Modelica Change Proposal MCP-0019

Flattening

(In Development)

Proposed Changes to the Modelica Language Specification

Version 3.3 Revision 1

Table of Contents

Preface

Chapter 1
Introduction

Chapter 2
Lexical Structure

Chapter 3
Operators and Expressions

Chapter 4
Classes, Predefined Types, and Declarations

Chapter 5
Scoping, Name Lookup, and Flattening

<text from original specification>

5.1 Flattening Context

<text from original specification>

5.2 Enclosing Classes

<text from original specification>

5.3 Static Name Lookup

<text from original specification>

5.3.1 Simple name lookup

When an element, equation, or section is flattened, any name is looked up sequentially in each member of the ordered set **of instance scopes (see section 5.6.1.2) corresponding to lexically enclosing classes** until a match is found or an enclosing class is encapsulated. In the latter case the lookup stops except for the predefined types, functions and operators defined in this specification.

Reference to variables successfully looked up in an enclosing class is only allowed for variables declared as constant. The values of modifiers are thus resolved in the **instance** scope of which the modifier appears; if the use is in a modifier on a short class definition,

This lookup in each **instance** scope is performed as follows

- Among declared named elements (`class_definition` and `component_declaration`) of the class (including elements inherited from base-classes).
- Among the import names of qualified import statements in the **instance** scope. The import name of `import A.B.C`; is `C` and the import name of `import D=A.B.C;` is `D`.
- Among the public members of packages imported via unqualified import-statements in the **instance** scope. It is an error if this step produces matches from several unqualified imports.

Import statements defined in inherited classes are ignored for the lookup, i.e. import statements are not inherited.

5.3.2 Composite name lookup

<text from original specification>

5.4 Instance Hierarchy Name Lookup of Inner Declarations

<text from original specification>

5.5 Simultaneous Inner/Outer Declarations

<text from original specification>

5.6 Flattening Process

In order to guarantee that elements can be used before they are declared and that elements do not depend on the order of their declaration (Section 4.3) in the enclosing class, the flattening proceeds in the following two major steps:

1. Instantiation process
2. Generation of the flat equation system

The result is an equation system of all equations/algorithms, initial equations/algorithms and instances of referenced functions. Modifications of constants, parameters and variables are included in the form of equations.

The constants, parameters and variables are defined by globally unique identifiers and all references are resolved to the identifier of the referenced variable. No other transformations are performed.

5.6.1 Instantiation

The instantiation is performed in two steps. First a class tree is created and then from that an instance tree for a particular model is built up. This forms the basis for derivation of the flat equation system.

An implementation may delay and/or omit building parts of these trees, which means that the different steps can be interleaved. If an error occurs in a part of the tree that is not used for the model to be instantiated the corresponding diagnostics can be omitted (or be given). However, errors that should only be reported in a simulation model must be omitted there, since they are not part of the simulation model.

5.6.1.1 The Class Tree

All necessary libraries including the model which is to be instantiated are loaded from e.g. file system and form a so called **class tree**. This tree represents the syntactic information from the class definitions. It contains also all modifications at their original locations in syntactic form. *[The class tree is built up directly during parsing of the Modelica texts. For each class a local tree is created which is then merged into the one big tree, according to the location of the class in the class hierarchy. This tree can be seen as the abstract syntax tree (AST) of the loaded libraries.]*. The builtin classes are put into the unnamed root of the class tree.

5.6.1.2 The Instance Tree

The output of the instantiation process is an **instance tree**. The instance tree consists of nodes representing the elements of a class definition from the class tree. For a component the subtree of a particular node is created using the information from the class of the component clause and a new modification environment as result of merging the current modification environment with the modifications from the current element declaration (see 7.2.3).

The instance tree has the following properties:

- It contains the instantiated elements of the class definitions, with redeclarations taken into account and merged modifications applied.
- Each instance knows its source class definition from the class tree and its modification environment.
- Each modification knows its instance scope.

The instance tree is used for lookup during instantiation. To be prepared for that, it has to be based on the structure of the class tree with respect to the class definitions. The builtin classes are instantiated and put in the unnamed root prior to the instantiation of the user classes, to be able to find them.

[The existence of the two separate trees (instance tree and class tree) is conceptual. Whether they really exist or are merged into only one tree or the needed information is held completely differently is an implementation detail. It is also a matter of implementation to have only these classes instantiated which are needed to instantiate the class of interest.]

A node in the instance tree is the instance scope for the modifiers and elements syntactically defined in the class it is instantiated from. The instance scope is the starting point for name lookup. *[If the name is not found the lookup is continued in the instance scope corresponding to the lexically enclosing class. Extends clauses are treated as unnamed nodes in the instance tree – when searching for an element in an instance scope the search also recursively examines the elements of the extends clauses. Except that inherited import-statements are ignored.]*

5.6.1.3 The Instantiation Procedure.

The instantiation is a recursive procedure with the following inputs:

- the class to be instantiated (current class)
- the modification environment with all applicable redeclarations and merged modifications (initially empty)
- a reference to the node of the instance tree, which the new instance should go into (parent instance)

The instantiation starts with the class to be instantiated, an empty modification environment, and a unnamed root node as parent node.

During instantiation all lookup is performed using the instance tree, starting from the instance scope of the current element. References in modifications and equations are resolved later (during generation of flat equation system) using the same lookup.

5.6.1.4 Steps of Instantiation

The element itself

A partially instantiated class or component is an element that is ready to be instantiated; a partially instantiated element (i.e. class or component) is comprised of a reference to the original element (from the class tree) and the modifiers for that element (including a possible redeclaration).

The possible redeclaration of the element itself takes effect.

The class of a partially instantiated component is found in the instance tree (using the redeclaration if any), modifiers merged to that class forming a new partially instantiated class that is instantiated as below.

The local contents of the element

For local classes and components in the current class, instance nodes are created and inserted into the current instance. Modifiers (including class redeclarations) are merged and associated with the instance and the element is partially instantiated. *[The partially instantiated elements are used later for lookup during the generation of the flat equation system and are instantiated fully, if necessary, using the stored modification environment.]*

Equations, algorithms, and annotations of the class and the component declaration are copied to the instance without merging. *[The annotations can be relevant for simulations, e.g. annotations for code generation (18.3.), simulation experiments (18.4) or functions(12.7.- 12.9).]*

The inherited contents of the element

Classes of extends clauses of the current class are looked up in the instance tree, modifiers (including redeclarations) are merged, the contents of these classes are partially instantiated using the new modification environment, and are inserted into the extends clause node, which is an unnamed node in the current instance that only contains the inherited contents.

The classes of extends-clauses are looked up before and after handling extends-clauses; and it is an error if those lookups generate different results.

At the end, the current instance is checked whether their children (including children of extends-clauses) with the same name are identical and only the first one of them is kept. *[This is important for function arguments where the order matters.]* It is an error if they are not identical.

Recursive instantiation of components

Components (local and inherited) are recursively instantiated.

[

As an example consider:

```
model M
    model B
      A a;
      replaceable model A=C;
      type E=Boolean;
    end B;
    B b(redeclare model A=D(p=1));
    partial model C
      E e;
    end C;
    model D
      extends C;
      parameter E p;
      type E=Integer;
    end D;
    type E=Real;
  end M;
```

To recursively instantiate M allowing the generation of flat equation system we have the following steps (not including checks):

1. *Instantiate M: which partially instantiates B, b, C, D, E.*
2. *Instantiate M.b:*
   - *2.1. First find the class B in M (the partially instantiated elements have correct name allowing lookup)*
   - *2.2. instantiate the partially instantiated M.B with the modifier "redeclare model A=D(p=1)"*
   - *2.3. partially instantiate M.b.a (no modifier), and M.b.A (with modifier "=D(p=1)")*
3. *Instantiate M.b.a*
   - *3.1. First find the class A in M.b (the partially instantiated elements have correct name allowing lookup)*
   - *3.2. Instantiate the partially instantiated M.b.A with the modifier "=D(p=1)".*
     - *3.2.1. Find the base-class "=D" from the modifier. This performs lookup for D in M, and finds the partially instantiated class D*
     - *3.2.2. Instantiate the base-class M.D with modifier p=1, and insert as unnamed node in M.b.A.*
       - *3.2.2.1. Partially instantiate the component p with modifier "=1"*
       - *3.2.2.2. Find the base-class "C" in M.D. Since there is no local element called "C" the search is then continued in M and finds the partially instantiated class M.C*
       - *3.2.2.3. Instantiate the base-class M.C as below*
4. *Instantiate the base-class M.C inserting the result into unnamed node in M.b.a*
   - *4.1. Partially instantiate "e"*

4.2. Instantiate "e" which requires finding "E". First looking for "E" in the un-named node for extends "M.C", and, since there is no local element "E" the search is then continued in "M" (which lexically encloses M.C) and finds "E" class inheriting from Real. The "e" is then instantiated using class "E" inheriting from "Real".

5. Instantiate M.b.a.p

5.1. First the class "E" in M.b.a finding "E" class inheriting from Integer.

5.2. Instantiate the "M.b.a.p" using the class "E" inheriting from Integer with modifier "=1"

5.3. Instantiate the base-class Integer with modifier "=1", and insert as unnamed node in "M.b.a.p".

An implementation can use different heuristics to be more efficient by re-using instantiated elements as long as the resulting flat equation system is identical.

Note that if "D" was consistently replaced by "A" in the example above the result would be identical (but harder to read due to two different classes called "A").

]

5.6.2 Generation of the flat equation system

During this process, all references by name in conditional declarations, modifications, dimension definitions, annotations, equations and algorithms are resolved to the real instance to which they are referring to, and the names are replaced by the global unique identifier of the instance. *[This identifier is normally constructed from the names of the instances along a path in the instance tree (and omitting the unnamed nodes of extends clauses), separated by dots. Either the referenced instance belongs to the model to be simulated the path starts at the model itself, or if not, it starts at the unnamed root of the instance tree, e.g. in case of a constant in a package.]*

[To resolve the names, a name lookup using the instance tree is performed, starting at the instance scope (unless the name is fully qualified) of the modification, algorithm or equation. If it is not found locally the search is continued at the instance of the lexically enclosing class of the scope [this is normally not equal to the parent of the current instance], and then continued with their parents as described in section 5.3. If the found component is an outer declaration, the search is continued using the direct parents in the instance tree (see section …). If the lookup has to look into a class which is not instantiated yet [or only partially instantiated], it is instantiated in place.]

The flat equation system consists of a list of variables with dimensions, flattened equations and algorithms, and a list of called functions which are flattened separately. A flattened function consists of algorithm or external clause and top-level variables – which recursively can contain other variables; the list of non-top level variables is not needed.

The instance tree is recursively walked through as follows for elements of the class (if necessary a partially instantiated component is first instantiated):

- At each visited component instance, the name is inserted into the variables list. Then the conditional declaration expression is evaluated if applicable.
    - The variable list is updated with the actual instance
    - The variability information and all other properties from the declaration are attached to this variable.
    - Dimension information from the declaration and all enclosing instances are resolved and attached to the variable to define their complete dimension.
    - If it is of record or simple type (Boolean, Integer, enumeration, Real, String, Clock, ExternalObject):
        - In the modifications of **value** attribute references are resolved using the instance scope of the modification. An equation is formed from a reference to the name of the instance and the resolved modification value of the instance, and included into the equation system. Except if

the value for an element of a record is overridden by the value for an entire record; section 7.2.3.

  - If it is of simple type (Boolean, Integer, enumeration, Real, String, Clock, ExternalObject):
    - In the modifications of **non-value** attributes, e.g. start, fixed etc. references are resolved using the instance scope of the modification. An equation is formed from a reference to the name of the instance appended by a dot and the attribute name and the resolved modification value of the instance, and included into the equation system.
  - If it is of a non-simple type the instance is recursively handled.

- If there are equation or algorithm sections in the class definition of the instance, references are resolved using the instance scope of the instance and are included in the equation system. Some references – in particular to non simple, non record objects like connectors in connect statements and states in transition statements are not resolved yet and handled afterwards.
- Instances of local classes are ignored.
- The unnamed nodes corresponding to extends-clauses are recursively handled.
- If there are function calls encountered during this process, the call is filled up with default arguments as defined in 12.4.1. These are built from the modifications of input arguments which are resolved using their instance scope. The called function itself is looked up in the instance tree. All used functions are flattened and put into the list of functions.
- Conditional components with false condition are removed afterwards and they are not part of the simulation model. *[Thus e.g. parameters don't need values in them. However, type-error can be detected.]*
- Each references is checked, whether it is a valid reference, e.g. the referenced object belongs to or is an instance, where all existing conditional declaration expressions evaluate to true or it is a constant in a package. *[Conditional components can be used in connect-statements, and if the component is conditionally disabled the connect-statement is removed.]*

This leads to a flattened equation system, except for connect and transition statements. These have to be transformed as described in chapter 9 and chapter 17. This may lead to further changes in the instance tree *[e.g. from expandable connectors (sections 9.1.3)]* and additional equations in the flattened equation system *[e.g. connect equations (sections 9.2), generated equations for state machine semantics (section 17.3.4)]*.

[After flattening, the resulting equation system is self contained and covers all information needed to transform it to a simulatable model, but the class and instance trees are still needed: in the transformation process, there might be the need to instantiate further functions, e.g. from `derivative` *annotation or from* `inverse` *annotation etc., on demand.]*

Chapter 6
Interface or Type Relationships

Chapter 7
Inheritance, Modification, and Redeclaration

Chapter 8
Equations

Chapter 9
Connectors and Connections

Chapter 10
Arrays

Chapter 11
Statements and Algorithm Sections

Chapter 12
Functions

Chapter 20
The Modelica Standard Library

Appendix A
Glossary

Appendix B
Modelica Concrete Syntax

Appendix C
Modelica DAE Representation

Appendix D
Derivation of Stream Equations