Modelica: A Reference Manual

– Or, Baby-Modelica Implementation Notes –

Baby-Modelica 3.4.0, 2021-03-18

Contents

1 Introduction

1.1 Baby-Modelica

Baby-Modelica is a simple frontend (parser+flattener) for the Modelica language specification 3.4. It parses a source code then performs simple syntactic transformations, and optionally dumps a flat model. Flattening is indispensable even for simple tools, because taking parameter values is not direct as they may be substituted multiple times by nested parameterization. Flattening is a major part of Modelica, and it resolves all the features as an object-oriented language. A generated flat model is hopefully to be re-readable by Modelica compilers.

Current status of Baby-Modelica is a pre-zero version, and we are working towards version zero. The version number of Baby-Modelica is the last digits, appended to the version number of the Modelica language specification.

This random memo accompanies the source code. This document was written as implementation notes but has grown with addendums to the specification. So, it is now renamed as a reference manual. This document often refers to the sections of the specification. We apologize if it is hard to read.

1.2 Useful documents

- *Modelica Language Specification* [1] is the source of the definitions. Note that some concepts are defined not in the text but in the glossary.
- *OpenModelica System Documentation* [2] explains the implementation of OpenModelica. See the section *Structure*, for Parse and Elaborate/Instantiate/Flatten.
- *DSblock: A neutral description of dynamic systems* [5] is an old description of structuring interfaces to solvers.
- *Translator to flat Modelica* [4] describes some notes about a translator from Modelica subset to flat Modelica, which is under development as a tool for AMESim.

- *SysML–Modelica Transformation* [3] describes a translator from SysML to Modelica. (The overview says bi-directional but it seems not). There is an accompanying paper [6].

2 Clarification to the specification

Some rules are assumed in the implementation on syntaxing.

2.1 Variable references in general

Composite names

A variable (instance) reference is in the form like $x[...].y[...].z[...]$. Subscripts are optional in a variable reference. A class reference is like $A.B.C$. Subscripts are not allowed in a class reference. Classes A, B, and C can be packages or non-packages. A constant refererence in a class is like $A.B.C.a[...].b[...].c[..]$, where all a, b, and c are constants and subscripts are optional.

A non-constant variable (instance) reference does not have any prefixes of classes nor a preceding dot.

A class reference may be with a preceding dot, which refers to the unnamed root. Or, a class reference may follow a variable reference, in that case classes are considered as packages.

Rule 1. There is a single namespace for classes and variables. It is allowed to refer to a constant in a package via a variable reference as $x.P.c$. But, it is illegal to refer to a class via a variable reference in a declaration as (illegal) $x.C\ v$. ▮

Fact 2. An enclosing class is considered as a package in the nesting of class definitions, and the scope of an enclosing class only includes the constants but variables are ignored. ▮

See Rule 19 for the binary roles of a class.

Fact 3. Array indexing is a part of a composite name. Array indexing to general expressions is ungrammatical. ▮

For example, (illegal) $\{1, 2, 3\}[1]$ is not allowed.

Predefined variables

Rule 4. A variable *time* is visible everywhere. Variables *end* are visible in array indexing. These are the only predefined variables. *time* is a variable, and thus, not defined in the unnamed-enclosing-class (it is a package). ▮

```
/*ILLEGAL*/
model M
    Real x;
    equation
    x = .time;
end M;
```

(?) Scopes

The scope is flat in most cases, spanning from the main (non-base) class to the base classes where the class and variable names are distinct. However, class names and constants are visible via enclosing classes, and hiding some of the names in an enclosing class may happen. An enclosing class is defined to each main and base class.

2.2 Importing

Importing makes visible the names of the definition/declaration elements of the specified package to the present class. The inclusion is from any classes but they are processed as packages. The inclusion is limited to the public elements. This inclusion recurses transitively to the base classes. This inclusion does not include the imported names in the base classes nor the names in the enclosing classes in the base classes.

Import-clause syntax

See Specification 13.2.1 *Importing Definitions from a Package*. A package name in an import-clause is considered as fully qualified. It can import only from packages. In the following, *package* is a sequence of identifiers *package* = $P_0. \ldots .P_{n-1}$. Each prefix in $P_0 \ldots .P_i$ refers to a class, and P_{n-1} refers to a package. *package* can be empty. Each P_i and N refers to an indentifier.

import N' = *package*.N (renaming import/renaming single definition import): makes a class (including a package) or a constant N visible as an identifier N'. There is no syntactic difference when N refers to a class/package or a constant.

import *package*.$*$ (unqualified import): imports each public name visible in the package including one in the base classes as "**import** N_i = *package*.N_i" for N_i. The package part can not be empty in this case.

import *package*.N (qualified import/single definition import): is equivalent to "**import** N = *package*.N".

import *package*.$\{N_0, N_1, ..., N_{m-1}\}$ (multiple definition import): is equivalent to "**import** N_i = *package*.N_i" for each i.

2.3 Extending

Extending includes the (definition/declaration) elements of the base classes to the present class. This inclusion recurses transitively to the base classes. This inclusion does not include the imported names in the base classes nor the names in the enclosing classes in the base classes. The visibility is nothing to do with the bases because it is either **public** or **protected**.

Checks for extends clauses

Rule 5. It is an error if a base class A of **extends** A is defined in the class A itself or in other base classes. This is broadly interpreted as a lookup of A skips extends-clauses. ▮

This may be a result of requiring the uniqueness of resolution of a class name. When another class A were defined in A in the code below, two occurrences of A would refer to the different classes, which contradicts to an intuitive assumption that the both A are the same.

```
model M
    extends A;
    A a;
end M;
```

Quote 1. (Specification 5.6.1.4 *Steps of Instantiation*) 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.

Mutual dependence of extends-clauses

Rule 6. Mutual dependence of extends-clauses is prohibited. ▮

(?) The specification does not likely state it explicitly. The specification states that inheritance is unified in cases where a class extends a common base class through distinct intermediate base classes. Mutual dependence could be resolved by identifying.

Extending a single base multiple times (?)

Rule 7. (?) Extending a single class multiple times (usually via a common base class) is allowed only if the modifiers to it are identical. It is an error if the modifiers differ. ▮

It is ambiguous what identical means. Textual equality may not be sufficient with class modifications. Consider "extends A(x=a)" and "extends A(x=b)" with "a=0" and "b=0".

(?) It may require unifying inherited elements. Classes and variables multiply inherited are unified.

> **Quote 2**. (Specification 5.6.1.4 *Steps of Instantiation*, in the paragraph *The inherited contents of the element*) At the end, the current instance is checked whether their children with the same name are identical and only the first one of them is kept. It is an error if they are not identical. (* A "child" means a component. *)

(* The implementation does not check the equivalence of modifiers, and arbitrary selects one modification. *)

Operator record

Operator records have restrictions on extends-clauses.

> **Quote 3**. (Specification 4.6 *Specialized Classes*, in the table) (?) It is not legal to extend from any of its enclosing scopes. (* *enclosing* likely refers to *enclosed* *).

No scopes of bases in a lookup of a base class

Rule 8. A lookup of a base class name skips bases of the present class. It applies to a lookup of both the first part and the remaining parts of a composite name. ▮

```
/*ILLEGAL*/

model A
    model B end B;
end A;

model M0
    extends B;
    extends A;
end M0;

model M1
    extends M1.B;
    extends A;
end M1;
```

Remark Note the first case is accepted with warnings in some implementations, and the both cases will be accepted, when the extends-clauses are swapped, i.e., when **extends** *A* comes early.

Unifying defined classes

Rule 9. It is not an error if classes with the same name are defined multiple times in the main class and its bases, when the definitions are considered identical. Identical classes are a sort of unified. It is defined in the specification that classes are identical if they are textually identical after possible modifications. ▮

For example, in Fluid/Interfaces, the declarations of *Medium* need be unified, when *PartialPump* extends both *PartialTwoPort* and *PartialLumpedVolume*.

```
partial model PartialTwoPort
    replaceable package Medium =
        Modelica.Media.Interfaces.PartialMedium;
    ......
end PartialTwoPort;

partial model PartialLumpedVolume
    replaceable package Medium =
        Modelica.Media.Interfaces.PartialMedium;
    ......
end PartialLumpedVolume;
```

PartialPump is defined in *Modelica.Fluid.Machines.BaseClasses*.
PartialTwoPort and *PartialLumpedVolume* are defined in *Modelica.Fluid.Interfaces*.

Extending simple types

Fact 10. Extending simple types is legal either by a short class definition or by an extends-clause. But adding elements is illegal. ▮

Restrictions on elements

> **Quote 4**. (Specification 4.5.2 *Restriction on combining base-classes and other elements*) It is not legal to combine other components or base-classes with an extends from an array class, a class with non-empty base-prefix, a simple type...

2.4 A lookup of a class name

Resovling class names is better understood not by a scope but by a lookup procedure. A lookup of a class name happens in import-clauses, extends-clauses, component declarations, and composite name references in expressions. A lookup of import-clauses and extends-clauses has a different procedure, and is performed in this order because one depends on the predecessor. A lookup is performed after applying renaming modifiers/redeclarations to the present class.

A lookup of a composite class name starts from a lookup of the first part followed by lookups of the remaining parts. A lookup of the remaining parts is performed in the class found in the previous lookup.

A lookup considers the scopes of the following:

- declared elements (s0). A scope of the declared elements of the present class is closed in the class definition.

- imported names (s1). A scope of the imported names transitively extends to the declared elements of the base classes.
- base classes (s2). A scope of the base classes transitively extends to the declared elements of the base classes.
- an enclosing class (s3). A scope of an enclosing class consists of the declared elements, the imported names, and the base classes, and transitively extends to its enclosing classes.

Fact 11. A lookup first tries to find in the declared elements (s0), and if it finds a name, other scopes are skipped. See Qoute 5. ▮

Especially, this rule is needed to search for *Icons* in *.Modelica* which extends *.Modelica.Icons.Package*. The scope of *.Modelica* should include *.Modelica.Icons.Package* but it avoids a cycle by skipping the base classes.

Fact 12. A lookup of a first part of a name for an import-clause and an extends-clause is not affected by redeclarations. See Rule 39 and Rule 40. A lookup respects class redeclarations for an extends-clause in some implementations. ▮

Fact 13. A lookup of the remaining parts respects class redeclarations in modifiers and in elements. It also respects outer prefixes. ▮

2.5 A lookup for importing

A package name here denotes a class name of an import-clause.

Rule 14. A lookup of the first part of a package name is performed in the root of the namespace (the unnamed-enclosing-class), since the package name is considered as fully-qualified. The root has no imports nor bases (s0).

A lookup of the remaining parts is performed in the declared elements and in the base classes, but not in the imported names nor in the enclosing classes (s0+s2). A lookup of a remaining part has a restriction. See Rule 18. ▮

2.6 A lookup for extending

Rule 15. A lookup of the first part of a name of a base class does not include the base classes, but does include the imported names and the enclosing classes (s0+s1+s3). Note that the base classes of the imported classes and the base classes of the enclosing classes are included. The names in the enclosing classes can be hidded by the other names.

A lookup of the remaining parts is performed in the declared elements and in the base classes, but not in the imported names nor in the enclosing classes (s0+s2). A lookup of a remaining part has a restriction. See Rule 18. ▮

Remark Some implementations find a base class through another extends-clause, although it reports the code is illegal.

> **Quote 5**. (Specification 5.6.1.4 *Steps of Instantiation*; in the paragraph "The inherited contents of the element") ... 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.

A lookup in the base classes is only necessary for error detection. A lookup in the base classes can simply be ignored in a lookup of a class of an extends-clause.

Ignoring base classes

A lookup of a base class name searches in the imported names but may not search in the base classes in some cases.

Rule 16. (A lookup of a base class) A lookup of a base class name ignores the base classes in the present class and the element classes transitively, where the present class is the class in which the lookup starts. A lookup also likely ignores mutually dependent extends-clauses. ▮

Ignoring the base classes in the present class is a consequence of the uniqueness rule of extends-clauses (in Qoute 5). It may also be deduced from the flattening process (Specification 5.6 *Flattening Process*). The description in *The local contents of the element* states the element classes are inserted into the instance tree whose extends-clauses seems to be empty at the time. The lookup of a class of an extends-clauses of the present class is perfomed with that state of the instance tree.

2.7 A lookup of a component class

A lookup of a class name of a component declaration, or a variable name or a class name in an equation or a statement starts at the class, where a component/equation/statement is in.

Rule 17. A lookup of the first part of a component class name is performed in the declared elements, in the imported names, in the base classes, and the enclosing classes (s0+s1+s2+s3).

A lookup of the remaining parts is performed in the declared elements and in the base classes, but not in the imported names nor in the enclosing classes (s0+s2). ▮

2.8 A short-cut in a lookup

A cycle in a lookup for importing/extending

It is common to import/extend a package defined in the class or in its subpackages. Such a package reference, when it appears in a remaining part of a composite name, may make a circular dependency. It is due to the rule for a lookup of a remaining part, in Rule 14 or Rule 15.

Consider processing a toplevel package M below. A lookup of $M.I.P$ in M first finds M, and then a lookup continues for I in M. It may need to look into bases of M, because I is a remaining part of the name. However, it is problematic because resolving the bases of M is underway.

```
package M
    package I
        package P end P;
    end I;
    extends M.I.P;
end M;
```

A similar code can be found in MSL 3.2.3, where M as Modelica, I as Icon, and P as Package.

Similarly for importing, searching C in $M.M.W$ may need to look into the imported definitions.

```
within M.M;
package W
    package C ... end C;
    import M.M.W.C.s;
end W;
```

A similar code can be found in MSL 3.2.3, where $M.M$ as Modelica.Media, W as Water, and C as ConstantPropertyLiquidWater.

A short-cut in a lookup for an element class

Rule 18. Lexically enclosing classes (the relation that is not extended by importing and extending) cannot affect the enclosed class by modifications. Thus, it is safe to use the textual definition for such classes. ▮

For example, consider the case M extends $M.I.P$ above. As $M.I$ is textually defined, it is not necessary to process M first to take the definition of $M.I$. Thus, it avoids a potential cycle in the definition.

2.9 Binary roles of a class

A class is used both as a type of an instance or as a package. It is usually instantiated directly or via extends-clauses. It is considered as a package, when a class name appears in the middle of a composite name. Semantics of a class differs in these uses.

Model nesting

A non-package class can contain a class, and it is possible to instantiate an enclosed one. Here, to instantiate a class means to declare a variable of it or to use it as a target model.

Rule 19. It is legal to instantiate a class enclosed by a non-package class, but the accessible variables in the enclosing class are restricted to constants. This includes variables used in modifiers. That is, an enclosing class is processed as a package in such a case, where declaring variables is permitted but using them is prohibited. ▮

It means that it is not possible to share a lexically visible variable from enclosed classes as usually expected.

In the example below, assume instatiating M, where a class M contains a class $M.A$. Only constants in M are accessible from A, and it is illegal. If x were a constant, it is legal. This restriction also exists when using a class $M.A$ as a target model.

```
/*ILLEGAL*/
model M
    /*constant*/ Real x=10;
    model A
        Real v=x;
    end A;
    A a;
end M;
```

Processing a class as a package

Rule 19 implies a fact:

Fact 20. An enclosing class of a class is processed as a package in a way that is different from a usual way. Variable declarations and outer class definitions are treated as non-existent when a class is processed as a package. (?) Outer constants are included. (?) Is it legal to prefix constants by outer? ▮

Note that a class is processed as a package during a lookup of intermediate names in a composite name.

Rule 21. Binary roles means allowance of illegal elements. For example, outer elements are illegal in a package, but it is allowed unless it is used. ▮

2.10 Modifications

Modification syntax

There is no syntactic difference of a modification on a class or a variable: $A(...)$ or $x(...)$. A class form applies to a declaration **class** A ...**end** A or **class** A = ..., and a variable form applies to a declaration $B\ x$.

Simple examples are:

```
model A
    model B = C(..0..);
end A;

model M
    model D = A(B(..1..));
    /* D.B = C(..0.. + ..1..); */
end M;
```

Fact 22. The LHS of modifiers is an identifier (possibly, some parenthesized modifiers may be attached to the LHS). Especially, the LHS does not admit array indexing. ▮

Modifications to enumerations or der-classes

Rule 23. (?) Modifications to der-classes are illegal. ▮

The simple types accept modifiers to their attributes. (?) For other predefined types such as *Clock*, *StateSelect*, *ExternalObject*, *AssertionLevel*, and *Connections*, it is not checked. Note that the grammar in the specification defines initializers (e.g., in $C\ x = e$) as a form of modification, and they modify the *value* attribute.

Merging modifiers

Muliple modifications can be applied to the same class.

Fact 24. (?) Modifications applied to the same class have the same **each**, **final**, **redeclare**, and **replaceable** statuses. ▮

Resolving names in modifiers

Resolving names in modifiers differs in the LHS and the RHS of modifiers. The LHS of modifiers means x in $x = e$ or in $x(e)$. Note that the LHS is an identifier always. Also, it differs between modifiers in class refinings and extends-clauses.

A rule of resolving names in the LHS of modifiers is:

Rule 25. Resolving the LHS in modifiers is done in the scope of the modified class (a in A in the example below). See Example: Scope for modifiers in Specification 7.2 *Modifications*. The LHS in the modifiers is visible as a derived class in an extends-clause, while it is external in a class refining. Thus, protected elements are not visible in a class refining. See (Specification 4.1 *Access Control – Public and Protected Elements*). ▮

A rule of resolving names in the RHS of modifiers is:

Rule 26. Resolving the RHS is done in the surrounding scope of the present class as usual. It applies to both classes and variables. It implies name resolution is postponed till expanding base classes. ▮

For example, a and x in the modifier $A(a = x)$ are visible, where a is resolved in the LHS rule, and x in the RHS rule.

```
/*LEGAL*/
model A
    Real a=0;
    constant Real x=1;
end A;

model M0
    extends X;
    model X=A(a=x);
end M0;

model M1
    extends A(a=x);
end M1;
```

It should be noted that modifiers to extends-redeclarations have different scoping rules. See Rule 50.

Value modifiers

The components of the simple types are specially call as attributes.

Fact 27. An attribute named *value* is defined as a parameter in the simple types, which holds a value of the type in the usual sense (the simple types are Real, Integer, Boolean, String, and enumerations). A modifier to the value is usually used to define a new type of a simple type with a default value. Specifying both a value modifier and an initializer modifier simultaneously is illegal. ▮

```
/*ILLEGAL*/
model M
    Real v (value=1.0)=2.0;
end M;
```

Independence of importing and extending from modifications

Fact 28. Resolving a class of an import-clause or extends-clause can be performed ignoring modifications. It is because a replaceable class is not allowed for an import-clause or an extends-clause. ▮

Note redeclarations (both class and variable) are type parameterization.

> **Quote 6**. (Specification 5.6.1.4 *Steps of Instantiation*) The possible redeclaration of the element itself takes effect (* in the step of "the element itself" *).

> **Quote 7**. (Specification 5.6.1.4 *Steps of Instantiation*) Extends clauses are not looked up, but empty extends clause nodes are created and inserted into the current instance ...

Initializers

Initializers attached to declarations like $C\ x = e$ are modifiers. An initializer modifier to classes is a set of modifiers to each of the components. However, an initializer modifier to the simple types is treaded as a modifier to the value attribute. Despite the fact that the simple types are composite classes, their components (specially called as attributes) are treated differently from other classes. See Quote 52.

Fact 29. An initializer modifier to a simple type only modifies the value attribute. ▮

Rule 30. An initializer modifier needs to have the same components in the RHS and LHS. Especially, an initializer modifier by a base instance is not allowed when some components are added. ▮

An initializer modifier *x1 = x0* below is illegal because of an addition of s. It is also illegal even if s were a constant. Additions of non-component definitions are allowed.

```
/*ILLEGAL*/

model B
    Real b=10;
end B;

model S
    extends B;
    Real s=20;
end S;

model M
    B x0;
    S x1=x0;
end M;
```

Rule 31. An initializer modifier and modifiers can coexist. Modifiers are ignored in that case. ▮

A modifier ($x = 20$) is ignored in the following example.

```
/*LEGAL*/
model A
    Real x=10;
    Real y=10;
end A;

model M
    A x0;
    A x1(x=20)=x0;      /* x1.x=10 */
end M;
```

2.11 Ordering of modifications

Modifier ordering

Rule 32. A latter application of modifiers has a precedence. The ordering is defined by a way how implementations should look at the modifications. A component declaration is considered latter than a modifier application. ▮

For example, $x0 = 10$ is effective over $X(value = 20)$.

```
model A
    model X=Real;
    X x0=10;            /* a.x0=10 */
    X x1;               /* a.x1=20 */
end A;

model M
    A a (X(value=20));
end M;
```

Quote 8. (Specification 4.4.2 *Component Declaration Static Semantics*) The new environment is the result of merging • the modification of enclosing class element-modification with the same name as the component • the modification of the component declaration in that order.

Redeclaration ordering

Fact 33. Ordering of modifications is inner to outer as strict applications. ▮

In the following example, a redeclaration $X = D$ in elements is applied later than and effective over $X = E$ in a modifier.

```
model C Real v=10; end C;
model D Real v=20; end D;
model E Real v=30; end E;

model A
    replaceable model X=C;
    X a;        /* X=D */
end A;

model M
    redeclare replaceable model X=D;
    extends A (redeclare replaceable model X=E);
end M;
```

In the following example, a redeclaration $X = E$ in M is applied later than and effective over $X = D$ in B.

```
model C Real v=10; end C;
model D Real v=20; end D;
model E Real v=30; end E;

model A
    replaceable model X=C;
    X a;        /* X=E */
end A;

model B
    redeclare replaceable model X=D;
    extends A;
```

```
end B;

model M
    redeclare replaceable model X=E;
    extends B;
end M;
```

Modifiers over redeclarations

Rule 34. A modifier attached to a replaceable is effective after a redeclaration on an instance. In contrast, A modifier is dropped after a redeclaration on a class. ▮

For example, a modifier $x = 30$ to a variable is effective after a redeclaration $D\ a$.

```
model C Real v=10; end C;
model D Real v=20; end D;

model A
    replaceable C a (v=30);
end A;

model M
    extends A (redeclare D a);
    /* a.v=30 */
end M;
```

In contrast, a modifier $v = 30$ to a class is dropped by a redeclaration $X = D$.

```
model C Real v=10; end C;
model D Real v=20; end D;

model A
    replaceable model X=C (v=30);
    X a;
end A;

model M
    redeclare model X=D;
    extends A;
    /* a.v=20 */
end M;
```

Remark Furthermore, in some implementations, a modifier $v = 30$ to a class is kept when a redeclaration is in a modifier.

```
model C Real v=10; end C;
model D Real v=20; end D;

model A
    replaceable model X=C (v=30);
    X a;
end A;
```

```
model M
    extends A (redeclare model X=D);
    /* a.v=30 */
end M;
```

2.12 Redeclarations

A class redeclaration redefines a class, and a component redeclaration redeclares a variable to be another class. Or, a redeclaration changes dimension sizes. Redefinitions and redeclarations may be jointly called redeclarations.

> **Quote 9**. (Specification 7.2 *Modifications*) A more dramatic change is to modify the type and/or the prefixes and possibly the dimension sizes of a declared element. This kind of modification is called a redeclaration...

Fact 35. **redeclare** is always required for a class definition in modifiers. ▮

Simply, it is an error to omit **redeclare** in modifiers.

```
/*ILLEGAL*/

model C Real v=10; end C;
model D Real v=20; end D;

model A
    X a;
end A;

model M
    extends A (/*redeclare*/ class X=D);
end M;
```

A target of a redeclaration

Note that a target (replaceable) class of a redeclaration cannot be defined at the same class, which results in duplicate definitions.

Rule 36. Redeclaring a class/variable in class elements applies to its base classes. Redeclaring a class/variable in modifiers applies to the main class and its base classes. In both cases, a replaceable class/variable in bases can be redeclared. It means that redeclaring a class definition is visible from base classes, although a usual class definition is not visible. ▮

Fact 37. Co-locating redeclare and replaceable definitions is duplicate definitions and an error. Thus, a replaceable definition should be an inherited class (which is defined in a base class). ▮

Redeclaring $X = D$ in M is visible from a base class A.

```
model C Real v=10; end C;
model D Real v=20; end D;

model A
    replaceable model X=C;
    X a; /* X=D */
```

```
end A;

model M
    redeclare model X=D;
    extends A;
end M;
```

Fact 38. A redeclaration replaces a definition of a replaceable. A replaceable defined outside is not visible. ▮

```
/*ILLEGAL*/

model C Real v=10; end C;
model D Real v=20; end D;

replaceable model X=C;

model A
    X a;
end A;
model M
    extends A (redeclare model X=D);
end M;
```

Imported names being not replaceable

Rule 39. Imported names are not replaceable. ▮

An imported X is not repalceable.

```
/*ILLEGAL*/

model C Real v=10; end C;
model D Real v=20; end D;

model A
    replaceable model X=C;
end A;

model B
    import X=A.X;
    X a;
end B;

model M
    redeclare model X=D;
    extends B;
end M;
```

To make it replaceable in the code above, it needs to be rewritten as **import** A; and **replaceable class** X = $A.X$; in B.

Base names being not replaceable

Rule 40. A base class in an extends-clause cannot be replaceable, implying it cannot be redeclared. (?) Each part of a composite name be non-replaceable. ▮

> **Quote 10**. (Specification 7.1.4 *Restrictions on Base Classes and Constraining Types to be Transitively Non-Replaceable*) The class name used after extends for base-classes and for constraining classes must use a class reference considered transitively non-replaceable.
>
> **Quote 11**. (Specification 7.3 *Redeclaration*) A redeclare construct in a modifier ... A redeclare construct as an element ... Both redeclare constructs work in the same way.
>
> **Quote 12**. (Specification 7.3.1 *The class extends Redeclaration Mechanism*) ... In contrast to normal extends it is not subject to the restriction that B should be transitively non-replaceable ...

Fact 41. There is a trick to extend a replaceable class. One can extend a replaceable class once redeclares it to be not replaceable. ▮

It is used for *BaseProperties* in Media.Examples.TwoPhaseWater. It can be found near line 2542, Media/package.mo (in MSL 3.2.3).

```
package TwoPhaseWater
  extends Modelica.Media.Water.StandardWater;
  redeclare model extends BaseProperties
    "Make StandardWater.BaseProperties non replaceable in order that
     inheritance is possible in model ExtendedProperties"
  end BaseProperties;

  model ExtendedProperties
    extends BaseProperties;
  ...
```

Remark The both of the following code are illegal. Some implementations accept the first code. X is redeclared, and A extends $X = D$. some implementations accept a redeclaration in modifiers, but do not accept a redeclaration in elements. Some implementations accept the second code. The redeclaration is ignored. X is not redeclared, and A extends $X = C$.

```
/*ILLEGAL*/

model C Real v=10; end C;
model D Real v=20; end D;

model A
    replaceable model X=C;
    extends X; /* X=D */
end A;

model M
    A a (redeclare model X=D);
end M;
```

```
/*ILLEGAL*/

model C Real v=10; end C;
model D Real v=20; end D;

model A
    replaceable model X=C;
    extends X; /* X=C */
end A;

model M
    redeclare model X=D;
    extends A;
end M;
```

Redeclarations of dimension sizes

Redeclaring dimension sizes works non-**replaceable** elements.

> **Quote 13**. (Specification 7.3.3 *Restrictions on Redeclarations*) Array dimensions may be redeclared; provided the sub-typing rules in 6.3 are satisfied. [This is one example of redeclare of non-replaceable elements.]

Redeclarations to a class and a component

Fact 42. Redeclaring a class and redefining a component are independent and both are effective. ▮

For example, $X\ a$ is converted to $D\ a$.

```
model A Real v=10; end A;
model B Real v=20; end B;
model C Real v=30; end C;
model D Real v=40; end D;

model H
    replaceable model X=A;
    replaceable model Y=B;
    replaceable X a;
end H;

model M
    redeclare Y a;
    redeclare model X=C;
    redeclare model Y=D;
    extends H; /* a.v=40 */
end M;
```

2.13 Redeclarations in modifiers and in elements

Redeclarations can be written in class elements (in-element) or in modifiers (in-modifier). They are sightly different syntactically/semantically. That is, an in-element redeclaration accepts **inner**/**outer**, but an in-modifier redeclaration not (see Fact 45). An in-element redeclaration affects a base class, which do not happen in usual languages.

Fact 43. Coexisting an in-element redeclaration and an in-modifier redeclaration in an extends-clause which would apply to the same class/variable is illegal. ▮

This is a rephrasing of the following quote.

> **Quote 14**. (Specification 7.3 *Redeclaration*) The redeclare construct as an element ... cannot be combined with a modifier of the same element in the extends-clause.

This does not apply to a combination of a class and a component, nor a non-redeclaration modifier. The modifier below has an effect as $a.v = 30$.

```
model C Real v=10; end C;
model D Real v=20; end D;

model A
    replaceable model X=C;
    X a; /* a.v=30 */
end A;

model M
    redeclare replaceable model X=D;
    extends A (a.v=30);
end M;
```

Ordering of redeclarations

Rule 44. A redeclaration in a derived class is applied later to a redeclaration in a modifier to a base class (in-element vs. in-modifier). A redeclaration in a derived class is applied later to a redeclaration in a base class (in-element vs. in-element). ▮

Redeclaring $X = D$ in M has precedence to a modifier $X = E$.

```
model C Real v=10; end C;
model D Real v=20; end D;
model E Real v=30; end E;

model A
    replaceable model X=C;
    X a; /* X=D */
end A;

model M
    redeclare replaceable model X=D;
    extends A (redeclare replaceable model X=E);
end M;
```

Redeclaring $X = E$ in M has precedence to $X = D$ in B.

```
model C Real v=10; end C;
model D Real v=20; end D;
model E Real v=30; end E;

model A
    replaceable model X=C;
```

```
    X a; /* X=E */
end A;

model B
    redeclare replaceable model X=D;
    extends A;
end B;

model M
    redeclare replaceable model X=E;
    extends B;
end M;
```

Redeclarations with prefixes

Fact 45. Replaceables accept prefixes inner/outer. In-element redeclarations accept inner/outer, but in-modifier redeclarations do not. ▮

2.14 Merging parts of a redeclaration

A redeclaration merges a part of a replaceable (original) declaration. It is called inheriting in the specification.

Merging prefixes in a redeclaration

> **Quote 15**. (Specification 7.3 *Redeclaration*) In redeclarations some parts of the original declaration is automatically inherited by the new declaration.
>
> **Quote 16**. (Specification 6.3 *Interface Compatibility or Subtyping*) (i.e., no need to repeat 'parameter' in a redeclaration).

Fact 46. A variability prefix is merged. A prefix input or output is merged. Prefixes inner and/or outer are merged. A conditional part is merged. (See Specification 6.3 *Interface Compatibility or Subtyping*). ▮

The definition of merging is made precise by the class compatibility (See Quote 33).

Rule 47. (?) Visibility statuses must be equal. Final statuses must be equal (final variables can be replaceable). A variability prefix is replaced with regard to the variability ordering. An input or output prefix in a replaceable is added, when none are given in a redeclaration. Inner and/or outer prefixes in a replaceable are added, when none are given in a redeclaration. A conditional part is added. If both a replaceable and a redeclaration have a conditional part, they must be semantically the same. ▮

That is, *parameter* on a replaceable replaces one if a redeclaration is *discrete* or *continuous*, but does not replace *constant*.

Merging modifiers in a redeclaration

The implementation follows the rules of merging modifiers, when modifiers are in a redeclaration. Note that it may not be compatible with some implementations.

Rule 48. Modifiers in a redeclaration is merged to ones in a replaceable with modifiers in a redeclaration being applied later. ▮

More on merging parts of a redeclaration

Remark Some implementations do not merge inner/outer prefixes by a redeclaration in elements. That is, an outer prefix attached to X in A is dropped in a redeclaration in B, in the following example.

```
model C Real v=10; end C;
model D Real v=20; end D;
model E Real v=30; end E;

model A
    outer replaceable model X=C;
    X a;
end A;

model B
    redeclare model X=D;
    extends A;
end B;

model M
    inner model X=E;
    B b;        /* b.a.v=20 */
end M;
```

Remark In some implementations, a redeclaration in modifiers and one in elements work differently. One in elements drops a modifier attached on a replaceable, but one in modifiers does not.

A modifier $v = 30$ to a class C is dropped by a redeclaration in elements, in the following code.

```
model C Real v=10; end C;
model D Real v=20; end D;

model A
    replaceable model X=C (v=30);
    X x;
end A;

model A0=A (redeclare model X=D);

model A1
    redeclare model X=D;
    extends A;
end A1;

model A2
    extends A (redeclare model X=D);
end A2;

model M
    A0 a0;      /* a0.x.v=30 */
    A1 a1;      /* a1.x.v=20 */
```

```
    A2 a2;      /* a1.x.v=30 */
end M;
```

A modifier $v = 30$ to a component x is dropped by a redeclaration in elements, in the following code.

```
model C Real v=10; end C;
model D Real v=20; end D;

model A
    replaceable C x (v=30);
end A;

model A0=A (redeclare D x);

model A1
    redeclare D x;
    extends A;
end A1;

model A2
    extends A (redeclare D x);
end A2;

model M
    A0 a0;      /* a0.x.v=30 */
    A1 a1;      /* a1.x.v=20 */
    A2 a2;      /* a1.x.v=30 */
end M;
```

2.15 Extends-redeclarations

An extends-redeclaration defines a new class based on an inherited class and replaces it. It redefines an inherited class as other redeclarations.

A base being replaceable

Fact 49. An extends-redeclaration applies to an inherited class. It implies that it searches for a base class in the base classes of the present class where it is defined. While a base class usually should not be replaceable, a base class of an extends-redeclaration needs to be replaceable. ▮

> **Quote 17**. (Specification 7.3.1 *The class extends Redeclaration Mechanism*) A class declaration of the type "**redeclare class extends** B(...)", ..., replaces the inherited class B with another declaration that extends the inherited class ...

It is not straightforwardly translated to a combination of an extends-clause and a redeclaration. It is because the scoping rule of modifiers is unusual. In addition, an extends-redeclaration extends an original base and replaces it, but a redeclaration may hide an original base.

It is illegal to define one without **redeclare**, such as **class extends** A ... **end** A. It is simply a duplicate definition, when it were allowed.

A scope of modifiers

Rule 50. The scope of the RHS of a modifier to an extends-redeclaration is the class that is to be redeclaring (the base and the body of the class). But, it does not include the class in which a modifier appears (the ususal scope). Note that, the scope of the RHS of a modifier is not the same as the LHS, because the LHS is limited to the base. ▮

The modifier $(a = x + y)$ is legal in the following example, because x and y are visible. In contrast, z declared in M is not visible, if it were used in the modifier.

```
model E
    replaceable model A
        Real a=10;
        Real x=20;
    end A;
end E;

model M
    extends E;
    redeclare model extends A (a=x+y)
        Real y=30;
    end A;
    Real z=40;
    A m;
end M;
```

This occurs in MSL. A parameter *preferredMediumStates* is visible in modifiers, near line 154, Media/Water/package.mo (in MSL 3.2.3). But, *preferredMediumStates* is defined in *BaseProperties* but not in *WaterIF97_base* nor its base *PartialTwoPhaseMedium*.

```
partial package WaterIF97_base
  extends Interfaces.PartialTwoPhaseMedium(...);
  redeclare replaceable model extends BaseProperties(
    h(stateSelect=if ph_explicit and preferredMediumStates
           then StateSelect.prefer
           else StateSelect.default), ...)
```

The definition of *preferredMediumStates* can be found at line 241, Media/package.mo (in MSL 3.2.3).

```
package Interfaces
  partial package PartialMedium
    replaceable partial model BaseProperties
      parameter Boolean preferredMediumStates=false
```

2.16 Inner/outer declarations

Inner-outer matching makes a class definition or a variable declaration depend on instantiations. It makes a name defined in an outer instance visible in an inner instance.

An inner-outer relation

Inner-outer refers to a relation of component embedding. A word *instance hierarchy* is also used to refer to it. Assume class M includes declarations $A\ a$ and $B\ b$, and B includes a declaration $A\ c$. By instantiating $M\ m$, m is outer and $m.b$ is inner. When the variable $A\ a$ is prefixed by **inner** and $A\ c$ by **outer**, the references $m.a$ and $m.b.c$ are identical.

Note that an enclosing class relation is not an embedding relation. They are different concepts.

Rule 51. Inner/outer prefixes applies to both classes and variables. An outer definition/declaration imports an identifier defined in some outer instance and makes it visible in the present class. An inner prefix indicates a definition/declaration to match to an outer one. An outer class/variable can be declared multiple times in a class and its bases. They refer to the same inner class/variable. ▮

An outer prefixes an identifier, and an inner prefixes a class definition or a variable declaration.

Fact 52. (?) An inner-outer relation on classes does not extend outward to lexically enclosing classes. ▮

In the following example, the model class is M. The outer $P.X$ is defined in a package P and it cannot have an inner-outer relation. Note the hierarchies are $M \ni P.B$ and $P > P.B$ ($\ni$ for component embedding, $>$ for lexical enclosing).

```
/*ILLEGAL*/

model A0 Real x=10; end A0;
model A1 Real x=20; end A1;

model P
    outer model X = A0;
    model B
       X a;
    end B;
end P;

model M
    inner model X = A1;
    P.B b;
    Real y = b.a.x;
end M;
```

Class compatibility

There is a compatibility restriction among classes of inner-outer.

Rule 53. A relation *A inner-outer B* is allowed for classes, if A is a subclass of B, $A <: B$. (?) What for other classes? (?) What for variable declarations? ▮

```
inner model C = A;
outer model C = B;
model A extends B; end A;
```

An example of inner/outer prefixes on classes is given in the specification as function definitions (functions are a kind of classes). See Specification 5.4.1 *Example of Field Functions using Inner/Outer*.

Syntax restrictions of modifiers

Fact 54. Outer class definitions are short class definition. Modifiers to outer definitions/declarations are usually ignored unless there is no matching inner. Inner definitions/declarations have no restrictions. Simultaneous inner/outer class definitions are short class definitions. Modifiers to simultaneous inner/outer definitions/declarations apply to the inner part unless there is no matching inner. ▮

See Specification 5.5 *Simultaneous Inner/Outer Declarations*.

Outers in a class during lookups

Fact 55. Outer classes and constants are treated as non-existent when a class is processed as a package. ▮

See also Fact 20.

Extending outers

Fact 56. It is legal to extend an outer class. ▮

Simultaneous inner/outer

Fact 57. A simultaneous inner/outer definition/declaration can be regarded as being with inner and outer separately. The only definite effect is that the present class sees an outer class/variable. ▮

See Specification 5.5 *Simultaneous Inner/Outer Declarations*.

Note that the instance created for a simultaneous inner/outer definition is not visible in the class it is defined (exactly speaking, in case when there is some inner which an outer part matches).

2.17 Visibility

Rule 58. A protected class is visible from an enclosed class. ▮

In syntax, **public**, **protect**, **equation**, and **algorithm** construct a section separated by themselves until the end of class elements. It is public, when sections may be started with none of them. That is, sections of equations and algorithms have no distinction of public/protected.

(?) (*CHECK*) Check the visibility of import/extends-clauses and redeclarations.

2.18 Enclosing classes

A lookup of a name in a class sometimes continues to the enclosing class of the class.

An enclosing class of a definition-body

Fact 59. An enclosing class is a class which defines the definition-body of the requested class (assigning a name does not change the enclosing class). An enclosing class may be a modified one, and thus, it is not just the lexical relation. An enclosing class respects the relation of a definition-body, that is, each main/base class has a separate enclosing class. ▮

Scopes not extending to enclosing classes

Elements from enclosing classes are not visible through import- or extends-clauses. The following code is illegal, because v is visible from A, but not visible in the derived class $M0$. Also, v is not visible in $M1$.

```
/*ILLEGAL*/

package P
    constant Real v=10;
    model A end A;
end P;

model M0
    extends P.A;
    Real z=v;
end M0;

model M1
    import P.A;
    Real z=v;
end M1;
```

A scope to each base class

An enclosing class is associated to each base class. Different classes are visible by the same name in a class.

```
package B0
    type T = Real(value=10);
    model B1
        T b;
    end B1;
end B0;

package D0
    type T = Real(value=20);
    model D1
        extends B0.B1;
        T d;
    end D1;
end D0;

model M
    D0.D1 m; /* m.b=10, m.d=20 */
end M;
```

An enclosing class is the defining class

In the following, the enclosing class of $D0.B1$ is $D0|B0$, a base of $D0$ (as denoting a base part of a class by $D0|B0$).

```
package B0
    type T = Real(value=10);
    model B1
        T b;
    end B1;
end B0;

package D0
    extends B0(T(value=20));
end D0;

model M
    D0.B1 m; /* m.b=20 */
end M;
```

An enclosing class of a reclaration

A redeclaration works as expected, as it does not change the scope.

```
package B0
    type T = Real(value=10);
    model B1
        replaceable model B2
            T x;
        end B2;
        B2 b;
    end B1;
end B0;

package D0
    type T = Real(value=20);
    model D1
        extends B0.B1;
        redeclare model B2=D2;
        model D2
            T x;
        end D2;
    end D1;
end D0;

model M
    D0.D1 m;  /* m.b.x=20 */
end M;
```

2.19 Restrictions of within

Processing a target model

A target model is usually loaded from a user specified file. It is possible to modify a target model, when it is defined in an enclosing classes which has modifiers on it. (?) It probably is not an intention.

Rule 60. (?) It is forbidden to declare a class in an enclosing class which is defined by refining, where an enclosing class is specified via **within**. ▮

That is, A is not defined by **package** $A = ...$ when using **within** A.

```
(*CHECK*)
/* File A: */
package A = B(M(x=10));

/* File M: */
within A;
model M
    Real x=0;
end M;
```

2.20 Class prefixes

Fact 61. (?) **encapsulated** can be attached to any class definitions in the syntax rules, but it only has a meaning to long-class-specifier and meaningless to short-class-specifier and der-class-specifier. ▮

Fact 62. **encapsulated** is only effective for main (non-base) classes. ▮

Fact 63. **final** to a class is unclear. It might disable modifiers or might disable extending. ▮

2.21 Variability

Variability of components is specified by keywords for **constant**, **parameter**, and **discrete**, but *continuous* when nothing is specified. Variability is ordered by the inclusion relation:

$$constant \subset parameter \subset discrete \subset continuous\ .$$

Continuous means unconstrained, and declaring continuous integers is valid. The ordering is used in class compatibility. It is described in Specification 3.8 *Variability of Expressions* and Specification 6.3 *Interface Compatibility or Subtyping.*

Fact 64. The ordering strict. However, it can be treated as *parameter* = *constant* for translation and simulation, although parameters and constants are distinguished at a surface level. Note that parameters, in addition to constants, in packages are visible. Values of parameters and constants may be specified by modifiers, unless they are final. ▮

Constants cannot be changed at runtime after translation of a model by definition. However, parameters are also hard to be changed at runtime, because parameters are treated as constants in simplification of constants.

> **Quote 18**. (Specification 3.8.1 *Constant Expressions*) The value of a constant can be modified after it has been given a value, unless the constant is declared final...
>
> **Quote 19**. (Specification 4.4.4 *Component Variability Prefixes discrete, parameter, constant*) A constant variable is similar to a parameter with the difference that constants cannot be changed after translation and usually not changed after they have been given a value.
>
> **Quote 20**. (Specification 7.2.6 *Final Element Modification Prevention*) [Setting the value of a parameter in an experiment environment is conceptually treated as a modification.

Fact 65. Variability is not orthogonal to number types. Simple types except Real are discrete. However, declaring such as continuous integers is allowed (and typical because continuous is the default). ▮

See Specification 3.8.3 *Discrete-Time Expressions* for discrete values.

2.22 Connectors

A connector is a type or a record that is marked to be used in connect equations.

Sides of connectors

A side is an aspect of a connector whether it is connected in a declaring class. An outside connector is an internal connection in the declaring class, and the flow direction is considered reversed (the value is negated) for that connection. An opposite is an inside connector. A connector is considered split to the inside/outside ones, but they share the same variables. The side is taken into account in connect equations but not in the cardinality operator.

Fact 66. Determination of a side of a connector must respect the place where the connector is declared, and inner/outer declarations are ignored for that. An outside connector is distinguished by the use of a connector that is declared and used in the class. That is, the first part of a composite name is a declared connector. ▮

For example, c in connect $(c, ...)$ is an outside connector when it appears in the class declaring c. And also, $c.m.d$ in connect $(c.m.d, ...)$ is an outside connector when it appears in the class declaring c, where m is a possibly empty list of components (connectors or non-connectors). (*CHECK*) It is an outside connector in the case when a non-connector intervenes like $c.m.d$.

A connection set

A connector is a record that may contain simple types, records, operator records, and connectors (but not models specifically). A connect equation connects variables including component connectors declared in the connector.

A connection set consists of tuples of a variable and its side. The set is defined as a symmetric and transitive closure of a relation that contains the pairs in connect equations. The variables are simple types or operator records (overdetermined records), and a record in a connector is broken down to its components, but an overdetermined record (or every operator record (?)) is treated as a single element.

(?) Connect equations on arrays

(?) Array sizes are determined?

Connection restrictions

Connect equations are fixed at translation-time.

> **Quote 21**. (Specification 9.3 *Restrictions of Connections and Connectors*) • The connect equations (and the special functions for overdetermined connectors) may only be used in equations and may not be used inside if-equations with non-parametric condition, or in when-equations. [For-equations always have parameter expressions for the array expression.]

> **Quote 22**. (Specification 8.3.2 *For-Equations - Repetitive Equation Structures*) The expression of a for-equation shall be a vector expression. ... The expression of a for-equation shall be a parameter expression.

Expandable connectors

An expandable connector is a set of connectors, and it is only connected to expandable connectors. A component will be included when it is used. All components in an expandable connector are implicitly considered as connectors. An expandable connector is first considered empty until a component (even a declared one) is included by using it explicitly.

A direct reference of a component in a connect equation signifies an addition of a component to an expandable connector. A reference is $m...m.c.d$, with m a possibly empty component, c an expandable connector, and d a component.

> **Quote 23**. (Specification 9.1.3 *Expandable Connectors*) • expandable connectors can only be connected to other expandable connectors.
>
> **Quote 24**. (Specification 9.1.3 *Expandable Connectors*) • All components in an expandable connector are seen as connector instances even if they are not declared as such
>
> **Quote 25**. (Specification 9.1.3 *Expandable Connectors*) Before generating connection equations non-parameter scalar variables and non-parameter array elements declared in expandable connectors are marked as only being potentially present.

(*CHECK*) Declared components of an expandable connector cannot be used in connect equations unless they are connected to connectors not in some expandable connectors.

Stream connectors

See Specification 15 *Stream Connectors*.

A stream connector is a connector containing stream variables. A stream connector contains a single scalar real flow variable and some reals of (or arrays of) stream variables. The record components of a stream variable are considered as stream variables.

inStream (v) and actualStream (v) take a stream variable (which may be an array).

Overdetermined types/records

A type/record defining equalityConstraint (x, y) is called overdetermined type/record. The Connections class defines some predefined functions to reduce overdetermined constraints.

(?) Operator record components

(?) A connection component of an operator record is used as an aggregate in generated equations with defined eqaulity, summation, or negation.

Unconnected connectors

Flow variables are set to zero for unconnected connectors. Conditional connectors are unconnected if their conditions are false. See Section 2.23.

cardinality operator

The cardinality (c) operator returns the count of the uses of connect equations on the inside part of the argument connector. It is not the size of a connection set. Cardinality of an embedded connector is counted, when it is indirectly used in connect equations. For example, cardinality $(c.d)$ counts an equation connect $(c, \_)$.

cardinality only appears in equation sections (only in assertions). cardinality does not take expandable connectors, components of expandable connectors, nor arrays of connectors. cardinality is a translation-time constant. However, cardinality (c) cannot be replaced by a constant in general, because a connector reference is not a translation-time constant (array indices can be non-constants).

The clarification a bit differs from the specification:

> **Quote 26**. (Specification 3.7.2 *Derivative and Special Purpose Operators with Function Syntax*) Returns the number of (inside and outside) occurrences of connector instance c in a connect-equation as an Integer number. See also Section 3.7.2.3.

> **Quote 27**. (Specification 3.7.2.3 *cardinality (deprecated)*) The cardinality is counted after removing conditional components. and may not be applied to expandable connectors, elements in expandable connectors, or to arrays of connectors (but can be applied to the scalar elements of array of connectors). The cardinality operator should only be used in the condition of assert and if-statements – that do not contain connect (and similar operators – see section 8.3.3).

2.23 Conditional components

A declaration may have a condition to enable/disable it. A disabled non-connector component is unusable. A conditional connector can be used only in connect equations, and connect equations with a disabled connector are considered removed. Connectors in a disabled component are disabled.

Conditional non-connectors

A disabled non-connector component is unusable. Thus, usually, conditional non-connector components are used for the purpose to observe particular states in a simulation.

Restrictions of conditional connectors

Conditional components are only used in connections.

> **Quote 28**. (Specification 4.4.5 *Conditional Component Declaration*) A component declared with a condition-attribute can only be modified and/or used in connections

> **Quote 29**. (Specification 5.6.2 *Generation of the flat equation system*) [Conditional components can be used in connect-statements, and if the component is conditionally disabled the connect-statement is removed.]

Conditions of conditionals are translation-time constants.

> **Quote 30**. (Specification 4.4.5 *Conditional Component Declaration*) The expression must be a Boolean scalar expression, and must be a parameter-expression

Conditions of conditionals are retained after component redeclarations.

> **Quote 31**. (Specification 4.4.5 *Conditional Component Declaration*) A redeclaration of a component may not include a condition attribute; and the condition attribute is kept from the original declaration

Conditional components cannot be used in modifiers of declarations. It is indicated by the example in Specification 4.4.5 *Conditional Component Declaration*. Conditional components cannot have ":" as an array dimension. It is described in Specification 6.3 *Interface Compatibility or Subtyping*.

Conditional declarations

Fact 67. A condition of a conditional component does not affect the scope. Particularly, a variable with a condition can be a target of modifications regardless of the condition. ▮

Is the value of x dependes on a condition?

```
(*CHECK*)
package A
    Real x=1;
    model M
        Real x=2 if (...);
        /* ? x */ /* x dependes on a condition? */
    end M;
end A;
```

(* (?) Check if an inner is affected by its condition. *)

2.24 Class names

Names of classes are almost insignificant after collecting elements of main/base classes. Records are exceptions which are refered to by names at runtime.

Classes of packages/instances can be nameless if modifiers are applied but not assigned by a name. However, dotted forms of variable references can be used to refer to packages/instances (like "x.y.P"), because only used classes need to be referred to. In addition, each class (a main/base part of a class) is generally associated to three names: • a name given to a lexically appearing definition body, • a name lastly assigned by a short class definition, and • a name indicating where the definition body is taken from. The second name by a short class definition is the name of a usual sense and needed to refer to a record name (needed in instances). The third name indicating its origin is needed to obtain an enclosing class (needed in packages).

2.25 Class equality

Generally, an equality or identity of a class is not significant, but class compatibility is used, instead. An identity of a class is not important even for records, which would be meaningful since they are used in function calls or copying in algorithms. See Fact 73 for records. In most cases, class equality is treated as a textual equality, which is required when removing duplicate class definitions.

input and **outer** are not a part of a type.

> **Quote 32**. (Specification 4.5.1 *Short Class Definitions*) A base-prefix applied in the short-class definition does not influence its type, but is applied to components declared of this type or types derived from it; see also section 4.5.2.

(?) (*CHECK*) The equality of functions is by textual (defined?). The overloading resolution is based on a set of functions (requiring uniqueness), thus, the equality of functions needs be defined.

2.26 (Class compatibility)

A redefinition/redeclaration can be applied to compatible classes.

Compatibility

Specification 6.3 *Interface Compatibility or Subtyping* defines compatibility, which applies when a class is redeclared. It is APPROXIMATELY summarized here. The conditions of $A <: B$, A is a subtype of B.

> **Quote 33**. (Specification 6.3 *Interface Compatibility or Subtyping*) A is a subtype of the type of B, iff [intuitively all important elements of B must be present in A].

- A extends C iff B extends C, where C is an operator record.
- A extends C_A iff B extends C_B, where C_A and C_B are ExternalObject with the same name.
- If B is non-replaceable, A must be non-replaceable. (* Why negatively expressed? *)
- If B is transitively non-replaceable, A must be transitively non-replaceable.
- $\forall C_B \in B, \exists C_A \in A,\ C_A <: C_B$ where C_B and C_A are components of the same name and visibility in each corresponding class.
- If B is replaceable, OK.
- If B is neither transitively non-replaceable nor replaceable, A must be linked (?) to the same class ((?) linked).
- If B is a non-replaceable long class definition, A must be a long class definition.
- Analogical (flow, effort (default), or stream) should be the same.
- variability$A \leq$ variability$B$, with respect to the variability ordering.
- Prefixes input and output must be the same.
- Prefixes inner, outer, and inner-outer must be the same.
- If B is final, A must be final and has semantically the same value. (This rule does not apply to redeclarations.)
- The number of array dimensions in A and B must be the same.
- Conditional components are only compatible with conditional components, where the conditions are semantically the same.
- Class kinds must satisfy kinds compatibility.
- An enumeration is only compatible with an enumeration. A compatible enumeration (except for enumeration(:)) must have the same literals in the same order. Or, $\forall A,\ A <:$ *enumeration* (:).
- B is a built-in iff A is a built-in and $A = B$.

  declared variability???

Supplementary notes

- Plug-compatibility is stricter than compatibility.
- Compatibility functions is stricter than compatibility.
- A modifier must be compatible or plug-compatible with the element being modified.
- *A* is an operator record when it extends an operator record.
- *A* is an ExternalObject when it extends an ExternalObject.

Rule 68. (?) The conditions on prefixes are checked before inheriting the prefixes from a replaceable to a redeclaration. ▮

Plug-compatibility

Plug-compatibility is defined in Specification 6.4 *Plug-Compatibility or Restricted Subtyping.*

Function compatibility

Function compatibility is defined in Specification 6.5 *Function-Compatibility or Function-Subtyping for Functions.*

2.27 Class kinds

The table in Specification 4.6 *Specialized Classes* shows the class kinds. Every kind is syntactically the same as a class, but some restrictions on its contents apply to the class with regard to its kind.

Class kinds

The following is the copy of the table in Specification 4.6 *Specialized Classes.*

model is a general collection of elements.

class is a restricted collection that contains extends-clauses, class definitions, and annotations. Note that it was a synonym of **model** by the old definition, and some implementations accept other elements. See Quote 35.

block is a restricted model, whose connectors are directed by **input**/**output**.

package contains similar elements as **record** that is restricted to contain constants/parameters but no variables. (*CHECK*) It can contain protected sections.

type is one of the simple types, arrays of a type, or one extending a type.

record is a record in a usual sense. A component of a record is a type, a record, or an array of them. It contains no equation/algorithm sections nor protected sections (but it can contain functions).

operator record is a record, but is restricted for operator-overloading. It can contain the same elements as records, but also it can contain operators.

operator contains similar elements as **package**, that is restricted to contain only functions as definitions. An operator is only defined in an operator record.

(operator function) is **operator** with one function.

connector is either a type or a record, but it is allowed to connect on it. It is actually a subkind of them. The use of a keyword **connector** defines a record, when it is used in a long class definition. It gives a connector-status to a type/record, when it is used in a short class definition. Connector declarations are implicit in expandable connectors.

expandable connector is a connector.

function is a function in a usual sense. It is syntactically a package (not an instance). It can contain either one algorithm section or one external interface. It does not contain equation sections nor initial algorithm sections. It can contain component declarations (variables) of **type**, **record**, **operator record**, and **function**. Only functions can contain an external interface. See (Specification 12.2 *Function as a Specialized Class*). (?) Functions can contain only input/output variables. A function is described in Section 2.39.

(* **function** is not stated as a restriction of **package** while **operator function** is. *)

Kind restrictions

The contents of a class are: import-clauses, extends-clauses, class definitions (class/enumeration/der), component declarations (variable/parameter/constant), equation/algorithm sections (with/without initial), and external interfaces. Any class can contain import-clauses, extends-clauses, and other class definitions. The other contents are restricted by its kind. In summary, the relations of content restrictions are:

kind	variable	parameter constant	equation	algorithm	similar kinds
type					
model	✓	✓	✓	✓	block, (class)
class					
record	✓	✓			operator record
package		✓			
connector	✓	✓			expandable connector
operator	✓	✓			operator function
function				✓	

Fact 69. (?) The classes that contain state variables are: (**class**), **model**, **block**, **record**, **operator record**, **connector**, **expandable connector**, **function**. Conversely, the classes that do not contain states are: **package**, **operator**, and **operator function**. ▮

Extending and class kinds

The table in Specification 7.1.3 *Restrictions on the Kind of Base Class* shows the restrictions on derived vs. base class relations. It shows that one class can extend some class. A can extend B, if $A \in kind_A$ and $B \in kind_B$ and $kind_B <:_k kind_A$ (note it is swapped).

- **class** $<:_k$ *any*
- **block** $<:_k$ **model**
- **record** $<:_k$ **model** (redundant)
- **record** $<:_k$ **block**
- **type** $<:_k$ **connector**
- **record** $<:_k$ **connector**
- **operator record** $<:_k$ **connector**

- **expandable connector** $<:_k$ **connector**
- **function** $<:_k$ **operator function**

The excerpt is the entries that are highlighted by shading in the table. We omit the reflexive relation on the same kind: $kind_A <:_k kind_A$.

(?) Kind compatibilty

(?) It is unsure that the extending relation is a part of the subtyping relation. See Section 2.26 for the subtyping relation. (?) A kind is not compatible (accepting redeclarations) to other kinds.

Kind restrictions in short class definitions

Fact 70. Short class definitions follow the base class restrictions (in Specification 7.1.3 *Restrictions on the Kind of Base Class*). For example, it is legal to define a boolean as a connector. "**connector** *BooleanInput* = **input Boolean**". ∎

It is a consequnce of the description of rewriting a short class definition to a class definition with an extends-clause:

> **Quote 34**. (Specification 4.5.1 *Short Class Definitions*) A class definition of the form ... is identical ... to the longer form.

The class kind

Remark The **class** kind is defined to be more restricted than in Specification 7.1.3 *Restrictions on the Kind of Base Class*.

> **Quote 35**. (Specification 7.1.3 *Restrictions on the Kind of Base Class*) A class may only contain class-definitions, annotations, and extends-clauses (having any other contents is deprecated).

By the deprecated definition, it is notable that a more restricted kind can be extend by **class**.

Fact 71. It is required to check base classes respect the restriction. ∎

2.28 Predefined classes

Predefined classes are called predefined types in the specification, but some are not types.

Simple types

A simple type is one of Real, Integer, Boolean, String, and *enumeration*. Enumerations are types. And, thus, an enumeration is defined by a short class definition using the **type** class kind. The names Real, Integer, Boolean, and String are not keywords but cannot be redefined.

> **Quote 36**. (Specification 4.8 *Predefined Types and Classes*) The names Real, Integer, Boolean and String are reserved such that it is illegal to declare an element with these names.

Each simple type has modifiable attributes and thus actually represents a set of types. Variable declarations have effective attributes of simple types, but values of expressions have no attributes, and their values are primitive types like usual reals or integers. See Section 2.37.

There are some predefined conversion functions among the simple types (not for all pairs). Integer : *enumeration* → Integer, *enumeration* : Integer → *enumeration*

Remark There are no predefined conversion functions between Boolean and Integer. However, array binding with these index types assumes mapping between these types, such as false=1 and true=2, etc.

Note that there are no constructors to the simple types. See Fact 82.

The least required ranges of the domains of types are defined by Modelica.Constants.inf for Real, and Modelica.Constants.Integer inf for Integer. (?) The behavior on overflows is unspecified.

Integer, String, enumerations

Fact 72. The class names of Integer, String, and *enumeration* can be used as conversion functions. Since there is a single namespace for classes and functions, these names are overloaded. ∎

See Specification 3.7.1 *Numeric Functions and Conversion Functions*. Integer and names of enumerations convert between integers and enumerations.

An enumeration definition by **enumeration** (:) is an unspecified list, which must be redeclared (Specification 4.8.5.4 *Unspecified enumeration*).

Ordering of simple types

See relational operators in Section 2.34.

Predefined classes

Predefined type is simple types and other classes defined in Specification 4.8.8 *Other Predefined Types*. They are Clock, StateSelect, ExternalObject, AssertionLevel, Connections, and classes for graphical annotations.

Clock is a class.

StateSelect is an enumeration.

ExternalObject is a partial class.

AssertionLevel is an enumeration.

Connections is a package.

Connections

Connections is a package is used to remove overdetermined equations. It is described in Specification 9.4 *Equation Operators for Overconstrained Connection-Based Equation Systems*. A type/record defining *equalityConstraint* (x, y) is called an overdetermined type/record. Connections works with overdetermined types/records. It contains equation operators *branch*, *root*, and *potentialRoot* (appear as equations), and boolean-valued operators *isRoot* and *rooted* (appear as functions).

Connecting connectors may induce an undirected graph of overdetermined types/records in connectors as nodes and equality as edges. Such a graph may have cycles when the equations are overdetermined. A graph is reduced to trees by removing some of the edges by replacing them with weaker contraints. Operators in Connections are used to specify trees in the graph.

branch $(m.c.r, m.c.r)$ makes an edge between overdetermined types/records. *root* $(m.c.r)$ specifies a root of a tree of overdetermined types/records. **connect** $(m.c, m.c)$ creates weak edges for the overdetermined types/records in the connectors. Some weak edges may be marked (as removed), and the equality of marked edges are replaced by *equalityConstraint* (x, y) equations.

2.29 Records

A record is a record in a usual sense. Restrictions on components of a record are described in Section 2.27.

Class equality

Fact 73. An identity of a record class is not clearly defined. The equality of the lists of the defined names may be required. ▮

Some implementations may accept the following code:

```
record R0
    Real v;
end R0;

record R1
    Integer v;
end R1;

model M
    R1 x1 (v=20);
    R0 x0 = x1;
end M;
```

See also Section 2.25. (*CHECK*) Function calls may require stricter equality.

Casting

A record constructor can take a class instance, and it is called casting. See Specification 12.6.1 *Casting to Record*.

equalityConstraint

A function *equalityConstraint* can be defined in types and records. *equalityConstraint*(...) = 0 is added as equations, when it is defined (0 in the RHS is used for an array of zeros). See Specification 9.4.1 *Overconstrained Equation Operators for Connection*.

2.30 Arrays

Component array declarations

Fact 74. Dimensions of component arrays are translation-time constants, while dimensions of other arrays (appearing in functions) can be runtime values. ▮

> **Quote 37**. (Specification 4.4.2 *Component Declaration Static Semantics*) Array dimensions shall be non-negative parameter expressions, or the colon operator
>
> **Quote 38**. (Specification 10 *Arrays*) The number of dimensions of an array is fixed and cannot be changed at run-time... However, the sizes of array dimensions can be computed at run-time

An extent in an array dimension is non-negative, that is, it can be zero. Those variables with size zero, obviously, cannot be used.

Constants in array dimensions are fairly arbitrary expressions. They can refer to other components. But, constant-folding may fail, when a dimension depends on a value in the array, which directly or indirectly causes cycles in the dependency. For example, *dim* in $x[dim]$ may not refer to a constant $x[1].c$.

Array dimension ordering

Fact 75. Multi-dimensional arrays are arrays of arrays. The ordering of an array dimension is "row-major". ▮

See code examples in (Specification 10.1 *Array Declarations*):

> **Quote 39**. (Specification 10.1 *Array Declarations*) It is possible to mix the two declaration forms although it might be confusing.

```
Real[3,2] x[4,5]; // x has type Real[4,5,3,2];
```

Note that accessing the example array by $x[-,-]$ returns an array of $[3,2]$, which means, partial indexing is applied from the left in the dimension. This ordering is row-major, despite that Fortran is column-major.

The definition of the fill operator by recursion also implies the row-major ordering (Specification 10.3.3 *Specialized Array Constructor Functions*).

Rectangle-ness of array dimension

An array is rectangular, and an array of arrays has the same size in each element. This applies to the uses of nested array constructors.

Non-integer array dimension

Enumeration types and Boolean can be used as a dimension, and their elements are used in indexing. They can also be used as ranges in for-loops. See code examples in (Specification 10.1 *Array Declarations*):

```
type TwoEnums = enumeration(one,two);
Real[TwoEnums] y;
```

An index of enumeration types is ordered by the declaration order. An index of Boolean is ordered as $false < true$. While indexing by different types is illegal, arrays are bound or assigned without regard to the index types.

> **Quote 40**. (Specification 10.1 *Array Declarations*) Binding equations of the form x1 = x2 as well as declaration assignments of the form x1 := x2 are allowed for arrays independent of whether the index types of dimensions are subtypes of Integer, Boolean, or enumeration types

> **Quote 41**. (Specification 10.1.1 *Array Dimension Lower and Upper Index Bounds*) • An array dimension indexed by Boolean values has the lower bound false and the upper bound true. • An array dimension indexed by enumeration values of the type E=enumeration(e1, e2, ..., en) has the lower bound E.e1 and the upper bound E.en.

Function promotions to arrays

Functions are promoted to array arugments. In promoting functions, array arguments are promoted to higher dimensinal arrays.

Redeclaring array dimensions

Array dimensions may be redeclared.

> **Quote 42**. (Specification 7.3.3 *Restrictions on Redeclarations*) Modification on array dimensions are redeclarations.

Empty arrays

Fact 76. An array indexed with *lower bound* > *upper bound*, such as $x[1{:}0]$, is an empty array. ∎

See Specification 10.7 *Empty Arrays* for empty arrays. This may actually appear (under some condition) in Interfaces.PartialMedium.BaseProperties (in Media/package.mo).

```
replaceable partial model BaseProperties
...
equation
  if standardOrderComponents then
    Xi = X[1:nXi];
```

A fill with a zero dimension can be used to create an empty array, such as fill ("", 0).

Homogeneity of array elements

Instances of an array elements are similar, in that, the classes of the elements are similar, and the number of dimenions of them are the same, because redeclarations (in-element or in-modifier) cannot depend on an index, although the values of constants can be different, and the dimesion of arrays can be different. The classes may have conditional components, but they are of similar classes even though they can be disabled. This ensures the identity of promoting operators to arrays. (?) Class similarity is defined by class compatibility.

(?) Array class definition

An array type definition for simple types **type** B = Real [4] is legal. (?) In contrast, a definition like **model** $B = A$ [4] does not work.

Array package definition

Fact 77. An array package definition like **package** $B = A$ [4] is illegal. ∎

Some implementations simply ignore subscripts, that is, B is an alias of A.

2.31 Array expressions

Array indexing

Fact 78. Array indexing is only in *identifier* $[i0, i1, ...]$. General forms *expression* $[i0, i1, ...]$ are not allowed. ∎

Array ranges

- : - and - : - : - create a one dimensional array (vector) of Real, Integer, Boolean, and enumerators. For numbers, they create Integer vectors if all elements are Integers, or Real vectors otherwise. Triple argument ones are not defined for Boolean and enumerations. They may create an empty vector. See Specification 10.4.3 *Vector Construction* and also Specification 3.2 *Operator Precedence and Associativity.*

Array constructors

An expression {...} creates an array, which is one-dimensional if the elements are scalar or multidimensional otherwise. An array constructor cannot be empty by the syntax definition, although an array can have zero in its dimension (it ensures the type an array constructor returns).

Predefined functions on arrays

ndims $(A) = 0$ and size $(A) = \{\}$ (an empty array) for a scalar A (See Specification 10.3.1 *Array Dimension and Size Functions*).

Functions are defined to create certain arrays: identity, diagonal, zeros, ones, fill, and linspace (See Specification 10.3.3 *Specialized Array Constructor Functions*). Note that identity, zeros and ones return an integer array, and linspace returns a real array.

Some can be easily defined by others.

$$\begin{aligned}
&\text{identity}\,(n) = \text{diagonal}\,(\text{ones}\,(n))\ ,\\
&\text{zeros}\,(...) = \text{fill}\,(0, ...)\ ,\\
&\text{ones}\,(...) = \text{fill}\,(1, ...)\ ,\\
&\text{linspace}\,(lb, ub, n) = lb : ub : (ub - lb/(n-1))\ ,\text{and}\\
&\text{fill}\,(e, n_0, ...) = \text{fill}\,(\text{fill}\,(e, ...), n_0)\ .
\end{aligned}$$

2.32 Iterators of for

Iterators are used in for-equations, for-statements, reductions, and comprehensions. An iterator ranges in Real, Integer, Boolean, and an enumeration, and its range is specified by a vector expression or by a type of Boolean or an enumeration.

Range expressions of iterators

For-iterators with multiple iterators are defined by expanding to nested ones.

> **Quote 43**. (Specification 11.2.2.3 *Nested For-Loops and Reduction Expressions with Multiple Iterators*) it can be expanded into the usual form by replacing each "," by 'loop for' and adding extra 'end for'.

Range expressions of for-equations are translation-time constants, and they can include iterators in the nested ranges.

> **Quote 44**. (Specification 8.3.2 *For-Equations - Repetitive Equation Structures*) The expression of a for-equation shall be a parameter expression.

It is allowed to refer to an iterator in vector expressions in a single for-equation.

```
equation
for i in 1:s, j in 1:i-1 loop
    x[i,j]=j;
end for;
```

The following quotes of the scope description might conflict with the handling by nesting iterators.

> **Quote 45**. (Specification 8.3.2 *For-Equations - Repetitive Equation Structures*) It is evaluated once for each for-equation, and is evaluated in the scope immediately enclosing the for-equation.

Implicit iterator ranges

A range expression of an iterator is optional, and the range is inferred from the uses of an iterator. Implicit ranges are accepted for arrays with dimensions of Integer, Boolean, and enumeration. Implicit iterator ranges are allowed in any *for*-iteration in equations, statements, reductions, and array constructors.

It is assumed that the range of an implicit iterator is inferred by the range of the subscript where an iterator occurs.

See the example in Specification 4.8.5 *Enumeration Types*.

```
/* 4.8.5 */
type DigitalCurrentChoices = enumeration(zero, one);
Real x[DigitalCurrentChoices];
for e loop
    x[e] := 0.;
end for;
```

See the example in Specification 11.2.2.3 *Nested For-Loops and Reduction Expressions with Multiple Iterators*.

```
/* 11.2.2.3 */
Real x[4,3];
algorithm
for j, i in 1:2 loop
// The loop-variable j takes the values 1,2,3,4 (due to use)
// The loop-variable i takes the values 1,2 (given range)
x[j,i] := j+i;
end for;
```

Fact 79. Boolean and enumerations can be an index space for arrays. However, Interger, even with min/max attributes specified, cannot be an index space. ▮

```
/*ILLEGAL*/
type R = Integer(min=3,max=5);
Real x[R];
```

2.33 Overloaded operators

Operators can be overloaded on the arguments, one of which is an operator record or an array of it.

Overloading

Fact 80. Functions are not overloaded. Functions are classes and having the same name is a duplication. ▮

Some predefined functions are overloaded. They include simple type names of Integer, String, and *enumeration* (see Fact 72). They also include ones taking array arguments of varying dimensions, including ndims, size, and reductions (min, max, sum, and product). Also note that, functions which take indefinite number of arguments cannot be defined as functions.

List of operators

The operators can be overloaded. See Specification 14 *Overloaded Operators*. The set of the overloaded operators is fixed.

```
-   not
+   -   *   /   ^   ==   <>   >   <   >=   <=   and   or
constructor   0
String
```

The `-` appears twice as a unary and a binary. Note that the list lacks unary `+`, becuase it is not overloaded.

Fact 81. Operators except constructors need be defined in an operator record on which the operators take the arguments. Constructors need be defined in an operator record on which the constructors return the value. Binary operators can be defined in one of the left or right operands. (?) If functions to an operator are defined in both the left and right operands, they should be equal (equality of functions may be textual). ▮

This rule is implied by the matching rules in Specification 14.4 *Overloaded Binary Operations* and Specification 14.5 *Overloaded Unary Operations*.

Fact 82. Constructors are not defined on the simple types. ▮

```
/*ILLEGAL*/
Real x = Real(10);
```

Resolution rules

The rules of resolving overloading are described in Specification 14.4 *Overloaded Binary Operations* and Specification 14.5 *Overloaded Unary Operations*.

The application of an unary operator $\oplus a$ with a of class A is determined as follows. The tests are tried in the specified order. $\tilde{A}.\oplus$ denotes a set of functions of the operator $\oplus$ in the operator record $\tilde{A}$ with arrays stripped off from A.

- If A is a simple type or an array of it, apply a predefined operator.
- Apply a function $f \in \tilde{A}.\oplus$, whose argument type is A. If f is not unique, it is illegal.
- Apply a function f by promoting it to an array argument.

The application of a binary operator $a \oplus b$ with a of class A and b of class B is determined as follows. The tests are tried in the specified order.

- If A and B are simple types or arrays of them, apply a predefined operator.
- Apply a function $f \in \tilde{A}.\oplus$ or $f \in \tilde{B}.\oplus$, whose argument types are $A \times B$. If f is not unique, it is illegal.
- Apply $f \in \tilde{A}.\oplus$ with trying type conversions applicable on the second argument.
- Apply $f \in \tilde{B}.\oplus$ with trying type conversions applicable on the first argument.
- Apply a function f by promoting it to array arguments.

Operators

The names of the operators need be quoted by ' to make them recognized as ordinary identifiers, for example, '+'. (?) It is not clear quoting is redundant for 'constructor' and 'String'.

Refer to sections, 14.2 for constructors, 14.3 for conversions, 14.4 for binaries, and 14.5 for unaries.

Operators on arrays

Arrays are the general entity including scalars, vectors and matrices. Scalars are arrays with no dimensions, vectors with one dimension, and matrices with two dimensions.

Fact 83. Overloaded operators on scalars are promoted to arrays. ▮

> **Quote 46**. (Specification 14 *Overloaded Operators*) The operator or operator function must be encapsulated;

> **Quote 47**. (Specification 14 *Overloaded Operators*) It is not legal to extend from an operator record, except as a short class definition modifying the default attributes...

(?) Default constructors

2.34 Predefined operators

Operators are overloaded on the arguments of a simple type or an array of it. The set of operators is different from the one for operator records.

List of operators

```
+   -   .+   .-   not
=   +   -   *   /   ^   .+   .-   .*   ./   .^   and   or
==   <>   >   <   >=   <=
```

Equalities

(?) Equality = The equality and assignment are element-wise. Classes which admit equality are simple types or records whose components admit equality. Arrays admit equality when the class of the elements admits equality.

Operators on numbers

The numbers (or the numerical types) are subtypes of either Real or Integer. Conversion from an Integer to a Real is applied, when an Integer expression is passed to an operator requiring a Real. The usual types for the arithmetic operators (+ - * /) are assumed: $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ and $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$. The type of the operator ^ is $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$.

Note that subtypes are mentioned for Real and Integer, because Real and Integer are defined as *type* and thus it is possible to extend them.

> **Quote 48**. (Specification 10.6 *Scalar, Vector, Matrix, and Array Operator Functions*) In all contexts that require an expression which is a subtype of Real, an expression which is a subtype of Integer can also be used; the Integer expression is automatically converted to Real.

Operators on numbers and strings are defined for arrays in Specification 10.6 *Scalar, Vector, Matrix, and Array Operator Functions*.

Unary + - .+ .- The operators + and .+ on numbers are the identity and element-wise. The operators - and .- on numbers are negation and element-wise. These are the same with/without a dot.

Binary + - The operators + and - on numbers are the addition and the subtraction, and element-wise.

Binary .+ .- The operators .+ and .- on numbers are similar to + and -. The difference to ones without a dot is that if one (not both) of the arguments is a scalar, it is promoted to an array.

Binary * The operator * on numbers is the multiplication. If one of the arguments is a scalar, it is the scalar product. 1-dimensional (vector) and 2-dimensional (matrix) cases are, as usual, an inner product, a matrix-vector product, or a matrix-matrix product. Higher dimensional multiplication is illegal.

Binary .* The operator .* on numbers is the element-wise multiplication. If one of the arguments is a scalar, it is promoted to an array, that is, it is a scalar product.

Binary / The operator / on numbers is the element-wise division. The RHS should be a scalar.

Binary ./ The operator / on numbers is the element-wise division. If one of the arguments is a scalar, it is promoted to an array.

Binary ^ The operator ^ on numbers is the exponentiation. The arguments are a scalar-scalar pair or a matrix-scalar pair. If the arguments are scalar-scalar, arguments of Integers are promoted to Reals. If the arguments are matrix-scalar, it is the matrix exponentiation where the scalar argument must be a non-negative integer.

Binary .^ The operator .^ on numbers is the element-wise exponentiation. If one of the arguments is a scalar, it is promoted to an array. Arguments of Integers are promoted to Reals.

Operators on booleans

Booleans and or not The operators work on boolean values and element-wise.

Operators on strings

Binary + The operator + on strings is the concatenation and element-wise.

Relational operators

Relationals == <> < <= > >= Relational operators are defined on the scalar simple types only, but not for arrays. Moreover, equality on class instances is not defined.

> **Quote 49**. (Specification 3.5 *Equality, Relational, and Logical Operators*) Relational operators ... are only defined for scalar operands of simple types.

The ordering of elements of simple types is defined in Specification 3.5 *Equality, Relational, and Logical Operators*. On numbers, the arguments may be promoted from Integer to Real, if the types of the arguments differ. Boolean is ordered as *false* < *true*. A set of enumerators is ordered by the definition order. The ordering in String is defined by the libc function *strcmp* $(x, y) \bowtie 0$ (replacing $\bowtie$ with a relational operator).

Remark The predefined operators are defined for the simple types, although Specification 14.3 *Overloaded String Conversions* and Specification 14.4 *Overloaded Binary Operations* refer to the predefined types. The predefined types excluding the simple types cannot have operators, because they are not operator records.

Remark *strcmp* compares strings as unsigned bytes (not unicode strings).

2.35 Type conversions

Function argument conversions are defined in Specification 12.4.6 *Scalar Functions Applied to Array Arguments*.

Fact 84. Type conversions are defined by constructors. Conversions may be chained, but it is illegal that they make a cycle. ▮

> **Quote 50**. (Specification 14.2 *Overloaded Constructors*) For a pair of operator record classes C and D and components c and d of these classes both of C.'constructor' (d) and D.'constructor' (c) shall not both be legal.

2.36 Constant expressions

A dimension of a component array needs to be translation-time constants, and thus, folding constants is essentially required. A vast set of expressions are allowed for array dimensions, and especially for example, they include array indexing because it is an operator. They include user defined functions, too. Note that we refer here by constants to both constants and parameters (parameter is a wider set in the specification).

> **Quote 51**. (Specification 3.8.2 *Parameter Expressions*) a function or operator with parameter subexpressions is a parameter expression.

Specification 3.8.1 *Constant Expressions* and Specification 3.8.2 *Parameter Expressions* list constant expressions (functions and variables) which can take non-constant arguments:

> initial terminal der edge change sample pre
>
> ndims(A) cardinality(c) end size(A) size(A, j) Connections.isRoot(A.R) Connections.rooted(A.R)

2.37 Attributes

The attributes are parameters predefined in the simple types. They can be modified by names through modifiers, but cannot be accessed by names using a dot notation. They can only be accessed in a subclass of simple types. The *value* attribute can be accessed by referencing a variable.

> **Quote 52**. (Specification 4.8 *Predefined Types and Classes*) Attributes cannot be accessed using dot notation, and are not constrained by equations and algorithm sections. E.g. in Real x(unit="kg")=y; only the values of x and y are declared to be equal, but not their unit attributes, nor any other attribute of x and y.

A dot notation is used for enumerations when accessing an enumerator element.

2.38 Expressions

(?) pure

pure (...) is not well specified.

> **Quote 53**. (Specification 12.3 *Pure Modelica Functions*) pure(impureFunctionCall(...)) - which allows calling impure functions in any pure context,

pure accepts only one argument, although it can accept a list of arguments by the syntax definition.

Predefined operators

See Section 2.34.

Result types

A result type of an expression can be determined by subexpressions, like C Language.

2.39 Functions

A function is a function in a usual sense. A function as a class kind is described in Section 2.27. A function is syntactically a package, and no instance is created. It contains variables that are inputs and outputs, although it is a package.

Arguments to a function are simple types, records (including connectors, operator records) and arrays of them, and functions.

> **Quote 54**. (Specification 12.2 *Function as a Specialized Class*) • A function may only contain components of the restricted classes type, record, operator record, and function; i.e. no model or block components.

(?) Array dimensions

A dimension of an array variable in a function can be a runtime value. A part of a dimension of a type defined elsewhere is a translation-time constant.

> **Quote 55**. (Specification 12.2 *Function as a Specialized Class*) • The dimension sizes not declared with (:) of each array result or array local variable [i.e., a non-input components] of a function must be either given by the input formal parameters, or given by constant or parameter expressions, or by expressions containing combinations of those...

(?) Local variables

(?) An input/output or local variable declaration does not accept general form of modifiers. It accepts only initializers (by =).

> **Quote 56**. (Specification 12.2 *Function as a Specialized Class*) • A formal parameter or local variable may be initialized through a binding (=) of a default value in its declaration, ...

(?) Restriction

> **Quote 57**. (Specification 12.2 *Function as a Specialized Class*) • A function may not be used in connections...

(?) No results

An output of an empty array is used to return no values. See an example in Specification 9.4.3.1 *An Overdetermined Connector for Power Systems*.

```
function equalityConstraint
    input AC_Angle theta1;
    input AC_Angle theta2;
    output Real residue[0] "No constraints"
    algorithm
```

2.40 Equations

if-equation

Each branch of **if** is tested sequentially, and the first true branch is taken.

for-equation

A range of **for** is a vector expression (real or integer) that is a translation-time constant, or types of Boolean or an enumeration. See Section 2.32.

2.41 Semantics remarks to grammar rules

IDENT There is a single namespace for classes and variables, and IDENT can refer to a class name or a variable name which is not distinguished in the grammar rules.

name is a sequence of IDENT. In *type-specifier*, IDENT in *name* refers to a class. In *element-modification*, IDENT refers to a variable. In *function-argument*, IDENT refers to a class, and a sequence of them refers to a function. Note that functions are classes.

In *enumeration-literal*, *declaration*, *for-indices*, *component-reference*, and *named-argument*, IDENT refers to a variable.

In *external-function-call*, *class-specifier*, and *der-class-specifier*, IDENT refers to a class. In *import-clause*, IDENT refers to a class, whose last IDENT can refers to a class or a variable.

2.42 Additional rules

(?) Steps of class name resolution

Class name resolution in a class may not proceed to process component classes at once. It may stop at resolving classes of import/extends-clauses, and postpone resolving component classes.

The following is code fragments in the MSL.

```
within;
package Modelica
  extends Modelica.Icons.Package;

within Modelica;
package Icons
  operator record TypeComplex = Complex;
```

Name resolution may proceed as in the follow steps. But, at the last step, the base-classes are not ready to lookup in (.Modelica) yet.

```
1. Process (.Modelica)
2. Resolve base-classes in (.Modelica)
    - Lookup base-class (Modelica.Icons.Package) in (.Modelica)
    - Lookup (Modelica) in (.)
    - Lookup (Icons) in (.Modelica)
3. Process (.Modelica.Icons) (* to find Package in it *)
4. Resolve component classes in (.Modelica.Icons)
    - Lookup component class (Complex) in (.Modelica.Icons)
    - Lookup (Complex) in (.Modelica) (* go up by enclosing class *)
    - Lookup (Complex) in base-classes of (.Modelica)
```

No non-each redeclarations

It is not possible to apply distinct redeclarations to elements of an array. Splitting of a redeclaration is not defined, and there is no way to drop each in the following.

```
A a[10] (each redeclare class X=B);
```

This implies class redeclarations can be handled without determining (folding constants, etc.) the array indices.

2.43 Memo

- Classes needing special handling: connectors, states in transitions, etc.
- Compilers can ignore "package.order" files. It mainly be used to display classes in some order.
- (?) Is there no problem on using a file name "A/B/package.mo" for package descriptions (a partial support is that "package" is usually not used as a class name).
- Arrays are described in (Specification 4.5.1 *Short Class Definitions*).
- Class *Complex* is defined in the root, not in *.Modelica*.
- Modifiers to a class do not directly applies to an extends-redeclaration but to a replaced class.
- (?) **protected** inner/outer may be illegal.
- (?) inner/outer compatibility of arrays.
- Connections .branch, .root, .potentialRoot, .isRooted.

Fact 85. A class definition needs to be obtained by following the hierarchy, because a class may be modified by its enclosing classes (by replacing itself or its parts). ▮

3 List of keywords and predefined names

This appendix summarizes some list of keywords and predefined names, etc.

3.1 Modelica keywords

(Specification 2.3.3 *Modelica Keywords*)

Syntactic keywords

algorithm and annotation block break class connect connector constant constrainedby der discrete each else elseif elsewhen encapsulated end enumeration equation expandable extends external false final flow for function if import impure in initial inner input loop model not operator or outer output package parameter partial protected public pure record redeclare replaceable return stream then true type when while within

Reserved type names

Real Integer Boolean String

In addition to the syntactic keywords above, the type names are reserved and cannot be redefined (See Quote 36). These names have overloaded meanings.

(*CHECK*) Check the status of "constructor".

3.2 Predefined classes

(Specification 4.8 *Predefined Types and Classes*)

Classes

Real Integer Boolean String *enumeration* Clock StateSelect ExternalObject AssertionLevel Connections

enumeration means names of any enumeration. All enumerations are members of simple types. It may be referred to as *EnumTypeName* in the specification.

Attributes

start fixed nominal unbounded

3.3 Predefined variables

time

(Specification 3.6.7 *Built-in Variable time*)

time

time has a special scoping rule.

end

end

Variables *end* have a special scoping rule.

3.4 Operators with special syntax

Arithmetic operators (on scalars or arrays of Real, Integer)

(Specification 10.6 *Scalar, Vector, Matrix, and Array Operator Functions*)

```
+    -    .+    .-    *    /    ^    .*    ./    .^    =    :=
```

The descriptions span multiple sections: Specification 10.6.2 *Array Element-wise Addition, Subtraction, and String Concatenation*, Specification 10.6.3 *Array Element-wise Multiplication*, Specification 10.6.4 *Matrix and Vector Multiplication of Numeric Arrays*, Specification 10.6.5 *Division of Scalars or Numeric Arrays by Numeric Scalars*, Specification 10.6.6 *Array Element-wise Division*, Specification 10.6.7 *Exponentiation of Scalars of Numeric Elements*, Specification 10.6.8 *Scalar Exponentiation of Square Matrices of Numeric Elements*.

Relational operators (on scalars of Real, Integer, Boolean, String)

(Specification 10.6.10 *Relational Operators*)

```
==    <>    <    <=    >    >=
```

Boolean operators (on scalars or arrays of Boolean)

(Specification 10.6.11 *Boolean Operators*)

```
and    or    not
```

Class constructors

```
0    constructor
```

These are used in defining constructors.

Array constructor

(Specification 10.4 *Vector, Matrix and Array Constructors*)

```
{...}
```

This can be written as array (...).

Matrix constructor

(Specification 10.4.2 *Array Concatenation*)

```
[...]
```

"," is a column separator, and ";" is a row separator. This is a restricted notation of cat (...) for two dimensional cases.

String literal

```
"..."
```

String operators

```
+
```

Array indexing operator

(Specification 10.5 *Array Indexing*)

```
_ [...]
```

Record access operator

```
.
```

Function call

```
_ (...)
```

Array range

(Specification 10.4.3 *Vector Construction*)

```
_:_    _:_:_
```

See Section 2.31.

Tuples

(Specification 8.3.1 *Simple Equality Equations*, Specification 11.2.1.1 *Assignments from Called Functions with Multiple Results*)

```
(...)
```

Tuples are comma separated expressions. Tuples are only used in equalities and assignments for functions returning multiple values.

3.5 Predefined functions

(Specification 3.7 *Built-in Intrinsic Operators with Function Syntax*)

Predefined functions are defined in the unnamed root, and it can be accessed with a preceding dot $.abs\,(v)$ for $abs\,(v)$. Array constructor admits an iterator syntax (array). Reductions admit a reduction syntax in addition to a function call syntax (min, max, sum, and product). der and pure are keywords but they can also be used by a function call syntax.

Numeric functions and conversion functions

(Specification 3.7.1 *Numeric Functions and Conversion Functions*)

abs(v) sign(v) sqrt(v) Integer(e) *enumeration*(i) String(...)

Event triggering mathematical functions

(Specification 3.7.1.1 *Event Triggering Mathematical Functions*)

div(x, y) mod(x, y) rem(x, y) ceil(x) floor(x) integer(x)

Predefined mathematical functions

(Specification 3.7.1.2 *Built-in Mathematical Functions and External Built-in Functions*)

sin(x) cos(x) tan(x) asin(x) acos(x) atan(x) atan2(y, x) sinh(x) cosh(x) tanh(x) exp(x) log(x) log10(x)

Derivative and special purpose operators

(Specification 3.7.2 *Derivative and Special Purpose Operators with Function Syntax*)

der(expr) delay(expr, delayTime, delayMax) delay(expr, delayTime) cardinality(c) homotopy(actual, simplified) semiLinear(x, positiveSlope, negativeSlope) inStream(v) actualStream(v) spatialDistribution(in0, in1, x, positiveVelocity, initialPoints, initialValues) getInstanceName()

der (...) accepts only one argument, although it can accept a list of arguments by the syntax definition.

inStream(v) and actualStream(v) introduce a new state variable and equations related to a given steam variable v.

Event-related operators

(Specification 3.7.3 *Event-Related Operators with Function Syntax*)

initial() terminal() noEvent(expr) smooth(p, expr) sample(start, interval) pre(y) edge(b) change(v) reinit(x, expr)

Assert

(Specification 8.3.7 *assert*)

assert(condition, message, level)

Terminate

(Specification 8.3.8 *terminate*)

terminate(s)

Array dimension and size functions

(Specification 10.3.1 *Array Dimension and Size Functions*)

ndims(A) size(A, i) size(A)

See 2.31.

Dimensionality conversion functions

(Specification 10.3.2 *Dimensionality Conversion Functions*)

scalar(A) vector(A) matrix(A)

Array constructor

(Specification 10.4.1 *Array Constructor with Iterators*)

array(...)

This can be written as {...}.

Specialized array constructor functions

(Specification 10.3.3 *Specialized Array Constructor Functions*)

identity(n) diagonal(v) zeros(n1, n2, n3, ...) ones(n1, n2, n3, ...)
fill(e, n1, n2, n3, ...) linspace(x1, x2, n)

See 2.31.

Array concatenation

(Specification 10.4.2 *Array Concatenation*)

cat(k, A, B, C, ...)

This can be written as [...] in the two dimensional case.

Matrix and vector algebra functions

(Specification 10.3.5 *Matrix and Vector Algebra Functions*)

transpose(A) outerProduct(v1, v2) symmetric(A) cross(x, y) skew(x)

Reduction functions and operators

(Specification 10.3.4 *Reduction Functions and Operators*)

min(A) min(x, y) min(e(i, ..., j) for i in u, ..., j in v) max(A) max(x, y)
max(e(i, ..., j) for i in u, ..., j in v) sum(A) sum(e(i, ..., j) for i in u, ..., j in v)
product(A) product(e(i, ..., j) for i in u, ..., j in v)

pure

(Specification 12.3 *Pure Modelica Functions*)

pure(...)

See Section 2.38

3.6 Predefined elements

Functions in types and records

equalityConstraint(x, y)

3.7 Synchronous language

Clock class

class Clock;

Discrete states

(Specification 16.4 *Discrete States*)

previous(u)

Base-clock conversion operators

(Specification 16.5.1 *Base-clock conversion operators*)

sample(u, c) hold(u)

Sub-clock conversion operators

(Specification 16.5.2 *Sub-clock conversion operators*)

subSample(u, factor) superSample(u, factor) shiftSample(u, shiftCounter, resolution) backSample(u, backCounter, resolution) noClock(u)

Other operators

(Specification 16.10 *Other Operators*)

firstTick(u) interval(u)

3.8 State machines

Transitions

(Specification 17.1 *Transitions*)

transition(from, to, condition, immediate, reset, synchronize, priority) initialState(state) activeState(state) ticksInState() timeInState()

4 Mandatory annotations (maybe)

There are some annotations predefined (chapter 18): annotations for graphical representation, annotations for code generation (18.3), annotations for simulation experiments (18.4), annotations for functions (12.7, 12.8 and 12.9). Some of them are likely mandatory to translators.

Derivative and inverses

(Specification 12.7.1 *Using the Derivative Annotation*, Specification 12.8 *Declaring Inverses of Functions*)

derivative inverse

These directives are placed in function body elements.

(Redeclaration choices)

(Specification 7.3.4 *Annotation Choices for Suggested Redeclarations and Modifications*)

choices choice

(?) The choices directive is associated to replaceables or modifiers, and is placed at replacable and non-replacable elements by short class definitons or declarations.

Fortran interface

(Specification 12.9.1.2 *Arrays*)

arrayLayout

External libraries

(Specification 12.9.4 *Annotations for External Libraries and Include Files*)

Library Include IncludeDirectory LibraryDirectory

Code generation

(Specification 18.3 *Annotations for Code Generation*)

Evaluate HideResult Inline LateInline GenerateEvents smoothOrder

Evaluate and *HideResult* affect component declarations. They are attached to component declarations, class definitions, and a base class of class definitions. *Evaluate* is ignored for non-parameter components.

Inline, *LateInline*, *GenerateEvents*, and *smoothOrder* are placed in elements of function definitions.

smoothOrder = *n* is placed in algorithm sections of functions. It takes optional argument *normallyConstant*.

Simulation setting

(Specification 18.4 *Annotations for Simulation Experiments*)

experiment StartTime StopTime Interval Toleranc

(?) *experiment* is placed in class body elements.

Singleton instances

(Specification 18.5 *Annotation for single use of class*)

singleInstance

(?) *singleInstance* is palced in class body elements.

Diagnostics messages

(Specification 18.7 *Annotations for the Graphical User Interface*)

missingInnerMessage obsolete unassignedMessage

obsolete applies to classes.

Version setting

(Specification 18.8.2 *Version Handling*)

version conversion uses

5 Some noteworthy excerpts

5.1 Terms with unnamed

There are many uses of *unnamed* in the specification.

unnamed enclosing class in 5.2 Enclosing Classes

unnamed root of the class tree in Specification 5.6.1.1 *The Class Tree*

unnamed nodes in the instance tree in Specification 5.6.1.2 *The Instance Tree* (unnamed nodes are generated by extends clauses)

unnamed root of the instance tree in Specification 5.6.2 *Generation of the flat equation system*

unnamed top-level package in Specification 13.2.1.1 *Lookup of Imported Names*

unnamed top-level scope in Specification 13.2.4.1 *Example of Searching MODELICAPATH*

unnamed element of a class definition in Specification *Glossary* As "extends clause: an unnamed element of a class definition"

5.2 Quotes about class name lookups

Qualified package names

Quote 58. (Specification 13.2.1 *Importing Definitions from a Package*) Here packagename is the fully qualified name of the imported package ... (Specification 13.2.1.1 *Lookup of Imported Names*) ... Lookup of the name of an imported package or class, ..., deviates from the normal lexical lookup by starting the lexical lookup of the first part of the name at the top-level.

Remaining parts lookups A lookup of the remaining parts of a name includes the elements from the base classes. It is because the lookup is in the temporarily flattened class.

Quote 59. (Specification 5.3.2 *Composite Name Lookup*, 4th-item) If the identifier denotes a class, that class is temporarily flattened ... The rest of the name ... is looked up among the declared named elements of the temporary flattened class. If the class does not satisfy the requirements for a package, the lookup is restricted to encapsulated elements only.

And, ... becuase the temporarily flattened classis usual flattening with empty modifications.

Quote 60. (Specification 5.3.2 *Composite Name Lookup*, in the following comment) [The temporary class flattening performed for composite names follow the same rules ..., except that the environment is empty. ...]

And, the partially flattened classincludes the elements from the base classes.

Quote 61. (Specification 5.2 *Enclosing Classes*) During flattening, the enclosing class of an element being flattened is a partially flattened class. [For example, this means that a declaration can refer to a name inherited through an extends-clause.]

6 Supplemental glossary

6.1 Words specific to the implementation

Some words are coined for the implementation or used with different meaning. The marker $^{\top}$ indicates the word is with stricter meaning. The marker $^{\perp}$ indicates the word with different meaning.

analogical is used to refer to a set of flow, effort, and stream.

array, scalar, vector, matrix . Scalars are arrays with no dimensions, vectors with one dimension, and matrices with two dimensions. There are no separate vector nor matrix types.

component$^{\top}$ and **element**$^{\top}$ component $\subseteq$ element as a set in a class definition. Components are variable declarations. Elements are both variable declarations and class definitions.

definition/declaration are often distinguished to mean a class definition and a variable declaration. It adheres to the uses in the syntax rules. It is the same for redefinition/redeclaration. Declaration may be used to refer to both.

definition-body is a set of elements of a definition that is not short-class-definition.

lexically enclosing class is used as an enclosing class without modifications and **encapsulated** considered.

instance/state/variable are used interchangeably.

internal connection is an aspect of a connector, which refers to a connection to an outside connector.

main class refers to a class which is not a base class.

merging modifiers is to merge a part of prefixes and modifiers in a redeclaration. It is called inheriting in the specification.

predefined type$^{\top}$ is simple types and classes defined in Specification 4.8.8 *Other Predefined Types*. "Built-in type" is synonymous and avoided.

present class is a class at attention. It is called a current class in the specification.

refining is a modifier application, including one by a short class definition.

resolve is determining the class to which a class name refers.

simple type$^{\top}$ is a set of Real, Integer, Boolean, String, and *enumeration*. The implementation also includes types extending them. (It does not includes der's).

(* They are called in many ways. A BASIC DATA TYPE is Real, Integer, Boolean and String. A PREDEFINED VARIABLE TYPE is Real, Integer, Boolean, and String. A SIMPLE TYPE is Real, Integer, Boolean, String and enumerations. A TYPE is Real, Integer, Boolean, String, enumerations, arrays of a type, and ones extending a type. *)

subject is a name of a package or an instance. It is used as a key to an associated class definition. For example, "tank.level" is a subject of a state, ".Modelica.Fluid" is a subject of a package.

$\in$ indicates an element is in a class or in a class of a state variable.

attributes are parameters defined in predefined types (Real, Integer, Boolean, and String) and enumerations.

base-prefix = input or output (see syntax rules).

elemental redeclaration is a redeclaration that appears in class elements, in contrast to in modifiers.

environment (in Specification 7.1 *Inheritance – Extends Clause*) = modification environment.

instance (in Specification 5.4 *Instance Hierarchy Name Lookup of Inner Declarations*) is a state, declared as a variable of a certain class.

instance hierarchy (in Specification 5.4 *Instance Hierarchy Name Lookup of Inner Declarations*) is an embedding relation of instances. (Related words: instance hierarchy, instance scope). A hierarchy is created by an instance declaration in a class definition. It is distinct from lexically enclosings.

instance scope (in Specification 5.6.1.2 *The Instance Tree*, Specification 5.3.1 *Simple Name Lookup*) Instance scope is a scope excluding enclosing classes. It consists of declared elements, declared elements from base classes, and imported names.

instance tree = instance hierarchy.

(instance) model hierarchy$^{\perp}$ (in Specification 5.4 *Instance Hierarchy Name Lookup of Inner Declarations*) = instance hierarchy.

instantiation$^{\perp}$ (in Specification 5.6.1 *Instantiation*) refers to applying modifications. It makes a partially instantiated element be fully instantiated.

(instantiation) partial instantiation (in Specification 5.6.1.4 *Steps of Instantiation*) is an intermediate state of flattening a class where modifiers are associated (but not applied) to an element but redeclarations are applied. This state is necessary to enable redeclarations to be applied before handling extends-clauses.

> **Quote 62**. (Specification 5.6.1.4 *Steps of Instantiation*) 2.1. ... (the partially instantiated elements have correct name allowing lookup)

(instantiation) partial instantiation (in Specification 5.6.1.4 *Steps of Instantiation*) is to make a pair of a class element and a modification environment, where redeclarations are applied. A produced pair is called a partially instantiated element.

> **Quote 63**. (Specification 5.6.1.4 *Steps of Instantiation*) ... a partially instantiated element (...) is comprised of a reference to the original element (from the class tree) and the modifiers for that element.

modification environment^{+} (or **flattening environment**) is a set of modifiers attached to a *partially instantiated element*.

> **Quote 64**. (Specification 7.2.2 *Modification Environment*) The modification environment contains arguments which modify elements of the class.

flattening context is a pair of a modification environment and a set of enclosing classes (Specification 5.1 *Flattening Context*).

(flattening) partial flattening^{+} = *partial instantiation*, as instantiation is a part of flattening.

(flattening) partially flattened enclosing class is simply *partially flattened + enclosing class* (See Specification 7.1 *Inheritance – Extends Clause*).

(flattening) partially flattened enclosing class From the quote, it means the extends-clauses are resolved. Thus, the names in the base classes can be looked up.

> **Quote 65**. (Specification 5.2 *Enclosing Classes*) During flattening, the enclosing class of an element being flattened is a partially flattened class. [For example, this means that a declaration can refer to a name inherited through an extends-clause.]

(flattening) temporary flattening (in Specification 5.3.2 *Composite Name Lookup*) is a flattening with an empty modification environmemnt.

(flattening) modification environmemnt ?.

simple name and **composite name** .

built-in classes = (Synonyms) predefined classes.

import name (in Specification 5.3.1 *Simple Name Lookup*) is an element imported by a qualified import.

> **Quote 66**. (Specification 5.1 *Flattening Context*) Flattening is made in a context which consists of a modification environment (Section 7.2.2) and an ordered set of enclosing classes.

temporary flattened class is a flattened class with an empty modification environment. (* the following part is uncertain *) It includes the declared elements, elements from the extends-clauses, and the elements from the import-clauses, but not the elements from the enclosing classes.

> **Quote 67**. (Specification 5.3.3) ... the class A is temporarily flattened with an empty environment (i.e. no modifiers, see Section 7.2.2) ...

class tree (in Specification 5.6.1.1 *The Class Tree*) is class definitions, hierarchically organized.

local class/component (in Specification 5.6.1.4 *Steps of Instantiation*) It is a class/variable defined/declared in elements, not including inherited classes. It, probably, excludes imported, base, and enclosing classes.

lookup of base name for extends.

> **Quote 68**. (Specification 7.1 *Inheritance – Extends Clause*) The name of the base class is looked up in the partially flattened enclosing class.

The sentence *The found base class is flattened with ...* has the recurrence of the word flattened as *arguments of all enclosing class environments that match names in the flattened base class* in the following itemization.

ordering of the merging rules

redeclaration A redeclaration apprears in three ways (Specification 7.3 *Redeclaration*): • in modifiers; • in class definition elements; • in extends-redeclaration.

For an element-redeclaration, a class/variable must be inherited. The modifiers on the extends-clause of the base class are ignored (Specification 7.3 *Redeclaration*).

simple types$^+$ are boolean, integer, enumeration, real, string, Clock, and *externally defined object*. (The definition in glossary lacks the last two).

specialized class (Specification 4.6 *Specialized Classes*) lists a list of classes with specific kinds (e.g, **model**, **block**).

unnamed top-level package = ?

7 Issues need to be checked in MSL

7.1 Missing each

An each seems missing for p, at line 850, Fluid/Interfaces.mo in MSL 3.2.3:

```
partial model PartialDistributedVolume
  ......
  Medium.BaseProperties[n] mediums(
    each preferredMediumStates=true,
    p(start=ps_start),
    each h(start=h_start),
    each T(start=T_start),
    each Xi(start=X_start[1:Medium.nXi]));
```

7.2 Mismatch of final statuses

Final statuses do not match in a redeclaration in Fluid/Vessels:

```
package BaseClasses
    partial model PartialLumpedVessel
      replaceable model HeatTransfer =
          Modelica.Fluid.Vessels.BaseClasses.HeatTransfer.IdealHeatTransfer
        ...;

      HeatTransfer heatTransfer(
        redeclare final package Medium = Medium,
        ...);

  partial model PartialVesselHeatTransfer
    extends Modelica.Fluid.Interfaces.PartialHeatTransfer;
    ...

  model IdealHeatTransfer
    extends PartialVesselHeatTransfer;
    ...
```

and in Fluid/Interfaces.mo:

```
partial model PartialHeatTransfer
  replaceable package Medium=Modelica.Media.Interfaces.PartialMedium
  ...
```

8 Baby-Modelica implementation notes

8.1 Limitations and general remarks

- The frontend is written in SML '97 (Standard ML). The code is developed mainly with Poly/ML (on SunOS 5.11/amd64), but it may work with SML/NJ or MLton. Sometimes the code is twisted because SML does not allow forward references of names.
- The frontend uses a modified BYACC parser generator (not ml-yacc), which can be obtained separately.
- The file coding is ASCII, although it is defined as UTF-8 in Specification 13.2.2 *Mapping Package/Class Structures to a Hierarchical File System*.
- Error checking/reporting is none. The parser does not record source line number information. No information is provided for errors in a model.
- Constant folding is very weak. Folding constants at translation-time is mandatory for array dimensions.
- The frontend may be confused by the uses of characters "`.`" and "`@`" in names in a model. It internally uses them to separate identifiers.
- The frontend treats **class** as a synonym of **model**, that follows the old definition.
- The frontend does not implement **pure** expressions and errs on uses of them.
- It does not parse the *Synchronous Language* and *State Machines*.
- It does not handle overdetermined connectors with *equalityConstraint*.
- Internal errors (assertions) raise the Match exception. But, other errors are often synonyms of the Match exception.
- The frontend does not check the representable values of Integer. Escpecially, Integer_inf may raise the Overflow exception. The definition of ModelicaServices.Integer_inf in MSL 3.2.3 is in 32 bits, and it is no problem for 64 bit SML implementations.
- (* Array subscripts ":" to connector references are not handled. *).
- (* The code for expanding for-equations and finding implicit iterator ranges is not tested at all. *).
- The package Connectors cannot be extended. It is handled nominally.
- (?) Translation-time constants cannot depend on variables in expandable connectors.

9 Baby-Modelica – Parsing

Parsing is simply done by a BYACC generated parser.

9.1 Code tips

- The rule set in *Modelica Concrete Syntax* defined in the specification is used essentially verbatim.
- The keywords **end** and **initial** appearing in the *primary* expression of the syntax rules cause conflicts in BYACC.

9.2 Parsing of end and initial

The keywords **end** and **initial** appearing in the *primary* expression of the syntax rules cause conflicts in BYACC. **end** may appear as a predefined variable as well as a keyword. **initial** may also appear as a predefined function as well as a keyword. For **end**, see Specification 10.5.2 *Indexing with end*. For **initial**, see Specification 8.6 *Initialization, initial equation, and initial algorithm*.

The token **end** only appears as a variable in subscript expressions. The lexer recognizes **end** as an identifier in subscript expressions, or as a keyword elsewhere. The parser tells the lexer to switch the token type of **end** as a keyword or an identifier. The switching code is placed at the safe points where a look-ahead of the parser never looks at **end**.

The token **initial** only appears as a keyword either in **initial equation** or **initial algorithm**. The lexer handles each sequence of **initial equation** and **initial algorithm** as a single token. The lexer judges by a look-ahead of a token next to **initial**. Otherwise, the lexer returns an identifier for **initial**.

10 Baby-Modelica – Resolving

Resolving handles modifier applications in a model for elaboration.

10.1 Code tips

- The implementation directly handles a short class definition as a class *refining* which represents modifier applications, although it is defined as an expansion to a class definition with an extends-clause in Specification 4.5.1 *Short Class Definitions*.
- Mutual extendings are ignored (not errors).
- The implementation does not save modified classes (even though they are named), and repeats the same modifications each time a package/instance is processed.

10.2 Internal class naming

Most of the classes do not have names, because a modifier application creates a new class but it may not have a name unless explicitly given. Also, name association may be changed by inner-outer matching and redeclarations.

Thus, the implementation refers to a class by a package/instance name, supplementally paired with a class name. A pair is used becuase only main (non-base) classes are named in this way, but names of base classes are needed to distinguish them. For example, it may refer to one base class with a pair *tank*|*PartialLumpedVessel*, where *tank* is an instance of *OpenTank* and *PartialLumpedVessel* is a base of it.

A class being named by a package/instance name is appropriate, because only classes used are need to be accessed. Note that all class bodies (a list of elements) have names, with the exception of a body of an extends-redeclaration, for which the base class name is used as a name for it.

10.3 Rewriting import-clauses

The parser rewrites an import-clause to an explicitly named form to reduce the number of variations. This rewriting is performed in the parser. For example, it rewrites:

```
import pkg.n;
==>
import n = pkg.n;

import pkg.{n0,n1,n2};
==>
import n0 = pkg.n0;
import n1 = pkg.n1;
import n2 = pkg.n2;
```

10.4 Handling of modifiers

Modifier applications

The syntaxer associates an application of modifiers to a declaration which is converted to a modification. Association to a class definition C is by temporarily creating a syntax element *refine* $(C, ...)$ with modifiers.

Redeclarations

The syntaxer handles a redeclaration by replacing a replaceable. Similarly, it handles an extends-redeclaration by replacing a replaceable class and making it a base class.

A class redefinition may need be applied to multiple occurrences of a replaceable in class elements, because the syntaxer delays merging elements in a main and base classes. See Rule 9.

Scoped modifiers

Names and expressions are attached with an environment which records the scope where they occur, especially for the RHS of modifiers. This makes modifiers to be moved inside another class which are associated to. The scope is represented by either an instance name or a class name. This scoping is necessary to delay the name resolution of modifiers attached to extends-clauses. Such modifiers have to be moved inside a base class before resolving the modifiers, because it lacks the information of the base classes at that time.

A value attribute

An initializer modifier to the simple types and a modifier to the *value* attribute are synonymous. The implementation handles them by converting an initializer modifier to a *value* modfier. See Fact 27 about a *value* attribute.

Initializer modifiers

An initializer modifier like $x = e$ is converted to modifiers corresponding to components. That is, $x = e$ is converted to $x\,(c_0 = e.c_0, c_1 = e.c_1, ...)$ for each component c_i of the class x. It does not check x and e have the same set of components; x can be of fewer components than e. It will make many copy expressions of e. (?) It will be problematic when the expression e has effects.

10.5 Modifcations and lookups

Modifications are shallowly applied to the class, and then lookups of the class names are performed. Here, shallow application means that each modifier is pushed inside the present class and it is attached as a modifier to a corresponding element. The process does not immediately

recurse. Name lookups need to be performed after modification application, because redeclaration modifications affect lookups of names of bases classes. Thus, modifications and lookups are performed alternatingly.

10.6 Lookups of base classes

The resolving of a class of an extends-clause may involve resolving classes of the parts of a component name. During the search, many new resolving processes may be started for intermediate packages. Resolving processes may interlace (which may visit a class being under resolving), and the search sees classes with unfinished resolving. The implementation keeps track of a process by the three states of loaded, importing-resolved, and extending-resolved.

The implementation skips searching in the bases, when a lookup in the bases has a cycle of dependency. See the section 2.8.

10.7 Other specific behabiors

Class prefixes

The implementation accepts **encapsulated** only to a long-class-specifier but errs/ignores to a short-class-specifier and a der-class-specifier. See Fact 61.

The implementation ignores **final** to a class. See Fact 63.

Folding constants

The implementation uses int of SML for Integer values. It may raises exceptions on overflows in folding constants.

Constants and parameters

The implementation does not distinguish constants and parameters. Both are treated as translation-time constants. See Fact 64.

Variables time and end

The implementation treats time and end as if they were defined in the unnamed-enclosing-class, because a look up of a name ends there. It violates the Rule 4.

11 Baby-Modelica – Instantiating

11.1 Instantiating by one big pass

Instantiating is a bit complex procedure. It works top down in order to handle modifiers and inner-outer matching, while it is usually natural to work bottom up because each submodel only knows and accesses their components but does not care its users. In addition, chains of dependency in name resolutions and simultaneous constant folding complicate the procedure. That is, determining array dimensions needs folding constants which extensively accesses constants in the components not yet instantiated.

Processing packages also works top down, although processing is requested on demand at references. It needs to work top down, because it is necessary first to process the enclosing package to obtain a class definition when the enclosing package has modifiers. Here, the enclosing relation is considered as extended by importing and extending.

Note that lexically enclosing classes cannot affect the enclosed class (See Fact 18). Thus, it is safe to use the textual definition for such class references. For example, Modelica.Fluid.Vessels.OpenTank

is as textually defined, and it does not need to process Modelica.Fluid.Vessels first to take the definition. Actually, the implementation depends on this shortcutting, that is necessary to process ".Modelica", or it would make a cycle otherwise.

11.2 Name resolutions in steps

Name resolution needs be processed from the root for the hierarchies of both packages and instances. The implementation splits name resolution in two steps. The first step resolves classes of importing/extending. The second step resolves classes of components.

(?) There is a simple dependency in processing the *.Modelica* package. *.Modelica.Icons.Package* is a base of it. Apparently, searching it needs to search *Icons* and *.Modelica*. See Section 11. Assume it would try continuing component processing. *Complex* is used in *.Modelica.Icons*. It results in searching *Complex* in *.Modelica*, again.

11.3 Instances

The syntaxer does not distinguish classes and instances, and a class is an instance when it is associated to a state variable. It is like classless languages as Self.

11.4 Handling arrays with modifiers

The implementation uses a pseudo function *split* when a non-each modifier is moved in a class definition that is an array. For example, a modifier will be moved in the class definition in the example below. The indexing $v[i]$ is represented by a *split*. The index i ranges in {1, ..., 10}.

```
A a[10] (x=v)
==>
model A' = A (x=v[i])
A'[10] a;
```

> **Quote 69**. (Specification 7.2.5 *Modifiers for Array Elements*) If the modified element is a vector and the modifier does not contain the each-prefix, the modification is split...

split is semantically an array indexing by a constant index. But, *split* is not a proper syntax in Modelica, because it only defines an array indexing on identifers but not on expressions.

11.5 Handling inner/outer declarations

The syntaxer performs an inner-outer matching when it makes a list of visible names in an instance. It leaves an alias for an outer variable pointing to an inner variable. An alias is necessary to retain the component relation for a connector (a place where a connector is declared) which determines the flow direction. Outer aliases are replaced after handling connectors. Replacing an outer alias is performed by ignoring subscripts of a variable reference because they are irrelevant.

11.6 Record class identity

A record class needs identity for the purposes such as passing arguments to functions. The implementation identifies a class by the definition that is (last) modified by redeclarations. Redeclarations may change the components of a class, and the implementation treats redeclarations create a separate class but non-redeclarations do not create a new one. The implementation creates an identity by concatenating an identity of a package and a name given after redeclarations.

See Fact 73. The implementation does not identify functions this way, because modifications of values change the function.

12 Baby-Modelica – Syntaxing

Syntaxing performs simple semantic transformations to make a flat model. It rewrites connect equations to equations. In that rewriting, it expands if-equations and for-equations that include connect equations.

12.1 Implicit Iterator Ranges

A range of an implicit iterator is taken as a subscript part of an expression that includes an iterator. It checks the occurrence only but ignores the surrounding expressions if any. See Section 2.32.

12.2 Expandable connectors

Instantiation of expandable connectors is delayed, while other (usual) classes are instantiated by descending the model hierarchy. Expanding expandable connectors is performed in steps starting from a connector at the shallowest position.

12.3 Connector operators

Operator applications on or related to connectors, inStream (v), actualStream (v), and cardinality (c) are replaced by new variables with some associated equations. See a *mix* variable in Specification 15 *Stream Connectors*. Variables are suffixed by `_mix_in_` for inStream (v), `_actual_in_` for actualStream (v), and `_cardinality_` for cardinality (c).

13 Baby-Modelica – Flattening

13.1 Variable references

A flattener moves array dimensions to the last part. It prints $a[i].b[j].c[k]$ as something like $a\_b\_c[i, j, k]$.

14 Baby-Modelica – Miscellaneous tips

14.1 Odd facts

- It is said that the RML compiler of OpenModelica is written in SML '97 (Standard ML). RML is Relational Meta-Language. Refer to OpenModelica for MetaModelica and RML.

14.2 Code indentation (emacs setting)

Code is indented by emacs sml-mode.el (6.9). It is with minor modifications to adjust spacing in the first mrule after `"case x of pat => (\n` to align with other mrules. And similarly after `"fun f = (\n"`.

```
--- sml-mode.el.org
+++ sml-mode.el
@@ -530,14 +530,20 @@
     (`(:after . "struct") 0)
     (`(:after . "=>") (if (smie-rule-hanging-p) 0 2))
     (`(:after . "in") (if (smie-rule-parent-p "local") 0))
-    (`(:after . "of") 3)
+    (`(:after . "of") 2)
-    (`(:after . ,(or `"(" `"{" `"[")) (if (not (smie-rule-hanging-p)) 2))
+    (`(:after . ,(or `"(" `"{" `"["))
+     (if (not (smie-rule-hanging-p))
+         2
+       (if (or (smie-rule-parent-p "|") (smie-rule-parent-p "of"))
+           (+ sml-indent-level -2)
+           (+ sml-indent-level 0))))
     (`(:after . "else") (if (smie-rule-hanging-p) 0)) ;; (:next "if" 0)
     (`(:after . ,(or `"|" `"d|" `";" `",")) (smie-rule-separator kind))
     (`(:after . "d=")
-     (if (and (smie-rule-parent-p "val") (smie-rule-next-p "fn")) -3))
+     (if (and (smie-rule-parent-p "val") (smie-rule-next-p "fn")) -3 2))
     (`(:before . "=>") (if (smie-rule-parent-p "fn") 3))
-    (`(:before . "of") 1)
+    (`(:before . "of") 2)
     ;; FIXME: pcase in Emacs<24.4 bumps into a bug if we do this:
     ;;(`(:before . ,(and `"|" (guard (smie-rule-prev-p "of")))) 1)
     (`(:before . "|") (if (smie-rule-prev-p "of") 1 (smie-rule-separator kind)))
```

References

[1] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.4*, April 2017.

[2] Peter Fritzson et al. OpenModelica system documentation, version 2014-02-01. Technical report, Open Source Modelica Consortium, February 2014. URL: https://openmodelica.org/svn/OpenModelica/trunk/doc/OpenModelicaSystem.pdf.

[3] OMG (Object Management Group). SysML–Modelica transformation, version 1.0. Technical report, OMG, November 2012. URL: https://www.omg.org/spec/SyM/1.0.

[4] LMS International. *Translator to flat Modelica*, May 2007.

[5] Martin Otter. DSblock: A neutral description of dynamic systems, version 3.3. Technical report, DLR at Oberpfaffenhofen, April 1994.

[6] Christiaan J.J. Paredis, Yves Bernard, Roger M Burkhart, Hans-Peter de Koning, Sanford Friedenthal, Peter Fritzson, Nicolas F Rouquette, and Wladimir Schamai. An overview of the SysML-Modelica transformation specification. *INCOSE International Symposium*, 20(1):709–722, 2010. DOI: 10.1002/j.2334-5837.2010.tb01099.x.