Proceedings
of the 4th International Modelica Conference,
Hamburg, March 7-8, 2005,
Gerhard Schmitz (editor)

J. Mauss
DaimlerChrysler AG, Berlin, Germany
Modelica Instance Creation
pp. 509-517

Paper presented at the 4th International Modelica Conference, March 7-8, 2005,
Hamburg University of Technology, Hamburg-Harburg, Germany,
organized by The Modelica Association and the Department of Thermodynamics, Hamburg University of Technology

All papers of this conference can be downloaded from
http://www.Modelica.org/events/Conference2005/

Modelica Instance Creation

Jakob Mauss
jakob.mauss@dcx.com
DaimlerChrysler Research & Technology
Alt-Moabit 96a, 10559 Berlin, Germany

Abstract

This paper is about instance creation in Modelica. Despite the conceptual simplicity of Modelica's object-oriented framework, instance creation in Modelica requires surprisingly complicated procedures. Hence, it takes considerable effort to develop a Modelica processor for extracting all variables, equations and algorithms from a given Modelica class. This paper is meant to reduce this effort by presenting key representations and algorithms for instance creation. To ease reading and verification, instance creation is developed for a fragment of Modelica, called SimpleModelica. Building on the representations and procedures given here, the implementation of instance creation (flattening) for full Modelica should be straightforward. However, that ultimate procedure is not given here, since it is loaded with technical details, described in the (100 pages) Modelica language specification.

1 SimpleModelica

The syntax of SimpleModelica is defined as follows

```
class_definition : [ encapsulated ] class IDENT class_specifier
class_specifier :  { element ";" } end IDENT |
                   "=" name [ class_modification ]
element : import_clause | extends_clause |
                   class_definition | component_declaration
import_clause :  import ( IDENT "=" name | name ["." "*"] )
extends_clause : extends name [ class_modification ]
component_declaration : name IDENT [ modification ]
modification : class_modification [ "=" expression ] | "=" expression
class_modification : "(" [ argument { "," argument } ] ")"
argument : element_modification | element_redeclaration
element_modification : name [ modification ]
element_redeclaration : redeclare
                   (class_definition | component_declaration)
expression : NUMBER | STRING | true | false | name
name : IDENT [ "." name ]
```

As usual, [x] stands for zero or one, and {x} for zero or more occurences of x, while | denotes alternatives.

SimpleModelica is a proper subset of Modelica. Omissions w.r.t. Modelica are: arrays, most prefixes, equation and algorithm section, class categories, expressions involving functions, comments, and annotations. An example of SimpleModelica is

```
class Ele1000 = Ele(Resistor.r=1000);
class Ele
   class Resistor
      Real r = 1;
   end Resistor;

   class Circuit
      Resistor r1;
      Ele.Resistor r2;
   end Circuit;
end Ele;
```

These class definitions will be used throughout the paper as illustrating example.

2 Representations

The procedure for instance creation operates on data structures as defined by the UML diagram shown in Fig 1. An instance of the shown classes is called *term* here, while an instance of a Modelica class is simply called *instance*.

2.1 Abstract syntax tree

A Modelica parser may use the classes shown in double-framed boxes to create an abstract syntax tree (AST) from a given SimpleModelica class definition. Fig 2 shows the AST that such a parser creates for the class definitions Ele and Ele1000 given above.

In the algorithm for instance creation, the Modelica class tree is represented by the constant `ROOT`, which is a ClassDefinition with no id and no parent that contains all top-level class definitions (typically packages).

Moreover, the constant `GLOBAL` is a ClassDefinition with no id and no parent that contains following built-in ClassDefinitions

(1) primitive types RealType, IntegerType, StringType, BooleanType, e.g.

```
class RealType
end RealType;
class StringType
end StringType;
```

(2) predefined types defined using these primitive types, e.g.

```
class Real
  RealType value; //accessed without dot-notation
  StringType unit;
  RealType min;
  RealType max;
end Real;
```

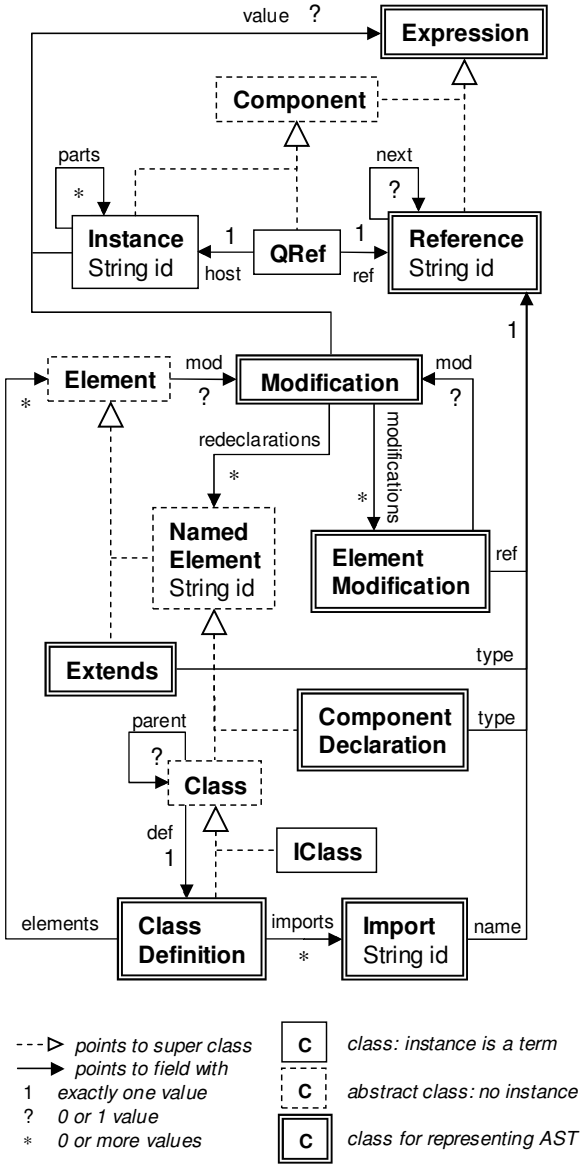

Fig 1: Classes used to implement instance creation

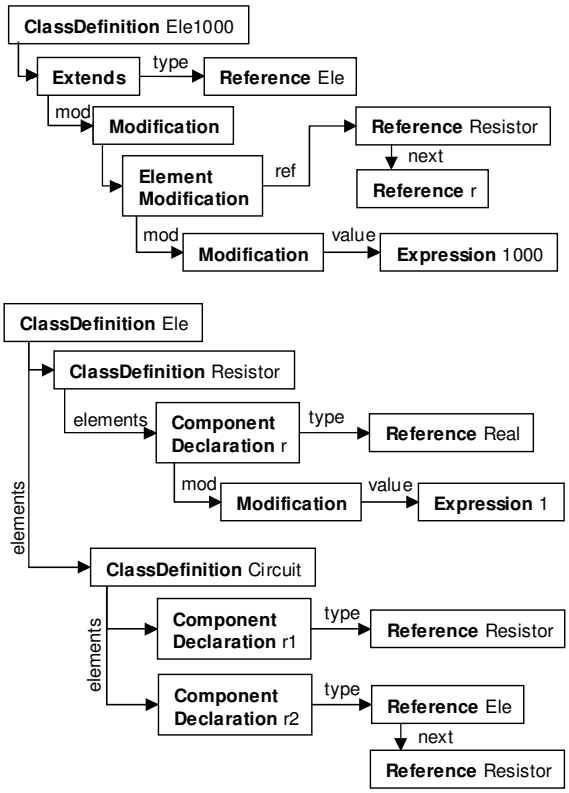

Fig 2: AST for classes Ele1000 and Ele

2.2 Implicit classes: IClass

An *implicit class* is a class that has no explicit definition in the Modelica class tree. In the example above, class Ele1000.Circuit is implicit, because the AST of Ele1000 does not contain a ClassDefinition named Circuit. Class Circuit is only inherited to Ele1000 through its base class Ele. Nevertheless, class Ele1000.Circuit can be instantiated, and the resulting instance differs from the result of instantiating class Ele.Circuit: Ele1000.Circuit.r1.r = 1000 while Ele.Circuit.r1.r = 1.

In other words: Ele1000.Circuit and Ele.Circuit are two different classes. In general, a class A may modify its base classes, which may modify all elements inherited by A from these base classes, including inherited classes.

To deal with implicit classes, a Modelica class is represented here by a tuple <ClassDefinition def, Modification mod, Class parent>, see Class in Fig 1. A ClassDefinition def from the Modelica class tree is complemented with a Modification mod that modifies this definition and a parent class that overrides the definition's parent. Subclasses of Class are ClassDefinition (for which mod is always null, and def refers to the ClassDefinition itself) to represent classes from the Modelica class tree, and IClass to represent implicit classes. The example Modelica classes are hence represented by the following terms

Modelica class	term is a	def, mod, parent
Ele	ClassDef.	Ele, null, ROOT
Ele.Resistor	ClassDef.	Ele.Resistor, null, Ele
Ele.Circuit	ClassDef.	Ele.Circuit, null, Ele
Ele1000	ClassDef.	Ele1000, null, ROOT
Ele1000.Resistor	IClass	Ele.Resistor, (r =1000), Ele1000
Ele1000.Circuit	IClass	Ele.Circuit, null, Ele1000

A ClassDefinition is created by the Modelica parser, while an IClass is created by the procedure for class name lookup during instance creation.

2.3 Qualified references: QRef

Modelica supports a *use before declare* policy for the components of a class. Moreover, there may be cyclic dependencies between components: to instantiate component a, we may need access to component b, and vice versa. This raises the question what kind of object the lookup of a component reference should return during instance creation. The referenced component (an instance) may not yet exist. To cope with this, we decide that lookup of a component may also return a pair <host, ref> where host is an Instance, and ref is a Reference, such that ref.id is the id of an instance in host.parts. This pair is called *qualified reference*, QRef for short. A QRef asserts that the host contains - or will contain - the referenced instance, and it represents this referenced instance. The host may still be under construction at the time the QRef is created, i.e. may not yet actually contain the referenced instance. Using QRefs, we can represent an instance before it actually exists. In contrast to a Reference, the meaning of a QRef does not depend on its context, but only on the host. This is why QRef s are called 'qualified'.

2.4 Qualified and unqualified modifications

A Modification is called *qualified*, if all its References have been replaced by QRefs. A qualified modification does not depend on its context, because all references have been looked up in some scope. In other words, the meaning of an unqualified modification depends on its context, while the meaning of a qualified modification does not.

ComponentDeclaration and Extends have unqualified modifications, while the modification of IClass is qualified or null, and the modification of ClassDefinition is always null. Example

```
class P
  class Ele2000 =Ele(Resistor.r=r2k);
  Real r2k(unit="Ohm")=2000;
end P
```

The extends clause "`=Ele(Resistor.r=r2k)`" of the short class definition Ele2000 is not qualified, i.e. the Modelica parser returns an Extends that contains the Reference "r2k".

The same holds for Modifications occurring in component declarations, such as `(unit="Ohm")=2000`.

In contrast, the Class returned by class lookup of P.Ele2000.Resistor is an IClass with parent P.Ele2000 and with the qualified modification (r = QRef(host = x, ref = "r2k")), where x is an instance of class P. Recall that a short class definition such as P.Ele2000 does not define its own scope, and hence "r2k" has to be looked up in the scope of P.

2.5 Instances

The objective of instance creation is to derive an object - called instance - that contains all inherited and locally declared components of a given Modelica class, with all occurring modifications applied. Fig 3 shows an instance of class Ele1000.Circuit.

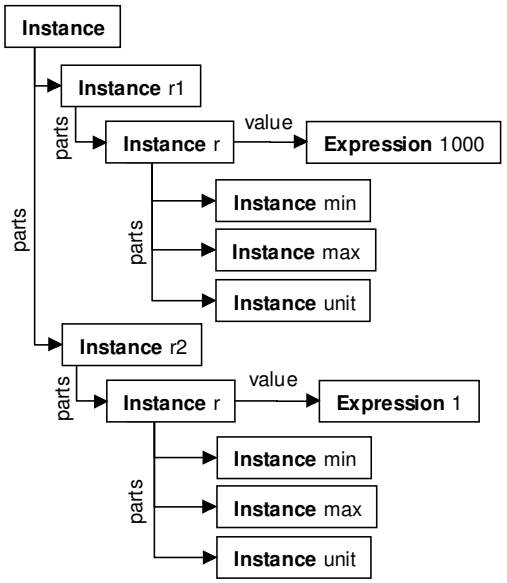

Fig 3: Instance of Modelica class Ele1000.Circuit

3 Algorithm for Instance Creation

Informally, instance creation as implemented below proceeds as follows

- take the term representing the class to be instantiated, e.g. class definition Ele.Circuit shown in Fig 2,
- replace each extends clause found in the class by the parts of an instance of the specified base class,
- replace each component declaration found in the class by an instance of the component type.
- The term resulting from all these replacements is called instance, see e.g. Fig 3.

In this context, a key algorithm is the procedure to lookup (resolve) a reference such as 'Resistor'. Lookup retrieves or computes the referenced component (a named instance) or class.

3.1 Name lookup

The following two procedures show how lookup works in principle, roughly on the level of detail as in the Modelica specification [1]. However, the procedure for instance creation given in section 3.2 uses an extended version, that (1) collects and merges class modifications encountered during lookup to support implicit classes (IClass) and (2) returns a Component (QRef) instead of a ComponentDeclaration, in case the reference refers to a component. Unfortunately, these extensions make the code for lookup less easy to understand. Therefore we also present a simplified version of name lookup here.

lookupName(Class c, Reference ref, Boolean isFirst) → NamedElement

```
1.  x ← lookupIdent(c, ref.id)
2.  if (x = null and isFirst)
3.    x ← import(c.def, ref.id)
4.  end if
5.  if (x ≠ null)
6.    if (ref.next = null) return x
7.    else if (x isa Class)
8.      return lookupName(x, ref.next, false)
9.    else
10.     assert x isa ComponentDeclaration
11.     xc ← the ClassDefinition that contains x
12.     type ← lookupName(xc, x.type, true)
13.     return lookupName(type, ref.next, false)
14.   end if
15. else if (isFirst)
16.     if (c.def is encapsulated or c = ROOT)
17.       return lookupName(GLOBAL, ref, true)
18.     else if (c.parent ≠ null)
19.       return lookupName(c.parent, ref, true)
20.     end if
21. end if
22. error "ref not found"
```

This procedure looks up the given reference in the scope of the given class, and either returns the first named element (class definition or component declaration) found, or signals a "ref not found" error.

The procedure searches the sequence of parents, until an encapsulated class or the unnamed root class of the Modelica class tree is reached. In both cases, search is continued (line 17) in the global scope that contains the predefined elements, such as Real, String, and time. Only the first identifier (as indicated by the isFirst argument) of a name is looked up using the import clauses of the class, see lines 2, 3, 4. E.g. when looking up reference A.B in the scope of class C, then A may be imported by C, but import clauses of A are ignored when looking for B in the scope of A in line 8.

lookupIdent(Class c, String id) → NamedElement

```
1.  if (c.def.elements contains
2.    a NamedElement e with e.id = id)
3.    return e
4.  else
5.    for each Extends ext in c.def.elements
6.      if (ext.type.id = id) return null
7.      end if
8.      base ← lookupName(c, ext.type, true)
9.      e ← lookupIdent(base, id)
10.     if (e ≠ null) return e
11.     end if
12.   end for
13.   return null
14. end if
```

Searches class c and its base classes for a named element with the given id. If id names a local or inherited named element of c, returns that element, returns null otherwise. This search does neither use imports nor parent classes. A tricky part of the algorithm is the test in line 6, which terminates a circular attempt to lookup a base class of c.

3.2 Instance creation for SimpleModelica

This section presents a set of procedures that implement instance creation for SimpleModelica.

instantiate(Reference name) → Instance

```
1.  c ← lookupClass(ROOT, name, null)
2.  ic ← elaborate(c, new Instance())
3.  ic ← replaceQRefs(ic)
4.  ic ← removeDuplicates(ic)
5.  return ic
```

This procedure instantiates the given class and returns the resulting instance. The class is specified by name. This way, also an implicit class (such as Ele1000.Circuit) can be instantiated. In line 1., the class name is looked up in the scope of the unnamed root of the Modelica class tree. In line 2., the entire instance tree is created. However, component references occurring in modifications are replaced by QRefs, not by the referenced instances, see section 2.3. In line 3., ic is the root of the completed instance tree. All referenced instances should have been created by then. Consequently, all QRefs can now be replaced by the referenced instances. In line 4., duplicate instances added through multiple inheritance are removed from the instance.

elaborate(Class c, Instance host) → Instance

```
1.  for each Extends ext in c.def.elements
2.    base ← getClass(c, ext, c.mod, host)
3.    host ← elaborate(base, host)
4.  end for
5.  for each ComponentDeclaration decl
6.  in c.def.elements
7.    if (redeclare(c.mod, decl.id) ≠ null)
8.      decl ← redeclare(c.mod, decl.id)
9.    end if
10.   qmod ← select(c.mod, decl.id)
11.   type ← getClass(c, decl, qmod, host)
12.   comp ← elaborate(type, new Instance())
13.   comp.id ← decl.id
14.   add comp to host.parts
15. end for
16. host.value ← c.mod.value
17. return host
```

This procedure adds for each inherited or locally declared component of the given class c an elaborated instance to the given host. In an elaborated instance, each component reference is represented by a QRef (see section 2.3), but not yet by the referenced instance. During elaboration, modifications are merged in the correct order. Redeclaration of a component is processed in lines 7 - 9.

getClass(Class c, Element e, Modification qm, Instance host) → Class

```
1.  type ← lookupClass(c, e.type, host)
2.  qmod ← qualify(c, e.mod, host)
3.  qmod ← merge(qm, qmod)
4.  return createClass(type, qmod)
```

This auxilliary procedure returns a base class (if e is an Extends) or component type (if e is a ComponentDeclaration) used during elaboration or lookup.

createClass(Class c, Modification qmod) → Class

```
1.  qmod ← merge(qmod, c.mod)
2.  if (qmod = c.mod) return c
3.  else return new IClass(c.def, qmod, c.parent)
4.  end if
```

This auxilliary procedure merges the qualified modification of the given class with the given qualified modification qmod, where elements of qmod override elements of c.mod. Returns either the given class (e.g. if qmod = null), or a new IClass (see 2.2).

lookupClass(Class c, Reference name, Instance host) → Class

```
1.  x ← lookup(c, name, true, host)
2.  if (x isa Class) return x
3.  else error "not a class"
4.  end if
```

Look for the given class name in the scope of class c. The host is either null, or an elaborated instance of c. The given host may be under construction, i.e. not yet completely elaborated. If lookup should require to instantiate c (e.g. to access a component of c that occurs in a modification, see r2k in the example in section 2.4) the host, if given, is used. Otherwise, c is elaborated on demand.

lookup(Class c, Reference ref, Boolean isFirst, Instance host) → Class or Component

```
1.  x ← lookup(c, ref.id, host)
2.  if (x = null and isFirst)
3.    x ← import(c.def, ref.id)
4.  end if
5.  if (x ≠ null)
6.    if (ref.next = null) return x
7.    else if (x isa Class)
8.      return lookup(x, ref.next, false, null)
9.    else
10.     assert x isa QRef
11.     return new QRef(x.host, ref)
12.   end if
13. else if (isFirst)
14.     if (c.def is encapsulated or c = ROOT)
15.       return lookup(GLOBAL, ref, true, null)
16.     else if (c.parent ≠ null)
17.       return lookup(c.parent, ref, true, null)
18.     end if
19. end if
20. error "ref not found"
```

Similar to procedure lookupName, defined in section 3.1. However, if ref names a component, the component is returned (as QRef), not its declaration.

Again, if isFirst is false, then only locally declared or inherited elements are found, i.e. import clauses, the parent of c as well as the global scope are ignored.

The next procedure looks up in class c for a local, inherited or redeclared class or component with the given id. Returns null, if no such class or component is found. This does not use imports or c's parent. If id names a component, the component is returned (represented by a QRef), and not (like procedure lookupIdent defined in section 3.1) the corresponding component declaration.

Lines 5 and 25 handle the case that id names a class that is redeclared by the qualified modification c.mod. If id names a component which is redeclared by c.mod, this redeclaration is either treated during elaboration of c in line 11, or in lines 19-20 in case the component is inherited from a base.

If id names a class inherited to c, then c becomes the new parent of this class (line 24).

lookup(Class c, String id, Instance host) → Class or Component

```
1.  if (c.def.elements contains
2.      a NamedElement e with e.id=id)
3.    if (e isa ClassDefinition)
4.      if (redeclare(c.mod, id) ≠ null)
5.        e ← redeclare(c.mod, id)
6.      end if
7.      return createClass(e, select(c.mod, id))
8.    else
9.      assert e isa ComponentDeclaration
10.     if (host = null)
11.       host ← elaborate(c, new Instance())
12.     end if
13.     return new QRef(host, new Reference(id))
14.   end if
15. else
16.   for each Extends ext in c.def.elements
17.     if (ext.type.id=id) return null
18.     end if
19.     base ← getClass(c, ext, c.mod, host)
20.     e ← lookup(base, id, host)
21.     if (e ≠ null)
22.       if (e isa Class)
23.         if (redeclare(c.mod, id) = null)
24.           e ← new IClass(e.def, e.mod, c)
25.         else e ← redeclare(c.mod, id)
26.         end if
27.         return createClass(e, select(c.mod, id))
28.       else return e
29.       end if
30.     end if
31.   end for
32.   return null
33. end if
```

A tricky part of this procedure is the test in line 17 which terminates a circular attempt to lookup a base class of c.

import(ClassDefinition c, String id) → Class or Component

```
1.  for each Import imp in c.imports
2.    if (imp matches "import A.B.C" and id="C")
3.      return lookup(ROOT, "A.B.C", true, null) else
4.    if (imp matches "import C = A.B" and id="C")
5.      return lookup(ROOT, "A.B", true, null) else
6.    if (imp matches "import A.B.*")
7.      ab ← lookupClass(ROOT, "A.B", null)
8.      e ← lookup(ab, id, null)
9.      if (e ≠ null) return e
10.     end if
11.   end if
12. end for
13. return null
```

Searches the import clauses of the given class definition for a named element with the given id. Returns the first matching class or component, or null if no match was found.

removeDuplicates(Instance host) → Instance

Remove all duplicate instances (instances with same id) from the given instance and return the resulting instance. Duplicates are caused by multiple inheritance. It is an error if two duplicate elements are not equivalent. Example:

```
class A Real x = 1; end A;
class B Real x = 2; end B;
class C
  extends A;
  extends B; // error
end C;
class D
  extends A;
  extends B(x = 1); // ok
end D;
```

Before application of removeDuplicates, instances of C and D contain a duplicate component x. The procedure removes x from D, but signals an error for C, because components x = 1 and x = 2 are not equivalent.

qualify(Class c, Modification mod, Instance host) → Modification

Lookup each Reference contained in the given unqualified modification in the scope of the given class, replace it with the resulting Class or QRef, and return the resulting qualified modification. See 2.4. The given host is either null or, if available, an elaborated instance of class c to be used as argument for name lookup in the scope of class c.

replaceQRefs(Instance host) → Instance

Replace each QRef contained in the given instance by the referenced Instance and return the resulting instance. It is an error if an instance referenced by a QRef is not found in the QRef's host. (For unrestricted Modelica, this method also performs dynamic lookup of the `inner` component, in case a QRef references an `outer` component.)

redeclare(Modification env, String id) → NamedElement

```
1.  if (env.redeclarations contains x with x.id = id)
2.    return x
3.  else return null
4.  end if
```

If the given modification redeclares an element with the given id, return the element. In the AST generated by a parser, the parent of a redeclared class is the class that contains the redeclaration.

Example:

```
class A = B(redeclare class C = D);
```

In the corresponding AST, the parent of C is A.

select(Modification env, String id) → Modification

```
1.  if (env.modifications contains x with x.ref.id = id)
2.    if (x.ref.next = null)
3.      return x.mod
4.    else
5.      mod ← new Modification()
6.      em ← new
7.        ElementModification(x.ref.next, x.mod))
8.      add em to mod.modifications
9.      return mod
10.   end if
11. else return null
12. end if
```

If the given modification modifies an element with the given id, this procedure returns the corresponding modification. Otherwise returns null.

Examples:

- select((R.r =10), "r") returns null
- select((R.r =10), "R") returns (r =10)
- select((r =10), "r") returns =10
- select((r), "r") returns null

merge(Modification env, Modification mod) → Modification

```
1.  if (env=null) return mod
2.  else if (mod=null) return env
3.  else
4.    result ← copy of env
5.    for each ElementModification em
6.    in mod.modifications
7.      if (select(env, em.ref.id) = null)
8.        add em to result.modifications
9.      end if
10.   end for
11.   for each NamedElement e
12.   in mod.redeclarations
13.     if (redeclare(env, e.id) = null)
14.       add e to result.redeclarations
15.     end if
16.   end for
17.   if (env.value = null)
18.     result.value ← mod.value
19.   end if
20.   return result
21. end if
```

Merge the given modifications, where elements in env beat (override, replace, update) elements in mod, and return the resulting merged modification.

The merge operation is associative, not commutative, and merge(null, m) = merge(m, null) = m for every Modification m. See [1] for a more detailed specification of the merge operation.

Examples:

- merge((x=1,y=2), (x(min=6)=3, z=4)=5) returns the modification (x=1, y=2, z=4)=5
- merge((x), (x=1)) returns (x=1)

3.3 Extension to Arrays

SimpleModelica can be extended to arrays by adding (updating resp.) the follwing syntactic definitions.

```
class_specifier :  { element ";" } end IDENT |
                   "=" name [ subscripts ] [ class_modification ]
component_declaration : name [ subscripts ] IDENT [ modification ]
element_modification : [ each ] reference [ modification ]
expression : NUMBER | STRING | true | false | reference |
                   "{" expression { "," expression } "}"
reference : IDENT [ subscripts ] [ "." reference ]
subscripts : "[" (":" | expression) { "," (":" | expression) } "]"
```

Example:

```
class P =Real[2](unit={"x","y"});
class A
   Real[n,n+1] a;
   Real[:] b(each min=1)={2,n,4};
   Integer n = 3;
end A
```

A challenge introduced by arrays is the need to evaluate expressions during instance creation.

- The parameter expressions that specify array size must be evaluated, and return positive integer sizes. E.g. to instantiate component a in class A, expressions n and n+1 must be evaluated.
- The modifier of an array must be split in order to get one single modifier for each array element. E.g., to instantiate b in class A, b's modification is split into three modifiers (min=1)=2, (min=1)=n, and min(=1)=4.

To represent arrays, we use a new class Array, which extends Instance (see Fig 1) and defines the fields elementType, subs, and mod where

- elementType is a Class
- subs is a qualified subscripts expression defining the array size, e.g. [3, 4]
- mod is an optional qualified array modification, e.g. (unit = {"r", "g", "b"}) = {1, 2, 3}.

Expansion of arrays is delayed until the class being instantiated has been elaborated and hence array size expressions can be evaluated. After expansion of an array a, the field a.parts (inherited to Array from Instance) contains the array elements.

A notable feature introduced by arrays are component references that cannot be resolved to a unique component during instance creation.

Example:

```
Real a[:] = {10, 20};
Real b = a[if (time<1) then 1 else 2];
```

The value of b cannot be identified with a unique array element during instance creation.

The following procedures extend instance creation as presented so far to arrays.

instantiate(Reference name) → Instance

```
1.  c ← lookupClass(ROOT, name, null)
2.  ic ← elaborate(c, new Instance())
3.  ic ← expandArrays(ic)
4.  ic ← replaceQRefs(ic)
5.  ic ← removeDuplicates(ic)
6.  return ic
```

The only difference to the procedure given in 3.2 is the inserted line 3, which expands arrays contained in ic by evaluating array size expressions and creating and inserting the corresponding array elements.

elaborate(Class c, Instance host) → Instance

```
1.  for each Extends ext in c.def.elements
2.    if (ext.subs ≠ null)
3.      return createArray(c, ext, c.mod, host)
4.    else
5.      base ← getClass(c, ext, c.mod, host)
6.      host ← elaborate(base, host)
7.    end if
8.  end for
9.  for each ComponentDeclaration decl
10. in c.def.elements
11.   if (redeclare(c.mod, decl.id) ≠ null)
12.     decl ← redeclare(c.mod, decl.id)
13.   end if
14.   qmod ← select(c.mod, decl.id)
15.   if (decl.subs ≠ null)
16.     comp ← createArray(c, decl, qmod, host)
17.   else
18.     type ← getClass(c, decl, qmod, host)
19.     comp ← elaborate(type, new Instance())
20.   end if
21.   comp.id ← decl.id
22.   add comp to host.parts
23. end for
24. host.value ← c.mod.value
25. return host
```

The only difference to the elaboration procedure given in 3.2 are lines 2-5 and 15-17, which treat the case that a short class definition or component declaration contains subscripts.

createArray(Class c, Element e, Modification qm, Instance host) → Array

```
1.  elementType ← lookupClass(c, e.type, host)
2.  qsubs ← qualify(c, e.subs, host)
3.  qmod ← merge(qm, qualify(c, e.mod, host))
4.  return new Array(elementType, qsubs, qmod)
```

This auxilliary procedure creates an array with the given fields. The returned array is not yet expanded, but it represents (is equivalent to) an expanded array.

expandArrays(Instance host) → Instance

```
1.  if (host isa Array)
2.    expr ← left-most expression in host.subs
3.    next ← host.subs without left-most expression
4.    n ← vectorSize(expr, host.mod)
5.    for i in 1 to n
6.      modi ← split(host.mod, i, n)
7.      if (next = null)
8.        ci ← createClass(host.c, modi)
9.        xi ← elaborate(ci, new Instance())
10.     else
11.       xi ← new Array(ci, next, modi)
12.     end if
13.     add expandArrays(xi) to host.parts
14.   end for
15. else
16.   for each Instance comp in host.parts
17.     replace comp by expandArrays(comp)
18.   end for
19. end if
20. return host
```

This procedure expands all arrays contained in the given host, and returns the resulting expanded instance.

split(Modification qmod, Integer i, Integer n) → Modification

Splits the given qualified modification into n parts and returns the ith part of it. Example:

- split((each unit="V"), 7, 10) returns (unit="V")
- split(={x, x+y, y}, 2, 3) returns = x+y

Note that Modelica arrays have a 1-based index, i.e. the first array index is 1 and not 0.

vectorSize(Expression qexpr, Modification qmod) → Integer

Determine vector size based on the given qualified integer-valued expression and the given qualified vector modification. This requires evaluation of qexpr, as well as evaluation of expressions occuring in qmod. Returns a positive integer, or signals an error. The following examples assume that parameter n evaluates to 3

- vectorSize(2, null) returns 2
- vectorSize(n, null) returns 3
- vectorSize(:, { x }) returns 1 without eval of x
- vectorSize(:, null) signals an error

3.4 Extension to unrestricted Modelica

To extend instance creation to full Modelica, the following remains to be done

- match **outer** references with the corresponding **inner** reference in the instance tree

- add qualified **equations** and **algorithms** to an instance, expand **for** clauses in equation sections and **connect** predicates i.e. generate the corresponding equations
- **validate** semantic constraints, e.g. (1) assertions associated with a **class category**, (2) **type constraints** of the Modelica type system, (3) **constraining clause** for redeclaration, (4) restrict modification to **public** and non-**final** elements of a class

The extension of the above algorithms to most of these features should be straightforward.

4 Application

Modelica's object-oriented approach to modeling opens new ways for systematically validating (models of) engineered systems, e.g. with respect to behavior in the presence of component faults and for alternative input scenarios. We have implemented instance creation as presented here for a large fragment of Modelica. This Java implementation (based on JavaCC, see [5]) is part of a bigger effort to develop a tool for automated simulation which, in a nutshell,

- instantiates a given annotated Modelica model
- extracts for each component of the system the corresponding fault modes (e.g. ok, stuckOpen, stuckClose) as indicated by specific annotations
- extracts information about the intended use of the system (inputs of the model)
- extracts information about specified, i.e. desired behavior of the system
- uses the extracted information to autonomously drive a large number of simulation runs, during which component faults are dynamically inserted and resulting behavior is classified w.r.t. specified behavior. The executable used for simulation is generated by Dymola.

Modelica applications like this one require access to instantiated Modelica classes. Modelica simulators such as Dymola do not currently offer an API to access instances, which currently forces application developers to implement instance creation on their own. This paper may help to reduce the required effort in the future.

5 Related Work

The *Modelica specification* [1] is available for free from www.modelica.org. The specification states that it defines the *static semantics* of Modelica in terms of a procedure for instance creation. Unfortunately, this is done in a quite informal way. No pseudo code is given, and no auxiliary representations (such as Instance, QRef, IClass) are explicitly defined. It would be helpful to complement the specification with a precise procedural definition of instance creation in the future. This paper may be a starting point.

Pelab at Linköping University has developed a *RML specification* [3] for a fragment of Modelica, which can be used to automatically generate [4] a procedure for instance creation. However, for a human reader interested in a procedural view on Modelica, e.g. to understand name lookup or to implement flattening, the RML specification (several thousand lines of declarative rules) is less helpful.

The design of Modelica was influenced by the *Theory of Objects* by Abadi and Cardelli [2]. This book defines various calculi (similar to Lambda-calculus) to model object-oriented languages. The calculi are composed from equational theories, called fragments. Based on these calculi, a procedure for instance creation in Modelica could perhaps be derived on formal grounds as follows

- define an equational theory E of objects
- direct the equations to convert E into a term rewrite system such that a term may be a Modelica class definition, and the irreducible term derived from that by a finite number or reductions is an instance of that class.

This is basically the idea underlying Pelab's RML specification. In this paper, we have chosen a less formal approach.

References

[1] Modelica Language Specification 2.1, www.modelica.org, 2004.

[2] Martin Abadi, Luca Cardelli: A Theory of Objects, Springer 1996.

[3] David Kågedal: A Natural Semantics Specification for the Equation-based Modeling Language Modelica. Master Thesis LiTH-IDA-Ex-98/48, Linköping University, 1998.

[4] David Kågedal and Peter Fritzson: Generating a Modelica compiler from natural semantics specifications. Proceedings of Summer Computer Simulation Conf. SCSC'98, 1998.

[5] JavaCC - Java Compiler Compiler, a Java Parser Generator, https://javacc.dev.java.net/