A *is a structure which is composed of other components, each of these components is an **immediate subcomponent** of* A*.*

Definition 4.5 (Subcomponent) *This is a recursive definition of **subcomponent**. A subcomponent of a component* A *is either*

1. *An immediate subcomponent of* A*.*
2. *A subcomponent of an immediate subcomponent of* A*.*

Example 4.3 *Consider the following Modelica model:*

```
model A
  Real x;
end A;

model B
  A a;
  Real y;
end B;
```

If an instance of `B` is created under the name, `B` will contain two subcomponents referred to as `B.a.x` and `B.y`. Only `B.y` will be an immediate subcomponent of `B`.

Definition 4.6 (Atomic component) *A component that is not composed of subcomponent is called an **atomic component**.*

4.3 Types

4.3.1 Overview of the type system in Modelica

The type system is based on ***classes***. A class is the basic unit of modeling. It is used to modularize the model description and to give the models a hierarchical structure. Another important concept in the type system is that of ***arrays***.

The type system used in Modelica is based on a type system described by Luca Cardelli[2].

The definition of what a type is has been the subject of discussion in the Modelica group, but the following definition is used in this specification.

Definition 4.7 (Type) *A type is a property of components and expressions. It is defined as one of the following:*

1. *A built-in type (Real, Integer, String or Boolean). These correspond to the Modelica predefined classes `RealType`, `IntegerType`, `StringType` and `BooleanType`.*
2. *A structured type, containing a set of public components (N, T), where N is an identifier and T is a type. No two components in the set can have the same identifier.*
3. *An array of a type, where the size of the array is either unknown or a non-negative integer.*

Note that there is no special provision for multi-dimensional arrays. They are regarded as equal to arrays of arrays. A Modelica implementation might do it the other way around, regard arrays of arrays as multidimensional arrays. The important thing is that they are the same. Synonyms for two classes of arrays are used both in this specification and the Modelica language definition.

Definition 4.8 (Array synonyms) *The term **vector** is a synonym for a one-dimensional array. The term **matrix** is a synonym for a two-dimensional array.*

Definition 4.9 (Incomplete types) *If a type includes an undefined array size, it is an **incomplete type**. If not, it is a **complete type**.*

All components must have a complete type. All expressions also have complete types. Incomplete types only appear as the input argument types of function (see the discussion in section 4.5).

`record`	No equations are allowed in the definition or in any of its components. A `record` instance may not be used in connections.
`type`	May only be declared as derived from one of the predefined types, or as a array of a `type`.
`connector`	No equations are allowed in the definition or in any of its components.
`model`	A `model` instance may not be used in connections.
`block`	Each public component must be declared with one of the modifiers `input` or `output`.
`package`	May only contain declarations of classes and constants.
`function`	Each public component must be declared with one of the modifiers `input` or `output`. No equations are allowed and only one algorithm section is allowed.

Table 4.1: Class restrictions

4.3.2 Type equivalence and subtypes

The concepts of equivalent types and subtypes appear in several places in the specification.

Definition 4.10 (Subtype) *A type* T_1 *is a* ***subtype*** *of another type* T_2 *if and only if one of the following hold:*

1. T_1 *and* T_2 *are the same built-in type*
2. T_1 *and* T_2 *are structured types, where all public components in* T_2 *appear in* T_1*. The type of the component in* T_1 *must be a subtype of the type of the corresponding component in* T_2*.*
3. T_1 *is an array of type* T_1' *and* T_2 *is an array of type* T_2'*, where* T_1' *is a subtype of* T_2'*. Also either the size of* T_2 *is undefined or the size of* T_1 *is equal to the size of* T_2*.*

Definition 4.11 (Supertype) *If* T_1 *is a subtype of* T_2*, then* T_2 *is a* ***supertype*** *of* T_1*.*

Definition 4.12 (Equivalent types) *Two types* T_1 *and* T_2 *are said to be* ***equivalent types*** *if and only if* T_1 *is a subtype of* T_2 *and* T_2 *is a subtype of* T_1*.*

4.3.3 Class restrictions

A class can either be declared with the `class` keyword, or with one of its restricted forms, listed below. The restricted forms are used to indicate the intended use of the class, and to impose certain restrictions on what the class definition can contain, and how it can be used.

A goal of the Modelica design is that a valid program can always be transformed into another valid program by replacing all occurrences of the restricted class keywords with the keyword `class`. For a class declared with the keyword `class` to be able to be used as if it was declared with a restricted keyword, it has to adhere to the restrictions for that keyword.

As an example, this means that if a variable is used in a connection, its type has to be either a class declared with `connector`, or a class declared with `class` which contains no equations. Another implication is that an invalid program can be transformed into a valid program by replacing all restricted keywords with `class`, e.g. when incorrectly trying to use a `record` in a connection, because the keyword `record` has the same restrictions as the `connector` keyword, except that `record` is explicitly not allowed in connections, but `class` is, as long as it fulfills the restrictions.

It is currently not known if this goal will be fully met. In most cases it could be done, but things like `package`, `function` and `type` might be problematic.

4.4 Arrays and matrices

The concept of arrays in Modelica have not been thoroughly examined as of writing. Most of the basic properties of the type system should be clear, but there are some points worth noting.

4.4.1 Overloaded array operations

The exact meaning of multiplication where one or more of the operands are arrays is currently being defined. Similarly for other operations. The semantics specified here should not be regarded as an absolute reference to what these operations should mean, but rather as a description of how the overloading of operators works.

4.5 Functions

The semantics of functions is yet to be defined. What is clear is that it should be possible to use positional arguments that depend on the order of the `input` components of the function definition. It should also be possible to use named arguments matching against the names of the components in the definition. Furthermore, all uses of a function should be type-checked to some extent.

How all this integrates into the type system and the general semantics of Modelica is yet to be determined. For simple cases a very simplistic semantic description can be used, but for some cases, where the sizes of array dimensions are unknown and depend of each other, either an extended type system is needed, or other extensions, such as that of implicit function instantiation, needs to be introduced.

For the purposes of this specification, a simplistic model where all formal parameter types need to be fully determinable from the definition is used. The algorithmic (runtime) semantics is left unspecified.

4.6 Connections

Connections between objects are introduced by the `connect` construct in the equation part of a class declaration. The connections are between components called ***connectors***

Definition 4.13 (Connector) *A **connector** is an instance of a class defined using the* `connector` *keyword.*

The `connect` construct takes two references to connectors, each of which is either an element of the same class as the `connect` construct or an element of one of its components. The connectors are connected to produce equations according to the rules below. Each connector reference has either the syntactic form `c`, where `c` is a connector instance in the class containing the `connect` construct, or `m.c`, where `m` is the name of a component or an array of components of the class containing the `connect` construct, and `c` is the name of a connector variable in the component `m`.

4.6.1 Connection sets

To create equations from the `connect` constructs, the connected variables are collected into ***connection sets***.

Definition 4.14 (flow variables) *A variable can be declared as a **flow variable**, using the prefix* `flow` *in the declaration. Variables not declared as flow variables are called **non-flow variables**.*

Definition 4.15 (Connection sets) *A connection set is a set* C *of variables connected by means of* `connect` *constructs. A connection set contains either flow or non-flow variables, but never both.*

Definition 4.16 (Inner and outer connectors) *In an object* M*, each connector element of that class is called an* outer connector *with respect to* M*. Each connector element of elements of* M *is called an* inner connector *with respect to* M*.*

When a `connect(a,b)` construct is encountered while instantiating a class `C`, the component names `a` and `b` are checked to be of the same type (see section 4.3.2). Then `a` and `b` are decomposed into their public primitive components. Each subcomponent of `a` forms a connection set with its corresponding subcomponent of `b`.

If several `connect(a,b)` constructs are encountered during instantiation of a class, the resulting connection sets are merged to a collection of connection sets C so that if a component x is in two different connection sets S_1 and S_2, the union $S_1 \cup S_2$ is a subset of one of the sets in C. Each component is in exactly one of the sets in C.

4.6.2 Equations

Each connection set is used to generate one or more equations. In the case of variables declared with the `flow` type modifier, the equation generated is a

```
connector C
  Real x;
  flow Real y;
end C;

model A
  C con1, con2;
equation
  connect(con1,con2);
end A;

model M
  C con;
  A a1, a2;
  Real y;
equation
  connect(con,a1.con1);
  connect(a1.con1,a2.con1);
  connect(a1.con2,a2.con2);
  y = con.y;
end M;
```

Figure 4.1: `connect` example

sum-to-zero equation, as in equation 4.1. The coefficient d_ν is 1 if the component ν was added to the connection set from an inner connector, and -1 if it was added from an outer connector.

$$\sum_{\nu \in C} d_\nu \nu = 0 \tag{4.1}$$

In the case of non-flow components a number of simple equations is generated. If the connection set contains n components $c_1 \ldots c_n$, equations to the effect of $c_1 = c_2 = \cdots = c_n$ are generated, as in 4.2.

$$\begin{aligned} c_1 &= c_2 \\ c_2 &= c_3 \\ &\vdots \\ c_{n-1} &= c_n \end{aligned} \tag{4.2}$$

4.6.3 Example

This section contains a short example of the use of connections in Modelica, and shows the corresponding connection sets and generated equations. The Modelica source is shown in figure 4.1 and declares a connector `C` and models `A` and `M`. The model that is instantiated at the top level is `M`, as it is the last model in the file.

Connection sets are generated at two different levels in this example. While instantiating the model `M`, the three `connect` statements creates the connection sets described in figure 4.2, and the two instances of the model `A` (components `a1` and `a2` in model `M`) create the connection sets in figure 4.3. Each component in the sets is marked with a label indicating whether they were added from an inner or an outer connector.

$$\{\texttt{con.x}^{\text{outer}}, \texttt{a1.con1.x}^{\text{inner}}, \texttt{a2.con1.x}^{\text{inner}}\}$$
$$\{\texttt{con.y}^{\text{outer}}, \texttt{a1.con1.y}^{\text{inner}}, \texttt{a2.con1.y}^{\text{inner}}\}$$
$$\{\texttt{a1.con2.x}^{\text{inner}}, \texttt{a2.con2.x}^{\text{inner}}\}\ \{\texttt{a1.con2.y}^{\text{inner}}, \texttt{a2.con2.y}^{\text{inner}}\}$$

Figure 4.2: Connection sets from `M`

$$\{\texttt{a1.con1.x}^{\text{outer}}, \texttt{a1.con2.x}^{\text{outer}}\}\{\texttt{a1.con1.y}^{\text{outer}}, \texttt{a1.con2.y}^{\text{outer}}\}$$
$$\{\texttt{a2.con1.x}^{\text{outer}}, \texttt{a2.con2.x}^{\text{outer}}\}\{\texttt{a2.con1.y}^{\text{outer}}, \texttt{a2.con2.y}^{\text{outer}}\}$$

Figure 4.3: Connection sets from `A`

The equations generated from these connection sets, together with the last equation in the model `M` are shown in figure 4.4. Since the model connects the same two connectors several times, the resulting set of equations will of course be overdetermined.

4.7 Parameters

The concept of ***parameters*** is important in simulation, but the semantics of parameters vs. constants in the modeling language is not so obvious.

In principle, parameters are constants that the user may modify before a simulation run is started. The value of a parameter is constant throughout the simulation, but it should be possible to change it before simulation in a simple way. In description of parameters the notion of compile-time is often used, indicating that parameters are not constant during compilation, but constants are.

All this means that it is unclear to what depth the semantic specification should go. Should it try to describe the semantics of a model while the parameters are still unknown? Or should it assume some default value for the

$$\begin{aligned}
\texttt{con.x} &= \texttt{a1.con1.x} \\
\texttt{a1.con1.x} &= \texttt{a2.con1.x} \\
-\texttt{con.y} + \texttt{a1.con1.y} + \texttt{a2.con1.y} &= 0 \\
\texttt{a1.con2.x} &= \texttt{a2.con2.x} \\
\texttt{a1.con2.y} + \texttt{a2.con2.y} &= 0 \\
\texttt{a1.con1.x} &= \texttt{a1.con2.x} \\
-\texttt{a1.con1.y} - \texttt{a1.con2.x} &= 0 \\
\texttt{a2.con1.x} &= \texttt{a2.con2.x} \\
-\texttt{a2.con1.y} - \texttt{a2.con2.x} &= 0
\end{aligned}$$

Figure 4.4: Equations

```
model SinSource
  parameter Real amplitude;
  Real x;
equation
  x = amplitude * sin(time);
end SinSource;
```

Figure 4.5: Non-structural parameter

```
model Stack
  parameter Integer levels;
  Real v[levels];
  Real x,y;
protected
  Integer i;
equation
  x = v[1];
  y = v[levels];
  for i in 2:levels loop
    v[i-1] = v[i] * 0.8;
  end for;
end Stack;
```

Figure 4.6: Structural parameter

parameters?

In this work, all parameters are regarded as constants, using the default value given in the model definition. In some cases, the parameters are still possible to change after translation, but in other cases they are not (see section 4.7.1 below). This means that parameters and constants are treated in the same way for most practical purposes. The output still indicates whether they were declared as parameters or constants.

4.7.1 Structural parameters

When a parameter is used to give variable values, such as the parameter `amplitude` in the example in figure 4.5, there will be no problems changing its value after translating the model to simple equations.

However, when the parameter is used for more complicated things, such as defining the size of an array, it is usually not possible to change its value after translation to simple equations. Consider the example in figure 4.6. The size of the array `v` is determined by the parameter `levels`. The equation section first defines two equations, and then there is a loop which defines another `levels` − 1 equations. This means that the number of equations in the result of the translation will depend on the value of the parameter `levels`. Simply changing the value of `levels` after the translation is done will only change the value, but the number of equations will remain the same, creating an erroneous model.

Parameters which change the structure of the equations are called ***structural parameters***. A parameter is a structural parameter if at least one of the following conditions are fulfilled:

1. It appears in the dimension size of a array declaration.
2. It is used to index an array in any expression.
3. It is used in a range expression (`a:b` or `a:b:c`).
4. Its value is defined in terms of another structural parameter.

The conditions above are not verified to be absolutely correct. For instance, the second condition is probably too restrictive.

As this semantic specification treats all parameters as constants, this distinction between structural and non-structural parameters is not important. But for simulation environment implementors, this will be important. Most implementations will include a compilation step before the user gets to set parameters and start the simulation. The question is how well these implementations will handle the case when the user changes the value of a structural parameter. This is currently regarded as quality-of-implementation issue, but it might be worthwhile to examine it further.

Chapter 5

The Target language: Flat Modelica

The formal semantic specification in appendix A specifies the semantics of Modelica by showing how it is translated into another, simpler, language. The translation is a sort of flattening of the object-oriented structure of the model. This chapter describes the format of the output of the translator.

5.1 Requirements

There are different requirements that can be placed on the output format, all of which are not possible to meet simultaneously.

1. The output should contain all the information gathered from the analysis of the Modelica source.
2. The output should be easily read by a human.
3. The output should be easily parsed by a computer.
4. The output should be able to be fed to an existing equation solver.
5. The target language needs to be able to express everything that can be expressed by the Modelica language

The target language needs to be able to express everything that can be expressed by the Modelica language, but it may do so in a much less structured way. The major part of the translator output consists of a flat list of equations which are the result of instantiating the main model of the source file and its subcomponents.

5.2 Flat Modelica

The language produced by the translator is similar to Modelica, but with a few restrictions and extensions.

The following restrictions apply:

- Flat models contain only components of predefined types. Predefined attributes, e.g. `start`, `unit`, `min`, `max`, `...` are declared as separate variables.
- A flat model has one declaration section, optionally followed by an equation section, which is followed by zero or more algorithm sections.
- No modifications are used, i.e. equation modifications made in the Modelica source code are stated in the equation section of the flat representation. Redeclarations are taken care of while flattening the model.
- No connect statements are used, i.e. the corresponding connection equations are stated.
- For loops are not used in equations. They are unrolled in the flat representation.

The following extensions are made:

- All array components are declared separately. If the Modelica model contains the declaration

  ```
  Real x[2,3];
  ```

  the flat model will contain the following declarations:

  ```
  Real x[1,1];
  Real x[1,2];
  Real x[1,3];
  Real x[2,1];
  Real x[2,2];
  Real x[2,3];
  ```

  This means that the brackets in declaration do not denote array size, as they do in normal Modelica.

  This is a consequence of how the semantic specification looks at the components. Using the simple `x[2,3]` could be done instead, as it is shorter and introduces no ambiguities.
- Identifiers may contain dots (`.`) (resulting from subcomponents in the originating Modelica code). Identifiers may also contain subscripts, as in `x[1]`. This is a consequence of the fact that all array components are declared separately.
- Automatic type conversions defined in the Modelica semantics, e.g. from an `Integer i` to `Real`, are stated using the target type name as conversion operator, i.e. `Real(i)`.
- A new postfix subscripting operator is introduced. In the flat model, it is necessary to be able to subscript expressions, which allows expressions like `expression[x]`. This is not allowed in Modelica.
- A special kind of equation `ident ::= expression` is used to indicate that `ident` is not equal to `expression`, but rather *defined as* `expression`.

This is used for splitting equations between two expressions which contain subcomponents. If one of the expressions is not a simple name, a temporary name is introduced for that expression, and the name is used instead. If an ordinary equation between the temporary name and the expression was introduced, this would not solve the problem of defining the semantics of the complex equation, but just create a new, similar problem.

From a mathematical point of view, there is no difference in the meaning of `::=` and =.

- Almost no overloading of operators occurs in the flat Modelica. If a model contains an array multiplied by a scalar, the flat model does not use the multiplication sign, but an operator that is specialized for that particular operation.

  The normal integer and real arithmetic operators are still overloaded, but they should not, really.

- Due to the extensions a flat model representation is not valid Modelica. Therefore, the class definition in the flat code is declared as `fclass` to differentiate it from Modelica's `class`.

The output of the translator also has the following properties:

- The order of equations is undefined.
- The order of statements in each algorithm section is kept intact, but the order in which the algorithm sections appear is undefined.

The flat Modelica does not exactly meet the requirements described in the previous section.

1. Most information gathered from the analysis is included. Some type information is lost, though. The arithmetic operations on scalars and the equality sign are still overloaded. This is to improve readability. Changing the syntax to include this information is very easy.

2. The output looks reasonably much like Modelica, so a person knowledgeable of Modelica syntax should have no problem understanding the output.

3. Parsing of the flat Modelica has not been explored. It should possible to write a parser that will accept and understand flat Modelica. One thing that may cause trouble is the extension where array subscripting may be allowed on arbitrary expressions.

   A modified version of the translator which produces valid Modelica is currently being developed. The output of this version will be less readable than the current Flat Modelica, but will allow us to feed the output of the translator to another Modelica system.

4. Feeding the output to an equation solver needs either a special parser for the solver or a filtering step to rewrite the flat Modelica to a syntax understood by the solver.

5. Everything in Modelica is expressible in flat Modelica, although obviously, structural information is lost.

An example of Flat Modelica can be seen in figure 6.8.

Chapter 6

Translation overview

The semantic specification is written as a translation from Modelica to a target language called flat Modelica (see chapter 5). The translation is performed in several steps, which are outlined in this chapter.

A schematic view of the translation steps are show in figure 6.1. The box labeled "RML" shows what parts of the translation are performed and handled by the RML specification.

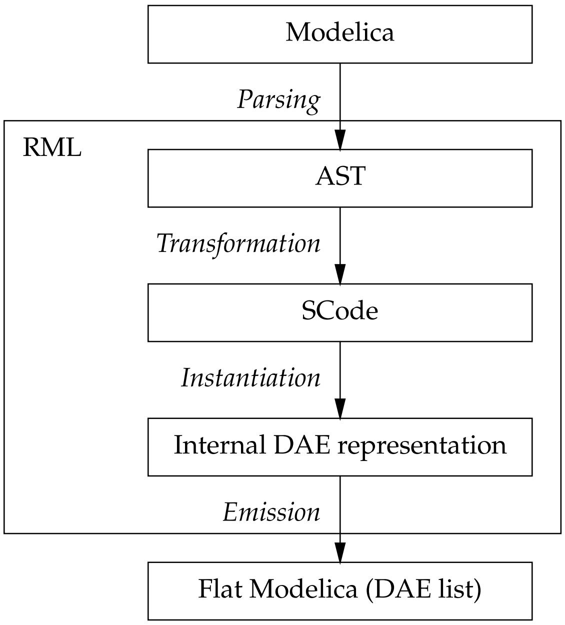

Figure 6.1: Translation steps

6.1 Lexical analysis and parsing

The lexical analyzer and parser are not really part of the semantic specification, and this report will not go into detail about the parsing of Modelica. It is based on the Modelica grammar as specified in the Modelica specification, and is written using a parser generator called PCCTS[10]. This generates a parser in C, which is separately compiled and linked with the RML specification.

The Modelica source file being translated is first passed to the lexical analyzer that scans the source for tokens, which it then passes on to the parser. The parser uses its grammar to build an RML representation of the contents of the source file, which is then used in the RML specification for further processing.

6.2 Abstract syntax

The ***abstract syntax*** is closely modeled after the syntactic structure, and is defined in the module `Absyn` (section A.3). It basically defines data structures for representing the structure of the source file (the ***abstract syntax tree***, or ***AST***), which are used by the parser to build a tree describing the source file. There is a datatype for representing a class definition, and the whole file is represented using the datatype `Program`, which is defined as a list of classes.

6.3 Preparing the AST: SCode

That the AST keeps the structure of the source file has several disadvantages when it comes to translating the program, especially if the translation rules should be easy to read for a human. To address this problem, a preparatory transformation pass is introduced which transforms the AST into an intermediate form, called ***SCode***. This intermediate form contains no more information than the abstract syntax, but merely restructures the tree representation in it. Besides some minor simplifications the SCode structure differs from the abstract syntax in the following important respects:

All components are described separately In the source and in the abstract syntax several components in a class definition can be declared at once, as in `Real x, y[17];`. In the SCode this is represented as two unrelated declarations, as if it had been written `Real x; Real y[17];`.

Class declaration sections In a Modelica class declaration the `public`, `protected`, `equation` and `algorithm` sections may be included in any number and in any order, with an implicit `public` section first. In the SCode these sections are collected so that all `public` and `protected` sections are combined into one section, while keeping the order of the elements. The information about which elements were in a `protected` section is stored with the element itself.

Modifications Modifications are stored using the data types in the `Mod` module (section A.10).

One might have thought that more work could be done at this stage, such as expression type analysis and name resolution. But due to the nature of the Modelica language, the only way to know anything about how the names will be resolved during instantiation is to do a more or less full instantiation. It is possible to analyze a class declaration and find out what the parts of the declaration would mean if the class was to be instantiated as-is, but since it is possible to modify much of the class while instantiating it that analysis would not be of much use.

6.4 Instantiation

The major part of the translation procedure, as described by the RML specification is the translation from SCode to hybrid differential-algebraic equations (hybrid DAEs).

A Modelica source file contains only definitions of models and other classes, which does not really have any semantic meaning in terms of equations, since the equations appear when a model definition is *used* to create an instance of that model.

To give a Modelica source file a meaning there is a convention that the last class, which should be a model, is instantiated as the top-level model. In a graphical modeling environment the situation will of course be different.

To go into detail exactly how the instantiation works here would be too lengthy, but a brief overview is given below.

6.4.1 Instantiation environment

When instantiating a class, certain information is available about what has been instantiated before, and what set of modifiers are currently applicable to the class being instantiated. Modelica uses lexical scoping (the language specification currently specifies dynamic scoping, but this will be changed in later versions), which means that when a class definition is saved in the environment, the current environment needs to be saved in the environment together with the class definition.

6.4.2 Class instantiation

Instantiation of a class is done by the relation `Inst.inst_class`. It goes through the class definition and instantiates all its elements, expanding all `extends` elements. The equations are added to the list of equations that are collected as the result of instantiation.

Before adding anything to the result, all names of components are converted to globally unique names, using the normal hierarchical naming scheme with dots between the name parts. If the topmost model contains a sub-model named `a`, which contains a variable `b`, the global name of this variable will be `a.b`.

```
model M
  real x[3], y[3];
equation
  x = y;
end M;
```

Figure 6.2: Array component equation

6.5 Static expression analysis

The module `Static` contains relations that perform static analysis and constant propagation on expressions. The primary reason to analyze expressions is to check their types, and make sure that there are no type errors. Constant propagation is used whenever the translator needs to know the actual value of an expression. The module is described in section A.17, with the following highlights.

- The relation `elab_exp` (A.17.2) analyzes an expression and produces two results: a new expression and a set of properties. The new expression is of type `Exp.Exp` (A.4) and is much more specific than the abstract syntax expressions. For each needed type conversion in the expression, an explicit type conversion operation is inserted, and the operators that are overloaded in the abstract syntax are replaced by non-overloaded counterparts.

  The set of properties describes the type of the expression (see section 4.3), and its ***variability***, which basically indicates whether the expression is constant or not.

- The relation `ceval` evaluates an expression and gives the result as a `Values.Value` (section A.16). This can obviously only be done for constant expressions.

6.6 Equation transformations

An equation in the equation section of a Modelica model can take several forms, and to describe the semantics of an equation, different transformations are applied.

6.6.1 Subcomponent equation expansion

When two arrays are equated, as in figure 6.2, the equation has the same meaning as if all the array elements were equated. Thus, in the output the equation is split into one equation for each pair of array elements, as in figure 6.3.

Likewise, if two structured components are equated, the equation is expanded to one equation for each pair of public components.

These two transformations causes problems in the output if either the left-hand side or the right-hand side of the equation is a complex expression and not a name of a component. The Modelica syntax does not allow the concept

```
model M
  real x[3], y[3];
equation
  x[1] = y[1];
  x[2] = y[2];
  x[3] = y[3];
end M;
```

Figure 6.3: Expanded array equations

FOUND_EQUATION	The class is now known to contain at least one equation.
NEWDEF	This class is not derived from another class with the `class A = B` construct.

Table 6.1: Class restriction inference events

of expression subscripting or of picking subcomponent from complex expressions. In the flat Modelica output (see chapter 5) this is solved in two different ways. For arrays, an expression subscripting operator is introduced, which allows things like `(A*B)[3,4]`. For expressions of a composite type, a new identifier is introduced, which is defined to be identical to the expression. Then this name is used to look up subcomponents. The name is constructed by concatenating the name `__TMP__` with a unique integer, forming a name that looks something like `__TMP__17`.

6.7 Class restriction inference

To be able to determine if a class definition or a variable can be used under certain circumstances it is necessary to check whether it adheres the restrictions currently imposed on it. If the class is declared with one of the restricted keywords listen in table 4.1, it is immediately known, but when the class is declared using the `class` keyword, the definition has to be scanned to see what restrictions it breaks.

This is done for each variable as it is instantiated, using the state machine in the file `classinf.rml` (see section A.5). While instantiating a class the current state is kept in a variable of type `ClassInf.State`. It is initialized with the keyword used to declare the class using the relation `ClassInf.start`. If the keyword was one of the restricted forms, the state will be initialized to a state that directly corresponds to that form, and it will never change to another state. If the keyword was `class`, the state is initialized to `ClassInf.UNKNOWN`.

Transitions in the state machine are triggered by events signaled during instantiation indicating that some piece of information has been encountered which affects the class restriction. The defined events are listed in table 6.1.

After instantiation of a class is complete, the state of the machine can be checked against a desired restriction to see if the class definition fulfills it, using the relation `valid` (A.5.5).

6.8 Representation of types

When instantiating, each component is assigned a type descriptor which is constructed from the type used in the component declaration combined with all applicable modifications.

The type descriptors are used in several places in the semantic specification, but two distinct types of usage can be noted.

Subcomponent lookup To look up components inside a component, a description of all its subcomponents are needed. This description must contain information about which components are protected. This kind of lookup is usually done in places where all type and component attributes are needed to determine such things as what components can be connected to each other.

Type checking of expressions Expressions are checked for type incompatibilities, which means that information about the types of components has to be known. In this context less information is needed. Either a component is of one of the built-in types, or it is a complex type, but then only the names and types of the public subcomponents need to be known.

These two needs could be met by two different data types, and a previous version of the specification did it this way. But having two data types that essentially does the same thing is more confusing than helpful, so they were reduced to only one.

6.9 DAE representation

The DAEs are collected in a data structure defined in the module `DAE` (section A.18). This structure is basically a linear list where each element is either a variable declaration, an equation or an algorithm.

6.10 Output of equations

As the last stage of the translation the internal representation of the hybrid DAEs are written out in the format described in chapter 5.

6.11 An example

To show how the translation of a Modelica model is done in a more visual manner, this section gives a small example of a model, and how it is translated to a flat representation.

6.11.1 The model

The source model is given in figure 6.4. It is a very simple electrical circuit, with an AC voltage source and two capacitors.

```
type Voltage = Real(unit="V");
type Current = Real(unit="A");

connector Pin
  Voltage      v;
  flow Current i;
end Pin;

partial model TwoPin "Superclass of elements with two electrical pins"
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;

model Capacitor "Ideal electrical capacitor"
  extends TwoPin;
  parameter Real C(unit="F") "Capacitance";
equation
  C*der(v) = i;
end Capacitor;

model VsourceAC "Sin-wave voltage source"
  extends TwoPin;
  parameter Voltage VA = 220 "Amplitude";
  parameter Real f(unit="Hz") = 50  "Frequency";
  constant  Real PI=3.141592653589793;
equation
  v = VA*sin(2*PI*f*time);
end VsourceAC;

model Circuit
  Capacitor  C1(C=0.01), C2(C=0.02);
  VsourceAC AC;
protected
  Current i;
equation
  connect (AC.p, C1.p);
  connect (AC.p, C2.p);
  connect (C1.n, AC.n);
  connect (C2.n, Ac.n);
  i = AC.i;
end Circuit;
```

Figure 6.4: Model source

```
model Circuit
  public Capacitor C1(C=0.01);
  public Capacitor C2(C=0.02);
  public VsourceAC AC;
  protected Current i;
equation
  connect (AC.p, C1.p);
  connect (AC.p, C2.p);
  connect (C1.n, AC.p);
  connect (C2.n, Ac.n);
  i = AC.i;
end Circuit;
```

Figure 6.5: SCode for `Circuit`

```
Real C1.p.v;
Real C1.p.i;
Real C1.n.v;
Real C1.n.i;
Real C1.v;
Real C1.i;
Real C1.C;
Real C2.p.v;
Real C2.p.i;
Real C2.n.v;
Real C2.n.i;
Real C2.v;
Real C2.i;
Real C2.C;
Real AC.p.v;
Real AC.p.i;
Real AC.n.v;
Real AC.n.i;
Real AC.v;
Real AC.i;
Real AC.VA;
Real AC.f;
Real AC.PI;
```

Figure 6.6: Global component names

6.11.2 First transformation

The first thing the translator does it to transform the abstract syntax tree representing the model into an SCode representation. This mainly effects the `Circuit` model definition, giving something that is described in figure 6.5, using a pseudo-Modelica representation.

6.11.3 Flattening

The main translation step does several things simultaneously. One important part is the flattening of the hierarchical structure. This means that all composite components are split up in subcomponents, and all atomic components will receive a new name that identifies it with a unique identifier in the global name space. For instance, the voltage variable in the positive connector in the capacitor named `C2` will get the global name `C2.p.v`.

The set of all components is listed with their global names in figure 6.6.

{`AC.p.v`, `C1.p.v`, `C2.p.v`} {`AC.p.i`, `C1.p.i`, `C2.p.i`}
{`AC.n.v`, `C1.n.v`, `C2.n.v`} {`AC.n.i`, `C1.n.i`, `C2.n.i`}

Figure 6.7: Connection sets

6.11.4 Connection equations

Generating equations from the connections in the model is done by collecting the connected variables into connection sets (see section 4.6.1). There will be one set for each set of connected voltage variables and one set for each set of connected current variables, giving the connection sets in figure 6.7. All connectors are inner connectors, which means that the connection sets will generate the equations given in figure 6.8.

6.11.5 Flat Modelica

The output of the translator is shown in figure 6.8, which is the Flat Modelica version of the source model. All the components are expanded into their simple components.

```
class Circuit
  Real C1.p.v;
  Real C1.p.i;
  Real C1.n.v;
  Real C1.n.i;
  Real C1.v;
  Real C1.i;
  parameter Real C1.C;

  Real C2.p.v;
  Real C2.p.i;
  Real C2.n.v;
  Real C2.n.i;
  Real C2.v;
  Real C2.i;
  parameter Real C2.C;
  Real AC.p.v;
  Real AC.p.i;
  Real AC.n.v;
  Real AC.n.i;
  Real AC.v;
  Real AC.i;
  parameter Real AC.VA;
  parameter Real AC.f;
  constant Real AC.PI;
  Real i;
equation
  C1.v = C1.p.v - C1.n.v;
  0 = C1.p.i + C1.n.i;
  C1.i = C1.p.i;
  C1.C*der(v) = C1.i;
  C2.v = C2.p.v - C2.n.v;
  0 = C2.p.i + C2.n.i;
  C2.i = C2.p.i;
  C2.C*der(v) = C2.i;
  AC.v = AC.p.v - AC.n.v;
  0 = AC.p.i + AC.n.i;
  AC.i = AC.p.i;
  AC.v = AC.VA*sin(2*AC.PI*AC.f*time);
  AC.p.v = C1.p.v;
  C1.p.v = C2.p.v;
  AC.p.i + C1.p.i + C2.p.i = 0.0;
  AC.n.v = C1.n.v;
  C1.n.v = C2.n.v;
  AC.n.i + C1.n.i + C2.n.i = 0.0;
  i = AC.i;
end Circuit;
```

Figure 6.8: Flat Modelica output

Chapter 7

Conclusions

In this chapter, some of the conclusions and experiences from writing the semantic specification are given. Also, some suggestions for future work are proposed.

7.1 Result

The result of this work is an RML semantic specification for Modelica which is incomplete, but works fairly well for the part of the language it handles.

Another result of working on the semantic specification is that many problems and unclear issues in the Modelica semantics have been resolved. This has not always lead to progress in the formal specification, since it sometimes has meant that it had to be rewritten due to changes in the semantics. But in a larger perspective, namely the progress of the Modelica language specifications, it has implied substantial progress.

7.2 Translator usability

Another goal was to produce a Modelica translator that could be used in real-world applications. This has become something of a secondary goal, but there are some issues worth mentioning.

Some properties of the formal semantic specification makes the output of the translator not so suitable for real use. For instance, solving matrix equations is often best done by keeping them as equations between matrices, but the translator translates them to one equation for each of the matrix elements.

There are some hidden options[1] in the translator which makes it possible to have it output something that is easier to pass on to existing solvers and simulation environments. This is not complete, and relies on some ugly tricks with the RML system. This code is not included in this report.

The performance of the translator has not been fully evaluated, although a simple performance test was done using a 1049–line Modelica model, contain-

[1]The command-line option `+m` makes the translator produce valid Modelica, `+a` prevents it from splitting arrays, and `+t` excludes type information in overloaded operators and implicit type conversions

ing 16 repetitions of a simple electrical circuit model. The test was performed on a Sun Ultra 1 workstation running Solaris 2.6. Running the translation 20 times with the standard output directed to `/dev/null`, it translated the file in 0.26 seconds on the average. This is not a complete test, but at least it hints that the performance is reasonable.

7.3 RML experiences

For practical purposes, writing an RML specification is very much like programming in a functional programming language or a logic programming language, but without the expressive power of those languages. This limitations of course make the programming more cumbersome, but on the upside, the resulting specification will hopefully be much more readable.

The newest version of the RML compiler works well, and even gives useful error messages. However, debugging a specification after it has passed through the compiler is not very fun. No debugger is available, and there is no built-in means of tracing the resolution process.

7.4 Future work

As already stated, the semantic specifications needs more work before it can be considered complete. It is possible that it even needs to be rewritten in large parts, to accommodate for new developments in the Modelica design process.

To make the translator usable in a production environment, more work is needed to allow the translator to be used with existing modeling tools and equation solvers.

Another important work is the development of a test suite that is used to test the translator to see that it does what it is supposed to. There currently exists a test suite that tests most of the features that are implemented, and some which are not yet implemented. It contains over 160 cases, but it needs to be expanded

Appendix A

Annotated formal semantics

This chapter contains the complete formal semantics of Modelica. This includes the RML source files, but the comments in the files have been converted to more readable text.

A.1 Overview

The specification is separated into a number of files, to modularize it and to make it easier to maintain and to make the structure more clear. Each file contains one module. The names of the files are the names of the module it contains, with all letters in lower case, and with the suffix `.rml` appended.

Source modules

A.2 Module **Main**

This is the main program in the Modelica specification.

```
module Main:
        relation main: string list => ()
end

with "parse.rml"
with "dump.rml"
with "explode.rml"
with "dae.rml"
with "inst.rml"
with "rtopts.rml"
```

Relation **translate_file**

This relation invokes the translator on a source file. The argument should be a list with a single file name.

```
relation translate_file : string list => () =

  rule  Parser.parse f => p &
        SCode.elaborate(p) => p' &
        Inst.instantiate(p') => d &
        DAE.dump d
        ----------
        translate_file [f]

  rule  print "# Too many arguments\n"
        ----------------------------
        translate_file _::_::_

  rule  print "# Supply a file name\n"
        ----------------------------
        translate_file []

  rule  print "# An error occured\n"
        ----------------------------
        translate_file _

end
```

Relation **main**

This is the main relation that the RML runtime system calls to start the translation.

```
relation main : string list => () =

  rule  RTOpts.args args => args' &
        translate_file args'
        --------------------
        main args

end
```

A.3 Module Absyn

This file defines the abstract syntax for Modelica. It mainly contains datatypes for constructing the abstract syntax tree (AST), and a few relations for printing the AST.

```
module Absyn:

  type Ident = string
```

A.3.1 Programs

```
  type Program = Class list
```

A program is simply a list of class definitions declared at top level in the source file.

A.3.2 Classes

```
  datatype Class = CLASS of Ident                        (* Name *)
                          * bool                         (* Partial *)
                          * Restriction                  (* Restricion *)
                          * ClassDef                     (* Body *)
```

A class definition consists of a name, a flag to indicate if this class is declared as `partial`, the declared class restriction, and the body of the declaration.

```
  datatype ClassDef = PARTS of ClassPart list
                    | DERIVED of Path
                               * ArrayDim option
                               * ElementArg list
```

The `ClassDef` type contains the definition part of a class declaration. The definition is either explicit, with a list of parts (`public`, `protected`, `equation` and `algorithm`), or it is a definition derived from another class. In the latter case this type contains the name of the derived class and an optional array dimension and a list of modifications.

```
  datatype ClassPart = PUBLIC of Element list
                     | PROTECTED of Element list
                     | EQUATIONS of Equation list
                     | ALGORITHMS of Algorithm list
```

A class definition contains several parts. There are public and protected component declarations, type definitions and `extends` clauses, collectively called elements. There are also equation sections and algorithm sections.

A.3.3 Elements

```
datatype Element = ELEMENT of bool                (* final *)
                          * Ident
                          * ElementSpec

datatype ElementSpec = CLASSDEF of bool           (* replaceable *)
                               * Class
                     | EXTENDS of Path * ElementArg list
                     | COMPONENTS of ElementAttributes
                                 * Path           (* type name *)
                                 * Component list
```

An element is something that occurs in a public or protected section in a class definition. There is one constructor in the `ElementSpec` type for each possible element type. There are class definitions (`CLASSDEF`), `extends` clauses (`EXTENDS`) and component declarations (`COMPONENTS`).

As an example, if the element `extends TwoPin;` appears in the source, it is represented in the AST as `EXTENDS(IDENT("TwoPin"),[])`.

```
datatype Component = COMPONENT of Ident           (* component name *)
                              * ArrayDim
                              * Modification option
```

Several component declarations can be grouped together in one `ElementSpec` by writing them on the same line in the source. This type contains the information specific to one component.

```
datatype Equation = EQ_EXPR of Exp         (* more later? *)
                  | EQ_IF of Exp
                         * Equation list         (* true branch *)
                         * (Exp * Equation list) list (* elseif branches *)
                         * Equation list         (* else branch *)
                  | EQ_EQUALS of Exp * Exp
                  | EQ_CONNECT of ComponentRef * ComponentRef
                  | EQ_FOR of Ident * Exp * Equation list
```

The `Equation` type describes one equation in an equation section.

```
datatype Algorithm = ALG_ASSIGN of ComponentRef * Exp
                   | ALG_IF of Exp
                          * Algorithm list      (* true branch *)
                          * (Exp * Algorithm list) list (* elseif *)
                          * Algorithm list      (* else branch *)
                   | ALG_FOR of Ident * Exp * Algorithm list
                   | ALG_WHILE of Exp * Algorithm list
                   | ALG_WHEN of Exp * Algorithm list
```

The `Algorithm` type describes one algorithm statement in an algorithm section. It does not describe a whole algorithm. The reason this type is named like this is that the name of the grammar rule for algorithm statements is `algorithm`.

A.3.4 Modifications

```
datatype Modification = CLASSMOD of ElementArg list * Exp option

datatype ElementArg = MODIFICATION of bool * ComponentRef * Modification
                    | REDECLARATION of bool * ElementSpec
```

Modifications are described by the `Modification` type. There are two forms of modifications: redeclarations and component modifications.

A.3.5 Component attributes

```
datatype ElementAttributes = ATTR of bool      (* flow *)
                                   * Variability      (* parameter *)
                                   * Direction
datatype Variability = VAR | DISCRETE | PARAM | CONST
datatype Direction = INPUT | OUTPUT | BIDIR
```

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

A.3.6 Array dimensions

```
type ArrayDim = Subscript list
```

Components in Modelica can be scalar or arrays with one or more dimensions. This datatype is used to indicate the dimensionality of a component or a type definition.

A.3.7 Expressions

```
datatype Exp = INTEGER of int
             | REAL of real
             | CREF of ComponentRef
             | STRING of string
             | BOOL of bool
             | BINARY of Exp * Operator * Exp
             | UNARY of Operator * Exp
             | LBINARY of Exp * Operator * Exp
             | LUNARY of Operator * Exp
             | RELATION of Exp * Operator * Exp
             | IFEXP of Exp * Exp * Exp
             | CALL of ComponentRef * Exp list
             | ARRAY of Exp list
             | MATRIX of Exp list list
             | RANGE of Exp * Exp option * Exp
```

The `Exp` datatype is the container of a Modelica expression.

```
datatype Operator = ADD   | SUB     | MUL     | DIV       | POW
                  | UPLUS | UMINUS
                  | AND   | OR
                  | NOT
                  | LESS  | LESSEQ  | GREATER | GREATEREQ | EQUAL | NEQUAL
```

A.3.8 Subscripts

```
datatype Subscript = NOSUB
                   | SUBSCRIPT of Exp
```

The `Subscript` datatype is used both in array declarations and component references. This might seem strange, but it is inherited from the grammar. The `NOSUB` constructor means that the dimension size is undefined when used in a declaration, and when it is used in a component reference it means a slice of the whole dimension.

A.3.9 Component references and paths

```
  datatype ComponentRef = CREF_QUAL of Ident * (Subscript list) * ComponentRef
                        | CREF_IDENT of Ident * (Subscript list)

  datatype Path = QUALIFIED of Ident * Path
                | IDENT of Ident
```

A component reference is the fully or partially qualified name of a component. It is represented as a list of identifier–subscript pairs. The type `Path`, on the other hand, is used to store references to class names, or names inside class definitions.

A.3.10 Restrictions

```
  datatype Restriction = R_CLASS
                       | R_MODEL
                       | R_RECORD
                       | R_BLOCK
                       | R_CONNECTOR
                       | R_TYPE
                       | R_PACKAGE
                       | R_FUNCTION
                       | R_PREDEFINED_INT
                       | R_PREDEFINED_REAL
                       | R_PREDEFINED_STRING
                       | R_PREDEFINED_BOOL
```

These constructors each correspond to a different kind of class declaration in Modelica, except the last four, which are used for the predefined types. The parser assigns each class declaration one of the restrictions, and the actual class definition is checked for conformance during translation. The predefined types are created in the `Builtin` module and are assigned special restrictions.

```
  relation path_string : Path => string
  relation cref_to_path : ComponentRef => Path
  relation element_spec_name : ElementSpec => Ident
  relation print_restr : Restriction => ()

end
```

Relation `path_string`

This relation simply converts a `Path` to a `string`.

```
relation path_string: Path => string =

  axiom path_string(IDENT(s)) => s

  rule  path_string(n) => ns &
        string_append(s,".") => s1 &
        string_append(s1,ns) => ss
        ------------------------------------
        path_string(QUALIFIED(s,n)) => ss

end
```

Relation `cref_to_path`

This relation converts a `ComponentRef` to a `Path`, if possible. If the component reference contains subscripts, it will silently fail.

```
relation cref_to_path : ComponentRef => Path =

  axiom cref_to_path CREF_IDENT(i,[]) => IDENT(i)

  rule  cref_to_path c => p
        -------------------
        cref_to_path CREF_QUAL(i,[],c) => QUALIFIED(i,p)

end
```

Relation `element_spec_name`

The `ElementSpec` type contans the name of the element, and this relation extracts this name.

```
relation element_spec_name : ElementSpec => Ident =

  axiom element_spec_name CLASSDEF(_,CLASS(n,_,_,_)) => n

  axiom element_spec_name COMPONENTS(_,_,[COMPONENT(n,_,_)]) => n

  rule  print "# element_spec_name EXTENDS\n"
        -------------------------------------
        element_spec_name EXTENDS(_,_) => fail

end
```

Relation `print_restr`

This is a utility relation for printing a `Restriction`. The code is excluded for brevity.

A.4 Module `Exp`

This file contains the module `Exp`, which contains data types for describing expressions, after they have been examined by the static analyzer in the module `StaticExp`. There are of course great similarities with the expression types in the `Absyn` module, but there are also several important differences.

No overloading of operators occur, and subscripts have been checked to see if they are slices. All expressions are also type consistent, and all implicit type conversions in the AST are made explicit here.

```
module Exp:

  with "absyn.rml"
```

A.4.1 Identifiers

```
  type Ident = string
```

Define `Ident` as an alias for `string` and use it for all identifiers in Modelica.

A.4.2 Basic types

```
  datatype Type = INT | REAL | BOOL | STRING
```

These types are not used as expression types (see the `Types` module for expression types). They are used to parameterize operators which may work on several simple types.

A.4.3 Expressions

```
  datatype Exp = ICONST of int
               | RCONST of real
               | SCONST of string
               | BCONST of bool
               | CREF of ComponentRef
               | BINARY of Exp * Operator * Exp
               | UNARY of Operator * Exp
               | LBINARY of Exp * Operator * Exp
               | LUNARY of Operator * Exp
               | RELATION of Exp * Operator * Exp
               | IFEXP of Exp * Exp * Exp
               | CALL of Absyn.Path * Exp list          (* Function call *)
               | ARRAY of Exp list
               | MATRIX of Exp list list
               | RANGE of Exp * Exp option * Exp
                 (* New constructors *)
               | CAST of Type * Exp
               | ASUB of Exp * int
```

The `Exp` datatype closely corresponds to the `Absyn.Exp` datatype, but is used for statically analyzed expressions. It includes explicit type promotions and typed (non-overloaded) operators. It also contains expression indexing with the `ASUB` constructor. Indexing arbitrary array expressions is currently not supported in Modelica, but it is needed here.

```
  datatype Operator = ADD         of Type
                    | SUB         of Type
                    | MUL         of Type
                    | DIV         of Type
                    | POW         of Type
                    | UMINUS      of Type
                    | UPLUS       of Type
                    | AND | OR
                    | NOT
                    | LESS        of Type
                    | LESSEQ      of Type
                    | GREATER     of Type
                    | GREATEREQ   of Type
                    | EQUAL       of Type
                    | NEQUAL      of Type
```

Operators which are overloaded in the abstract syntax are here made type-specific. The integer addition operator (`ADD(INT)`) and the real addition operator (`ADD(REAL)`) are two distinct operators.

A.4.4 Component references

```
  datatype ComponentRef = CREF_QUAL of Ident * (Subscript list) * ComponentRef
                        | CREF_IDENT of Ident * (Subscript list)

  datatype Subscript = WHOLEDIM
                     | SLICE of Exp
                     | INDEX of Exp
```

The `Subscript` and `ComponentRef` datatypes are simple translations of the corresponding types in the `Absyn` module.

A.4.5 Exported relations

```
  relation extend_cref : (ComponentRef, Ident, Subscript list) => ComponentRef
  relation int_subscripts : int list => Subscript list
  relation simplify : Exp => Exp
  relation print_exp : Exp => ()
  relation print_component_ref : ComponentRef => ()
  relation print_subscript: Subscript => ()
  relation print_list : ('a list, 'a => (), string) => ()

end

with "rtopts.rml"
```

Relation `extend_cref`

The `extend_cref` relation extends a `ComponentRef` by appending an identifier and a (possibly empty) list of subscripts. Adding the identifier `a` to the component reference `x.y[10]` would produce the component reference `x.y[10].a`, for instance.

```
relation extend_cref : (ComponentRef, Ident, Subscript list) => ComponentRef =

  axiom extend_cref (CREF_IDENT(i1,s1),i,s) => CREF_QUAL(i1,s1,CREF_IDENT(i,s))

  rule  extend_cref (c,i,s) => c'
        -------------------------
        extend_cref (CREF_QUAL(i1,s1,c), i, s) => CREF_QUAL(i1,s1,c')

end
```

A.4.6 Utility relations

These are utility relations used in some of the other relations.

Relation `int_subscripts`

This relation describes the relation between a list of integers and a list of `Exp.Subscript` where each integer is converted to an integer indexing expression.

```
relation int_subscripts : int list => Subscript list =

  axiom int_subscripts [] => []

  rule  int_subscripts xs => xs'
        ------------------------
        int_subscripts x::xs => INDEX(ICONST(x))::xs'

end
```

Relation `simplify`

This relation does some very basic simplification on expressions. It is not intended to be used to simplify expressions provided by the model, but to simplify unnecessarily complex expressions constructed during instantiation.

```
relation simplify : Exp => Exp =

  rule  simplify e => ARRAY(exps) &
        int_sub (i,1) => i' &
        list_nth(exps,i') => exp
        -----------------------
        simplify ASUB(e, i) => exp

  rule  simplify e => MATRIX(exps) &
        int_sub (i,1) => i' &
        list_nth(exps,i') => exp
        -----------------------
        simplify ASUB(e, i) => ARRAY(exp)

  rule  simplify e => IFEXP(c,t,f) &
        simplify ASUB(t,i) => t' &
        simplify ASUB(f,i) => f'
        ------------------------
        simplify ASUB(e, i) => IFEXP(c,t',f')

  rule  simplify e => CREF(CREF_IDENT(n,s)) &
        subscripts_append(s,i) => s'
        ------------------------
        simplify ASUB(e, i) => CREF(CREF_IDENT(n,s'))

  rule  simplify e => CREF(CREF_QUAL(n,s,c)) &
        simplify ASUB(CREF(c), i) => CREF(c')
        -------------------------------------
        simplify ASUB(e, i) => CREF(CREF_QUAL(n,s,c'))

  axiom simplify e => e

end
```

Relation `subscripts_append`

This relation takes a subscript list and adds a new subscript. But there are a few special cases. When the last existing subscript is a slice, it is replaced by the slice indexed by the new subscript.

```
relation subscripts_append : (Subscript list, int) => Subscript list =

  axiom subscripts_append([], i) => [INDEX(ICONST(i))]

  axiom subscripts_append([WHOLEDIM], i) => [INDEX(ICONST(i))]

  rule  simplify ASUB(e, i) => e'
        -------------------------
        subscripts_append([SLICE(e)], i) => [INDEX(e')]

  axiom subscripts_append([s as INDEX(_)], i) => [s, INDEX(ICONST(i))]

  rule  subscripts_append(ss, i) => ss'
        -------------------------------
        subscripts_append(s::ss, i) => s::ss'

end
```

A.4.7 Printing expressions

This module provides some relations to print data to the standard output. This is used for error messages, and for debugging the semantic description.

Relation `print_component_ref`

Print a `ComponentRef`.

Relation `print_subscript`

Print a `Subscript`.

Relation `print_exp`

This relation prints a complete expression.

A.5 Module `ClassInf`

This module deals with class inference, i.e. determining if a class definition adhers to one of the class restrictions, and, if specifically declared in a restrictied form, if it breaks that restriction.

The inference is implemented as a finite state machine. The relation `start` initializes a new machine, and the relation `trans` signals transitions in the machine. Finally, the state can be checked agains a restriction with the `valid` relation.

```
module ClassInf :

  with "absyn.rml"
```

A.5.1 Machine states

```
  datatype State = UNKNOWN       of string
                 | MODEL         of string
                 | RECORD        of string
                 | BLOCK         of string
                 | CONNECTOR     of string
                 | TYPE          of string
                 | PACKAGE       of string
                 | FUNCTION      of string
                 | HAS_EQUATIONS of string
                 | IS_NEW        of string
                 | TYPE_INTEGER  of string
                 | TYPE_REAL     of string
                 | TYPE_STRING   of string
                 | TYPE_BOOL     of string
```

A.5.2 Events

```
  datatype Event = FOUND_EQUATION     (* There are definitions inside *)
                                      (* the current definition *)
                 | NEWDEF             (* This is not a derived class *)
```

The `Event` type contains the different events during instantiation that signals a possible machine transition.

A.5.3 Exported relations

```
  relation start : (Absyn.Restriction, string) => State
  relation trans : (State, Event) => State

  relation valid : (State, Absyn.Restriction) => ()
  (* `assert_valid' is identical to 'valid'.  The only difference is *)
  (* that it prints an error message when it fails. *)
  relation assert_valid : (State, Absyn.Restriction) => ()

  (* Debug relation *)
  relation print_state : State => ()

end
```

A.5.4 Printing

Some relations for printing error and debug information about the state machine.

The code is excluded from the report.

Relation `print_state`

Relation `print_event`

A.5.5 Transitions

Relation `start`

This is the state machine initialization relation.

```
relation start : (Absyn.Restriction, string) => State =

  axiom start (Absyn.R_CLASS, s)                => UNKNOWN(s)
  axiom start (Absyn.R_MODEL, s)                => MODEL(s)
  axiom start (Absyn.R_RECORD, s)             => RECORD(s)
  axiom start (Absyn.R_BLOCK, s)              => BLOCK(s)
  axiom start (Absyn.R_CONNECTOR, s)          => CONNECTOR(s)
  axiom start (Absyn.R_TYPE, s)               => TYPE(s)
  axiom start (Absyn.R_PACKAGE, s)            => PACKAGE(s)
  axiom start (Absyn.R_FUNCTION, s)           => FUNCTION(s)
  axiom start (Absyn.R_PREDEFINED_INT, s)     => TYPE_INTEGER(s)
  axiom start (Absyn.R_PREDEFINED_REAL, s)    => TYPE_REAL(s)
  axiom start (Absyn.R_PREDEFINED_STRING, s)  => TYPE_STRING(s)
  axiom start (Absyn.R_PREDEFINED_BOOL, s)    => TYPE_BOOL(s)

end
```

Relation `trans`

This is the state machine transition relation. It describes the transitions between states at different events.

```
relation trans : (State, Event) => State =
```

Event `NEWDEF`

```
  axiom trans(UNKNOWN(s),        NEWDEF) => IS_NEW(s)
  axiom trans(MODEL(s),          NEWDEF) => MODEL(s)
  axiom trans(RECORD(s),         NEWDEF) => RECORD(s)
  axiom trans(BLOCK(s),          NEWDEF) => BLOCK(s)
  axiom trans(CONNECTOR(s),      NEWDEF) => CONNECTOR(s)

  rule  print "# In class " & print s & print "\n" &
        print "# 'type' can only be derived from predefined types\n"
        -----------------------------------------------
        trans(TYPE(s),           NEWDEF) => fail

  axiom trans(PACKAGE(s),        NEWDEF) => PACKAGE(s)
  axiom trans(FUNCTION(s),       NEWDEF) => FUNCTION(s)
  axiom trans(IS_NEW(s),         NEWDEF) => IS_NEW(s)
  axiom trans(TYPE_INTEGER(s),   NEWDEF) => TYPE_INTEGER(s)
  axiom trans(TYPE_REAL(s),      NEWDEF) => TYPE_REAL(s)
  axiom trans(TYPE_STRING(s),    NEWDEF) => TYPE_STRING(s)
  axiom trans(TYPE_BOOL(s),      NEWDEF) => TYPE_BOOL(s)
```

Event `FOUND_EQUATION`

```
  axiom trans(UNKNOWN(s),        FOUND_EQUATION) => HAS_EQUATIONS(s)
  axiom trans(IS_NEW(s),         FOUND_EQUATION) => HAS_EQUATIONS(s)
  axiom trans(MODEL(s),          FOUND_EQUATION) => MODEL(s)

  rule  print "# In class " & print s & print "\n" &
        print "# Equations not allowed in records\n"
        ----------------------------------------------
        trans(RECORD(s),         FOUND_EQUATION) => fail

  axiom trans(BLOCK(s),          FOUND_EQUATION) => BLOCK(s)

  rule  print "# In class " & print s & print "\n" &
        print "# Equations not allowed in connectors\n"
        ----------------------------------------------
        trans(CONNECTOR(s),      FOUND_EQUATION) => fail

  axiom trans(TYPE(s),           FOUND_EQUATION) => fail (* CORRECT? *)
  axiom trans(PACKAGE(s),        FOUND_EQUATION) => fail
  axiom trans(FUNCTION(s),       FOUND_EQUATION) => fail
  axiom trans(HAS_EQUATIONS(s),  FOUND_EQUATION) => HAS_EQUATIONS(s)

  rule  print "- trans failed: " & print_state st &
        print ", " & print_event ev & print "\n"
        -------------------------------------------------------
        trans(st, ev) => fail

end
```

Relation `valid`

This is the validity relation which determines if a state is valid according to one of the restrictions. This means, that if a class definition is to be used as, say, a connector, the state of the state machine is checked against the `Absyn.R_CONNECTOR` restriction using this relation to find out if it is an error to use this class definition as a connector.

```
relation valid : (State, Absyn.Restriction) => () =

  axiom valid(UNKNOWN(s), _)

  axiom valid(IS_NEW(s),        Absyn.R_CLASS)
  axiom valid(HAS_EQUATIONS(s), Absyn.R_CLASS)

  axiom valid(MODEL(s),         Absyn.R_MODEL)
  axiom valid(IS_NEW(s),        Absyn.R_MODEL)
  axiom valid(HAS_EQUATIONS(s), Absyn.R_MODEL)

  axiom valid(RECORD(s),        Absyn.R_RECORD)
  axiom valid(IS_NEW(s),        Absyn.R_RECORD)

  axiom valid(BLOCK(s),         Absyn.R_BLOCK)
  axiom valid(HAS_EQUATIONS(s), Absyn.R_BLOCK)

  axiom valid(CONNECTOR(_),     Absyn.R_CONNECTOR)
  axiom valid(IS_NEW(_),        Absyn.R_CONNECTOR)
  axiom valid(TYPE_INTEGER(_),  Absyn.R_CONNECTOR)
  axiom valid(TYPE_REAL(_),     Absyn.R_CONNECTOR)
  axiom valid(TYPE_STRING(_),   Absyn.R_CONNECTOR)
  axiom valid(TYPE_BOOL(_),     Absyn.R_CONNECTOR)

  axiom valid(TYPE(s),          Absyn.R_TYPE)
  axiom valid(TYPE_INTEGER(s),  Absyn.R_TYPE)
  axiom valid(TYPE_REAL(s),     Absyn.R_TYPE)
  axiom valid(TYPE_STRING(s),   Absyn.R_TYPE)
  axiom valid(TYPE_BOOL(s),     Absyn.R_TYPE)

  axiom valid(IS_NEW(s),        Absyn.R_PACKAGE)
  axiom valid(PACKAGE(s),       Absyn.R_PACKAGE)

  axiom valid(IS_NEW(s),        Absyn.R_FUNCTION)
  axiom valid(FUNCTION(s),      Absyn.R_FUNCTION)

end
```

Relation `assert_valid`

This relation has the same semantical meaning as the relation `valid`. However, it prints an error message when it fails.

```
relation assert_valid : (State, Absyn.Restriction) => () =

  rule  valid(st,re)
        ------------
        assert_valid(st,re)

  rule  print "# Restriction violation: " & print_state st &
        print " is not a " & Absyn.print_restr re & print "\n"
        -------------------------------------------
        assert_valid(st,re) => fail

end
```

A.6 Module `Types`

This file defines the type system. It contains an RML type called `Type` which defines types. It also contains relations for determining subtyping etc.

There are a few known problems with this module. It currently depends on `SCode.Attributes`, which in turn depends on `Absyn.ArrayDim`. However, the only things used from those modules are constants that could be moved to their own modules.

```
module Types :

  with "classinf.rml"
  with "absyn.rml"
  with "exp.rml"
  with "values.rml"
  with "explode.rml"
```

A.6.1 Identifiers

```
  type Ident = string
```

A.6.2 Variables

```
  datatype Var = VAR of Ident                    (* name *)
                    * Attributes        (* attributes *)
                    * bool                      (* protected *)
                    * Type                      (* type *)
                    * Binding                   (* equation modification *)

  datatype Attributes = ATTR of bool    (* flow *)
                            * SCode.Accessibility
                            * Absyn.Variability       (* parameter *)
                            * Absyn.Direction

  datatype Binding = UNBOUND
                   | EQBOUND of Exp.Exp * bool
                   | VALBOUND of Values.Value
```

A.6.3 Types

```
  datatype Type = T_INTEGER
                | T_REAL
                | T_STRING
                | T_BOOL
                | T_ARRAY of ArrayDim * Type
                | T_COMPLEX of ClassInf.State
                           * Var list
                | T_FUNCTION of FuncArg list * Type (* Only single-result *)
                | T_NOTYPE

  datatype ArrayDim = DIM of int option

  type FuncArg = Ident * Type
```

To generate the correct set of equations, the translator has to differentiate between the primitive types `Real`, `Integer`, `String`, `Boolean` and types directly derived from then from other, complex types. For arrays and matrices the type T_ARRAY is used, with the first argument being the number of dimensions, and the second being the type of the objects in the array. The Type type is used to store information about whether a class is derived from a primitive type, and whether a variable is of one of these types.

A.6.4 Exported relations

```
  relation basic_type : Type => bool
  relation equivtypes : (Type, Type) => bool
  relation subtype : (Type, Type) => bool
  relation lookup_component : (Type, Ident) => Var
  relation lift_array : (Type,int option) => Type
  relation unlift_array : Type => Type
  relation print_type : Type => ()

end
```

Relation **basic_type**

Test whether a type is one of the builtin types.

```
relation basic_type : Type => bool =

  axiom basic_type T_INTEGER => true
  axiom basic_type T_REAL => true
  axiom basic_type T_STRING => true
  axiom basic_type T_BOOL => true
  axiom basic_type T_ARRAY(_,_) => false
  axiom basic_type T_COMPLEX(_,_) => false

end
```

Relation **equivtypes**

This is the type equivalence relation. It is defined in terms of the subtype relation. Two types are considered equivalent if they are subtypes of each other.

```
relation equivtypes : (Type, Type) => bool =

  rule  subtype(t1, t2) => true &
        subtype(t2, t1) => true
        -----------------------
        equivtypes(t1, t2) => true

        (* default *)
  axiom equivtypes(t1, t2) => false

end
```

Relation **subtype**

Is the first type a subtype of the second type? This relation specifies the rules for subtyping in Modelica.

```
relation subtype : (Type, Type) => bool =

  axiom subtype (T_INTEGER,T_INTEGER) => true
  axiom subtype (T_REAL,T_REAL)       => true
  axiom subtype (T_STRING,T_STRING)   => true
  axiom subtype (T_BOOL,T_BOOL)       => true

  rule  subtype (t1,t2) => true
        -----------------------
        subtype(T_ARRAY(_,t1),T_ARRAY(DIM(NONE),t2)) => true

  rule  i1 = i2 &
        subtype (t1,t2) => true
        ---------------------------------------------------------
        subtype(T_ARRAY(DIM(SOME(i1)),t1),T_ARRAY(DIM(SOME(i2)),t2)) => true

  rule  subtype_varlist(els1, els2) => true
        -----------------------------------
        subtype(T_COMPLEX(st1,els1), T_COMPLEX(st2,els2)) => true

        (* default *)
  axiom subtype(_,_) => false

end
```

Relation **subtype_varlist**

This relation checks if the `Var` list in the first list is a subset of the list in the second argument. More precisely, it checks if, for each `Var` in the second list there is a `Var` in the first list with a type that is a subtype of the `Var` in the second list.

```
relation subtype_varlist : (Var list, Var list) => bool =

  axiom subtype_varlist(_, []) => true

  rule  varlist_lookup(l,n) => VAR(_,_,_,t1,_) &
        subtype(t1,t2) => true &
        subtype_varlist(l,vs) => true
        -----------------------------
        subtype_varlist(l, VAR(n,_,_,t2,_)::vs) => true

        (* default *)
  axiom subtype_varlist(_,_) => false

end
```

Relation **varlist_lookup**

Given a list of `Var` and a name, this relation finds any `Var` with the given name.

```
relation varlist_lookup : (Var list, Ident) => Var =

  rule  n = name
        --------
        varlist_lookup((v as VAR(n,_,_,_,_))::_, name) => v

  rule  varlist_lookup(vs, name) => v
        -----------------------------
        varlist_lookup(v::vs, name) => v

end
```

Relation `lookup_component`

This relation finds a subcomponent by name.

```
relation lookup_component : (Type, Ident) => Var =

  rule  basic_type(t) => true &
        lookup_in_builtin(t,n) => v
        ------------------------
        lookup_component(t,n) => v

  rule  lookup_component2(cs, id) => v
        ------------------------------
        lookup_component(T_COMPLEX(st,cs), id) => v

  rule  print "# Can't lookup components in arrays\n"
        ------------------------------
        lookup_component(T_ARRAY(dim,t), id) => fail

  rule  print "- Looking up " &
        print id &
        print " in noncomplex type\n"
        ----------------------------------------
        lookup_component(_, id) => fail

end
```

Relation `lookup_in_builtin`

Since builtin types are not represented as T_COMPLEX, special care is needed to be able to lookup the attributes (`start` etc) in them.

This is not a complete solution. The current way of mapping the both the Modelica type `Real` and the simple type `RealType` to `Types.T_REAL` is a bit problematic, since it doesn't make a difference between `Real` and `RealType`, which makes the translator accept things like `x.start.start.start`.

```
relation lookup_in_builtin : (Type,Ident) => Var =

  axiom lookup_in_builtin(T_REAL,"start")
          => VAR("start",
                 ATTR(false, SCode.RW, Absyn.PARAM, Absyn.BIDIR),
                 false, T_REAL, UNBOUND)

  axiom lookup_in_builtin(t,"start")
          => VAR("start",
                 ATTR(false, SCode.RW, Absyn.PARAM, Absyn.BIDIR),
                 false, t, UNBOUND)

  axiom lookup_in_builtin(T_INTEGER,"unit")
          => VAR("unit",
                 ATTR(false, SCode.RW, Absyn.PARAM, Absyn.BIDIR),
                 false, T_STRING, UNBOUND)

  axiom lookup_in_builtin(T_REAL,"unit")
          => VAR("unit",
                 ATTR(false, SCode.RW, Absyn.PARAM, Absyn.BIDIR),
                 false, T_STRING, UNBOUND)

        (* The rest of them should also be added *)

end
```

Relation `lookup_component2`

This relation finds a named `Var` in a list of `Vars`, comparing the name against the second argument to this relation.

```
relation lookup_component2 : (Var list , Ident) => Var =

  rule  n = m
        -----
        lookup_component2((v as VAR(n,_,_,_,_))::_, m) => v

  rule  lookup_component2(vs, n) => v
        -----------------------------
        lookup_component2(v::vs, n) => v

end
```

Relation `lift_array`

This relation turns a type into an array of that type. If the type already is an array, aonther dimension is simply added.

```
relation lift_array : (Type,int option) => Type =

  axiom lift_array (ty,i) => T_ARRAY(DIM(i),ty)

end
```

Relation `unlift_array`

This relation turns an array of a type into that type.

```
relation unlift_array : Type => Type =

  axiom unlift_array T_ARRAY(DIM(_),ty) => ty

end
```

Relation `print_type`

This relation prints a textual description of a Modelica type. If the type is not one of the primitive types, it simply prints `composite`. The actual code is expluded from the report.

A.7 Module `SCode`

This module contains data structures to describe a Modelica model in a more convenient way than the `Absyn` module does. The most important relation in this module is the `elaborate` relation which turns an abstract syntax tree into an SCode representation.

```
module SCode :

  with "absyn.rml"
```

Some definitions are borrowed from `Absyn`

```
  type Ident = Absyn.Ident
  type Path = Absyn.Path
  type Subscript = Absyn.Subscript
  type Restriction = Absyn.Restriction
```

A.7.1 Modifications

```
  datatype Mod = MOD of bool * (SubMod list) * (Absyn.Exp option)
               | REDECL of bool * Absyn.ElementSpec
               | NOMOD
  datatype SubMod = NAMEMOD of Ident * Mod         (* A named component *)
                  | IDXMOD of Subscript list * Mod (* An array element *)
```

Modifications are represented in an more structured way than in the `Absyn` module. Modifications using qualified names (such as in `x.y = z`) are normalized (to `x(y = z)`). And a special case when arrays are subscripted in a modification.

A.7.2 Programs

```
  type Program = Class list
```

As in the AST, a program is simply a list of class definitions.

A.7.3 Classes

```
  datatype Class = CLASS of Ident               (* Name *)
                         * bool                 (* Partial *)
                         * Restriction          (* Restricion *)
                         * ClassDef             (* Parts *)

  datatype ClassDef = PARTS of Element list
                            * Equation list
                            * Algorithm list
                    | DERIVED of Path
                              * Absyn.ArrayDim option
                              * Mod
```

The major difference between these types and their `Absyn` counterparts is that the `PARTS` constructor contains separate lists for elements, equations and algorithms.

```
(* - Equations *)
datatype Equation = EQ_EXPR of Absyn.Exp      (* more later? *)
                  | EQ_IF of Absyn.Exp        (* conditional *)
                          * Equation list     (* true branch *)
                          * Equation list     (* false branch *)
                  | EQ_EQUALS of Absyn.Exp * Absyn.Exp
                  | EQ_CONNECT of Absyn.ComponentRef * Absyn.ComponentRef
                  | EQ_FOR of Ident * Absyn.Exp * Equation list
```

These are almost identical to the Absyn versions. In EQ_IF, the elseif branches are represented as normal else branches with a single if statement in them.

A.7.4 Algorithms

```
datatype Algorithm = ALGORITHM of Absyn.Algorithm list
```

The Absyn module uses the terminology from the grammar, where algorithm means an algorithmic statement. But here, Algorithm means a whole algorithm section.

A.7.5 Elements

```
datatype Element = EXTENDS of Path * Mod
                 | CLASSDEF of Ident          (* name *)
                            * bool            (* final *)
                            * bool            (* replaceable *)
                            * Class
                 | COMPONENT of Ident         (* component name *)
                             * bool           (* final *)
                             * bool           (* protected *)
                             * Attributes
                             * Path           (* type name *)
                             * Mod
```

There are three types of elements in a declaration, represented by the constructors EXTENDS (for extends clauses), CLASSDEF (for local class definitions) and COMPONENT (for local variables).

A.7.6 Attributes

```
datatype Attributes = ATTR of Absyn.ArrayDim
                           * bool    (* flow *)
                           * Accessibility
                           * Absyn.Variability       (* parameter *)
                           * Absyn.Direction

datatype Accessibility = RW   (* read/write *)
                       | RO   (* read-only *)
                       | WO   (* write-only (not used) *)
```

These are attributes that apply to a declared component.

```
  relation elaborate : Absyn.Program => Program
  relation build_mod : (Absyn.Modification option, bool) => Mod
  relation print_mod : Mod => ()

end

with "dump.rml"
```

Relation **elaborate**

This relation takes an `Absyn.Program` and constructs a `Program` from it.

```
relation elaborate: Absyn.Program => Program =

  axiom elaborate([]) => []

  rule  elab_class(c) => c' &
        elaborate(cs) => cs'
        -----------------------------
        elaborate(c::cs) => (c'::cs')

end
```

Relation **elab_class**

This relations converts an `Absyn.Class` to a `Class`.

```
relation elab_class: Absyn.Class => Class =

  rule  elab_classdef d => d'
        -------------------------------------------
        elab_class(Absyn.CLASS(n,p,r,d)) => CLASS(n,p,r,d')

end
```

Relation **elab_classdef**

This relation converts an `Absyn.ClassDef` to a `ClassDef`. For the `DERIVED` case, the conversion is fairly trivial, but for the `PARTS` case more work is needed. The result contains separate lists for elements, equations and algorithms, which are mixed in the input.

```
relation elab_classdef: Absyn.ClassDef => ClassDef =

  rule  build_mod(SOME(Absyn.CLASSMOD(a, NONE)), false) => mod
        ------------------------------------------------
        elab_classdef(Absyn.DERIVED(p,d,a)) => DERIVED(p,d,mod)

  axiom elab_classdef(Absyn.PARTS([])) => PARTS([],[],[])

        (* Lots of unnecessary consing ahead... *)
  rule  elab_classdef(Absyn.PARTS(ps)) => PARTS(els,eqs,als) &
        elab_equations(l) => l' &
        list_append(eqs,l') => eqs'
        ------------------------------
        elab_classdef(Absyn.PARTS(Absyn.EQUATIONS(l)::ps))
          => PARTS(els,eqs',als)

  rule  elab_classdef(Absyn.PARTS(ps)) => PARTS(els,eqs,als) &
        let als' = ALGORITHM(l)::als
        ----------------------------
        elab_classdef(Absyn.PARTS(Absyn.ALGORITHMS(l)::ps))
          => PARTS(els,eqs,als')

  rule  elab_classdef(Absyn.PARTS(ps)) => PARTS(els,eqs,als) &
        elab_elist(es,false) => es' &
        list_append(els,es') => els'
        ------------------------------
        elab_classdef(Absyn.PARTS(Absyn.PUBLIC(es)::ps))
          => PARTS(els',eqs,als)

  rule  elab_classdef(Absyn.PARTS(ps)) => PARTS(els,eqs,als) &
        elab_elist(es,true) => es' &
        list_append(els,es') => els'
        ------------------------------
        elab_classdef(Absyn.PARTS(Absyn.PROTECTED(es)::ps))
          => PARTS(els',eqs,als)

end
```

Relation `elab_elist`

This relation converts a list of `Absyn.Element` to a list of `Element`. The boolean argument flags whether the elements are pretected.

```
relation elab_elist : (Absyn.Element list, bool) => Element list =

  axiom elab_elist([],_) => []

  rule  elab_element(e, prot) => e' &
        elab_elist(es, prot) => es' &
        list_append(e',es') => l
        ---------------------
        elab_elist(e::es, prot) => l

end
```

Relation `elab_element`

This relation converts an `Absyn.Element` to a list of `Elements`. The original element may declare several components at once, and those are separated to several declarations in the result.

```
relation elab_element : (Absyn.Element, bool) => Element list =

  rule  elab_elementspec(f,prot,s) => es
        --------------------------------
        elab_element(Absyn.ELEMENT(f,_,s), prot) => es

end
```

Relation `elab_elementspec`

This relation turns an `Absyn.ElementSpec` to a list of `Elements`. The boolean arguments say if the element is final and protected, respectively.

```
relation elab_elementspec: (bool,bool,Absyn.ElementSpec) => Element list =

  rule  elab_classdef(de) => de'
        -----------------------
        elab_elementspec(final,prot,Absyn.CLASSDEF(rp,Absyn.CLASS(n,pa,re,de)))
          => [CLASSDEF(n,final,rp,CLASS(n,pa,re,de'))]

  rule  build_mod(SOME(Absyn.CLASSMOD(args, NONE)), false) => mod &
        Absyn.path_string(n) => ns
        --------------------------
        elab_elementspec(final,prot,Absyn.EXTENDS(n,args))
          => [EXTENDS(n,mod)]

  axiom elab_elementspec(_,_,Absyn.COMPONENTS(_,_,[])) => []

  rule  elab_elementspec(final,prot,Absyn.COMPONENTS(attr,t,xs)) => xs' &
        build_mod(m,false) => mod
        --------------------------------------------------
        elab_elementspec(final,prot,
                         Absyn.COMPONENTS(attr as Absyn.ATTR(fl,pa,di),t,
                                          (Absyn.COMPONENT(n,d,m))::xs))
          => COMPONENT(n,final,prot,ATTR(d,fl,RW,pa,di),t,mod)::xs'

end
```

Relation `elab_equations`

This relation transforms a list of `Absyn.Equations` to a list of `Equationss`, by applying the `elab_equation` relation to each equation.

```
relation elab_equations : Absyn.Equation list => Equation list =

  axiom elab_equations [] => []

  rule  elab_equation e => e' &
        elab_equations es => es'
        ------------------------
        elab_equations e::es => e'::es'

end
```

Relation `elab_equation`

The translation of equations are straightforward, with one exception. `If` clauses are translated so that the SCode only contains simple `if-else` constructs, and no `elseif`.

```
relation elab_equation : Absyn.Equation => Equation =

  axiom elab_equation Absyn.EQ_EXPR(e) => EQ_EXPR(e)

  rule  elab_equations tb => tb' &
        elab_equations fb => fb'
        ------------------------
        elab_equation Absyn.EQ_IF(e,tb,[],fb) => EQ_IF(e,tb',fb')

  rule  elab_equation Absyn.EQ_IF(e,tb,[],[Absyn.EQ_IF(ee,ei,eis,fb)]) => eq
        ------------------------------------------------------------------
        elab_equation Absyn.EQ_IF(e,tb,(ee,ei)::eis,fb) => eq

  axiom elab_equation Absyn.EQ_EQUALS(e1,e2) => EQ_EQUALS(e1,e2)

  axiom elab_equation Absyn.EQ_CONNECT(c1,c2) => EQ_CONNECT(c1,c2)

  rule  elab_equations l => l'
        ----------------------
        elab_equation Absyn.EQ_FOR(i,e,l) => EQ_FOR(i,e,l')

end
```

A.7.7 Modification management

Relation `build_mod`

Builds an `SCode.Mod` from an `Absyn.Modification`. The boolean argument flags whether the modification is `final`.

```
relation build_mod : (Absyn.Modification option, bool) => Mod =

  axiom build_mod(NONE, _) => NOMOD

  axiom build_mod(SOME(Absyn.CLASSMOD([], e as SOME(_))), final)
          => MOD(final,[],e)

  rule  build_args(l) => subs
        ---------------------------------
        build_mod(SOME(Absyn.CLASSMOD(l,e)),final) => MOD(final,subs,e)

end

relation build_args : (Absyn.ElementArg list) => SubMod list =

  axiom build_args [] => []

  rule  build_args xs => subs &
        build_mod(SOME(mod),final) => mod' &
        build_sub(cref,mod') => sub
        ----------------------
        build_args((Absyn.MODIFICATION(final,cref,mod)::xs)) => (sub::subs)

  rule  build_args xs => subs &
        Absyn.element_spec_name spec => n
        --------------------------
        build_args((Absyn.REDECLARATION(final,spec)::xs))
          => (NAMEMOD(n,REDECL(final,spec))::subs)

end
```

Relation `build_sub`

This relation converts a `ComponentRef` into a number of nested `SUBMOD`s.

```
relation build_sub : (Absyn.ComponentRef,Mod) => SubMod =

        (* First some rules to prevent bad modifications *)

  rule  print "# Illegal modification of " & Dump.print_component_ref c &
        Dump.print_component_ref c & print "\n"
        ---------------------------------------
        build_sub(c as Absyn.CREF_IDENT(_,_::_),MOD(_,_::_,_)) => fail

  rule  print "# Illegal modification of " &
        Dump.print_component_ref c & print "\n"
        ---------------------------------------
        build_sub(c as Absyn.CREF_QUAL(_,_::_,_),MOD(_,_::_,_)) => fail

        (* Then the normal rules *)

  rule  build_sub_sub (ss,mod) => mod'
        ------------------------------
        build_sub(Absyn.CREF_IDENT(i,ss),mod) => NAMEMOD(i,mod')

  rule  build_sub(path,mod) => sub &
        let mod = MOD(false,[sub],NONE) &
        build_sub_sub(ss,mod) => mod'
        --------------------------
        build_sub(Absyn.CREF_QUAL(i,ss,path),mod) => NAMEMOD(i,mod')

end
```

Relation `build_sub_sub`

This relation is used to handle the case when a array component is indexed in the modification, so that only one or a limitied number of array elements should be modified.

```
relation build_sub_sub : (Subscript list, Mod) => Mod =

  axiom build_sub_sub ([], m) => m
  axiom build_sub_sub (l, m) => MOD(false,[IDXMOD(l,m)],NONE)

end
```

Relation `print_mod`

This relation prints a modification. The code is excluded from the report for brevity.

A.8 Module `Inst`

This module exports only one relation.

```
module Inst:

  with "explode.rml"
  with "dae.rml"

  relation instantiate: SCode.Program => DAE.DAElist

end
```

Other modules used by this module

```
with "exp.rml"
with "algorithm.rml"
with "values.rml"
with "absyn.rml"
with "prefix.rml"
with "env.rml"
with "lookup.rml"
with "builtin.rml"
with "dump.rml"
with "types.rml"
with "connect.rml"
with "classinf.rml"
with "staticexp.rml"
with "mod.rml"
with "rtopts.rml"
```

These type aliases are introduced to make the code a little more readable.

```
type Prefix = Prefix.Prefix
type Mod = Mod.Mod
type Ident = Exp.Ident
type Env = Env.Env
```

Relation `new_ident`

This relation creates a new, unique identifer. The same name is never returned twice.

```
relation new_ident : () => Exp.ComponentRef =

  rule  tick => i & int_string i => is & string_append("__TMP__",is) => s
        ----------------------------------------------------------------
        new_ident => Exp.CREF_IDENT(s,[])

end
```

Relation `select`

This utility relation selects one of two objects depending on a boolean variable.

```
relation select : (bool, 'a, 'a) => 'a =

  axiom select (true,x,_) => x
  axiom select (false,_,x) => x

end
```

Relation **`instantiate`**

To instantiate a Modelica program, an initial environment is built, containing the predefined types. Then the program is instantiated by the relation `inst_program`

```
relation instantiate: SCode.Program => DAE.DAElist =

  rule  Builtin.initial_env => env &
        inst_program(env,p) => l
        -----------------------------
        instantiate(p) => DAE.DAE(l)

end
```

Relation **`inst_program`**

Instantiating a Modelica program is the same as instantiating the last class definition in the source file. First all the class definitions is added to the environment without modifications, and then the last class is instantiated in the relation `inst_class`

```
relation inst_program : (Env, SCode.Program) => DAE.Element list =

  rule  print "# Empty program\n"
        ------------------------------
        inst_program(env,[]) => fail

  rule  (* Env.extend_frame_c(env,c) => env' & *)
        inst_class(env,Mod.NOMOD,Prefix.NOPRE,[],c) => (dae,csets,_,_)
        -----------------------------
        inst_program(env,[c as SCode.CLASS(n,_,_,_)])
          => [DAE.COMP(n,DAE.DAE(dae))]

  rule  inst_class_decl(env,Mod.NOMOD,Prefix.NOPRE,[],c) => (env',dae1) &
        (* Env.extend_frame_c(env,c) => env' & *)
        inst_program(env',cs) => dae2 &
        list_append(dae1,dae2) => dae
        ----------------------
        inst_program(env,c::(cs as (_::_))) => dae

end
```

Relation **`inst_class`**

Instantiating a class consists of the following steps:

- Create a new frame on the environment
- Initialize the class inference state machine
- Instantiate all the elements and equations
- Generate equations from the connection sets built during instantiation

```
relation inst_class: (Env, Mod, Prefix, Connect.Sets, SCode.Class)
          => (DAE.Element list, Connect.Sets, Types.Type, ClassInf.State) =

  rule  print "# Can't instantiate partial class " & print n & print "\n"
        ------------------------------------------------------
        inst_class(env,mod,pre, csets, SCode.CLASS(n,true,_,_)) => fail

  rule  Env.open_scope(env) => env' &
        ClassInf.start(r,n) => ci_state &
        inst_class_in(env', mod, pre, csets, ci_state, c, false)
          => (dae1,_,csets',ci_state', tys) &
        Connect.equations csets' => dae2 &
        list_append(dae1, dae2) => dae &
        mktype(ci_state',tys) => ty
        -------------------------------------------
        inst_class(env, mod, pre, csets, c as SCode.CLASS(n,_,r,_))
          => (dae, [], ty,ci_state')

  rule  print "- inst_class " & print n & print " failed\n"
        -----------------------------------------------------
        inst_class(_,_,_,_,SCode.CLASS(n,_,_,_)) => fail

end
```

Relation `inst_class_in`

This rule instantiates the contents of a class definition, with a new environment already setup.

```
relation inst_class_in: (Env, Mod, Prefix, Connect.Sets, ClassInf.State,
                         SCode.Class, bool)
          => (DAE.Element list, Env, Connect.Sets, ClassInf.State,
              Types.Var list) =

  rule  inst_classdef(env,mods,pre,csets,ci_state,d,r,prot)
          => (l,env', csets', ci_state', tys)
        ------------------------------------
        inst_class_in(env,mods,pre,csets,ci_state,
                      c as SCode.CLASS(n,_,r,d), prot)
          => (l,env', csets', ci_state', tys)

  rule  print "- inst_class_in failed\n"
        ---------------------------------
        inst_class_in(_,_,_,csets,_,_,_) => fail

end
```

Relation `inst_classdef`

There are two kinds of class definitions, either explicit definitions (`SCode.PARTS()`) or derived definitions (`SCode.DERIVED()`).

When instantiating an explicit definition, the elements are first instantiated, using `inst_element_list`, and then the equations and finally the algorithms are instantiated using `inst_equation` and `inst_algorithm`, respectively. The resulting lists of equations are concatenated to produce the result.

```
relation inst_classdef: (Env, Mod, Prefix, Connect.Sets, ClassInf.State,
                         SCode.ClassDef, Absyn.Restriction, bool)
          => (DAE.Element list, Env, Connect.Sets, ClassInf.State,
              Types.Var list) =
```

This rule describes how to instantiate an explicit class definition

```
  rule  ClassInf.trans(ci_state, ClassInf.NEWDEF) => ci_state' &
        inst_element_list(env,mods,pre,csets,ci_state', els)
          => (dae1,env', csets', ci_state'', tys) &
        inst_list(env',mods,pre,csets',ci_state'',inst_equation, eqs)
          => (dae2,env'',csets'', ci_state''') &
        inst_list(env'',mods,pre,csets'',ci_state''', inst_algorithm, als)
          => (dae3,env''',csets''', ci_state'''') &
        list_append(dae1,dae2) => dae5 &
        list_append(dae5,dae3) => dae
        -----------------------------
        inst_classdef(env,mods,pre, csets, ci_state,
                      SCode.PARTS(els,eqs,als), re, prot)
          => (dae,env''', csets''', ci_state'''', tys)
```

This rule describes how to instantiate a derived class definition

```
  rule  Lookup.lookup_class(env,cn) => (c as SCode.CLASS(cn2,_,r,_), cenv) &
        Mod.lookup_modification_p(mods,cn) => m &
        Mod.elab_mod(env,pre,mod) => mod' &
        ClassInf.start(r, cn2) => new_ci_state &
        Mod.merge(mods,m) => mods' &
        Mod.merge(mods',mod') => mods'' &
        inst_class_in(cenv, mods'', pre, csets, new_ci_state, c, prot)
          => (dae,env, csets', ci_state', tys) &

        (* Check for restriction violations *)
        ClassInf.assert_valid(ci_state', re)
        ----------------------------------------
        inst_classdef(env,mods,pre, csets, ci_state,
                      SCode.DERIVED(cn,ad,mod), re, prot)
          => (dae,env, csets', ci_state', tys)
```

If the class is derived from a class that can not be found in the environment, this rule prints an error message.

```
  rule  not Lookup.lookup_class(env,cn) => (_,_) &
        Absyn.path_string(cn) => cns &
        print "# unknown class: " & print cns & print "\n"
        ----------------------------------------
        inst_classdef(env,mods,pre,csets,ci_state,
                      SCode.DERIVED(cn,ad,mod), re, prot)
          => fail

  rule  print "- inst_classdef failed\n"
        ----------------------------------
        inst_classdef(_,_,_,_,_,_,_,_) => fail

end
```

Relation `inst_element_list`

Instantiate elements one at a time, and concatenate the resulting lists of equations.

```
relation inst_element_list : (Env, Mod, Prefix, Connect.Sets, ClassInf.State,
                              SCode.Element list)
          => (DAE.Element list, Env, Connect.Sets, ClassInf.State,
              Types.Var list) =

  axiom inst_element_list(env,_,_,csets,ci_state,[])
          => ([], env, csets, ci_state, [])

  rule  inst_element(env,mod,pre,csets,ci_state,el)
          => (dae1,env',csets',ci_state',tys1) &
        inst_element_list(env',mod,pre,csets',ci_state',els)
          => (dae2,env'',csets'',ci_state'',tys2) &
        list_append(tys1, tys2) => tys &
        list_append(dae1, dae2) => dae
        ------------------------------
        inst_element_list(env,mod,pre,csets,ci_state,el::els)
          => (dae, env'', csets'', ci_state'', tys)

  rule  print "- inst_element_list failed\n"
        ------------------------------------
        inst_element_list(_,_,_,_,_,_) => fail

end
```

Relation `inst_element`

This monster relation instantiates an element of a class definition. An element is either a class definition, a variable, or an `extends` clause.

```
relation inst_element : (Env, Mod, Prefix, Connect.Sets, ClassInf.State,
                         SCode.Element)
          => (DAE.Element list, Env, Connect.Sets, ClassInf.State,
              Types.Var list) =
```

extends

Handle "extends" elements by instantiating the class definition of the extended class.

```
  rule  Absyn.path_string(cn) => cns &

        Lookup.lookup_class(env,cn) => (c as SCode.CLASS(cn2,_,restr,def),
                                        cenv) &
        Mod.lookup_modification_p(mods,cn) => classmod &
        Mod.elab_mod(env,pre,m) => m' &

        (* Build the combind set of modifications *)
        (* classmod is the modifications stored with the class *)
        (* mods     is the modifications passed to the relation *)
        (* m        is the modification stored in the element *)
        Mod.merge(classmod,mods) => mods' &
        Mod.merge(mods',m') => mods'' &

        (* Can't use inst_class, as that creates a new frame *)
        inst_class_in(cenv,mods'',pre,csets,ci_state,c,false)
          => (dae,env',csets',ci_state',vars)
        ----------------------------------
        inst_element(env,mods,pre,csets, ci_state, SCode.EXTENDS(cn,m))
          => (dae,env',csets',ci_state',vars)
```

Rules to catch redeclarations and name collsions

```
     (* If a variable is declared multiple times, the first is used *)
rule  Lookup.lookup_ident_local(env,n) => _
      -------------------------------------
      inst_element(env,mods,pre,csets,ci_state,
                   SCode.COMPONENT(n,final,prot,_,_,_))
        => ([],env,csets,ci_state,[])

      (* Illegal redeclarations *)
rule  Lookup.lookup_ident_local(env,n) => _ &
      print "# Trying to redeclare the class " & print n &
      print " as a variable\n"
      ------------------------
      inst_element(env,mods,pre,csets,ci_state,
                   SCode.CLASSDEF(n,_,_,_))
        => fail

rule  Lookup.lookup_class(env,Absyn.IDENT(n)) => (_,_) &
      print "# Trying to redeclare the variable " & print n &
      print " as a class\n"
      ---------------------
      inst_element(env,mods,pre,csets,ci_state,
                   SCode.COMPONENT(n,_,_,_,_,_))
        => fail
```

A new class definition

Put it in the current frame in the environment

```
rule  Mod.lookup_modification_p(mods,Absyn.IDENT(n)) => classmod &
      inst_class_decl(env,classmod,pre,csets,c) => (env',dae)
      ------------------------------------------------------
      inst_element(env,mods,pre,csets,ci_state, SCode.CLASSDEF(n,_,_,c))
        => (dae,env',csets,ci_state,[])
```

A component

This is the rule for instantiating a model component. A component can be a structured subcomponent or a variable, parameter or constant. All of these are treated in a similar way.

Lookup the class name, apply modifications and add the variable to the current frame in the environment. Then instantiate the class with an extended prefix.

```
rule  Prefix.prefix_cref(pre,Exp.CREF_IDENT(n,[])) => vn &
```

The class definition is fetched from the environment. Then the set of modifications is calculated. The modificions is the result of merging the modifications from several sources. The modification stored with the class definition is put in the variable `classmod`, the modification passed to the relation is extracted and put in the variable `mm`, and the modification that is included in the variable declaration is in the variable `m`. All of these are merged so that the correct precedence rules are followed.

```
      Lookup.lookup_class(env,t) => (cl,cenv) &
      Mod.lookup_modification_p(mods,t) => classmod &
      Mod.lookup_comp_modification(mods,n) => mm &
      Mod.elab_mod(env,pre,m) => m' &

      Mod.merge(classmod,mm) => mod &
      Mod.merge(mod,m') => mod' &
```

If the element is `protected`, and an external modification is applied, it is an error.

```
        check_prot(prot, mm, vn) &

        Mod.mod_equation mod' => eq &
```

The variable declaration and the (optional) equation modification are inspected for array dimensions.

```
        elab_arraydim(env,ad,eq) => dims &
```

Instantiate the component

```
        inst_var(cenv,mod',pre,csets,n,cl,attr,dims,[])
          => (dae1,csets',ty) &
```

The environment is extended with the new variable binding.

```
        make_binding (env,attr,eq,cl) => binding &
        Env.extend_frame_v(env,Types.VAR(n,Types.ATTR(flow,acc,param,dir),
                                         prot,ty,binding)) => env' &
```

If the type is one of the simple, predifined types a simple variable declaration is added to the DAE.

```
        (* dae_declare (vn,cl,attr) => dae3 & *)

        let dae = dae1
        (*list_append(dae1, dae2) => dae4 &
        list_append(dae4, dae3) => dae*)
        -----------------------------------------------------------
        inst_element(env,mods,pre,csets,ci_state,
                     SCode.COMPONENT(n,final,prot,
                                     attr as SCode.ATTR(ad,flow,acc,param,dir),
                                     t,m))
          => (dae, env',csets',ci_state,
              [Types.VAR(n,Types.ATTR(flow,acc,param,dir), prot, ty, binding)])
```

If the class lookup in the previous rule fails, this rule catches the error and prints an error message about the unknown class.

Failure

```
   rule  not Lookup.lookup_class(env,t) => (cl,cenv) &
         Absyn.path_string(t) => s &
         print "# unknown class '" & print s & print "' while instantiating " &
         Prefix.prefix_path(Absyn.IDENT(n),pre) => n' &
         Absyn.path_string(n') => ns &
         print ns & print "\n"
         --------------------------
         inst_element(env,_,pre,csets,ci_state,
                      SCode.COMPONENT(n,final,prot,_,t,_))
          => ([],env,csets,ci_state,[])

   rule  Prefix.prefix_path(Absyn.IDENT(n),pre) => n' &
         Absyn.path_string(n') => ns &
         print "- inst_element COMPONENT(" & print ns & print ") failed\n"
         -----------------------------------------------------------------
         inst_element(_,_,pre,_,_,SCode.COMPONENT(n,_,_,_,_,_))
          => fail

end
```

Relation **`inst_var`**

A component element in a class may consist of several subcomponents or array elements. This relation is used to instantiate a component, instantiating all subcomponents and array elements separately.

```
relation inst_var : (Env.Env,Mod.Mod,Prefix.Prefix,Connect.Sets,Ident,
                     SCode.Class,SCode.Attributes,
                     int list, int list)
          => (DAE.Element list,Connect.Sets,Types.Type) =
```

The class is instantiated with the calculated modification, and an extended prefix.

```
  rule  list_reverse idxs => idxs' &
        Prefix.prefix_add(n,idxs',pre) => pre' &
        (* This is where redeclare should be handled *)
        inst_class(env,mod,pre',csets,cl) => (dae1,csets',ty,st) &

        Exp.int_subscripts idxs' => subs &
        Prefix.prefix_cref(pre,Exp.CREF_IDENT(n,subs)) => cr &
        inst_mod_equation(cr,ty,mod) => dae2 &
        dae_declare(cr,ty,SCode.ATTR([], flow, acc, vt, dir)) => dae3 &

        list_append(dae1,dae2) => daex &
        list_append(daex,dae3) => dae
        --------------------------
        inst_var(env,mod,pre,csets,n,cl,SCode.ATTR(_,flow,acc,vt,dir),
                 [],idxs)
          => (dae,csets',ty)

  rule  print "= inst_array mod: " & print n & Mod.print_mod mod & print "\n" &
        inst_array(env,mod,pre,csets,n,(cl,attr),1,dim,dims,idxs)
          => (dae, csets', ty) &
        Types.lift_array(ty,SOME(dim)) => ty'
        -------------------------------
        inst_var(env,mod,pre,csets,n,cl,attr,dim::dims,idxs)
          => (dae,csets',ty')

  rule  print "- inst_var failed: " & print n & print "\n"
        ------------------------------------------------
        inst_var(_,_,_,_,n,_,_,_,_) => fail

end
```

Relation **`inst_array`**

When an array is instantiated by `inst_var`, this relation is used to go through all the array elements and instantiate each array element separately.

```
relation inst_array : (Env.Env,Mod.Mod,Prefix.Prefix,Connect.Sets,Ident,
                       (SCode.Class*SCode.Attributes),int,int,int list,int list)
          => (DAE.Element list,Connect.Sets,Types.Type) =

  rule  int_gt(i,stop) => true
        ----------------------
        inst_array(env,mod,pre,csets,n,(cl,attr),i,stop,dims,idxs)
          => ([], csets, Types.T_NOTYPE)

  rule  Mod.lookup_idx_modification(mod,i) => mod' &
        inst_var(env,mod',pre,csets,n,cl,attr,dims,i::idxs)
          => (dae1,csets',ty) &
        int_add(i,1) => i' &
        inst_array(env,mod,pre,csets',n,(cl,attr),i',stop,dims,idxs)
          => (dae2, csets'',_) &

        list_append(dae1,dae2) => dae
        -----------------------------
        inst_array(env,mod,pre,csets,n,(cl,attr),i,stop,dims,idxs)
          => (dae,csets'',ty)

  rule  print "- inst_array failed: " & print n & print "\n"
        --------------------------------------------------
        inst_array(_,_,_,_,n,(_,_),_,_,_,_) => fail

end
```

Relation `elab_arraydim`

This relations examines both an `Absyn.ArrayDim` and an 'Mod.EqMod option' argument to find out the dimensions af a component. If no equation modifications is given, only the declared dimension is used.

When the size of a dimension in the type is undefined, the corresponding size in the type of the modification is used.

All this is accomplished by examining the two arguments separately and then using `complete_arraydime` or `compatible_arraydim` to check that that the dimension sizes are compatible and complete.

```
relation elab_arraydim : (Env.Env, Absyn.ArrayDim, Mod.EqMod option)
          => int list =

  rule  elab_arraydim_decl(env,ad) => dim &
        complete_arraydim(dim) => dim'
        ------------------------------
        elab_arraydim(env,ad,NONE) => dim'

  rule  elab_arraydim_decl(env,ad) => dim1 &
        elab_arraydim_type(t,ad) => dim2 &
        compatible_arraydim(dim1,dim2) => dim3
        ---------------------------------
        elab_arraydim(env,ad,SOME((e,Static.PROP(t,c)))) => dim3

  rule  elab_arraydim_decl(env,ad) => dim1 &
        elab_arraydim_type(t,ad) => dim2 &
        not compatible_arraydim(dim1,dim2) => dim3 &
        print "# Array dimension mismatch\n" &
        print "    expression " & Exp.print_exp e &
        print " has type " & Types.print_type t & print "\n" &
        print "    expected array dimensions [" &
        print_dim dim1 & print "]\n"
        ---------------------------------
        elab_arraydim(env,ad,SOME((e,Static.PROP(t,c)))) => fail

  rule  print "- elab_arraydim failed\n"
        -------------------------------------
        elab_arraydim(_,_,_) => fail

end
```

Relation **print_dim**

This relation prints array dimensions. The code is not included in the report.

Relation **elab_arraydim_decl**

Given an `Absyj.ArrayDim`, this relation evaluates all dimension size specifications, creating a list of (optional) integers. When the array dimension size is specified as **:**, the result will contain `NONE`.

```
relation elab_arraydim_decl : (Env.Env, Absyn.ArrayDim) => (int option) list =

  axiom elab_arraydim_decl(_,[]) => []

  rule  elab_arraydim_decl(env,ds) => l
        -------------------------------
        elab_arraydim_decl(env,Absyn.NOSUB::ds) => NONE::l

  rule  Static.elab_exp(env, d)
          => (e,Static.PROP(Types.T_INTEGER,true)) &
        Static.ceval(env,e) => Values.INTEGER(i) &
        elab_arraydim_decl(env,ds) => l
        -----------------------------------------
        elab_arraydim_decl(env,Absyn.SUBSCRIPT(d)::ds) => SOME(i)::l

  rule  Static.elab_exp(env, d)
          => (e,Static.PROP(Types.T_INTEGER,false)) &
        print "# Variable array sizes are not allowed\n" &
        print "    expression: " & Exp.print_exp e & print "\n"
        -----------------------------------------------------
        elab_arraydim_decl(env,Absyn.SUBSCRIPT(d)::ds) => fail

  rule  Static.elab_exp(env, d)
          => (e,Static.PROP(t,_)) &
        print "# Array dimensions must be integer\n" &
        print "    expression: " & Exp.print_exp e &
        print " (" & Types.print_type t & print ")\n"
        -----------------------------------------------------
        elab_arraydim_decl(env,Absyn.SUBSCRIPT(d)::ds) => fail

  rule  print "- elab_arraydim_decl failed\n"
        -------------------------------------
        elab_arraydim_decl(_,_) => fail

end
```

Relation **complete_arraydim**

This relation converts a list of optional integers to a list of integers. If one element of the list is `NONE`, this relation will fail.

This is used to check that an array specification contain fully specified array dimension sizes.

```
relation complete_arraydim : (int option) list => int list =

  axiom complete_arraydim [] => []

  rule  complete_arraydim xs => xs'
        ---------------------------
        complete_arraydim SOME(x)::xs => x::xs'

end
```

Relation `compatible_arraydim`

Given two, possibly incomplete, array dimension size specifications as list of optional integers, this relation checks whether they are compatible. Being compatible means that they have the same number of dimension, and for every dimension at least one of the lists specifies its size. If both lists specify a dimension size, they have to specify the same size.

```
relation compatible_arraydim : ((int option) list,(int option) list)
          => int list =

  axiom compatible_arraydim([],[]) => []

  rule  compatible_arraydim(xs,ys) => l
        -------------------------------
        compatible_arraydim(SOME(x)::xs,NONE::ys) => x::l

  rule  compatible_arraydim(xs,ys) => l
        -------------------------------
        compatible_arraydim(NONE::xs,SOME(y)::ys) => y::l

  rule  x = y &
        compatible_arraydim(xs,ys) => l
        -------------------------------
        compatible_arraydim (SOME(x)::xs,SOME(y)::ys) => x::l

  rule  print "- compatible_arraydim failed\n"
        --------------------------------------
        compatible_arraydim (_,_) => fail

end
```

Relation `elab_arraydim_type`

Find out the dimension sizes of a type. The second argument is used to know how many dimensions should be extracted from the type.

```
relation elab_arraydim_type : (Types.Type, Absyn.ArrayDim)
          => int option list =

  rule  elab_arraydim_type(t,ad) => l
        ---------------------------
        elab_arraydim_type(Types.T_ARRAY(Types.DIM(i),t), _::ad) => i::l

  axiom elab_arraydim_type(_,[]) => []

end
```

Relation `inst_class_decl`

The class definition is instantiated although no variable is declared with it. After instantiating it, it is checked to see if it can be used as a package, and if it can, then it is added as a variable under the same name as the class. This makes it possible to use a unified lookup mechanism. And since packages only can contain constants and class definition, instantiating a package does not do anything else.

```
relation inst_class_decl : (Env.Env, Mod.Mod, Prefix.Prefix,
                            Connect.Sets, SCode.Class)
          => (Env.Env, DAE.Element list) =

  rule  Env.extend_frame_c(env,c) => env' &
        implicit_instantiation(env',Mod.NOMOD,pre,csets,c) => (env'',dae)
        ----------------------------------
        inst_class_decl(env,mod,pre,csets,c as SCode.CLASS(n,_,restr,_))
          => (env'', dae)

end
```

relation implicit_instantiation

If a class definition is a function or a package, it is implicitly instantiated and added as a type binding under the same name as the class name.

```
relation implicit_instantiation : (Env.Env, Mod.Mod, Prefix.Prefix,
                                   Connect.Sets, SCode.Class)
          => (Env.Env, DAE.Element list) =

  rule  Prefix.prefix_add(n,[],pre) => pre' &
        inst_class(env,mod,pre',csets,c) => (dae, csets', ty, st) &
        Env.extend_frame_t(env, n, ty) => env'
        --------------------------------------
        implicit_instantiation(env,mod,pre,csets,
                               c as SCode.CLASS(n,_,Absyn.R_PACKAGE,_))
          => (env', dae)

  rule  Prefix.prefix_add(n,[],pre) => pre' &
        inst_class(env,mod,pre',csets,c) => (dae, csets', ty, st) &
        Env.extend_frame_t(env, n, ty) => env'
        --------------------------------------
        implicit_instantiation(env,mod,pre,csets,
                               c as SCode.CLASS(n,_,Absyn.R_FUNCTION,_))
          => (env', (*dae*) [])

  axiom implicit_instantiation(env,mod,pre,csets,c) => (env,[])

end
```

Relation **`dae_declare`**

Given a global component name, a type, and a set of attributes, this relation declares a component for the DAE result. Altough this relation returns a list of `DAE.Elements`, only one component is actually declared.

The relations `dae_declare2` and `dae_declare3` below are helper relations that perform parts of the task.

```
relation dae_declare : (Exp.ComponentRef,
                        Types.Type, SCode.Attributes) => DAE.Element list =

  rule  dae_declare2 (vn, ty, par) => dae
        -----------------------------------
        dae_declare (vn, ty, SCode.ATTR(_,_,_,par,_)) => dae

  rule  print "- dae_declare failed\n"
        ------------------------------
        dae_declare(_,_,_) => fail

end
```

```
relation dae_declare2 : (Exp.ComponentRef, Types.Type, Absyn.Variability)
          => DAE.Element list =

  rule  dae_declare3 (vn, ty, DAE.VARIABLE) => dae
        -----------------------------------------
        dae_declare2 (vn, ty, Absyn.VAR) => dae

  rule  dae_declare3 (vn, ty, DAE.DISCRETE) => dae
        -----------------------------------------
        dae_declare2 (vn, ty, Absyn.DISCRETE) => dae

  rule  dae_declare3 (vn, ty, DAE.PARAM) => dae
        -----------------------------------------
        dae_declare2 (vn, ty, Absyn.PARAM) => dae

  rule  dae_declare3 (vn, ty, DAE.CONST) => dae
        -----------------------------------------
        dae_declare2 (vn, ty, Absyn.CONST) => dae

  rule  print "- dae_declare2 failed\n"
        ------------------------------
        dae_declare2(_,_,_) => fail

end
```

```
relation dae_declare3 : (Exp.ComponentRef, Types.Type, DAE.VarKind)
          => DAE.Element list =

  axiom dae_declare3 (vn, Types.T_INTEGER, kind)
          => [DAE.VAR(vn, kind, DAE.INT)]

  axiom dae_declare3 (vn, Types.T_REAL, kind)
          => [DAE.VAR(vn, kind, DAE.REAL)]

  axiom dae_declare3 (vn, Types.T_BOOL, kind)
          => [DAE.VAR(vn, kind, DAE.BOOL)]

  axiom dae_declare3 (vn, Types.T_STRING, kind)
          => [DAE.VAR(vn, kind, DAE.STRING)]

  axiom dae_declare3 (c,ty,_) => []

end
```

Relation `inst_equation`

The DAE output of the translation contains equations which in most cases directly corresponds to equations in the source. Some of them are also generated from `commect` clauses.

This relation takes an equation from the source and generates DAE equations and connection sets.

```
relation inst_equation : (Env,Mod, Prefix, Connect.Sets, ClassInf.State,
                          SCode.Equation)
          => (DAE.Element list, Env, Connect.Sets, ClassInf.State) =
```

connect

Handle connct statements

```
  rule  inst_connect(csets,env,pre,c1,c2) => csets' &
        ClassInf.trans(ci_state, ClassInf.FOUND_EQUATION) => ci_state'
        ------------------------------------------------------------
        inst_equation(env,mods,pre, csets, ci_state, SCode.EQ_CONNECT(c1,c2))
          => ([],env,csets',ci_state')
```

assert

```
rule  Static.elab_exp(env,e) => (e', Static.PROP(t, c)) &
      inst_assertion (env, e', t, c, d, pre) => dae &
      ClassInf.trans(ci_state, ClassInf.FOUND_EQUATION) => ci_state'
      --------------------------------------------------------------
      inst_equation(env,mods,pre,csets, ci_state,
                   SCode.EQ_EXPR(Absyn.CALL(Absyn.CREF_IDENT("assert",[]),
                                             [e,Absyn.STRING(d)])))
        => (dae,env,csets,ci_state')
```

Normal equations

```
rule  print "- No expression equations yet\n" &
      ClassInf.trans(ci_state, ClassInf.FOUND_EQUATION) => ci_state'
      ------------------------------------
      inst_equation(env,mods,pre,csets,ci_state,SCode.EQ_EXPR(_)) => fail
```

The following rule handles shadowed (replaced) equations. If an equation has a simple name on the left-hand side, and that component has an equation modifier, this equation is discared.

```
rule  Lookup.lookup_ident_local(env,n)
        => Types.VAR(_,_,_,_,Types.EQBOUND(_,_)) &
      ClassInf.trans(ci_state, ClassInf.FOUND_EQUATION) => ci_state'
      ------------------------------------------------------
      inst_equation(env,mods,pre,csets,ci_state,
                   SCode.EQ_EQUALS(Absyn.CREF(Absyn.CREF_IDENT(n,[])), e2))
        => ([],env,csets,ci_state')

rule  Static.elab_exp(env,e1) => (e1',prop1) &
      Static.elab_exp(env,e2) => (e2',prop2) &
      Prefix.prefix_exp(env,e1',pre) => e1'' &
      Prefix.prefix_exp(env,e2',pre) => e2'' &
      inst_eq_equation(e1'',prop1,e2'',prop2) => dae &
      ClassInf.trans(ci_state, ClassInf.FOUND_EQUATION) => ci_state'
      --------------------------------------------------------------
      inst_equation(env, mods, pre, csets, ci_state, SCode.EQ_EQUALS(e1,e2))
        => (dae, env, csets, ci_state')
```

`if` statements

If statements are instantiated by evaluating the conditional expression, and selecting the branch that should be used.

```
rule  Static.elab_exp(env,e) => (e',Static.PROP(Types.T_BOOL,true)) &
      Static.ceval(env,e') => Values.BOOL(cond) &
      select(cond,tb,fb) => b &
      inst_list(env,mod,pre,csets,ci_state,inst_equation, b)
        => (dae,env',csets', ci_state')
      -------------------------------------------------------
      inst_equation(env,mod,pre,csets,ci_state,SCode.EQ_IF(e,tb,fb))
        => (dae,env',csets', ci_state')
```

`for` loops

The loop expression is evaluated to a constant array of integers, and then the loop is unrolled.

```
  rule  Lookup.lookup_var(env, Exp.CREF_IDENT(i,[]))
          => (Types.ATTR(false, SCode.RW, Absyn.VAR, _),
              Types.T_INTEGER, Types.UNBOUND) &
        Static.elab_exp (env,e)
          => (e',Static.PROP(Types.T_ARRAY(Types.DIM(_),
                                           Types.T_INTEGER),
                             true)) &
        (* FIXEM: Check bounds *)
        Static.ceval (env,e') => v &
        unroll(env,mod,pre,csets,ci_state,i,v,el) => (dae, csets') &
        ClassInf.trans(ci_state, ClassInf.FOUND_EQUATION) => ci_state'
        -----------------------------------------------------------
        inst_equation(env,mod,pre,csets,ci_state,SCode.EQ_FOR(i,e,el))
          => (dae, env,csets', ci_state')

  rule  not Lookup.lookup_var(env, Exp.CREF_IDENT(i,[]))
          => (Types.ATTR(false, SCode.RW, Absyn.VAR, _),
              Types.T_INTEGER, Types.UNBOUND) &
        print "# Invalid loop variable: " & print i & print "\n"
        -----------------------------------------------------------
        inst_equation(env,mod,pre,csets,ci_state,SCode.EQ_FOR(i,e,el))
          => fail

  rule  print "- inst_equation failed\n"
        ----------------------------------
        inst_equation(_,_,_,_,_,_) => fail

end
```

Relation `inst_eq_equation`

Equations follow the same typing rules as equality expressions. This relation adds the equation to the DAE.

```
relation inst_eq_equation : (Exp.Exp, Static.Properties,
                             Exp.Exp, Static.Properties) => DAE.Element list =

  rule  Static.match_prop(e1, p1, p2) => e1' &
        inst_eq_equation_2(e1', e2, t2) => dae
        ------------------------------------------
        inst_eq_equation(e1,p1 as Static.PROP(t1,_),
                         e2,p2 as Static.PROP(t2,_))
          => dae

  rule  Static.match_prop(e2, p2, p1) => e2'&
        inst_eq_equation_2(e1, e2', t1) => dae
        ------------------------------------------
        inst_eq_equation(e1,p1 as Static.PROP(t1,_),
                         e2,p2 as Static.PROP(t2,_))
          => dae

  rule  (* Types.equivtypes(t1,t2) => false & *)
        print "# Illegal types in equation\n"&
        print "    lhs: " & Exp.print_exp e1 &
        print " :: " & Types.print_type t1 & print "\n" &
        print "    rhs: " & Exp.print_exp e2 &
        print " :: " & Types.print_type t2 & print "\n"
        ----------
        inst_eq_equation(e1,Static.PROP(t1,_),
                         e2,Static.PROP(t2,_)) => fail

end
```

Relation `inst_eq_equation_2`

This is the second stage of `inst_eq_equation`, when the types are checked.

```
relation inst_eq_equation_2 : (Exp.Exp, Exp.Exp, Types.Type)
        => DAE.Element list =

  axiom inst_eq_equation_2(e1,e2,Types.T_INTEGER)
        => [DAE.EQUATION(e1,e2)]

  axiom inst_eq_equation_2(e1,e2,Types.T_REAL)
        => [DAE.EQUATION(e1,e2)]

  axiom inst_eq_equation_2(e1,e2,Types.T_STRING)
        => [DAE.EQUATION(e1,e2)]

  axiom inst_eq_equation_2(e1,e2,Types.T_BOOL)
        => [DAE.EQUATION(e1,e2)]

  rule  RTOpts.split_arrays => true &
        print "  Array equation: " &
        Exp.print_exp e1 & print " = " &
        Exp.print_exp e2 & print "\n" &
        inst_array_equation(e1,e2,ad,t) => dae
        ----------------------------------------
        inst_eq_equation_2(e1,e2,Types.T_ARRAY(ad,t)) => dae

  rule  RTOpts.split_arrays => false
        ----------------------------------------
        inst_eq_equation_2(e1,e2,Types.T_ARRAY(ad,t))
          => [DAE.EQUATION(e1,e2)]

  axiom inst_eq_equation_2(e1, e2, Types.T_COMPLEX(_,[])) => []

  rule  Exp.extend_cref(c1,n,[]) => c1' &
        Exp.extend_cref(c2,n,[]) => c2' &
        inst_eq_equation_2(Exp.CREF(c1'), Exp.CREF(c2'), t) => dae1 &
        inst_eq_equation_2(Exp.CREF(c1), Exp.CREF(c2),
                           Types.T_COMPLEX(cs,vs)) => dae2 &
        list_append(dae1, dae2) => dae
        ------------------------------
        inst_eq_equation_2(Exp.CREF(c1), Exp.CREF(c2),
                           Types.T_COMPLEX(cs,Types.VAR(n,_,_,t,_)::vs)) => dae
```

When the type of the expressions is a complex type, and the left-hand side of the equation is not a component reference, a new variable is introduced to be able to dereference components of the expression.

This is rather ugly, since it doesn't really solve the problem of describing the semantics. Now the semantics of composite equations are defined in terms of other composite equations. To make this a little cleaner, the equation that equates the new name to the expression is stored using DAE.DEFINE rather than DAE.EQUATION. This makes it a little clearer.

```
  rule  new_ident => n &
        dae_declare(n, t, SCode.ATTR([], false, SCode.RW,
                                     Absyn.VAR, Absyn.BIDIR))
          => decl &
        inst_eq_equation_2(Exp.CREF(n), e2, t) => dae1 &
        list_append(decl, DAE.DEFINE(n, e1)::dae1) => dae
        ------------------------------------------------------------
        inst_eq_equation_2(e1, e2 as Exp.CREF(_),
                           t as Types.T_COMPLEX(_,_)) => dae
```

When the right-hand siade is not a component reference a similar trick is applied. This also catched the case where none of the sides is a component reference

```
  rule  new_ident => n &
        dae_declare(n, t, SCode.ATTR([], false, SCode.RW,
                                     Absyn.VAR, Absyn.BIDIR))
          => decl &
        inst_eq_equation_2(e1, Exp.CREF(n), t) => dae1 &
        list_append(decl, DAE.DEFINE(n, e2)::dae1) => dae
        -------------------------------------------------------------
        inst_eq_equation_2(e1, e2, t as Types.T_COMPLEX(_,_)) => dae

  rule  print "- inst_eq_equation_2 failed\n"
        -------------------------------------
        inst_eq_equation_2(_,_,_) => fail

end
```

Relation **`inst_array_equation`**

This checks the array size and uses `inst_array_el_eq` to create equations for each array element.

```
relation inst_array_equation : (Exp.Exp, Exp.Exp,
                                Types.ArrayDim, Types.Type)
          => DAE.Element list =

  rule  print "# Cannot instantiate equations with arrays of unknown size\n" &
        print "    " & Exp.print_exp e1 & print " = " & Exp.print_exp e2&
        print "\n"
        ----------
        inst_array_equation(e1, e2, Types.DIM(NONE), t) => fail

  rule  inst_array_el_eq(e1, e2, t, 1, sz) => dae
        --------------------------------------------
        inst_array_equation(e1, e2, Types.DIM(SOME(sz)), t) => dae

  rule  print "- inst_array_equation failed\n"
        --------------------------------------
        inst_array_equation (_,_,_,_) => fail

end
```

Relation **`inst_array_el_eq`**

This relation loops recursively through all indexes in the two arrays and generates an equation for each pair of elements.

```
relation inst_array_el_eq : (Exp.Exp, Exp.Exp, Types.Type, int, int)
          => DAE.Element list =

  rule  int_le(i,sz) => true &
        Exp.simplify(Exp.ASUB(e1,i)) => e1' &
        Exp.simplify(Exp.ASUB(e2,i)) => e2' &
        inst_eq_equation_2(e1',e2',t) => dae1 &
        int_add(i,1) => i' &
        inst_array_el_eq(e1,e2,t,i',sz) => dae2 &
        list_append(dae1, dae2) => dae
        ------------------------------
        inst_array_el_eq(e1, e2, t, i, sz) => dae

  rule  int_le(i,sz) => false
        ---------------------
        inst_array_el_eq(e1, e2, t, i, sz) => []

  rule  print "- inst_array_el_eq failed\n"
        -----------------------------------
        inst_array_el_eq(_,_,_,_,_) => fail

end
```

Relation **`inst_assertion`**

Instantiate an assertion. There are not assertions in the output currently.

```
relation inst_assertion : (Env.Env, Exp.Exp, Types.Type,
                           bool, string, Prefix)
          => DAE.Element list =

        (* Constant assertions *)
  axiom inst_assertion (env, e, Types.T_BOOL, true, d, pre) => [(* lost *)]

  axiom inst_assertion (env, e, Types.T_BOOL, _, d, pre) => [(* lost *)]

  rule  print "# Assertions have to be of type Boolean\n" &
        print "  assertion: (" &
        Prefix.prefix_exp(env,e,pre) => e' & Exp.print_exp e' & print "): " &
        print d & print "\n"
        -----------------------------
        inst_assertion (env, e, _, _, d, pre) => fail

end
```

Relation **`unroll`**

Unrolling a loop is a way of removing the non-linear structure of the `for` clause by explicitly repeating the body of the loop once for each iteration.

```
relation unroll : (Env, Mod, Prefix, Connect.Sets, ClassInf.State,
                   Ident, Values.Value, SCode.Equation list)
          => (DAE.Element list, Connect.Sets) =

  axiom unroll (_,_,_,csets,_,_,Values.ARRAY([]),_) => ([], csets)

  rule  Env.open_scope env => env' &
        Env.extend_frame_v (env',
                            Types.VAR(i, Types.ATTR(false,
                                                    SCode.RO,
                                                    Absyn.CONST,
                                                    Absyn.BIDIR),
                                      true,
                                      Types.T_INTEGER,
                                      Types.VALBOUND(fst))) => env'' &
        inst_list(env'',mods,pre,csets,ci_state,inst_equation, eqs)
          => (dae1,env''',csets', ci_state') &
        unroll (env, mods, pre, csets', ci_state', i, Values.ARRAY(rest), eqs)
          => (dae2, csets'') &
        list_append(dae1,dae2) => dae
        -----------------------------
        unroll (env,mods,pre,csets,ci_state,i,Values.ARRAY(fst::rest),eqs)
          => (dae, csets'')

  rule  print "- unroll " & Values.print_val v & print " failed\n"
        -------------------------
        unroll (_,_,_,_,_,_,v,_) => fail

end
```

Relation **`inst_algorithm`**

Algorithms are converted to the representation defined in the module `Algorithm`, and the added to the DAE result.

This relation converts an algorithm section.

```
relation inst_algorithm : (Env, Mod, Prefix, Connect.Sets, ClassInf.State,
                           SCode.Algorithm)
          => (DAE.Element list, Env, Connect.Sets, ClassInf.State) =

  rule  inst_statements(env,statements) => statements'
        ---------------------------------------------
        inst_algorithm(env,_,_,csets,ci_state,SCode.ALGORITHM(statements))
          => ([DAE.ALGORITHM(Algorithm.ALGORITHM(statements'))],
              env,csets,ci_state)

  rule  print "- inst_algorithm failed\n"
        --------------------------------
        inst_algorithm(_,_,_,_,_,_) => fail

end
```

Relation **`inst_statements`**

This relation converts a list of algorithm statements.

```
relation inst_statements : (Env, Absyn.Algorithm list)
          => Algorithm.Statement list =

  axiom inst_statements(env,[]) => []

  rule  inst_statement(env,x) => x' &
        inst_statements(env,xs) => xs'
        ------------------------------
        inst_statements(env,x::xs) => x'::xs'

end
```

Relation **`inst_statement`**

This relation Looks at an algorithm statement and uses relations in the `Algorithm` module to build a representation of it that can be used in the DAE output.

```
relation inst_statement : (Env, Absyn.Algorithm) => Algorithm.Statement =

  rule  Static.elab_cref (env,cr) => (ce, cprop, acc) &
        Static.elab_exp (env,e) => (e', eprop) &
        Algorithm.make_assignment(ce,cprop,e',eprop,acc) => stmt
        --------------------------------------------------------
        inst_statement(env,Absyn.ALG_ASSIGN(cr,e)) => stmt

  rule  Static.elab_exp(env,e) => (e',prop) &
        inst_statements(env,tb) => tb' &
        inst_elseifs(env,eib) => eib' &
        inst_statements(env,fb) => fb' &
        Algorithm.make_if(e',prop,tb',eib',fb') => stmt
        --------------------------------------------
        inst_statement(env,Absyn.ALG_IF(e,tb,eib,fb)) => stmt

  rule  Static.elab_exp (env,e) => (e',prop) &
        inst_statements(env,sl) => sl' &
        Algorithm.make_for(i,e',prop,sl') => stmt
        ---------------------------------------
        inst_statement(env,Absyn.ALG_FOR(i,e,sl)) => stmt

  rule  Static.elab_exp (env,e) => (e',prop) &
        inst_statements(env,sl) => sl' &
        Algorithm.make_while(e',prop,sl') => stmt
        ---------------------------------------
        inst_statement(env,Absyn.ALG_WHILE(e,sl)) => stmt

  rule  Static.elab_exp (env,e) => (e',prop) &
        inst_statements(env,sl) => sl' &
        Algorithm.make_when(e',prop,sl') => stmt
        ---------------------------------------
        inst_statement(env,Absyn.ALG_WHEN(e,sl)) => stmt

  rule  print "- inst_statement failed\n"
        ---------------------------------
        inst_statement(_,_) => fail

end
```

Relation `inst_elseifs`

This relation helps `inst_statement` to handle `elseif` parts.

```
relation inst_elseifs : (Env.Env, (Absyn.Exp * Absyn.Algorithm list) list)
        => (Exp.Exp * Static.Properties * Algorithm.Statement list) list =

  axiom inst_elseifs (env,[]) => []

  rule  Static.elab_exp(env,e) => (e',prop) &
        inst_statements(env,l) => stmts &
        inst_elseifs(env,tail) => tail'
        -------------------------------
        inst_elseifs(env,(e,l)::tail) => ((e',prop,stmts)::tail')

  rule  print "- inst_elseifs failed\n"
        -------------------------------
        inst_elseifs(_,_) => fail

end
```

Relation `inst_connect`

```
relation inst_connect: (Connect.Sets, Env, Prefix,
                        Absyn.ComponentRef, Absyn.ComponentRef)
          => Connect.Sets =

  rule  Static.elab_cref(env, c1) => (Exp.CREF(c1'),prop1,acc) &
        Static.elab_cref(env, c2) => (Exp.CREF(c2'),prop2,acc) &
        Static.canon_cref(env, c1') => c1'' &
        Static.canon_cref(env, c2') => c2'' &
        Lookup.lookup_var_local(env,c1'')
          => (attr1 as Types.ATTR(flow1,_,_,_),ty1,_) &
        Lookup.lookup_var_local(env,c2'') => (attr2,ty2,_) &
```

Check that the types of the connectors are good.

```
        valid_connector(ty1) &
        valid_connector(ty2) &
        check_connect_types(c1'',ty1,attr1,c2'',ty2,attr2) &

        component_face(c1'') => f1 &
        component_face(c2'') => f2 &
        connect_components(sets, pre, c1'', f1, ty1, c2'', f2, ty2, flow1)
          => sets'
        ---------------------------------
        inst_connect(sets, env,pre,c1,c2) => sets'

  rule  print "# could not connect\n"
        ---------------------------------
        inst_connect(sets,_,_,_,_) => fail

end
```

Relation `valid_connector`

This relation tests whether a type is a eligible to be used in connections.

```
relation valid_connector : Types.Type => () =

  axiom valid_connector Types.T_REAL

  rule  ClassInf.valid(state,Absyn.R_CONNECTOR)
        ----------------------------------------
        valid_connector Types.T_COMPLEX(state,_)

  rule  print "# Can't connect objects of type " &
        Types.print_type t & print "\n"
        ------------------------------
        valid_connector t => fail
end
```

Relation `check_connect_types`

Check that the type and type attributes of two connectors match, so that they really may be connected.

```
relation check_connect_types : (Exp.ComponentRef, Types.Type,
                                Types.Attributes,
                                Exp.ComponentRef, Types.Type,
                                Types.Attributes) => () =

  rule  print "# Can't connect two input variables\n" &
        print "    while connecting " & Exp.print_component_ref c1 &
        print " to " & Exp.print_component_ref c2 & print "\n"
        ------------------------------------------------------
        check_connect_types(c1,_,Types.ATTR(_,_,_,Absyn.INPUT),
                            c2,_,Types.ATTR(_,_,_,Absyn.INPUT)) => fail

  rule  print "# Can't connect two output variables\n" &
        print "    while connecting " & Exp.print_component_ref c1 &
        print " to " & Exp.print_component_ref c2 & print "\n"
        ------------------------------------------------------
        check_connect_types(c1,_,Types.ATTR(_,_,_,Absyn.OUTPUT),
                            c2,_,Types.ATTR(_,_,_,Absyn.OUTPUT)) => fail

  rule  flow1 = flow2 &
        Types.equivtypes(t1, t2) => true
        -------------------------------
        check_connect_types(_,t1,Types.ATTR(flow1,_,_,_),
                            _,t2,Types.ATTR(flow2,_,_,_))

  rule  print "# Can't connect flow component " &
        Exp.print_component_ref c1 &
        print " to non-flow component " &
        Exp.print_component_ref c2 & print "\n"
        ------------------------------------------------------
        check_connect_types(c1,_,Types.ATTR(true,_,_,_),
                            c2,_,Types.ATTR(false,_,_,_)) => fail

  rule  print "# Can't connect non-flow component " &
        Exp.print_component_ref c1 &
        print " to flow component " &
        Exp.print_component_ref c2 & print "\n"
        ------------------------------------------------------
        check_connect_types(c1,_,Types.ATTR(false,_,_,_),
                            c2,_,Types.ATTR(true,_,_,_)) => fail

  rule  print "- check_connect_types(" & Exp.print_component_ref c1 &
        print " <-> " & Exp.print_component_ref c2 & print ") failed\n"
        ------------------------------------------------------
        check_connect_types(c1,_,_,c2,_,_) => fail

end
```

Relation `connect_components`

This relation connects two components and generates connection sets along the way. For simple components (of type `Real`) it adds the components to the set, and for complex types it traverses the subcomponents and recursively connects them to each other.

```
relation connect_components: (Connect.Sets,
                              Prefix,
                              Exp.ComponentRef, Connect.Face, Types.Type,
                              Exp.ComponentRef, Connect.Face, Types.Type,
                              bool) => Connect.Sets =
```

Flow type

```
  rule  Prefix.prefix_cref(pre, c1) => c1' &
        Prefix.prefix_cref(pre, c2) => c2' &
        Connect.add_flow(sets, c1', f1, c2', f2) => sets'
        -----------------------------------------
        connect_components(sets, pre, c1, f1, Types.T_REAL,
                           c2, f2, Types.T_REAL, true) => sets'
```

Non-flow type

```
  rule  Prefix.prefix_cref(pre, c1) => c1' &
        Prefix.prefix_cref(pre, c2) => c2' &
        Connect.add_equ(sets, c1', c2') => sets'
        -----------------------------------------
        connect_components(sets, pre, c1, _, Types.T_REAL,
                           c2, _, Types.T_REAL, false) => sets'
```

Complex types

```
  rule  Prefix.prefix_cref(pre, c1) => c1' &
        Prefix.prefix_cref(pre, c2) => c2' &
        connect_vars(sets,c1',f1,l1,c2',f2,l2) => sets'
        ----------------------------------------
        connect_components(sets,pre,c1,f1, Types.T_COMPLEX(_,l1),
                           c2,f2, Types.T_COMPLEX(_,l2),_) => sets'
```

Error

```
  rule  Prefix.prefix_cref(pre, c1) => c1' &
        Prefix.prefix_cref(pre, c2) => c2' &
        print "# Wrong type in connection\n"&
        print "  while making connectiion " &
        Exp.print_component_ref c1 & print " <-> " &
        Exp.print_component_ref c2 & print "\n" &
        print "  The type `" & Types.print_type t1 &
        print "' is not allowed in connections\n"
        ----------------------------------------
        connect_components(_,pre,c1,_,t1,c2,_,t2,_) => fail

end
```

Relation `connect_vars`

This relation connects two subcomponents by adding the component name to the current path and recursively connecting the components using the relation `connet_components`.

```
relation connect_vars : (Connect.Sets,
                         Exp.ComponentRef, Connect.Face, Types.Var list,
                         Exp.ComponentRef, Connect.Face, Types.Var list)
          => Connect.Sets =

  axiom connect_vars(sets,_,_,[],_,_,[]) => sets

  rule  Exp.extend_cref(c1, n, []) => c1' &
        Exp.extend_cref(c2, n, []) => c2' &
        check_connect_types(c1', ty1, attr1, c2', ty2, attr2) &
        connect_components(sets,Prefix.NOPRE,
                           c1',f1, ty1,
                           c2',f2, ty2, flow1) => sets' &
        connect_vars(sets',c1,f1,xs1,c2,f2,xs2) => sets''
        --------------------------------------
        connect_vars(sets,
                     c1,f1, Types.VAR(n,attr1 as Types.ATTR(flow1,_,vt1,_),
                                      _,ty1,_)::xs1,
                     c2,f2, Types.VAR(_,attr2 as Types.ATTR(flow2,_,vt2,_),
                                      _,ty2,_)::xs2)
          => sets''

end
```

Relation `mktype`

From a class inference state, and a list of subcomponents, this relation returns `Types.Type`. If the class inference state indicates that the type should be a built-in type, one of the built-in type constructors is used. Otherwise, a `T_COMPLEX` is built.

```
relation mktype : (ClassInf.State,Types.Var list) => Types.Type =

  axiom mktype(ClassInf.TYPE_INTEGER(_),_) => Types.T_INTEGER
  axiom mktype(ClassInf.TYPE_REAL(_),_)    => Types.T_REAL
  axiom mktype(ClassInf.TYPE_STRING(_),_)  => Types.T_STRING
  axiom mktype(ClassInf.TYPE_BOOL(_),_)    => Types.T_BOOL

  axiom mktype(st,l) => Types.T_COMPLEX(st,l)

end
```

Relation `inst_list`

This is a utility used to do instantiation of list of things, collecting the result in another list.

```
relation inst_list : (Env, Mod, Prefix, Connect.Sets, ClassInf.State,
                      (Env, Mod, Prefix, Connect.Sets, ClassInf.State,
                       'a) => ('b list, Env, Connect.Sets, ClassInf.State),
                      'a list)
          => ('b list, Env, Connect.Sets, ClassInf.State) =

  axiom inst_list(env,mod,pre,csets,ci_state,r,[]) => ([],env,csets,ci_state)

  rule  r(env,mod,pre,csets,ci_state,e) => (l,env',csets',ci_state') &
        inst_list(env',mod,pre,csets',ci_state',r,es)
          => (l',env'',csets'',ci_state'') &
        list_append(l,l') => l''
        --------------------------------------------
        inst_list(env,mod,pre,csets,ci_state,r,e::es)
          => (l'',env'',csets'',ci_state'')

end
```

Relation `component_face`

This relation determines whether a component reference refers to an inner or outer connector.

```
relation component_face : Exp.ComponentRef => Connect.Face =

  axiom component_face Exp.CREF_QUAL(_,_,Exp.CREF_IDENT(_,_)) => Connect.INNER

  axiom component_face Exp.CREF_IDENT(_,_) => Connect.OUTER

  rule  print "# While connecting " &
        Exp.print_component_ref c & print "\n" &
        print "# Only local or child connectors allowed.\n"
        ---------------------------------------------------
        component_face c => fail

end
```

Relation `inst_mod_equation`

This relation adds the equation in the declaration of a variable, if such an equation exists.

```
relation inst_mod_equation : (Exp.ComponentRef, Types.Type, Mod)
          => DAE.Element list =


  rule  inst_eq_equation(Exp.CREF(cr),
                         Static.PROP(ty1,false(*FIXME*)), e,prop2) => dae
        ----------------------------------------------
        inst_mod_equation(cr,ty1, mod as Mod.MOD(_,_,SOME((e,prop2))))
          => dae

  axiom inst_mod_equation(_,_,Mod.MOD(_,_,NONE)) => []
  axiom inst_mod_equation(_,_,Mod.NOMOD) => []

  rule  print "- inst_mod_equation failed\n"
        -------------------------------------
        inst_mod_equation (_,_,_) => fail

end
```

Relation `check_prot`

This relation is used to check that a protected element is not modified.

```
relation check_prot : (bool, Mod.Mod, Exp.ComponentRef) => () =

  axiom check_prot(false,_,cref)

  axiom check_prot(_,Mod.NOMOD,_)

  rule  print "# Trying to modify protected element " &
        Exp.print_component_ref cref & print "\n"
        -----------------------------------------
        check_prot(true,_,cref) => fail

end
```

Relation **make_binding**

This relation looks at the equation part of a modification, and if there is a declaration equation builds a `Types.Binding` for it.

```
relation make_binding : (Env.Env, SCode.Attributes, Mod.EqMod option, SCode.Class)
          => Types.Binding =

  axiom make_binding (_,_,NONE,_) => Types.UNBOUND

        (* Constant binding *)
  axiom make_binding (env, SCode.ATTR(_,_,_,Absyn.CONST,_), SOME((e,prop)), cl)
          => Types.EQBOUND(e,true)

        (* default *)

  axiom make_binding (_,_, SOME((e,Static.PROP(t,c))),_) => Types.EQBOUND(e,c)

  rule  print "- make_binding failed\n"
        -------------------------------
        make_binding(_,_,_,_) => fail

end
```

A.9 Module **Prefix**

When instantiating an expression, there is a prefix that has to be added to each variable name to be able to use it in the flattened equation set. These relations take care of adding the prefixes.

```
module Prefix:

  with "absyn.rml"
  with "exp.rml"
  with "env.rml"
  with "lookup.rml"

  datatype Prefix = NOPRE
                  | PRE of string * int list * Prefix

  relation prefix_add : (Exp.Ident, int list, Prefix) => Prefix
  relation prefix_cref : (Prefix, Exp.ComponentRef) => Exp.ComponentRef
  relation prefix_exp  : (Env.Env,Exp.Exp,Prefix) => Exp.Exp
  relation prefix_path : (Absyn.Path,Prefix) => Absyn.Path

end
```

Relation **prefix_add**

This relation is used to extend a prefix with another level. If the prefix `a.b[10].c` is extended with `d` and an empty subscript list, the resulting prefix is `a.b[10].c.d`. Remember that prefixes components are stored in the opposite order from the normal order used when displaying them.

```
relation prefix_add : (Exp.Ident, int list, Prefix) => Prefix =

  axiom prefix_add(i,s,p) => PRE(i,s,p)

end
```

Relation **prefix_path**

Prefix a `Path` variable by adding the supplied prefix to it and returning a new `Path`.

```
relation prefix_path: (Absyn.Path,Prefix) => Absyn.Path =

  axiom prefix_path(p,NOPRE) => p

  rule  prefix_path(Absyn.QUALIFIED(s,p),ss) => p'
        ---------------------------
        prefix_path(p,PRE(s,_,ss)) => p'

end
```

Relation **prefix_cref**

Prefix a `ComponentRef` variable by adding the supplied prefix to it and returning a new `ComponentRef`.

```
relation prefix_cref : (Prefix, Exp.ComponentRef) => Exp.ComponentRef =

  axiom prefix_cref (NOPRE, cref) => cref

  rule  Exp.int_subscripts s => s' &
        prefix_cref(xs, Exp.CREF_QUAL(i,s',cref)) => cref'
        ---------------------------------
        prefix_cref (PRE(i,s,xs), cref) => cref'

end
```

Relation `prefix_exp`

Add the supplied prefix to all component references in an expression.

```
relation prefix_exp: (Env.Env,Exp.Exp,Prefix) => Exp.Exp =

  axiom prefix_exp(_,e as Exp.ICONST(_),_) => e
  axiom prefix_exp(_,e as Exp.RCONST(_),_) => e
  axiom prefix_exp(_,e as Exp.SCONST(_),_) => e
  axiom prefix_exp(_,e as Exp.BCONST(_),_) => e

  rule  Lookup.lookup_var_local(env,p) => (_,_,_) &
        prefix_cref(pre,p) => p'
        --------------------------
        prefix_exp(env,Exp.CREF(p),pre) => Exp.CREF(p')

  rule  not Lookup.lookup_var_local(env,p) => (_,_,_)
        --------------------------
        prefix_exp(env,e as Exp.CREF(p),pre) => e

  rule  prefix_exp(env,e1,p) => e1' &
        prefix_exp(env,e2,p) => e2'
        -----------------------
        prefix_exp(env,Exp.BINARY(e1,o,e2),p) => Exp.BINARY(e1',o,e2')

  rule  prefix_exp(env,e1,p) => e1'
        -----------------------
        prefix_exp(env,Exp.UNARY(o,e1),p) => Exp.UNARY(o,e1')

  rule  prefix_exp(env,e1,p) => e1' &
        prefix_exp(env,e2,p) => e2'
        -----------------------
        prefix_exp(env,Exp.LBINARY(e1,o,e2),p) => Exp.LBINARY(e1',o,e2')

  rule  prefix_exp(env,e1,p) => e1'
        -----------------------
        prefix_exp(env,Exp.LUNARY(o,e1),p) => Exp.LUNARY(o,e1')

  rule  prefix_exp(env,e1,p) => e1' &
        prefix_exp(env,e2,p) => e2'
        -----------------------
        prefix_exp(env,Exp.RELATION(e1,o,e2),p) => Exp.RELATION(e1',o,e2')

  rule  prefix_exp(env,e1,p) => e1' &
        prefix_exp(env,e2,p) => e2' &
        prefix_exp(env,e3,p) => e3'
        -----------------------
        prefix_exp(env,Exp.IFEXP(e1,e2,e3),p) => Exp.IFEXP(e1',e2',e3')

  rule  prefix_explist(env,es,p) => es'
        -----------------------
        prefix_exp(env,Exp.CALL(f,es),p) => Exp.CALL(f,es')
```

```
  axiom prefix_exp(env,Exp.ARRAY([]),p) => Exp.ARRAY([])

  rule  prefix_explist(env,es,p) => es'
        -------------------------------
        prefix_exp(env,Exp.ARRAY(es),p) => Exp.ARRAY(es')

  axiom prefix_exp(env,Exp.MATRIX([]),p) => Exp.MATRIX([])

  rule  prefix_explist(env,x,p) => x' &
        prefix_exp(env,Exp.MATRIX(xs),p) => Exp.MATRIX(xs')
        ----------------------------------------
        prefix_exp(env,Exp.MATRIX(x::xs),p) => Exp.MATRIX((x'::xs'))

  rule  prefix_exp(env,start, p) => start' &
        prefix_exp(env,stop, p) => stop'
        ------------------------------
        prefix_exp(env,Exp.RANGE(start,NONE,stop), p)
          => Exp.RANGE(start',NONE,stop')

  rule  prefix_exp(env,start, p) => start' &
        prefix_exp(env,step, p) => step' &
        prefix_exp(env,stop, p) => stop'
        ------------------------------
        prefix_exp(env,Exp.RANGE(start,SOME(step),stop), p)
          => Exp.RANGE(start',SOME(step'),stop')

  rule  prefix_exp(env,e,p) => e'
        -----------------------
        prefix_exp(env,Exp.CAST(Exp.REAL,e), p) => Exp.CAST(Exp.REAL,e')

  rule  print "- prefix_exp failed\n" &
        print "  expression: " & Exp.print_exp e & print "\n"
        -----------------------------------------------------
        prefix_exp(_,e,_) => fail

end
```

Relation `prefix_explist`

This relation prefixes a list of expressions using the `prefix_exp` relation.

```
relation prefix_explist: (Env.Env,Exp.Exp list,Prefix) => Exp.Exp list =

  axiom prefix_explist(_,[],_) => []

  rule  prefix_exp(env,e,p) => e' &
        prefix_explist(env,es,p) => es'
        ---------------------------
        prefix_explist(env,e::es,p) => (e'::es')

end
```

A.10 Module **Mod**

Modifications are simply the same kind of modifications used in the `Absyn` module.

```
module Mod:

  with "absyn.rml"
  with "dae.rml"
  with "env.rml"
  with "exp.rml"
  with "prefix.rml"
  with "explode.rml"
  with "types.rml"
  with "staticexp.rml"
  with "values.rml"

  type Ident = string
```

A.10.1 Modification datatype

```
  type EqMod = (Exp.Exp * Static.Properties)
  datatype SubMod = NAMEMOD of Ident * Mod
                  | IDXMOD of int list * Mod
  and Mod = MOD of bool * (SubMod list) * EqMod option
          | REDECL of bool * Absyn.ElementSpec
          | NOMOD
```

This type is very similar to `SCode.Mod`. The main difference is that it uses `Exp.Exp` for the expressions. Expressions stored here are prefixed and type-checked.

```
  relation elab_mod : (Env.Env, Prefix.Prefix, SCode.Mod) => Mod

  relation lookup_comp_modification: (Mod, Absyn.Ident) => Mod
  relation lookup_idx_modification: (Mod, int) => Mod
  relation lookup_modification_p: (Mod,Absyn.Path) => Mod
  relation merge : (Mod, Mod) => Mod
  relation mod_equation : Mod => EqMod option

  relation print_mod : Mod => ()

end
```

Relation **elab_mod**

This relation elaborates on the expressions in a modification and turns them into global expressions. This is done because the expressions in modifications must be elaborated on in the context they are provided in, and not the context they are used in.

```
relation elab_mod : (Env.Env, Prefix.Prefix, SCode.Mod) => Mod =

  axiom elab_mod (_,_,SCode.NOMOD) => NOMOD

  rule  elab_submods (env,pre,subs) => subs'
        ------------------------------------
        elab_mod (env,pre,m as SCode.MOD(final,subs,NONE))
          => MOD(final,subs',NONE)

  rule  elab_submods (env,pre,subs) => subs' &
        Static.elab_exp(env,e) => (e',prop) &
        Prefix.prefix_exp(env,e',pre) => e''
        ------------------------------------
        elab_mod (env,pre,m as SCode.MOD(final,subs,SOME(e)))
          => MOD(final,subs',SOME((e'',prop)))

end
```

Relation `elab_submods`

This relation helps `elab_mod` by recusively elaborating on a list of submodifications.

```
relation elab_submods : (Env.Env, Prefix.Prefix, SCode.SubMod list)
          => SubMod list =

  axiom elab_submods (_,_,[]) => []

  rule  elab_submod (env,pre,x) => x' &
        elab_submods (env,pre,xs) => xs' &
        insert_submods (x',xs') => res
        ------------------------------
        elab_submods (env,pre,x::xs) => res

end
```

Relation `elab_submod`

This relation elaborates on a submodification, turning an `SCode.SubMod` into one or more `SubMod`s.

```
relation elab_submod : (Env.Env, Prefix.Prefix, SCode.SubMod) => SubMod list =

  rule  elab_mod (env,pre,m) => m'
        --------------------------
        elab_submod (env,pre,SCode.NAMEMOD(i,m)) => [NAMEMOD(i,m')]

  rule  Static.elab_subscripts (env,ss) => (ss',true) &
        elab_mod (env,pre,m) => m' &
        make_idxmods (ss',m') => smods
        ---------------------------
        elab_submod (env,pre,SCode.IDXMOD(ss,m)) => smods

end
```

Relation `make_idxmods`

From a list of list of integers, this relation creates a list of sub-modifications of the `IDXMOD` variety.

```
relation make_idxmods : (Exp.Subscript list, Mod) => SubMod list =

  axiom make_idxmods([Exp.INDEX(Exp.ICONST(x))],m) => [IDXMOD([x],m)]

  rule  make_idxmods(xs, m) => mods &
        prefix_idxmods(mods, x) => mods'
        --------------------------------
        make_idxmods(Exp.INDEX(Exp.ICONST(x))::xs, m) => mods'

  rule  print "= expand_slice\n" &
        expand_slice(x, xs, 1, m) => mods
        ---------------------------
        make_idxmods(Exp.SLICE(Exp.ARRAY(x))::xs, m) => mods

  rule  print "# Sorry, [:] slices are not handled in modifications\n"
        ------------------------------------------------------------
        make_idxmods(Exp.WHOLEDIM::xs, m) => fail

end
```

Relation `prefix_idxmods`

This relation adds a subscript to each `IDXMOD` in a list of submodifications.

```
relation prefix_idxmods : (SubMod list, int) => SubMod list =

  axiom prefix_idxmods([],_) => []

  rule  prefix_idxmods(mods, i) => mods'
        --------------------------------
        prefix_idxmods(IDXMOD(l,m)::mods, i) => IDXMOD(i::l,m)::mods'

end
```

Relation `expand_slice`

This relation goes through an array slice modification and creates an singly indexed modification for each index in the slice. For example, `x[2:3] = y` is changed into `x[2] = y[1]` and `x[3] = y[2]`.

```
relation expand_slice : (Exp.Exp list, Exp.Subscript list, int, Mod)
          => SubMod list =

  axiom expand_slice([], _, _, _) => []

  axiom expand_slice([], _, _, _) => []

  rule  Exp.simplify(Exp.ASUB(e,n)) => e' &
        Types.unlift_array(t) => t' &
        make_idxmods(Exp.INDEX(x)::ss,
                     MOD(final, [],
                         SOME((e',Static.PROP(t',const))))) => mods1 &
        int_add(n, 1) => n' &
        expand_slice(xs, ss, n', m) => mods2 &
        list_append(mods1, mods2) => mods
        ---------------------------------
        expand_slice(x::xs, ss, n,
                     m as MOD(final, [],
                              SOME((e,Static.PROP(t,const))))) => mods

  rule  print "# Illegal modification (" & print_mod mod & print ")\n"
        --------------------------------
        expand_slice(_,_,_,mod) => fail

end
```

Relation **`expand_list`**

This utility relation takes a list of integer values and a list of list of integers, and for each integer in the first and each list in the second list creates a list with that integer as head and the second list as tail. All resulting lists are collected in a list and returned.

```
relation expand_list : (Values.Value list, int list list) => int list list =

  axiom expand_list([],_) => []

  axiom expand_list(_,[]) => []

  rule  expand_list(xx,ys) => l1 &
        expand_list(xs,yy) => l2 &
        list_append(l1,l2) => l
        -----------------------
        expand_list(xx as (Values.INTEGER(x)::xs),yy as (y::ys))
                    => ((x::y)::l)

end
```

Relation **`insert_submods`**

This relation repeatedly calls `insert_submod` to incrementally insert several sub-modifications.

```
relation insert_submods : (SubMod list, SubMod list) => SubMod list =

  axiom insert_submods ([],_) => []

  rule  insert_submod (x,y) => x' &
        insert_submods (xs,y) => xs' &
        list_append(x', xs') => l
        -------------------------
        insert_submods (x::xs,y) => l

end
```

Relation **`insert_submod`**

This relation inserts a `SubMod` into a list of unique `SubMods`, while keeping the uniqueness, merging the submod if necessary.

```
relation insert_submod : (SubMod, SubMod list) => SubMod list =

  axiom insert_submod (sub,[]) => [sub]

  rule  n1 = n2 &
        merge (m1,m2) => m
        ----------------------
        insert_submod (NAMEMOD(n1,m1),NAMEMOD(n2,m2)::tail)
          => (NAMEMOD(n1,m)::tail)

  rule  i1 = i2 &
        merge (m1,m2) => m
        ----------------------
        insert_submod (IDXMOD(i1,m1),IDXMOD(i2,m2)::tail)
          => (IDXMOD(i1,m)::tail)

  axiom insert_submod (sub1, sub2) => sub1::sub2

end
```

A.10.2 Lookup

Relation `lookup_modification_p`

This relation extracts a modification from inside another modification, using a name to look up submodifications.

```
relation lookup_modification_p: (Mod,Absyn.Path) => Mod =

  rule  lookup_comp_modification (m,n) => mod
        --------------------------------
        lookup_modification_p (m,Absyn.IDENT(n)) => mod

  rule  lookup_comp_modification (m,n) => mod &
        lookup_modification_p (mod,p) => mod'
        -------------------------------------
        lookup_modification_p (m,Absyn.QUALIFIED(n,p)) => mod'

end
```

Relation `lookup_comp_modification`

This relation is used to look up an identifier in a modification.

```
relation lookup_comp_modification: (Mod, Absyn.Ident) => Mod =

  axiom lookup_comp_modification(NOMOD,_) => NOMOD

  axiom lookup_comp_modification(REDECL(_,_),_) => NOMOD

  rule  lookup_comp_modification2(subs,n) => mod
        -----------------------------------
        lookup_comp_modification(MOD(_,subs,_),n) => mod

  rule  print "# Mod.lookup_comp_modification(" &
        print_mod mod & print "," & print i &
        print ") failed\n"
        --------------------
        lookup_comp_modification(mod,i) => fail

end
```

Relation `lookup_comp_modification2`

This relation is just a helper to `lookup_comp_modification`.

```
relation lookup_comp_modification2 : (SubMod list,Absyn.Ident) => Mod =

  axiom lookup_comp_modification2([],_) => NOMOD

  rule  n = m
        --------------------------------------
        lookup_comp_modification2(NAMEMOD(n,mod)::_,m) => mod

  rule  lookup_comp_modification2(xs,n) => mod
        ---------------------------------
        lookup_comp_modification2(x::xs,n) => mod

  rule  print "- lookup_comp_modification2 failed\n"
        ---------------------------------------
        lookup_comp_modification2 (_,_) => fail

end
```

Relation `lookup_idx_modification`

This relation extracts modifications to an array element, using an integer to index the modification.

```
relation lookup_idx_modification: (Mod, int) => Mod =

  axiom lookup_idx_modification(NOMOD,_) => NOMOD

  axiom lookup_idx_modification(REDECL(_,_),_) => NOMOD

  rule  lookup_idx_modification2(subs,NONE,idx) => (mod',subs') &
        merge (MOD(f,subs',NONE),mod') => mod'' &

        index_eqmod (eq,[idx]) => eq' &
        merge (mod'',MOD(false,[],eq')) => mod'''
        -----------------------------------
        lookup_idx_modification(MOD(f,subs,eq),idx) => mod'''

  rule  print "# Mod.lookup_idx_modification(" &
        print_mod mod & print "," & int_string idx => s & print s &
        print ") failed\n"
        --------------------
        lookup_idx_modification(mod,idx) => fail

end
```

Relation `lookup_idx_modification2`

This relation does part of the job for `lookup_idx_modification`.

```
relation lookup_idx_modification2 : (SubMod list, EqMod option, int)
          => (Mod, SubMod list) =

  axiom lookup_idx_modification2([],_,_) => (NOMOD, [])

        (* FIXME: Redeclaration *)

  rule  x = y &
        lookup_idx_modification2(subs,eq,y) => (NOMOD,subs')
        --------------------------------------
        lookup_idx_modification2(IDXMOD([x],mod)::subs,eq,y)
          => (mod,subs')

  rule  x = y &
        lookup_idx_modification2(subs,eq,y) => (mod',subs')
        ---------------------------------
        lookup_idx_modification2(IDXMOD(x::xs,mod)::subs,eq,y)
          => (mod',IDXMOD(xs,mod)::subs')

  rule  not x = y &
        lookup_idx_modification2(subs,eq,y) => (mod',subs')
        ---------------------------------
        lookup_idx_modification2(IDXMOD(x::xs,mod)::subs,eq,y)
          => (mod',subs')

  rule  lookup_idx_modification2(xs,eq,idx) => (mod,xs')
        ---------------------------------
        lookup_idx_modification2(x::xs,eq,idx) => (mod,x::xs')

  rule  print "- lookup_idx_modification2 failed\n"
        ---------------------------------------
        lookup_idx_modification2 (_,_,_) => fail

end
```

Relation **index_eqmod**

If there is an equation modification, this relation can subscript it using the provided indexing expressions. This is used when a modification equates an array variable with an array expression. This expression will be expanded to produce one equation expression per array component.

```
relation index_eqmod : (EqMod option, int list) => EqMod option =

  axiom index_eqmod (NONE,_) => NONE

  axiom index_eqmod (e,[]) => e

  rule  Types.unlift_array t => t' &
        Exp.simplify Exp.ASUB(e,x) => exp &
        index_eqmod (SOME((exp,Static.PROP(t',c))),xs) => e
        ------------------------------------------
        index_eqmod (SOME((e,Static.PROP(t,c))),x::xs) => e

end
```

A.10.3 Merging

The merge relation merges to modifications to one. The first argument is the "outer" modification that should take precedence over the "inner" modifications.

Relation **merge**

This relation merges to modificiations into one. The first modifications takes precedence over the second.

```
relation merge : (Mod, Mod) => Mod =

  axiom merge(NOMOD,NOMOD) => NOMOD

  axiom merge(NOMOD,m) => m

  axiom merge(m,NOMOD) => m

  rule  print "- ignoring inner modification\n"
        --------------------------------------
        merge(m as REDECL(_,_),_) => m

  rule  print "# can't merge modification with a redeclaration\n"
        ---------------------------------------------------------
        merge(_,REDECL(_,_)) => fail

  rule  print "# trying to modify final element\n"
        ------------------------------------------
        merge(_,m as MOD(true,_,_)) => fail

  rule  merge_subs(subs1, subs2) => subs &
        merge_eq(ass1, ass2) => ass
        ----------------------------
        merge(MOD(final,subs1,ass1), MOD(false,subs2,ass2))
          => MOD(final,subs,ass)

  rule  print "# incomplete handling of modification merging\n"
        -----------------------------------------------------
        merge(outer,inner) => outer

end
```

Relation `merge_subs`

This relation merges to list of `SubMod`s.

```
relation merge_subs : (SubMod list, SubMod list)
          => SubMod list =

  axiom merge_subs(s1,[]) => s1

  rule  merge_subs2(s1,s) => (s1',s') &
        merge_subs(s1',s2) => ss
        ------------------------
        merge_subs(s1,s::s2) => s'::ss

end
```

Relation `merge_subs2`

This relation helps in the merging of two lists of `SubMod`s. It compares one `SubMod` against a list of other `SubMod`s, and if there is one with the same name, it is kept and the one `SubMod` given in the second argument is discarded.

```
relation merge_subs2 : (SubMod list, SubMod)
          => (SubMod list, SubMod) =

  axiom merge_subs2([],m) => ([],m)

        (* Modifications in the list take precedence *)

  rule  n1 = n2 & merge(m1,m2) => m
        ---------------------------
        merge_subs2(NAMEMOD(n1,m1)::ss, NAMEMOD(n2,m2))
          => (ss, NAMEMOD(n1,m))

  rule  i1 = i2 & merge(m1,m2) => m
        ---------------------------
        merge_subs2(IDXMOD(i1,m1)::ss, IDXMOD(i2,m2)) => (ss, IDXMOD(i1,m))

  rule  merge_subs2(ss, s2) => (ss',s)
        ------------------------------
        merge_subs2(s1::ss, s2) => (s1::ss',s)

end
```

Relation `merge_eq`

The outer modification, given in the first argument, takes precedence over the inner modifications.

```
relation merge_eq : (EqMod option, EqMod option) => EqMod option =

        (* Outer assignments take precedence *)

  axiom merge_eq(e as SOME((_,_)), _) => e

  axiom merge_eq(NONE,e) => e

end
```

Relation `mod_equation`

This relation simply extracts the equation part of a modification.

```
relation mod_equation : Mod => EqMod option =

  axiom mod_equation NOMOD => NONE
  axiom mod_equation REDECL(_,_) => NONE
  axiom mod_equation MOD(_,_,e) => e

end
```

A.10.4 Printing

Relation `print_mod`

This relation prints a modification. It uses a few other relation to do its stuff.
The relations are excluded from the report for brevity.

A.11 Module `Algorithm`

This file contains data types and relations for managing algorithms. The algorithms in the AST is analyzed by the `Inst` module which uses this module to represent the algorithms.

```
module Algorithm:

  with "exp.rml"
  with "types.rml"
  with "staticexp.rml"
  with "explode.rml"

  type Ident = string

  datatype Algorithm = ALGORITHM of Statement list
```

The `Algorithm` type corresponds to a whole algorithm secion. It is simple a list of algorithm statements.

```
  datatype Statement = ASSIGN of Exp.ComponentRef * Exp.Exp
                     | IF of Exp.Exp * Statement list * Else
                     | FOR of Ident * Exp.Exp * Statement list
                     | WHILE of Exp.Exp * Statement list
                     | WHEN of Exp.Exp * Statement list
```

There are four kinds of statements. Assignments (`a := b;`), if statements (`if A then B; elseif C; else D;`), for loops (`for i in 1:10 loop ...; end for;`) and when statements (`when E do S; end when;`).

```
  datatype Else = NOELSE
                | ELSEIF of Exp.Exp * Statement list * Else
                | ELSE of Statement list
```

An if statements can one or more `elseif` branches and an optional `else` branch.

```
  relation make_assignment : (Exp.Exp, Static.Properties,
                              Exp.Exp, Static.Properties,
                              SCode.Accessibility) => Statement

  relation make_if : (Exp.Exp,
                      Static.Properties,
                      Statement list,
                      (Exp.Exp * Static.Properties * Statement list) list,
                      Statement list) => Statement

  relation make_for : (Ident, Exp.Exp, Static.Properties,
                       Statement list) => Statement

  relation make_while : (Exp.Exp, Static.Properties,
                         Statement list) => Statement

  relation make_when : (Exp.Exp, Static.Properties,
                        Statement list) => Statement

end
```

Relation `make_assignment`

This relation creates an `ASSIGN` construct, and checks that the assignment is semantically valid, which means that the component being assigned is not constant, and that the types match.

```
relation make_assignment : (Exp.Exp, Static.Properties,
                            Exp.Exp, Static.Properties,
                            SCode.Accessibility) => Statement =

  rule  print "# Trying to assign to constant component\n" &
        print "    " & Exp.print_exp lhs &
        print " := " & Exp.print_exp rhs & print "\n"
        ------------------------------------------
        make_assignment (lhs,Static.PROP(lt,true),
                         rhs,Static.PROP(rt,_),_) => fail

  rule  print "# Can't assign to " & Exp.print_exp e & print "\n"
        ---------------------------------------------------------
        make_assignment(e,_,_,_, SCode.RO) => fail

  rule  Static.match_prop(rhs, rhprop, lhprop) => rhs'
        -------------------------------------
        make_assignment(Exp.CREF(c), lhprop as Static.PROP(_,false),
                        rhs, rhprop, _) => ASSIGN(c,rhs')

  rule  Types.equivtypes (lt,rt) => false &
        print "# Type mismatch in assignment\n" &
        print "    " & Exp.print_exp lhs &
        print " := " & Exp.print_exp rhs & print "\n" &
        print "    " & Types.print_type lt &
        print " := " & Types.print_type rt & print "\n"
        ------------------------------------------
        make_assignment (lhs,Static.PROP(lt,_),rhs,Static.PROP(rt,_),_) => fail

  rule  print "- Algorithm.make_assignment failed\n" &
        print "    " & Exp.print_exp lhs &
        print " := " & Exp.print_exp rhs & print "\n"
        --------------------------------------------
        make_assignment (lhs,_,rhs,_,_) => fail

end
```

Relation `make_if`

This relation creates an `IF` construct, checking that the types of the parts are correct.

```
relation make_if : (Exp.Exp,
                    Static.Properties,
                    Statement list,
                    (Exp.Exp * Static.Properties * Statement list) list,
                    Statement list) => Statement =

  rule  make_else(eib,fb) => else
        -------------------------
        make_if(e,Static.PROP(Types.T_BOOL,_),tb,eib,fb) => IF(e,tb,else)

  rule  print "# Type error in if conditional (" &
        Exp.print_exp e & print ")\n" &
        print "    Expected Boolean, got " &
        Types.print_type t & print "\n"
        ------------------------------
        make_if(e,Static.PROP(t,_),_,_,_) => fail

end
```

Relation `make_else`

```
relation make_else : ((Exp.Exp * Static.Properties * Statement list) list,
                      Statement list) => Else =
```

This removes empty else branches

```
  axiom make_else ([],[]) => NOELSE

  axiom make_else ([],fb) => ELSE(fb)

  rule  make_else (xs,fb) => else
        -------------------------
        make_else ((e,Static.PROP(Types.T_BOOL,_),b)::xs,fb)
          => ELSEIF(e,b,else)

  rule  print "# Type error in elseif conditional (" &
        Exp.print_exp e & print ")\n" &
        print "    Expected Boolean, got " &
        Types.print_type t & print "\n"
        -------------------------------
        make_else((e,Static.PROP(t,_),_)::_,_) => fail

end
```

Relation `make_for`

```
relation make_for : (Ident, Exp.Exp, Static.Properties,
                     Statement list) => Statement =

  axiom make_for(i,e,Static.PROP(Types.T_ARRAY(_,_),_),stmts) => FOR(i,e,stmts)

  rule  print "# Type error in for conditional.\n" &
        print "    Expected array, got " &
        Types.print_type t & print "\n"
        -------------------------------
        make_for(_,_,Static.PROP(t,_),_) => fail

end
```

Relation `make_while`

```
relation make_while : (Exp.Exp, Static.Properties,
                       Statement list) => Statement =

  axiom make_while(e,Static.PROP(Types.T_BOOL,_),stmts) => WHILE(e,stmts)

  rule  print "# Type error in while conditional.\n" &
        print "    Expected Boolean, got " &
        Types.print_type t & print "\n"
        -------------------------------
        make_while(_,Static.PROP(t,_),_) => fail

end
```

Relation `make_when`

```
relation make_when : (Exp.Exp, Static.Properties,
                      Statement list) => Statement =

  axiom make_when(e,Static.PROP(Types.T_BOOL,_),stmts) => WHEN(e,stmts)

  rule  print "# Type error in when conditional.\n" &
        print "    Expected Boolean, got " &
        Types.print_type t & print "\n"
        -------------------------------
        make_when(_,Static.PROP(t,_),_) => fail

end
```

A.12 Module **Env**

An environment is a stack of frames, where each frame contains a number of class and variable bindings.

```
module Env:

  with "absyn.rml"
  with "values.rml"
  with "explode.rml"
  with "types.rml"
  with "classinf.rml"
  with "exp.rml"

  type Ident = string

  datatype Frame = FRAME of (Ident * Item) list

  datatype Item = VAR of Types.Var
                | CLASS of SCode.Class * Env
                | TYPE of Types.Type

  type Env = Frame list

  val empty_frame : Frame
  val empty_env   : Env

  relation open_scope       : Env => Env
  relation extend_frame_c   : (Env, SCode.Class) => Env
  relation extend_frame_v   : (Env, Types.Var) => Env
  relation extend_frame_t : (Env,Ident,Types.Type) => Env

end
```

A.12.1 Values

```
val empty_frame = FRAME([])
val empty_env = []
```

A.12.2 Relations

Relation **open_scope**

Opening a new scope in the environment mans adding a new frame on top of the stack of frames.

```
relation open_scope: Env => Env =

  axiom open_scope(env) => empty_frame::env

end
```

Relation **extend_frame_c**

This relation adds a class definition to the environment.

```
relation extend_frame_c : (Env, SCode.Class) => Env =

  axiom extend_frame_c(env as (FRAME(items)::fs),c as SCode.CLASS(n,_,_,_))
        => ((FRAME((n,CLASS(c,env))::items)::fs))

end
```

Relation **extend_frame_v**

This relation adds a component to the environment.

```
relation extend_frame_v : (Env,Types.Var) => Env =

  axiom extend_frame_v(FRAME(items)::fs,v as Types.VAR(n,_,_,_,_))
          => (FRAME((n,VAR(v))::items)::fs)

end
```

Relation **extend_frame_t**

This relation adds a type to the environment. Types in the environment are used for looking up constants etc. inside class definitions, such as packages. For each type in the environment, there is a class definition with the same name in the environment.

```
relation extend_frame_t : (Env,Ident,Types.Type) => Env =

  axiom extend_frame_t(FRAME(items)::fs, n, t)
          => (FRAME((n,TYPE(t))::items)::fs)

end
```

A.13 Module `Lookup`

```
module Lookup:

  with "classinf.rml"
  with "types.rml"
  with "absyn.rml"
  with "exp.rml"
  with "env.rml"
  with "explode.rml"

  relation lookup_type       : (Env.Env, Absyn.Path) => (Types.Type)
  relation lookup_class      : (Env.Env, Absyn.Path) => (SCode.Class, Env.Env)
  relation lookup_var        : (Env.Env,Exp.ComponentRef)
          => (Types.Attributes,Types.Type,Types.Binding)
  relation lookup_var_local  : (Env.Env,Exp.ComponentRef)
          => (Types.Attributes, Types.Type, Types.Binding)
  relation lookup_ident_local : (Env.Env, SCode.Ident) => Types.Var

end

with "parse.rml"
```

A.13.1 Lookup relations

These relations look up class and variable names in the environment. The names are supplied as a path, and if the path is qualified, a variable named as the first part of the path is searched for, and the name is looked for in it.

Relation `lookup_type`

```
relation lookup_type: (Env.Env,Absyn.Path) => Types.Type =

  rule  lookup_type_in_env(env,path) => t
        --------------------------------------
        lookup_type(env, path) => t

  rule  lookup_package_type (env,pack,path) => t
        --------------------------------------
        lookup_type(env, Absyn.QUALIFIED(pack,path)) => t

end
```

Relation `lookup_class`

```
relation lookup_class: (Env.Env,Absyn.Path) => (SCode.Class, Env.Env) =

  rule  lookup_class_in_env(env,path) => (c, env)
        --------------------------------------
        lookup_class(env, path) => (c, env)

  rule  lookup_package_class (env,pack,path) => (c, env)
        --------------------------------------
        lookup_class(env, Absyn.QUALIFIED(pack,path)) => (c, env)

end
```

Relation `lookup_var`

```
relation lookup_var : (Env.Env,Exp.ComponentRef)
          => (Types.Attributes, Types.Type, Types.Binding) =

  rule  (* print " lookup_var: looking in frame\n" & *)
        lookup_var_f(items,ref) => (attr,ty,binding)
        --------------------------------------------
        lookup_var(Env.FRAME(items)::fs,ref) => (attr,ty,binding)

  rule  (* print " lookup_var: next frame\n" & *)
        lookup_var(fs,ref) => (attr,ty,binding)
        ---------------------------------------
        lookup_var(_::fs,ref) => (attr,ty,binding)

end
```

Relation **`lookup_var_local`**

This relation is very similar to `lookup_var`, but it only looks in the topmost environment frame, which means that it only finds names defined in the local scope.

```
relation lookup_var_local : (Env.Env,Exp.ComponentRef)
          => (Types.Attributes, Types.Type, Types.Binding) =

  rule  lookup_var_f(items,ref) => (attr,ty,binding)
        --------------------------------------------
        lookup_var_local(Env.FRAME(items)::fs,ref) => (attr,ty,binding)

end

relation lookup_ident_local : (Env.Env, SCode.Ident) => Types.Var =

  rule  lookup_var2(items, id) => fv
        ----------------------------
        lookup_ident_local(Env.FRAME(items)::_, id) => fv

end
```

A.13.2 Internal relations

Type lookup

```
relation lookup_type_in_env: (Env.Env,Absyn.Path) => (Types.Type) =

  rule  lookup_type_in_frame(items,id) => c
        ---------------------
        lookup_type_in_env(Env.FRAME(items)::fs,Absyn.IDENT(id)) => c

  rule  lookup_type_in_env(fs,id) => c
        ---------------------
        lookup_type_in_env(f::fs,id) => c

end
```

```
relation lookup_type_in_frame: ((SCode.Ident * Env.Item) list,
                                SCode.Ident)
          => Types.Type =

  rule  (* print "  lookup_type_in_frame(" & print id & print "): " &
           print n & print "\n" & *)
        id = n
        ---------------------------------------------------------
        lookup_type_in_frame((n,Env.TYPE(t))::_, id) => t

  rule  id = n & print "# Error while looking up type " &
        print id & print ": found a component\n"
        ------------------------------------
        lookup_type_in_frame((n,Env.VAR(_))::_,id) => fail

  rule  (* print "  lookup_type_in_frame(" & print id & print "): not " &
           print n & print "\n" & *)
        lookup_type_in_frame(fs,id) => t
        ------------------------------
        lookup_type_in_frame((n,_)::fs,id) => t

end
```

```
relation lookup_package_type : (Env.Env, string, Absyn.Path) => Types.Type =

  rule  locate_package (env,packname) => pack &
        print "- package lookup of types not implemented\n"
        (* lookup_type_in_package (pack, path) => t *)
        -------------------------------------
        lookup_package_type (env, packname, path) => (* t *) fail

end
```

Class lookup

```
relation lookup_class_in_env: (Env.Env,Absyn.Path) => (SCode.Class, Env.Env) =

  rule  lookup_class_in_frame(items,id) => (c, env)
        ---------------------
        lookup_class_in_env(Env.FRAME(items)::fs,Absyn.IDENT(id)) => (c, env)

  rule  lookup_class_in_env(fs,id) => (c, env)
        ---------------------
        lookup_class_in_env(f::fs,id) => (c, env)

end
```

```
relation lookup_class_in_frame: ((SCode.Ident * Env.Item) list,
                                 SCode.Ident)
          => (SCode.Class, Env.Env) =

  rule  (* print "  lookup_class_in_frame(" & print id & print "): " &
           print n & print "\n" & *)
        id = n
        --------------------------------------------------------
        lookup_class_in_frame((n,Env.CLASS(c, env))::_, id) => (c, env)

(*
  rule  id = n & print "# Error while looking up class " &
        print id & print ": found a component\n"
        ------------------------------------
        lookup_class_in_frame((n,Env.VAR(_))::_,id) => fail
*)

  rule  lookup_class_in_frame(fs,id) => (c, env)
        ------------------------------
        lookup_class_in_frame(_::fs,id) => (c, env)

end
```

```
relation lookup_package_class : (Env.Env, string, Absyn.Path)
          => (SCode.Class, Env.Env) =

  rule  locate_package (env,packname) => pack &
        lookup_class_in_package (pack, path) => (c, env)
        -------------------------------------
        lookup_package_class (env, packname, path) => (c, env)

end
```

```
relation lookup_var2: ((SCode.Ident * Env.Item) list,SCode.Ident)
          => Types.Var =

  rule  id = n
        --------------------------------------------
        lookup_var2((n,Env.VAR(fv))::_,id) => fv

  rule  lookup_var2(fs,id) => fv
        ------------------------
        lookup_var2(_::fs,id) => fv

end
```

Relation **`check_subscripts`**

This relation checks a list of subscripts agains type, and removes dimensions from the type according to the subscripting.

```
relation check_subscripts : (Types.Type, Exp.Subscript list)
          => Types.Type =

  axiom check_subscripts(t, []) => t

  rule  check_subscripts(t, ys) => t'
        ------------------------------
        check_subscripts(Types.T_ARRAY(dim,t),
                         Exp.WHOLEDIM::ys)
          => Types.T_ARRAY(dim,t')

  rule  check_subscripts(t, ys) => t' &
        list_length(se) => dim
        (* FIXME: Check range *)
        ------------------------------
        check_subscripts(Types.T_ARRAY(Types.DIM(SOME(sz)),t),
                         Exp.SLICE(Exp.ARRAY(se))::ys)
          => Types.T_ARRAY(Types.DIM(SOME(dim)),t')

  rule  int_gt(ind, 0) => true  &  int_le(ind, sz) => true &
        check_subscripts(t, ys) => t'
        ------------------------------
        check_subscripts(Types.T_ARRAY(Types.DIM(SOME(sz)),t),
                         Exp.INDEX(Exp.ICONST(ind))::ys) => t'

  rule  print "- check_subscripts failed (" &
        Types.print_type t & print ", [" &
        Exp.print_list(s,Exp.print_subscript,",") & print "])\n"
        -----------------------------------
        check_subscripts(t,s) => fail

end
```

```
relation lookup_in_var: (Types.Type, Exp.ComponentRef)
          => (Types.Attributes,Types.Type,Types.Binding) =

        (* Public components *)
  rule  Types.lookup_component(ty, id)
          => (Types.VAR(_,Types.ATTR(fl,acc,vt,di),false,ty',binding)) &
        check_subscripts(ty', ss) => ty''
        -------------------------------------------
        lookup_in_var(ty, Exp.CREF_IDENT(id,ss))
          => (Types.ATTR(fl,acc,vt,di),ty'',binding)

        (* Protected components *)
  rule  Types.lookup_component(ty, id)
          => (Types.VAR(_,_,true,_,_))
        ------------------------------
        lookup_in_var(ty, Exp.CREF_IDENT(id,ss))
          => fail

  rule  Types.lookup_component(ty, id)
          => (Types.VAR(_,Types.ATTR(fl,acc,vt,di),_,ty',_)) &
        check_subscripts(ty', ss) => ty'' &
        lookup_in_var(ty'', vs) => (attr, ty''',binding)
        --------------------------------------
        lookup_in_var(ty, Exp.CREF_QUAL(id,ss,vs)) => (attr,ty''',binding)

  rule  print "- lookup_in_var failed\n"
        --------------------------------
        lookup_in_var(_,_) => fail

end
```

Relation `lookup_var_f`

This relation looks in a frame to find a declared variable. If the name being looked up is qualified, the first part of the name is looked up, and `lookup_in_var` is used to for further lookup in the result of that lookup.

```
relation lookup_var_f : ((SCode.Ident * Env.Item) list,
                         Exp.ComponentRef)
          => (Types.Attributes, Types.Type, Types.Binding) =

  rule  lookup_var2(f, id)
          => Types.VAR(n, Types.ATTR(f,acc,vt,di),_,ty,bind) &
        check_subscripts(ty,ss) => ty'
        ----------------------------------
        lookup_var_f(f,Exp.CREF_IDENT(id,ss))
          => (Types.ATTR(f,acc,vt,di),ty',bind)

  rule  lookup_var2(f, id) => Types.VAR(n,Types.ATTR(f,acc,vt,di),_,ty,_) &
        check_subscripts(ty,ss) => ty' & (* FIXME: Check for non-array *)
        lookup_in_var(ty', ids) => (attr,ty'',binding)
        ----------------------------------
        lookup_var_f(f,Exp.CREF_QUAL(id,ss,ids)) => (attr,ty'',binding)

end
```

Package handling

Relation `locate_package`

This relation finds a package in a file and loads it.

It appends `.mo` to the name of the package and tries to load a file of that name. This relation should be memoized, as it currently reloads the package each time it is used.

```
relation locate_package : (Env.Env, string) => SCode.Class =

  rule  lookup_class(env, Absyn.IDENT(pack)) => (c, env) &
        assert_package c
        ----------------
        locate_package (env, pack) => c

  rule  string_append (pack, ".mo") => filename &
        print "Loading " & print filename & print "...\n" &
        Parser.parse filename => ((c as Absyn.CLASS(classname,_,_,_))::_) &
        pack = classname &
        SCode.elaborate [c] => [c'] &
        assert_package c'
        ---------------------------------------------
        locate_package(env, pack) => c'

end
```

Relation `assert_package`

This relation checks that a class definition is a package. This breaks the everything-can-be-generalized-to-class principle, since it requires that the keyword `package` is used in the package file.

```
relation assert_package : SCode.Class => () =

        (* Break the generalize-to-class rule *)
  axiom assert_package SCode.CLASS(_,_,Absyn.R_PACKAGE,_)

end
```

Relation `lookup_class_in_package`

This relation searches a class definition for a named class definition.

The class is returned together with an empty environment, which might cause problems. This should be solved using implicit instantiation, but that is future work.

```
relation lookup_class_in_package : (SCode.Class, Absyn.Path)
          => (SCode.Class, Env.Env) =

  rule  lookup_class_in_elements (els, path) => c
        -----------------------------------------
        lookup_class_in_package (SCode.CLASS(_,_,_,
                                               SCode.PARTS(els,_,_)),
                                 path) => (c, Env.empty_env)

end
```

Relation `lookup_class_in_elements`

This relation helps `lookup_class_in_package` to do its thing.

```
relation lookup_class_in_elements : (SCode.Element list, Absyn.Path)
          => SCode.Class =

  rule  n1 = n2
        -------
        lookup_class_in_elements (SCode.CLASSDEF(n1,_,_,c)::_,
                                  Absyn.IDENT(n2))
          => c

  rule  n1 = n2 &
        assert_package c &
        lookup_class_in_package (c, path) => (c', env)
        ----------------------------
        lookup_class_in_elements (SCode.CLASSDEF(n1,_,_,c)::_,
                                  Absyn.QUALIFIED(n2, path))
          => c'

  rule  lookup_class_in_elements (es, path) => c
        -------------------------------------
        lookup_class_in_elements (_::es, path) => c

end
```

A.14 Module **Builtin**

This module defines the builtin types, variables and functions in Modelica. The only exported relation is `initial_env`.

Currently, the only attribute defined in the built-in types is `unit`. The others can easily be added.

```
module Builtin:

  with "absyn.rml"
  with "explode.rml"
  with "env.rml"

  val rl_type   : SCode.Class
  val int_type  : SCode.Class
  val str_type  : SCode.Class
  val bool_type : SCode.Class

  val real_type : SCode.Class

  relation initial_env : () => Env.Env

end
```

```
with "exp.rml"
with "types.rml"
with "classinf.rml"
```

A.14.1 The primitive types

```
val rl_type = SCode.CLASS("RealType",false,Absyn.R_PREDEFINED_REAL,
                          SCode.PARTS([],[],[]))
val int_type = SCode.CLASS("IntegerType",false,Absyn.R_PREDEFINED_INT,
                           SCode.PARTS([],[],[]))
val str_type = SCode.CLASS("StringType",false,Absyn.R_PREDEFINED_STRING,
                           SCode.PARTS([],[],[]))
val bool_type = SCode.CLASS("BooleanType",false,Absyn.R_PREDEFINED_BOOL,
                            SCode.PARTS([],[],[]))
```

These are the primitive types that are used to build the types `Real`, `Integer` etc.

```
val unit = SCode.COMPONENT("unit",true,false,
                              SCode.ATTR([],
                                         false,
                                         SCode.RW,
                                         Absyn.VAR,
                                         Absyn.BIDIR),
                              Absyn.IDENT("StringType"),
                              SCode.MOD(false,[],SOME(Absyn.STRING(""))))
```

This `unit` component is used in several places below, and it is declared once here to make the definitions below easier to read.

A.14.2 The **Real** type

```
val real_type =
SCode.CLASS("Real",false,Absyn.R_PREDEFINED_REAL,
            SCode.PARTS
            ([unit],
             [],
             []))
```

A.14.3 The `Integer` type

```
val integer_type =
SCode.CLASS("Integer",false,Absyn.R_PREDEFINED_INT,
            SCode.PARTS
            ([unit],
             [],
             []))
```

A.14.4 The `String` type

```
val string_type =
SCode.CLASS("String",false,Absyn.R_PREDEFINED_STRING,
            SCode.PARTS
            ([],
             [],
             []))
```

A.14.5 The `Boolean` type

```
val boolean_type =
SCode.CLASS("Boolean",false,Absyn.R_PREDEFINED_BOOL,
            SCode.PARTS
            ([],
             [],
             []))
```

A.14.6 The `time` variable

```
val time_var =
Types.VAR("time", Types.ATTR(false, SCode.RO, Absyn.VAR, Absyn.BIDIR),
          false, Types.T_REAL, Types.UNBOUND)
```

A.14.7 Some assorted function types

```
val nil2real = Types.T_FUNCTION([],
                                Types.T_REAL)

val real2real = Types.T_FUNCTION([("x",Types.T_REAL)],
                                 Types.T_REAL)

val real_real2real = Types.T_FUNCTION([("x",Types.T_REAL),("y",Types.T_REAL)],
                                      Types.T_REAL)

val bool2bool = Types.T_FUNCTION([("x",Types.T_BOOL)],
                                 Types.T_BOOL)

val real_real2bool = Types.T_FUNCTION([("x",Types.T_REAL),("y",Types.T_REAL)],
                                      Types.T_BOOL)
```

A.14.8 Initial environment

Relation `initial_env`

The initial environment where instantiation takes place is built up using this relation. It creates an empty environment and adds all the built-in definitions to it.

```
relation initial_env =

  rule  Env.open_scope(Env.empty_env) => env1 &
        Env.extend_frame_c(env1, rl_type) => env2 &
        Env.extend_frame_c(env2, int_type) => env3 &
        Env.extend_frame_c(env3, str_type) => env4 &
        Env.extend_frame_c(env4, bool_type) => env5 &
        Env.extend_frame_c(env5, real_type) => env6 &
        Env.extend_frame_c(env6, integer_type) => env7 &
        Env.extend_frame_c(env7, string_type) => env8 &
        Env.extend_frame_c(env8, boolean_type) => env9 &
        Env.extend_frame_c(env9, boolean_type) => env10 &

        Env.extend_frame_v(env10, time_var) => env11 &

        (* non-functions *)
        Env.extend_frame_t(env11, "inital", nil2real)                => env12 &
        Env.extend_frame_t(env12, "terminal", nil2real)              => env13 &
        Env.extend_frame_t(env13, "event", bool2bool)                => env14 &
        Env.extend_frame_t(env14, "switch", bool2bool)               => env15 &
        Env.extend_frame_t(env15, "timeEvent", real_real2bool)       => env16 &
        Env.extend_frame_t(env16, "der", real2real)                  => env17 &

        (* non-differentiable functions *)
        Env.extend_frame_t(env17, "div", real_real2real)             => env18 &
        Env.extend_frame_t(env18, "rem", real_real2real)             => env19 &
        Env.extend_frame_t(env19, "ceil", real2real)                 => env20 &
        Env.extend_frame_t(env20, "floor", real2real)                => env21 &

        (* differentiable functions *)
        Env.extend_frame_t(env21, "abs", real2real)                  => env22 &
        Env.extend_frame_t(env22, "sign", real2real)                 => env23 &

        (* Not in the report *)
        Env.extend_frame_t(env23, "sin", real2real)                  => env24 &
        Env.extend_frame_t(env24, "cos", real2real)                  => env25
        ----------------------------------------------
        initial_env () => env25

end
```

A.15 Module Connect

Connections generate connection sets which are constructed during instantiation. When a connection set is generated, it is used to create a number of equations.

This module performs connection set management.

```
module Connect :

  with "exp.rml"
  with "staticexp.rml"
  with "dae.rml"

  datatype Face = INNER | OUTER
```

This type indicates whether a connector is an inner or an outer connector.

```
  datatype Set = EQU of Exp.ComponentRef list
               | FLOW of (Exp.ComponentRef * Face) list

  type Sets = Set list
```

A connection set is represented using the Set type.

```
  relation add_equ : (Sets, Exp.ComponentRef, Exp.ComponentRef) => Sets
  relation add_flow : (Sets,
                       Exp.ComponentRef, Face,
                       Exp.ComponentRef, Face) => Sets
  relation print_sets : Sets => ()
  relation equations : Sets => DAE.Element list

end
```

A.15.1 Adding

The two relations add_eq and add_flow addes a variable to a connection set. The first relation is used to add a non-flow variable, and the second is used to add a flow variable. When two component are to be added to a collection of connection sets, the connections sets containg the components have to be located. If no such set exists, a new set containing only the new component is created.

If the connection sets containing the two components are not the same, they are merged.

```
relation add_equ : (Sets, Exp.ComponentRef, Exp.ComponentRef) => Sets =

  rule  find_equ_set(ss,r1) => s1 &
        find_equ_set(ss,r2) => s2 &
        merge(ss,s1,s2) => ss'
        ----------------------
        add_equ(ss,r1,r2) => ss'

end
```

```
relation add_flow : (Sets, Exp.ComponentRef, Face, Exp.ComponentRef, Face)
          => Sets =

  rule  find_flow_set(ss,r1,d1) => s1 &
        find_flow_set(ss,r2,d2) => s2 &
        merge(ss,s1,s2) => ss'
        ----------------------
        add_flow(ss,r1,d1,r2,d2) => ss'

end
```

A.15.2 Equation generation

Relation **`equations`**

From a number of connection sets, this relation generates a list of equations.

```
relation equations : Sets => DAE.Element list =

  axiom equations [] => []

  rule  equ_equations cs => dae1 &
        equations ss => dae2 &
        list_append(dae1,dae2) => dae
        -----------------------------
        equations EQU(cs)::ss => dae

  rule  flow_equations cs => dae1 &
        equations ss => dae2 &
        list_append(dae1,dae2) => dae
        -----------------------------
        equations FLOW(cs)::ss => dae

end
```

Relation **`equ_equations`**

A non-flow connection set contains a number of components. Generating the equation from this set means equating all the components. For n components, this will give n-1 equations.

For example, if the set contains the components `x`, `y.a` and `z.b`, the equations generated will me `x = y.a` and `y.a = z.b`.

```
relation equ_equations : Exp.ComponentRef list => DAE.Element list =

  axiom equ_equations [_] => []

  rule  equ_equations y::cs => eq
        ------------------------
        equ_equations x::y::cs
          => DAE.EQUATION(Exp.CREF(x), Exp.CREF(y))::eq
end
```

Relation **`flow_equations`**

Generating equations from a flow connection set is a little trickier that from a non-flow set. Only one equation is generated, but it has to consider whether the comoponents were inner or outer connectors.

This relation uses `flow_sum` to create the sum of all components (some of which will be negated), and the returns the equation where this sum is equal to 0.0.

```
relation flow_equations : (Exp.ComponentRef * Face) list => DAE.Element list =

  rule  flow_sum cs => sum
        ------------------
        flow_equations cs => [DAE.EQUATION(sum, Exp.RCONST(0.0))]

end
```

Relation `flow_sum`

This relation creates an exression expressing the sum of all components in the given list. Before adding the component to the sum, it is passed to `sign_flow` which will negate all outer connectors.

```
relation flow_sum : (Exp.ComponentRef * Face) list => Exp.Exp =

  rule  sign_flow(c,f) => exp
        -------------------
        flow_sum [(c,f)] => exp

  rule  sign_flow(c,f) => exp1 &
        flow_sum cs => exp2
        ------------------
        flow_sum ((c,f)::cs) => Exp.BINARY(exp1, Exp.ADD(Exp.REAL), exp2)

end
```

Relation `sign_flow`

This relation takes a name of a component and a `Face`, returns an expression. If the face is `INNER` the expression simply contains the component reference, but if it is `OUTER`, the expression is negated.

```
relation sign_flow : (Exp.ComponentRef, Face) => Exp.Exp =

  axiom sign_flow (c,INNER) => Exp.CREF(c)
  axiom sign_flow (c,OUTER) => Exp.UNARY(Exp.UMINUS(Exp.REAL),Exp.CREF(c))

end
```

A.15.3 Lookup

These relations are used to find and create connection sets.

Relation `find_equ_set`

This relation finds a non-flow connection set that contains the component named by the second argument. If no such set is found, a new set is created.

```
relation find_equ_set : (Sets, Exp.ComponentRef) => Set =

  rule  new_equ_set c => s
        ------------------
        find_equ_set([],c) => s

  rule  find_in_set(s,c)
        ----------------
        find_equ_set(s::_,c) => s

  rule  find_equ_set(ss,c) => s
        -------------------
        find_equ_set(_::ss,c) => s

end
```

Relation `find_flow_set`

This relation finds a flow connection set that contains the component named by the second argument. If no such set is found, a new set is created.

```
relation find_flow_set : (Sets, Exp.ComponentRef, Face) => Set =

  rule  new_flow_set(c,d) => s
        ------------------
        find_flow_set([],c,d) => s

  rule  find_in_set(s,c)
        ----------------
        find_flow_set(s::_,c,d) => s

  rule  find_flow_set(ss,c,d) => s
        -------------------
        find_flow_set(_::ss,c,d) => s

end
```

Relation `find_in_set`

This relation checks if a componet already appears in a given connection set.

```
relation find_in_set : (Set, Exp.ComponentRef) => () =

  rule  find_in_set_equ(cs,c)
        ---------------------
        find_in_set(EQU(cs), c)

  rule  find_in_set_flow(cs, c)
        -----------------------
        find_in_set(FLOW(cs), c)

end
```

Relation `find_in_set_equ`

This is a version of `find_in_set` which is specialized on non-flow connection sets

```
relation find_in_set_equ : (Exp.ComponentRef list, Exp.ComponentRef) => () =

  rule  Static.eq_cref(c1,c2)
        ------------------
        find_in_set_equ(c1::_,c2)

  rule  find_in_set_equ(cs,c2)
        ----------------------
        find_in_set_equ(_::cs,c2)

end
```

Relation **find_in_set_flow**

This is a version of `find_in_set` which is specialized on flow connection sets

```
relation find_in_set_flow : ((Exp.ComponentRef * Face) list, Exp.ComponentRef)
          => () =

  rule  Static.eq_cref(c1,c2)
        ------------------
        find_in_set_flow((c1,_)::_,c2)

  rule  find_in_set_flow(cs,c2)
        ----------------------
        find_in_set_flow(_::cs,c2)

end
```

Relation **new_equ_set**

This relation creates a new non-flow connection set containing only the given component.

```
relation new_equ_set : Exp.ComponentRef => Set =

  axiom new_equ_set c => EQU([c])

end
```

Relation **new_flow_set**

This relation creates a new-flow connection set containing only the given component.

```
relation new_flow_set : (Exp.ComponentRef, Face) => Set =

  axiom new_flow_set(c,d) => FLOW([(c,d)])

end
```

A.15.4 Merging

The result of merging two connection sets is the intersection of the two sets.

```
relation merge : (Sets, Set, Set) => Sets =

  rule  s1 = s2
        -------
        merge(ss,s1,s2) => ss

  rule  list_append(cs1,cs2) => cs &
        remove_set2(ss,s1,s2) => ss'
        ----------------------------
        merge(ss, s1 as EQU(cs1), s2 as EQU(cs2)) => EQU(cs)::ss'

  rule  list_append(cs1,cs2) => cs &
        remove_set2(ss,s1,s2) => ss'
        ----------------------------
        merge(ss, s1 as FLOW(cs1), s2 as FLOW(cs2)) => FLOW(cs)::ss'

end
```

Relation **remove_set2**

This relation removes the two sets given in the second and third argument from the collection of sets given in the first argument.

```
relation remove_set2 : (Sets, Set, Set) => Sets =

  axiom remove_set2([],_,_) => []

  rule  s = s1 &
        remove_set(ss,s2) => ss'
        ------------------------
        remove_set2(s::ss, s1, s2) => ss'

  rule  s = s2 &
        remove_set(ss,s1) => ss'
        ------------------------
        remove_set2(s::ss, s1, s2) => ss'

  rule  remove_set2(ss,s1,s2) => ss'
        ------------------------
        remove_set2(s::ss, s1, s2) => s::ss'

end
```

Relation **remove_set**

This relation removes one set from a list of sets.

```
relation remove_set : (Sets, Set) => Sets =

  axiom remove_set([],_) => []

  rule  s = s1
        ------
        remove_set(s::ss, s1) => ss

  rule  remove_set(ss,s1) => ss'
        ------------------------
        remove_set(s::ss, s1) => s::ss'

end
```

A.15.5 Printing

These are a few relations used for printing a description of the connection sets. The implementation is excluded from the report for brevity.

Relation `print_sets`

Prints a description of a number of connection sets to the standard output.

A.16 Module `Values`

The module `Values` contains data structures for representing constant Modelica values. These include integer, real, string and boolean values, and also arrays of any dimensionality and type. Multidimensional arrays are represented as arrays of arrays.

The code is excluded from the report, since they convey no semantic information.

```
module Values :

  datatype Value = INTEGER of int
                 | REAL of real
                 | STRING of string
                 | BOOL of bool
                 | ARRAY of Value list

  relation val_string : Value => string
  relation print_val : Value => ()

end
```

Relation `val_string`

This relation returns a textual representation of a value.

Relation `val_list_string`

This relation returns a textual representation of a list of values, separating each value with a comman.

Relation `print_val`

This relation prints a value.

A.17 Module `Static`

This module does static analysis and constant evaluation of expressions. The analyzed expressions are built using the constructors in the `Exp` module. Also, a set of properties of the expressions is calculated during analysis.

```
module Static :

  with "absyn.rml"
  with "exp.rml"
  with "explode.rml"
  with "types.rml"
  with "env.rml"
  with "values.rml"
  with "classinf.rml"
  with "lookup.rml"

  type Ident = string
```

A.17.1 Expression properties

```
  datatype Properties = PROP of Types.Type (* type *)
                              * bool       (* constant *)
```

The datatype `Properties` contain information about an expression. The properties are created by analyzing the expressions.

```
  (* Expression analysis *)
  relation elab_exp : (Env.Env, Absyn.Exp) => (Exp.Exp, Properties)
  relation elab_cref : (Env.Env, Absyn.ComponentRef)
          => (Exp.Exp, Properties, SCode.Accessibility)
  relation elab_subscripts : (Env.Env, Absyn.Subscript list)
          => (Exp.Subscript list, bool)

  (* Constant propagation *)
  relation ceval : (Env.Env, Exp.Exp)     => Values.Value
  relation canon_cref : (Env.Env, Exp.ComponentRef) => Exp.ComponentRef

  (* Property matching (type conversions etc.) *)
  relation match_prop : (Exp.Exp, Properties, Properties) => Exp.Exp
  relation deoverload : ((Exp.Operator * Types.Type list * Types.Type) list,
                         (Exp.Exp * Types.Type) list,
                         Absyn.Exp)
          => (Exp.Operator, Exp.Exp list, Types.Type)
  relation operators : Absyn.Operator
          => (Exp.Operator * Types.Type list * Types.Type) list

  (* Utility relations *)
  relation eq_cref : (Exp.ComponentRef, Exp.ComponentRef) => ()
  relation value_exp : Values.Value => Exp.Exp

end

with "dump.rml"
```

A.17.2 Expression elaboration

Relation `elab_exp`

Static analysis of expressions means finding out the properties of the expression. These properties are described by the `Properties` type, and include the type and the variability of the expression. This relation performs analysis, and returns an `Exp.Exp` and the properties.

```
relation elab_exp : (Env.Env, Absyn.Exp) => (Exp.Exp, Properties) =

  axiom elab_exp(_, Absyn.INTEGER(x))
          => (Exp.ICONST(x), PROP(Types.T_INTEGER,true))
  axiom elab_exp(_, Absyn.REAL(x))
          => (Exp.RCONST(x), PROP(Types.T_REAL,   true))
  axiom elab_exp(_, Absyn.STRING(x))
          => (Exp.SCONST(x), PROP(Types.T_STRING, true))
  axiom elab_exp(_, Absyn.BOOL(x))
          => (Exp.BCONST(x), PROP(Types.T_BOOL,   true))

  rule  elab_cref(env, cr) => (exp,prop,_)
        ---------------------------------------
        elab_exp(env, Absyn.CREF(cr)) => (exp,prop)
```

Binary and unary operations

```
  rule  elab_exp (env,e1) => (e1', PROP(t1, c1)) &
        elab_exp (env,e2) => (e2', PROP(t2, c2)) &
        bool_and (c1,c2) => c &
        operators op => ops &
        deoverload (ops, [(e1',t1),(e2',t2)], exp) => (op',[e1'',e2''],rtype)
        ---------------------------------------------------------------------
        elab_exp (env,exp as Absyn.BINARY(e1,op,e2))
          => (Exp.BINARY(e1'',op',e2''),PROP(rtype,c))

  rule  elab_exp (env,e) => (e',PROP(t,c)) &
        operators op => ops &
        deoverload (ops, [(e',t)], exp) => (op', [e''], rtype)
        ------------------------------------------------------
        elab_exp (env,exp as Absyn.UNARY(op,e))
          => (Exp.UNARY(op',e''),PROP(rtype,c))

  rule  elab_exp (env,e1) => (e1', PROP(t1, c1)) &
        elab_exp (env,e2) => (e2', PROP(t2, c2)) &
        bool_and (c1,c2) => c &
        operators op => ops &
        deoverload (ops, [(e1',t1),(e2',t2)], exp) => (op',[e1'',e2''],rtype)
        ---------------------------------------------------------------------
        elab_exp (env,exp as Absyn.LBINARY(e1,op,e2))
          => (Exp.LBINARY(e1'',op',e2''),PROP(rtype,c))

  rule  elab_exp (env,e) => (e',PROP(t,c)) &
        operators op => ops &
        deoverload (ops, [(e',t)], exp) => (op', [e''], rtype)
        ------------------------------------------------------
        elab_exp (env,exp as Absyn.LUNARY(op,e))
          => (Exp.LUNARY(op',e''),PROP(rtype,c))

  rule  elab_exp (env,e1) => (e1', PROP(t1, c1)) &
        elab_exp (env,e2) => (e2', PROP(t2, c2)) &
        bool_and (c1,c2) => c &
        operators op => ops &
        deoverload (ops, [(e1',t1),(e2',t2)], exp) => (op',[e1'',e2''],rtype)
        ----------------------------------------------
        elab_exp (env,exp as Absyn.RELATION(e1,op,e2))
          => (Exp.RELATION(e1'',op',e2''),PROP(rtype,c))
```

Conditional expressions

```
  rule  elab_exp (env,e1) => (e1', prop1) &
        elab_exp (env,e2) => (e2', prop2) &
        elab_exp (env,e3) => (e3', prop3) &
        elab_ifexp(e1',prop1,e2',prop2,e3',prop3) => (e,prop)
        ------------------------------------------------------
        elab_exp (env,Absyn.IFEXP(e1,e2,e3)) => (e,prop)
```

Function calls

```
  rule  elab_call(env,fn,args) => (e,prop)
        -------------------------------------
        elab_exp (env,Absyn.CALL(fn,args)) => (e,prop)
```

Array-related expressions

```
  rule  elab_exp (env, start) => (start', PROP(Types.T_INTEGER, c_start)) &
        elab_exp (env, stop) => (stop', PROP(Types.T_INTEGER, c_stop)) &
        bool_and (c_start, c_stop) => const &
        elab_range_type (env,start',NONE,stop',const) => t
        --------------------------------------------------
        elab_exp (env, Absyn.RANGE(start, NONE, stop))
          => (Exp.RANGE(start',NONE,stop'), PROP(t, const))

  rule  elab_exp (env, start) => (start', PROP(Types.T_INTEGER, c_start)) &
        elab_exp (env, step) => (step', PROP(Types.T_INTEGER, c_step)) &
        elab_exp (env, stop) => (stop', PROP(Types.T_INTEGER, c_stop)) &
        bool_and (c_start, c_step) => c1 &
        bool_and (c1, c_stop) => const &
        elab_range_type (env,start',SOME(step'),stop',const) => t
        ---------------------
        elab_exp (env, Absyn.RANGE(start, SOME(step), stop))
          => (Exp.RANGE(start',SOME(step'),stop'), PROP(t, const))

  rule  elab_array (env, es) => (es', PROP(t, const)) & list_length es' => l
        --------------------------------------------------------------
        elab_exp (env, Absyn.ARRAY(es))
          => (Exp.ARRAY(es'),
              PROP(Types.T_ARRAY(Types.DIM(SOME(l)), t),
                   const))

  rule  elab_matrix (env, es) => (es', t, dim, const) & list_length es' => l
        --------------------------------------------------------------------
        elab_exp (env, Absyn.MATRIX(es))
          => (Exp.MATRIX(es'),
              PROP(Types.T_ARRAY(Types.DIM(SOME(l)),
                                 Types.T_ARRAY(Types.DIM(SOME(dim)), t)),
                   const))

  rule  print "- elab_exp failed: " & Dump.print_exp e & print "\n"
        ---------------------------------------------------------
        elab_exp(_,e) => fail

end
```

Relation `elab_range_type`

```
relation elab_range_type : (Env.Env, Exp.Exp,
                            Exp.Exp option, Exp.Exp, bool) => Types.Type =

  rule  ceval (env,start) => Values.INTEGER(startv) &
        ceval (env,stop) => Values.INTEGER(stopv) &
        int_sub (stopv,startv) => n & int_add (n,1) => n'
        ------------------------------------------------
        elab_range_type (env,start,NONE,stop,true)
          => Types.T_ARRAY(Types.DIM(SOME(n')), Types.T_INTEGER)

  rule  ceval (env,start) => Values.INTEGER(startv) &
        ceval (env,step) => Values.INTEGER(stepv) &
        ceval (env,stop) => Values.INTEGER(stopv) &
        int_sub (stopv,startv) => n &
        int_div (n,stepv) => n' &
        int_add (n',1) => n''
        ------------------------------------------------
        elab_range_type (env,start,SOME(step),stop,true)
          => Types.T_ARRAY(Types.DIM(SOME(n'')), Types.T_INTEGER)

  axiom elab_range_type (_,_,_,_,false)
          => Types.T_ARRAY(Types.DIM(NONE), Types.T_INTEGER)

end
```

Relation `elab_array`

```
relation elab_array : (Env.Env, Absyn.Exp list)
          => (Exp.Exp list, Properties) =

  rule  elab_exp (env,e) => (e',prop)
        ----------------------------
        elab_array (env,[e]) => ([e'],prop)

  rule  elab_exp (env,e) => (e',PROP(t1,c1)) &
        elab_array (env,es) => (es',PROP(t2,c2)) &
        t1 = t2 &
        bool_and (c1,c2) => c
        --------------------------------
        elab_array (env,e::es) => (e'::es',PROP(t1,c))

  rule  elab_exp (env,e) => (e',PROP(t1,c1)) &
        elab_array (env,es) => (es',PROP(t2,c2)) &
        not t1 = t2 &
        print "# Incompatible types in array expression\n" &
        print "  " & Dump.print_exp e & print " is of type " &
        Types.print_type t1 & print "\n  while the elements [" &
        Dump.print_list (es, Dump.print_exp, ",") &
        print "] are of type " & Types.print_type t2 & print "\n"
        ----------------------------------------------------
        elab_array (env,e::es) => fail

  rule  print "- elab_array failed\n"
        ----------------------------
        elab_array (_,_) => fail

end
```

Relation `elab_matrix`

```
relation elab_matrix : (Env.Env, Absyn.Exp list list)
        => (Exp.Exp list list, Types.Type, int, bool) =

  rule  list_length(el) => dim &
        elab_array(env,el) => (el', PROP(t, c))
        ---------------------------------
        elab_matrix(env,[el]) => ([el'], t, dim, c)

  rule  list_length(el) => dim1 &
        elab_array(env,el) => (el',PROP(t1,c1)) &
        elab_matrix(env,els) => (els',t2,dim2,c2) &
        t1 = t2 & dim1 = dim2 &
        bool_and(c1,c2) => c
        --------------------
        elab_matrix(env,el::els) => (el'::els',t1,dim1,c)
```

Error messages

```
  rule  print "- How the f*ck did you write an empty matrix?\n"
        -----------------------------------------------------
        elab_matrix(_,[]) => fail

  rule  elab_array(env,el) => (el',PROP(t1,c1)) &
        elab_matrix(env,els) => (els',t2,dim2,c2) &
        not t1 = t2 &
        print "# Incompatible types in matrix rows\n" &
        print "    [" & Dump.print_list(el,Dump.print_exp,",") &
        print "] is a row of " & Types.print_type t1 & print "\n" &
        print "    the rest of the matrix is of " &
        Types.print_type t2 & print "\n"
        --------------------------------
        elab_matrix(env,el::els) => fail

  rule  list_length(el) => dim1 &
        elab_matrix(env,els) => (els',t2,dim2,c2) &
        not dim1 = dim2 &
        print "# Incompatible row length in matrix expression\n" &
        print "    [" & Dump.print_list(el,Dump.print_exp,",") &
        print "] is a row of size " &
        int_string dim1 => s & print s & print "\n" &
        print "    the rest of the matrix rows are of size " &
        int_string dim2 => s & print s & print "\n"
        --------------------------------------------
        elab_matrix(env,el::els) => fail

end
```

Relation **`elab_call`**

This relation elaborates on a function call. It converts the name to a `Path`, and used the `elab_call_args` to do the rest of the work.

```
relation elab_call : (Env.Env, Absyn.ComponentRef, Absyn.Exp list)
        => (Exp.Exp, Properties) =

  rule  Absyn.cref_to_path fn => fn' &
        elab_call_args (env,fn',args) => (e,prop)
        -----------------------------------
        elab_call (env,fn,args) => (e,prop)

  rule  print "- elab_call failed\n"
        ---------------------------
        elab_call (env,fn,args) => fail

end
```

Relation `elab_call_args`

Given the name of a function and a list of expressions to be used as actual arguments in a function call to that function, this relation finds the function definition and matches the actual arguments to the formal parameters.

```
relation elab_call_args : (Env.Env, Absyn.Path, Absyn.Exp list)
        => (Exp.Exp, Properties) =

        (* This rule finds user-defined functions. *)

  rule  Lookup.lookup_type(env,fn) => (t as Types.T_COMPLEX(st,vl)) &
        (* ClassInf.valid(st, Absyn.R_FUNCTION) & *)
        function_params(vl) => (in,out as [(_,rt)]) &
        elab_input_args(env, args, in) => (args', c)
        --------------------------------------------
        elab_call_args(env,fn,args) => (Exp.CALL(fn,args'), PROP(rt, c))

        (* This rule finds the built-in functions *)

  rule  Lookup.lookup_type(env,fn) => (t as Types.T_FUNCTION(params,restype)) &
        elab_input_args(env, args, params) => (args',c)
        -----------------------------------------------
        elab_call_args (env,fn,args) => (Exp.CALL(fn,args'), PROP(restype, c))


  rule  not Lookup.lookup_type(env,fn) => _ &
        print "# Couldn't find function " & Dump.print_path fn & print "\n"
        -------------------------------------------------------------------
        elab_call_args(env,fn,_) => fail

  rule  print "- elab_call_args failed\n"
        ---------------------------------
        elab_call_args(_,_,_) => fail

end
```

Relation `function_params`

A function definition is just a class definition where all public components are declared as either `input` or `output`. This relation find all those components and separates them into two separate lists.

```
relation function_params : Types.Var list => ((Ident * Types.Type) list,
                                              (Ident * Types.Type) list) =

  axiom function_params [] => ([],[])

        (* Ignore protected components *)
  rule  function_params vs => (in, out)
        ------------------------------
        function_params((Types.VAR(_,_,true,_,_))::vs) => (in, out)

  rule  function_params vs => (in, out)
        ------------------------------
        function_params (Types.VAR(n,Types.ATTR(_,_,_,Absyn.INPUT),
                                   false,t,Types.UNBOUND)::vs)
          => ((n,t)::in, out)

  rule  function_params vs => (in, out)
        ------------------------------
        function_params (Types.VAR(n,Types.ATTR(_,_,_,Absyn.OUTPUT),
                                         false,t,Types.UNBOUND)::vs)
          => (in, (n,t)::out)

  rule  print "# Components in functions must be INPUT or OUTPUT\n" &
        print "    component: " & print n & print "\n"
        ----------------------------------------------
        function_params((v as Types.VAR(n,Types.ATTR(_,_,_,Absyn.BIDIR),_,_,_))
                        ::vs) => fail

  rule  print "- function_params failed\n"
        ----------------------------------
        function_params _ => fail

end
```

Relation `elab_input_args`

This relation elaborates on a number of expressions and matches them to a number of `Types.Var` object, applying type conversions on the expressions when necessary to match the type of the `Types.Var`.

```
relation elab_input_args : (Env.Env, Absyn.Exp list, (Ident * Types.Type) list)
          => (Exp.Exp list, bool) =

  axiom elab_input_args(_, [], []) => ([], true)

  rule  elab_exp(env, e) => (e',PROP(t, c1)) &
        match_type(e', t, vt) => e'' &
        elab_input_args(env, es, vs) => (args', c2) &
        bool_and(c1, c2) => c
        -------------------------------------
        elab_input_args(env, e::es, (_,vt)::vs) => (e''::args', c)

  rule  elab_exp(env, e) => (e',PROP(t,_)) &
        print "# Argument (" & Dump.print_exp e &
        print ") doesn't match parameter type\n" &
        print "    parameter type: " & Types.print_type vt & print "\n" &
        print "    argument type: " & Types.print_type t & print "\n"
        -------------------------------------------------------------
        elab_input_args(env, e::es, (_,vt)::vs) => fail

end
```

Relation `elab_cref`

Elaborate on a component reference. Check the type of the component referred to, and check if the environment contains either a constant binding for that variable, or if it contains an equation binding with a constant expression.

```
relation elab_cref : (Env.Env, Absyn.ComponentRef)
          => (Exp.Exp, Properties, SCode.Accessibility) =

  rule  elab_cref_subs (env,c) => (c', const) &
        Lookup.lookup_var (env,c') => (Types.ATTR(_,acc,variability,_),
                                       t,
                                       binding) &
        elab_cref2 (env, c', acc, variability, t, binding)
          => (exp,const,acc')
        (* FIXME subscript_cref_type (exp,t) => t' &  *)
        ---------------------------------
        elab_cref(env, c) => (exp, PROP(t, const), acc')

  rule  elab_cref_subs (env,c) => (c', const) &
        print "# Unknown component: " & Dump.print_component_ref c & print "\n"
        -----------------------------------------------------
        elab_cref(env, c) => fail

  rule  print "- elab_cref failed\n"
        ------------------------------
        elab_cref (_,_) => fail

end
```

Relation `elab_cref2`

This relation check whether the component reference found in `elab_cref` has a binding, and if that binding is constant. If the binding is a `VALBOUND` binding, the value is substituted.

```
relation elab_cref2 : (Env.Env,
                       Exp.ComponentRef,
                       SCode.Accessibility, Absyn.Variability,
                       Types.Type, Types.Binding)
          => (Exp.Exp,bool,SCode.Accessibility) =

  axiom elab_cref2 (_,cr, acc, Absyn.VAR, _,_) => (Exp.CREF(cr),false, acc)
  axiom elab_cref2 (_,cr, acc, Absyn.DISCRETE, _,_) => (Exp.CREF(cr),false,acc)

  rule  value_exp v => e
        ----------------
        elab_cref2 (_,cr,_,_,t, Types.VALBOUND(v)) => (e,true,SCode.RO)

  axiom elab_cref2 (env,cr,acc,_,t, Types.EQBOUND(exp,true))
          => (Exp.CREF(cr),true,acc)

  rule  print "# Constant or parameter with a non-constant initializer\n" &
        print "# component: " & Exp.print_component_ref cr & print " = " &
        Exp.print_exp exp & print "\n"
        ------------------------------
        elab_cref2 (_,cr,_, _,_,Types.EQBOUND(exp,false)) => fail

  rule  print "# Constant or parameter without a value\n" &
        print "# component: " & Exp.print_component_ref cr & print "\n"
        ------------------------------------------------
        elab_cref2 (_,cr,_,_,_,Types.UNBOUND) => fail

  rule  print "- elab_cref2 failed (component: " &
        Exp.print_component_ref cr & print ")\n"
        ----------------------------------------
        elab_cref2 (_,cr,_,_,_,_) => fail

end
```

Relation **elab_cref_subs**

This relation elaborates on all subscripts in a component reference.

```
relation elab_cref_subs : (Env.Env, Absyn.ComponentRef)
         => (Exp.ComponentRef, bool) =

  rule  elab_subscripts (env, ss) => (ss', const)
        ---------------------------------------------
        elab_cref_subs(env,Absyn.CREF_IDENT(id,ss))
          => (Exp.CREF_IDENT(id,ss'), const)

  rule  elab_subscripts (env, ss) => (ss', const1) &
        elab_cref_subs(env,subs) => (cr, const2) &
        bool_and(const1, const2) => const
        ------------------------------------
        elab_cref_subs(env,Absyn.CREF_QUAL(id,ss,subs))
          => (Exp.CREF_QUAL(id,ss',cr), const)

end
```

Relation **elab_subscripts**

This relation converts a list of `Absyn.Subscript` to a list of `Exp.Subscript`, and checks if all subscripts are constant.

```
relation elab_subscripts : (Env.Env, Absyn.Subscript list)
         => (Exp.Subscript list, bool) =

  axiom elab_subscripts (_,[]) => ([], true)

  rule  elab_subscript (env,sub) => (sub',const1) &
        elab_subscripts (env,subs) => (subs',const2) &
        bool_and (const1, const2) => true &
        ceval_subscripts(env, sub'::subs') => ss
        -----------------------------------
        elab_subscripts (env,sub::subs) => (ss,true)

end
```

Relation **elab_subscript**

This relation converts an `Absyn.Subscript` to an `Exp.Subscript`.

```
relation elab_subscript : (Env.Env, Absyn.Subscript)
         => (Exp.Subscript, bool) =

  axiom elab_subscript (_,Absyn.NOSUB) => (Exp.WHOLEDIM, true)

  rule  elab_exp (env,sub) => (sub', PROP(ty, const)) &
        elab_subscript_type (ty, sub, sub') => sub''
        --------------------------------------------
        elab_subscript (env,Absyn.SUBSCRIPT(sub)) => (sub'', const)

end
```

Relation **elab_subscript_type**

This relation is used to find the correct constructor for `Exp.Subscript` to use for an indexing expression. If an integer is given as index, `Exp.INDEX()` is used, and if an integer array is given, `Exp.SLICE()` is used.

```
relation elab_subscript_type : (Types.Type, Absyn.Exp, Exp.Exp)
          => Exp.Subscript =

  axiom elab_subscript_type (Types.T_INTEGER,_,sub) => Exp.INDEX(sub)

  axiom elab_subscript_type (Types.T_ARRAY(_,Types.T_INTEGER),_,sub)
          => Exp.SLICE(sub)

  rule  print "# Subscript is not an integer or integer array\n" &
        print "    expression: " & Dump.print_exp e &
        print " :: " & Types.print_type t & print "\n"
        -----------------------------------------------------
        elab_subscript_type (t,e,_) => fail

end
```

Relation **subscript_cref_type**

If a component of an array type is subscripted, the type of the component reference is of lower dimensionality than the component. This relation shows the relation between the component type and the component reference expression type.

This relation might actually not be needed.

```
relation subscript_cref_type : (Exp.Exp, Types.Type) => Types.Type =

  rule  subscript_cref_type2 (c, t) => t'
        ---------------------------------
        subscript_cref_type (Exp.CREF(c), t) => t'

  axiom subscript_cref_type (e, t) => t

end
```

```
relation subscript_cref_type2 : (Exp.ComponentRef, Types.Type) => Types.Type =

  axiom subscript_cref_type2 (Exp.CREF_IDENT(_,[]), t) => t

  rule  subscript_type (t,subs) => t'
        ----------------------------------------
        subscript_cref_type2 (Exp.CREF_IDENT(_,subs), t) => t'

  rule  subscript_cref_type2 (c, t) => t'
        ---------------------------------
        subscript_cref_type2 (Exp.CREF_QUAL(_,_,c), t) => t'

end
```

Relation **subscript_type**

Given an array dimensionality and a list of subscripts, this relation reduces the dimensionality.

This does not handle slices or check that subscripts are not out of bounds.

```
relation subscript_type : (Types.Type, Exp.Subscript list)
          => Types.Type =

  axiom subscript_type (t,[]) => t

  rule  subscript_type (t, subs) => t'
        ----------------------------------
        subscript_type (Types.T_ARRAY(Types.DIM(_),t),
                        Exp.INDEX(_)::subs) => t'

  rule  subscript_type (t, subs) => t'
        ------------------------------
        subscript_type (Types.T_ARRAY(dim,t),
                        Exp.SLICE(_)::subs)
          => Types.T_ARRAY(dim,t')

  rule  subscript_type (t, subs) => t'
        ------------------------------
        subscript_type (Types.T_ARRAY(dim,t),
                        Exp.WHOLEDIM::subs)
          => Types.T_ARRAY(dim,t')

  rule  print "- subscript_type failed (" & Types.print_type t &
        print " , [...])\n"
        -------------------
        subscript_type (t,_) => fail

end
```

Relation `elab_ifexp`

This relation elaborates on the parts of an if expression.

```
relation elab_ifexp : (Exp.Exp, Properties,
                       Exp.Exp, Properties,
                       Exp.Exp, Properties)
          => (Exp.Exp, Properties) =

  rule  Types.equivtypes(t2,t3) => true &
        const_ifexp(e1,c1,c2,c3) => c
        -----------------------------
        elab_ifexp(e1,PROP(Types.T_BOOL,c1),
                   e2,PROP(t2,c2),
                   e3,PROP(t3,c3)) => (Exp.IFEXP(e1,e2,e3), PROP(t2, c))

  rule  not t1 = Types.T_BOOL &
        print "# conditional in if expression has to be boolean\n" &
        print "    got type: " & Types.print_type t1 & print "\n" &
        print "    expression: " & Exp.print_exp e1 & print "\n"
        ------------------------------------------------------
        elab_ifexp(e1,PROP(t1,c1),e2,PROP(t2,c2),e3,PROP(t3,c3)) => fail

  rule  Types.equivtypes(t2,t3) => false &
        print "# Type mismatch in if-expression\n" &
        print "    true branch : " & Exp.print_exp e2 &
        print " :: " & Types.print_type t2 & print "\n" &
        print "    false branch: " & Exp.print_exp e3 &
        print " :: " & Types.print_type t3 & print "\n"
        ----------------------------------------------
        elab_ifexp(e1,PROP(Types.T_BOOL,c1),
                   e2,PROP(t2,c2),e3,PROP(t3,c3)) => fail

  rule  print "- elab_ifexp failed\n"
        -----------------------------
        elab_ifexp(_,_,_,_,_,_) => fail

end
```

Relation `const_ifexp`

Tests wether an `if` expression is constant. This is done by first testing if the conditional is constant, and if so evaluating it to see which branch should be tested for constant-ness.

This will miss some occations where the expression actually is constant, as in the expression `if x then 1.0 else 1.0`.

```
relation const_ifexp : (Exp.Exp, bool, bool, bool) => bool =

  axiom const_ifexp(_,true,true,true) => true
  axiom const_ifexp(_,_,_,_) => false

end
```

Relation `ceval`

This relations is used when the value of a constant expression is needed. It takes an environment and an expression and calculates its value.

```
relation ceval : (Env.Env, Exp.Exp) => Values.Value =

  axiom ceval (_,Exp.ICONST(x)) => Values.INTEGER(x)
  axiom ceval (_,Exp.RCONST(x)) => Values.REAL(x)
  axiom ceval (_,Exp.SCONST(x)) => Values.STRING(x)
  axiom ceval (_,Exp.BCONST(x)) => Values.BOOL(x)

  rule  ceval_list (env,es) => es' &
        list_length es' => l
        ------------------------------
        ceval (env,Exp.ARRAY(es)) => Values.ARRAY(es')

  rule  ceval_cref (env,c) => v
        ---------------------------
        ceval (env,Exp.CREF(c)) => v

  rule  print "# Can't call functions at compile time\n" &
        print "    expression: " & Exp.print_exp e & print "\n"
        -----------------------------------------------------
        ceval (env, e as Exp.CALL(_,_)) => fail
```

Numerical

```
  rule  ceval (env, lh) => Values.REAL(lhv) &
        ceval (env, rh) => Values.REAL(rhv) &
        real_add(lhv, rhv) => sum
        ------------------------
        ceval (env, Exp.BINARY(lh, Exp.ADD(Exp.REAL), rh))
          => Values.REAL(sum)

  rule  ceval (env, lh) => Values.INTEGER(lhv) &
        ceval (env, rh) => Values.INTEGER(rhv) &
        int_add(lhv, rhv) => sum
        ------------------------
        ceval (env, Exp.BINARY(lh, Exp.ADD(Exp.INT), rh)) => Values.INTEGER(sum)
```

```
rule  ceval (env, lh) => Values.REAL(lhv) &
      ceval (env, rh) => Values.REAL(rhv) &
      real_sub(lhv, rhv) => sum
      -------------------------
      ceval (env, Exp.BINARY(lh, Exp.SUB(Exp.REAL), rh)) => Values.REAL(sum)

rule  ceval (env, lh) => Values.INTEGER(lhv) &
      ceval (env, rh) => Values.INTEGER(rhv) &
      int_sub(lhv, rhv) => sum
      ------------------------
      ceval (env, Exp.BINARY(lh, Exp.SUB(Exp.INT), rh))
        => Values.INTEGER(sum)
```

```
rule  ceval (env, lh) => Values.REAL(lhv) &
      ceval (env, rh) => Values.REAL(rhv) &
      real_mul(lhv, rhv) => sum
      -------------------------
      ceval (env, Exp.BINARY(lh, Exp.MUL(Exp.REAL), rh)) => Values.REAL(sum)

rule  ceval (env, lh) => Values.INTEGER(lhv) &
      ceval (env, rh) => Values.INTEGER(rhv) &
      int_mul(lhv, rhv) => sum
      ------------------------
      ceval (env, Exp.BINARY(lh, Exp.MUL(Exp.INT), rh))
        => Values.INTEGER(sum)
```

```
rule  ceval (env, exp) => Values.REAL(x) &
      real_neg (x) => x'
      ---------------------
      ceval (env, Exp.UNARY(Exp.UMINUS(Exp.REAL), exp))
        => Values.REAL(x')

rule  ceval (env, exp) => Values.INTEGER(x) &
      int_neg (x) => x'
      ---------------------
      ceval (env, Exp.UNARY(Exp.UMINUS(Exp.INT), exp))
        => Values.INTEGER(x')
```

```
rule  ceval (env, exp) => Values.REAL(x) &
      real_neg (x) => x'
      ---------------------
      ceval (env, Exp.UNARY(Exp.UPLUS(Exp.REAL), exp))
        => Values.REAL(x')

rule  ceval (env, exp) => Values.INTEGER(x) &
      int_neg (x) => x'
      ---------------------
      ceval (env, Exp.UNARY(Exp.UPLUS(Exp.INT), exp))
        => Values.INTEGER(x')
```

Logical

```
  rule  ceval (env, lh) => Values.BOOL(lhv) &
        ceval (env, rh) => Values.BOOL(rhv) &
        bool_and(lhv, rhv) => x
        ------------------------
        ceval (env, Exp.LBINARY(lh, Exp.AND, rh))
          => Values.BOOL(x)

  rule  ceval (env, lh) => Values.BOOL(lhv) &
        ceval (env, rh) => Values.BOOL(rhv) &
        bool_or(lhv, rhv) => x
        ------------------------
        ceval (env, Exp.LBINARY(lh, Exp.OR, rh))
          => Values.BOOL(x)

  rule  ceval (env,e) => Values.BOOL(b) &
        bool_not(b) => b'
        -----------------
        ceval (env, Exp.LUNARY(Exp.NOT, e)) => Values.BOOL(b')
```

Relations

```
  rule  ceval (env, lhs) => lhs' &
        ceval (env, rhs) => rhs' &
        ceval_relation (lhs',relop,rhs') => v
        -------------------------------------
        ceval (env, Exp.RELATION(lhs,relop,rhs)) => v
```

```
  rule  ceval (env, start) => Values.INTEGER(start') &
        ceval (env, stop) => Values.INTEGER(stop') &
        ceval_range(start', 1, stop') => arr
        -------------------------------------------
        ceval (env, Exp.RANGE(start, NONE, stop)) => Values.ARRAY(arr)

  rule  ceval (env, start) => Values.INTEGER(start') &
        ceval (env, step) => Values.INTEGER(step') &
        ceval (env, stop) => Values.INTEGER(stop') &
        ceval_range(start', step', stop') => arr
        -------------------------------------------
        ceval (env, Exp.RANGE(start, SOME(step), stop)) => Values.ARRAY(arr)

  rule  ceval (env,e) => Values.INTEGER(i) &
        int_real i => r
        ---------------
        ceval (env, Exp.CAST(Exp.REAL,e)) => Values.REAL(r)

  rule  print "- ceval " & Exp.print_exp e & print " failed\n"
        -----------------------------------------------------
        ceval (_,e) => fail

end
```

Relation ceval_relation

Performs the relation check and gives a boolean result.

```
relation ceval_relation : (Values.Value, Exp.Operator, Values.Value)
          => Values.Value =

  rule  ceval_relation (v2, Exp.LESS(t), v1) => v
        -----------------------------------------
        ceval_relation (v1, Exp.GREATER(t), v2) => v
```

Integers

```
  rule  int_lt (i1,i2) => b
        -------------------
        ceval_relation (Values.INTEGER(i1),
                        Exp.LESS(Exp.INT),
                        Values.INTEGER(i2)) => Values.BOOL(b)

  rule  int_eq (i1,i2) => b
        -------------------
        ceval_relation (Values.INTEGER(i1),
                        Exp.EQUAL(Exp.INT),
                        Values.INTEGER(i2)) => Values.BOOL(b)

  rule  int_ne (i1,i2) => b
        -------------------
        ceval_relation (Values.INTEGER(i1),
                        Exp.NEQUAL(Exp.INT),
                        Values.INTEGER(i2)) => Values.BOOL(b)
```

Reals

```
  rule  real_lt (i1,i2) => b
        -------------------
        ceval_relation (Values.REAL(i1),
                        Exp.LESS(Exp.REAL),
                        Values.REAL(i2)) => Values.BOOL(b)

  rule  real_eq (i1,i2) => b
        -------------------
        ceval_relation (Values.REAL(i1),
                        Exp.EQUAL(Exp.REAL),
                        Values.REAL(i2)) => Values.BOOL(b)

  rule  real_ne (i1,i2) => b
        -------------------
        ceval_relation (Values.REAL(i1),
                        Exp.NEQUAL(Exp.REAL),
                        Values.REAL(i2)) => Values.BOOL(b)

  rule  print "- ceval_relation failed\n"
        ---------------------------------
        ceval_relation (_,_,_) => fail

end
```

Relation `ceval_range`

This relation evaluates a range expression. It only handles integers.

```
relation ceval_range : (int, int, int) => Values.Value list =

  rule  int_gt(start,stop) => true
        -------------------------
        ceval_range (start,_,stop) => []

  rule  int_gt(start,stop) => false & (* redundant *)
        int_add (start, step) => next &
        ceval_range (next, step, stop) => l
        -----------------------------------
        ceval_range (start, step, stop) => Values.INTEGER(start)::l

end
```

Relation `ceval_list`

This relation does a constant evaluation on a number of expressions.

```
relation ceval_list : (Env.Env, Exp.Exp list) => Values.Value list =

  rule  ceval (env, exp) => v
        ----------------------------
        ceval_list (env, [exp]) => [v]

  rule  ceval (env,exp) => v &
        ceval_list (env, exps) => vs
        ------------------------------
        ceval_list (env, exp::exps) => v::vs

end
```

```
relation ceval_cref : (Env.Env, Exp.ComponentRef) => Values.Value =

  rule  Lookup.lookup_var (env, c) => (_,_,binding) &
        ceval_cref_binding (env,binding) => v
        -------------------------------------
        ceval_cref (env,c) => v

        (* default *)

  rule  print "# No constant binding for " &
        Exp.print_component_ref c & print "\n"
        -------------------------------------
        ceval_cref (_,c) => fail

end

relation ceval_cref_binding : (Env.Env, Types.Binding) => Values.Value =

  axiom ceval_cref_binding (env,Types.VALBOUND(v)) => v

  rule  print "- ceval_cref_binding failed (UNBOUND)\n"
        -------------------------------------
        ceval_cref_binding (env,Types.UNBOUND) => fail

  rule  ceval (env,exp) => v
        ------------------------
        ceval_cref_binding (env,Types.EQBOUND(exp,true)) => v

  rule  print "- ceval_cref__binding failed (nonconstant EQBOUND(" &
        Exp.print_exp exp & print ")\n"
        --------------------------------
        ceval_cref_binding (env,Types.EQBOUND(exp,false)) => fail

end
```

```
relation value_exp : Values.Value => Exp.Exp =

  axiom value_exp (Values.INTEGER(x)) => Exp.ICONST(x)
  axiom value_exp (Values.REAL(x))    => Exp.RCONST(x)
  axiom value_exp (Values.STRING(x))  => Exp.SCONST(x)
  axiom value_exp (Values.BOOL(x))    => Exp.BCONST(x)
  axiom value_exp (Values.ARRAY([]))  => Exp.ARRAY([])

  rule  value_exp x => x' &
        value_exp Values.ARRAY(xs) => Exp.ARRAY(xs')
        -------------------------------------------
        value_exp (Values.ARRAY(x::xs)) => Exp.ARRAY(x'::xs')

  rule  print "- value_exp failed\n"
        ----------------------------
        value_exp _ => fail

end
```

```
relation value_type : Values.Value => Types.Type =

  axiom value_type Values.INTEGER(x) => Types.T_INTEGER
  axiom value_type Values.REAL(x)    => Types.T_REAL
  axiom value_type Values.STRING(x)  => Types.T_STRING
  axiom value_type Values.BOOL(x)    => Types.T_BOOL

  rule  print "- value_type failed\n"
        ----------------------------
        value_type _ => fail

end
```

Relation **canon_cref**

This relation relates a `Exp.ComponentRef` to its canonical form, which is when all subscripts are evaluated to constant values. If Such an evaluation is not possible, there is no canonical form and this relation fails.

```
relation canon_cref : (Env.Env, Exp.ComponentRef) => Exp.ComponentRef =

  rule  ceval_subscripts (env,ss) => ss'
        --------------------------------
        canon_cref (env, Exp.CREF_IDENT(n,ss)) => Exp.CREF_IDENT(n,ss')

  rule  ceval_subscripts (env,ss) => ss' &
        canon_cref (env,c) => c'
        ------------------------
        canon_cref (env, Exp.CREF_QUAL(n,ss,c)) => Exp.CREF_QUAL(n,ss',c')

end
```

Relation **ceval_subscripts**

This relation relates a list of subscripts to their canonical forms, which is when all expressions are evaluated to constant values.

```
relation ceval_subscripts : (Env.Env, Exp.Subscript list)
          => Exp.Subscript list =

  axiom ceval_subscripts (_,[]) => []

  rule  ceval_subscript (env,sub) => sub' &
        ceval_subscripts (env,subs) => subs'
        ------------------------------------
        ceval_subscripts (env,sub::subs) => (sub'::subs')

end
```

Relation **ceval_subscript**

This relation relates a subscript to its canonical forms, which is when all expressions are evaluated to constant values.

```
relation ceval_subscript : (Env.Env, Exp.Subscript) => Exp.Subscript =

  axiom ceval_subscript (env, Exp.WHOLEDIM) => Exp.WHOLEDIM

  rule  ceval (env, e1) => v1 &
        value_exp v1 => e1'
        ----------------------------------------
        ceval_subscript (env, Exp.INDEX(e1)) => Exp.INDEX(e1')

  rule  ceval (env, e1) => v1 &
        value_exp v1 => e1'
        ----------------------------------------
        ceval_subscript (env, Exp.SLICE(e1)) => Exp.SLICE(e1')

end
```

A.17.3 Equality relations

Relation **`eq_cref`**

This relation checks if two component references can be considered equal and fails if not. Two component references are equal if all corresponding identifiers are the same, and if the subscripts are equal, according to the relation `eq_subscripts`.

```
relation eq_cref : (Exp.ComponentRef, Exp.ComponentRef) => () =

  rule  n1 = n2 &
        eq_subscripts (s1,s2)
        ---------------------
        eq_cref (Exp.CREF_IDENT(n1,s1), Exp.CREF_IDENT(n2,s2))

  rule  n1 = n2 &
        eq_subscripts (s1,s2) &
        eq_cref (c1,c2)
        ---------------
        eq_cref (Exp.CREF_QUAL(n1,s1,c1), Exp.CREF_QUAL(n2,s2,c2))

end
```

Relation **`eq_subscripts`**

Two list of subscripts are equal if they are of equal length and all their elements are pairwise equal according to the relation `eq_subscript`.

```
relation eq_subscripts : (Exp.Subscript list, Exp.Subscript list) => () =

  axiom eq_subscripts([],[])

  rule  eq_subscript(s1,s2) &
        eq_subscripts(ss1,ss2)
        ----------------------
        eq_subscripts(s1::ss1, s2::ss2)

end
```

Relation **`eq_subscript`**

This relation test whether two subscripts are equal. Two subscripts are equal if they have the same constructor, and if all corresponding expressions are either syntactically equal, or if they have the same constant value.

```
relation eq_subscript : (Exp.Subscript, Exp.Subscript) => () =

  axiom eq_subscript(Exp.WHOLEDIM,Exp.WHOLEDIM)

  rule  s1 = s2
        -------------------------------
        eq_subscript(Exp.INDEX(s1),Exp.INDEX(s2))

  rule  print "- eq_subscript failed\n"
        -------------------------------
        eq_subscript(_,_)

end
```

A.17.4 Argument type casting and operator de-overloading

If a function is called with arguments that don't match the expected parameter types, implicit type conversions are performed in some cases. Usually it is an integer argument that is promoted to a real.

Many operators in Modelica are overloaded, meaning that they can operate on several different types of arguments. To describe what it means to add, say, an integer and a real number, the expressions have to be de-overloaded, with one operator for each distinct operation.

Relation `elab_arglist`

Given a list of parameter types and an argument list, this relation tries to match the two, promoting the type of arguments when necessary.

```
relation elab_arglist : (Types.Type list, (Exp.Exp * Types.Type) list)
        => Exp.Exp list =

  axiom elab_arglist ([],[]) => []

  rule  match_type(arg, atype, pt) => arg' &
        elab_arglist (pts, args) => args'
        ---------------------------------
        elab_arglist (pt::pts, (arg, atype)::args) => arg'::args'

end
```

Relation `match_prop`

This is basically a wrapper aroune `match_type`. It matches an expression with properties with another set of properties. If necessary, the expression is modified to match. The only relevant property is the type.

```
relation match_prop : (Exp.Exp, Properties, Properties) => Exp.Exp =

  rule  match_type(e, gt, et) => e'
        ---------------------------
        match_prop(e, PROP(gt,_), PROP(et,_)) => e'

end
```

Relation `match_type`

This relation matches an expression with an expected type, and converts the expression to the expected type if necessary.

```
relation match_type : (Exp.Exp, Types.Type, Types.Type) => Exp.Exp =

  rule  Types.subtype(e_type, expected_type) => true
        --------------------------------------------
        match_type(e, e_type, expected_type) => e

  rule  Types.subtype(e_type, expected_type) => false &
        type_convert(e, e_type, expected_type) => e'
        --------------------------------------------
        match_type(e, e_type, expected_type) => e'

end
```

Relation `type_convert`

This relations converts the expression in the first argument to the type specified in the third argument. The current type of the expression is given in the second argument.

If no type conversion is possible, this relation fails.

```
relation type_convert : (Exp.Exp, Types.Type, Types.Type) => Exp.Exp =

  axiom type_convert(e, Types.T_INTEGER, Types.T_REAL) => Exp.CAST(Exp.REAL,e)

end
```

Relation `deoverlooad`

Given several lists of parameter types and one argument list, this relation tries to find one list of parameter types which is compatible with the argument list. It uses `elab_arglist` to do the matching, which means that automatic type conversions will be made when necessary. The new argument list, together with a new operator that corresponds to the parameter type list is returned.

The basic principle is that the first operator that matches is chosen.

The third argument to the relation is the expression containing the operation to be deoverloaded. It is only used for error messages.

```
relation deoverload : ((Exp.Operator * Types.Type list * Types.Type) list,
                       (Exp.Exp * Types.Type) list,
                       Absyn.Exp)
          => (Exp.Operator, Exp.Exp list, Types.Type) =

  rule  elab_arglist (params, args) => args'
        ------------------------------------
        deoverload ((op, params, rtype)::_, args, _) => (op, args', rtype)

  rule  deoverload (xs, args, exp) => (op, args', rtype)
        ------------------------------
        deoverload (_::xs, args, exp) => (op, args', rtype)

  rule  print "# Can't resolve type of expression: " &
        Dump.print_exp exp & print "\n"
        ------------------------------
        deoverload([], args, exp) => fail

end
```

Relation `operators`

This relation relates the operators in the abstract syntax to the de-overaloaded operators in the SCode.

```
relation operators : Absyn.Operator
          => (Exp.Operator * Types.Type list * Types.Type) list =
```

Arithmetical operators

```
  axiom operators Absyn.ADD
        => [(Exp.ADD(Exp.INT),
             [Types.T_INTEGER, Types.T_INTEGER],
             Types.T_INTEGER),
            (Exp.ADD(Exp.REAL),
             [Types.T_REAL, Types.T_REAL],
             Types.T_REAL),
            (Exp.ADD(Exp.STRING),
             [Types.T_STRING, Types.T_STRING],
             Types.T_STRING)]

  axiom operators Absyn.SUB
        => [(Exp.SUB(Exp.INT),
             [Types.T_INTEGER, Types.T_INTEGER],
             Types.T_INTEGER),
            (Exp.SUB(Exp.REAL),
             [Types.T_REAL, Types.T_REAL],
             Types.T_REAL)]

  axiom operators Absyn.MUL
        => [(Exp.MUL(Exp.INT),
             [Types.T_INTEGER, Types.T_INTEGER],
             Types.T_INTEGER),
            (Exp.MUL(Exp.REAL),
             [Types.T_REAL, Types.T_REAL],
             Types.T_REAL)]

  axiom operators Absyn.DIV
        => [(Exp.DIV(Exp.REAL),
             [Types.T_REAL, Types.T_REAL],
             Types.T_REAL)]

  axiom operators Absyn.UMINUS
        => [(Exp.UMINUS(Exp.INT),
             [Types.T_INTEGER],
             Types.T_INTEGER),
            (Exp.UMINUS(Exp.REAL),
             [Types.T_REAL],
             Types.T_REAL)]

  axiom operators Absyn.UPLUS
        => [(Exp.UPLUS(Exp.INT),
             [Types.T_INTEGER],
             Types.T_INTEGER),
            (Exp.UPLUS(Exp.REAL),
             [Types.T_REAL],
             Types.T_REAL)]
```

Logical operators

```
axiom operators Absyn.AND
        => [(Exp.AND,
             [Types.T_BOOL, Types.T_BOOL],
             Types.T_BOOL)]

axiom operators Absyn.OR
        => [(Exp.OR,
             [Types.T_BOOL, Types.T_BOOL],
             Types.T_BOOL)]

axiom operators Absyn.NOT
        => [(Exp.NOT,
             [Types.T_BOOL],
             Types.T_BOOL)]
```

Relational operators

```
  axiom operators Absyn.LESS
          => [(Exp.LESS(Exp.INT),
               [Types.T_INTEGER, Types.T_INTEGER],
               Types.T_BOOL),
              (Exp.LESS(Exp.REAL),
               [Types.T_REAL, Types.T_REAL],
               Types.T_BOOL)]

  axiom operators Absyn.LESSEQ
          => [(Exp.LESSEQ(Exp.INT),
               [Types.T_INTEGER, Types.T_INTEGER],
               Types.T_BOOL),
              (Exp.LESSEQ(Exp.REAL),
               [Types.T_REAL, Types.T_REAL],
               Types.T_BOOL)]

  axiom operators Absyn.GREATER
          => [(Exp.GREATER(Exp.INT),
               [Types.T_INTEGER, Types.T_INTEGER],
               Types.T_BOOL),
              (Exp.GREATER(Exp.REAL),
               [Types.T_REAL, Types.T_REAL],
               Types.T_BOOL)]

  axiom operators Absyn.GREATEREQ
          => [(Exp.GREATEREQ(Exp.INT),
               [Types.T_INTEGER, Types.T_INTEGER],
               Types.T_BOOL),
              (Exp.GREATEREQ(Exp.REAL),
               [Types.T_REAL, Types.T_REAL],
               Types.T_BOOL)]

  axiom operators Absyn.EQUAL
          => [(Exp.EQUAL(Exp.INT),
               [Types.T_INTEGER, Types.T_INTEGER],
               Types.T_BOOL),
              (Exp.EQUAL(Exp.REAL),
               [Types.T_REAL, Types.T_REAL],
               Types.T_BOOL),
              (Exp.EQUAL(Exp.STRING),
               [Types.T_STRING, Types.T_STRING],
               Types.T_BOOL),
              (Exp.EQUAL(Exp.BOOL),
               [Types.T_BOOL, Types.T_BOOL],
               Types.T_BOOL)]

  axiom operators Absyn.NEQUAL
          => [(Exp.NEQUAL(Exp.INT),
               [Types.T_INTEGER, Types.T_INTEGER],
               Types.T_BOOL),
              (Exp.NEQUAL(Exp.REAL),
               [Types.T_REAL, Types.T_REAL],
               Types.T_BOOL),
              (Exp.NEQUAL(Exp.STRING),
               [Types.T_STRING, Types.T_STRING],
               Types.T_BOOL),
              (Exp.NEQUAL(Exp.BOOL),
               [Types.T_BOOL, Types.T_BOOL],
               Types.T_BOOL)]


  rule  print "- operators(" & Dump.op_symbol op => s &
        print s & print ") failed\n"
        ----------------------------
        operators op => fail

end
```

A.18 Module **DAE**

This module defines types for collecting DAE equations together with declarations of variables.

A.18.1 Module header

```
module DAE:

  with "absyn.rml"
  with "dump.rml"
  with "exp.rml"
  with "explode.rml"
  with "algorithm.rml"

  type Ident = string

  datatype VarKind = VARIABLE | DISCRETE | PARAM | CONST

  datatype Type = REAL | INT | BOOL | STRING

  datatype Element = VAR of Exp.ComponentRef * VarKind * Type
                   | DEFINE of Exp.ComponentRef * Exp.Exp
                   | EQUATION of Exp.Exp * Exp.Exp
                   | ALGORITHM of Algorithm.Algorithm
                   | COMP of Ident * DAElist

  datatype DAElist = DAE of Element list

  relation dump: DAElist => ()

end
```

A.18.2 Relations

The only relations in this module are for printing, and are not included in the report.

Relation **dump**

This relation prints the DAE in the standard output format.

Relation **pp_statement**

Prettyprint an algorithm statement

A.19 Module `Dump`

Printing routines for debugging of the AST. These relations do nothing but print the data structures to the standard output.

The implementation of the relations are excluded from the report, as they occupy a lot of space and do not convey any useful semantic information.

```
module Dump:

  with "absyn.rml"

  type Ident = string

  (* Dump routines for debugging of AST *)

  relation dump: Absyn.Program => ()
  relation print_exp: Absyn.Exp => ()
  relation print_component_ref: Absyn.ComponentRef => ()
  relation print_path : Absyn.Path => ()
  relation print_subscripts : Absyn.Subscript list => ()
  relation print_arraydim: Absyn.ArrayDim => ()
  relation print_equation: Absyn.Equation => ()
  relation print_algorithm: Absyn.Algorithm => ()
  relation print_list : ('a list, 'a => (), string) => ()
  relation op_symbol : Absyn.Operator => string

end
```

Appendix B

rmldoc

```
#!/home/x97davka/arch/sparc-sun-solaris5.5.1/bin/pike
// -*- pike -*-

object comstart = Regexp("^[ \t]*(\\(|)\\*\\*(.*)$");
object procinst = Regexp("^[ \t]*\\(\\*!([a-z]*)");

string untabify(string s)
{
  string s1, s2;
  while(sscanf(s,"%[^\t]\t%s", s1, s2) == 2)
    s = s1 + String.strmult(" ", 8 - sizeof(s1)%8) + s2;
  return s;
}

string tex_escape(string s)
{
  array a;
  string ss = "";
  s = replace(s,
      ({ "_"  , "{"  , "}"  , "\\" }),
      ({ "\\_", "\\{", "\\}", "\\\\" }));
  while(a=Regexp("([^`]*)`([^']*)'(.*)")->split(s))
  {
    ss += a[0]+"\\code{"+a[1]+"}";
    s = a[2];
  }
  return ss + s;
}

#define OUT(x) outbuf += (x)

void convert(string f)
{
  array(string) lines = Stdio.read_file(f) / "\n";
  string state = "start", buf = "", desc = "";
  string fn = (f/"/")[-1];
  string module = fn - ".rml";
  array a;
  int ignorecode = 0, lno = 0;
  string outbuf = "";

  foreach(lines, string line)
  {
    int empty = Regexp("^[ \t]*$")->match(line);
    lno++;
    // write(sprintf("+%s\n",line));
    if(a = procinst->split(line))
    {
      switch(a[0]) {
      case "ignorecode":
ignorecode = 1; break;
      case "includecode":
```

```
ignorecode = 0; break;
      default:
werror(sprintf("%s:%d: unknown processing instruction: %s\n",
       f, lno, a[0]));
      }
    }
    else if(a = comstart->split(line))
    {
      // write("-com\n");
      string s, newstate;
      switch(state) {
      case "start":
      case "comment":
      case "itemize":
break;
      case "code2":
OUT("\\end{boxedverbatim}\n\n");
break;
      case "code1":
OUT("\n");
break;
      }
      newstate = "comment";
      // Remove extra stuff at the end of the line
      s = a[1];
      if(s[sizeof(s)-2..] == "*)")
s = s[..sizeof(s)-3];
      while(sizeof(s) && s[-1] == ' ') s = s[..sizeof(s)-2];
      // Escape TeX special characters
      s = tex_escape(s);

      if(s==")")
;
      else if(s[..2] == " - ")
      {
OUT("\n\\subsection{"+s[3..]+"}\n\n");
      }
      else if(s[..3] == "  o ")
      {
newstate = "itemize";
if(state != "itemize")
  OUT("\\begin{itemize}\n");
OUT("\\item "+s[4..]+"\n");
      }
      else if(s[..3] == "    " && state == "itemize")
      {
newstate = "itemize";
OUT("      "+s[4..]+"\n");
      }
      else if(s[..1] == "  ")
OUT("\\subsubsection{"+s[2..]+"}\n");
      else if(a = Regexp("^ *([a-zA-Z]*):[ \t]*(.*)")->split(s))
      {
switch(lower_case(a[0])) {
case "file":
  if(a[1] != fn)
  {
    werror("Filename mismatch in file "+f+
   ": claims to be '"+a[1]+"'\n");
    fn = (a[1]/"/")[-1];
    module = fn - ".rml";
  }
  break;
case "module":
  module = a[1];
  break;
case "description":
  desc = a[1];
  break;
case "relation":
  OUT("\\subsubsection{Relation \\code{"+a[1]+"}}\n"
      "\\label{src:"+module+":"+lower_case(a[1]-"\\_")+"}\n");
  break;
case "purpose":
```

```
  OUT("\\textbf{Purpose:} "+a[1]+"\n\\vspace{1ex}\n");
  break;
case "fixme":
  OUT("\\fixme{"+a[1]+"}\n");
  break;
case "rcs":
  OUT("% "+a[1]+"\n");
  break;
default:
  werror(sprintf("%s:%d: Warning: ignored header %s\n",
 f,lno,a[0]));
}
      }
      else
OUT(s+"\n");
      if(state == "itemize" && newstate != "itemize")
OUT("\\end{itemize}\n");
      state = newstate;
    }
    else if(!ignorecode)
    {
      // write("-code: "+state+"\n");
      switch (state) {
      case "code1":
      case "code2":
break;
      case "start":
      case "comment":
// OUT("{\\small\\begin{verbatim}\n");
buf = "";
break;
      }
      line = untabify(line);
      if(!empty)
      {
if(state!="code2")
  OUT("\\begin{boxedverbatim}\n");
state = "code2";
      }
      else if (state != "code2")
state = "code1";

      if(state == "code2")
if(empty)
  buf += "\n";
else
{
  OUT(buf+line+"\n");
  buf = "";
}
    }
  }

  if(state == "code2")
    // OUT("\\end{verbatim}}\n");
    OUT("\\end{boxedverbatim}\n\n");

  write("\\sourcefile{"+module+"}{"+desc+"}\n"+outbuf);
}

int main(int argc, array argv)
{
  foreach(argv[1..], string f)
    convert(f);
}
```

Bibliography

[1] Idonex AB. The pike programming language. `http://pike.idonex.se/`.

[2] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[3] A. V. Aho, R. Sethi, , and J.D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[4] The Modelica design group. Modelica$^{\text{TM}}$ - a unified object-oriented language for physical systems modeling version 1. Available on the World Wide Web at `http://www.Dynasim.se/Modelica/`, September 1997.

[5] The Modelica design group. Modelica$^{\text{TM}}$ - a unified object-oriented language for physical systems modeling version 1f. Available on the World Wide Web at `http://www.Dynasim.se/Modelica/`, July 1998.

[6] Leslie Lamport. *LATEX: A Document Preparation System*. Addison-Wesley, Reading, MA, 1986.

[7] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[8] Modelica home page. `http://www.dynasim.se/Modelica/`.

[9] Frank G. Pagan. *Formal Specification of Programming Languages, a Panoramic Primer*. Prentice Hall, Englewood Cliffs, 1981.

[10] Terence John Parr. *Language Translation Using PCCTS & C++, A Reference Guide*. Automata Publishing Company, san Jose, CA 95129, 1996.

[11] Mikael Pettersson. RML—A new language and implementation for natural semantics. In *Proceedings of PLILP'94*, 1994.

[12] Mikael Pettersson. *Compiling Natural Semantics*. PhD thesis, Linköping University, December 1995.