Operator Overloading in Modelica 3.1

Hans Olsson[1], Martin Otter[2], Hilding Elmqvist[1], Dag Brück[1],
[1]Dassault Systèmes, Lund, Sweden (Dynasim)
[2]German Aerospace Centre (DLR), Institute for Robotics and Mechatronics, Germany
Hans.Olsson@3ds.com, Martin.Otter@DLR.de,
Hilding.Elmqvist@3ds.com, Dag.Bruck@3ds.com

Abstract

The constructor and operator overloading introduced in Modelica 3.1 is discussed. The goal is that elementary operators like "+" or "*" can be overloaded for records. This makes it possible to define and use, in a convenient way, complex numbers, polynomials, transfer functions, state space systems, etc. The chosen approach is different to other languages: (a) Only scalar operations need to be overloaded. Array operations are then automatically available, so the growth of the number of overloaded functions is avoided. (b) Automatic type casts between different data types is performed using overloaded constructor functions. Again this reduces the number of overloaded functions. (c) The approach is conservative and only allows overloading if no ambiguity is present, in order to not introduce pitfalls into the language. This is reached by basing the overloading on disjoint sets of matching functions and not on a priority match.

Keywords: overloading, automatic overloading of arrays, overloading without ambiguities.

1 Introduction

Operator overloading is a well known concept in computer science and is available in languages such as Ada (ANSI 1983), C++ (ISO 1998), C#, Mathematica, Matlab and Python. In 2002-2005 the Modelica Association has worked on operator overloading for the Modelica language and several different versions have been designed by different people, especially to avoid some of the known problems of overloading from other languages. The work was then suspended for some years to concentrate on the improved safety in Modelica 3.0. Work has restarted in 2008: Based on a prototype implementation in Dymola and by applying this prototype to the Beta version of the Modelica_LinearSystems2 library *(Baur et. al. 2009)*, the 7th design version from 2005 was revised considerably and finally resulted in a version that has been included in Modelica 3.1 *(Modelica 2009)*.

The overloading introduced in Modelica 3.1 is seen as a first step and more features might be introduced later, based on the gained experience. The design is conservative and restrictive in order to reduce the probability to introduce pitfalls in the language. For example, ambiguities are not allowed. This is opposed to other languages where ambiguities are often resolved by priorities in function matches. An important, new feature is that it usually suffices to overload scalar operations and that array operations are automatically mapped to the overloaded scalar operations. The benefit is that explosive growth of the number of overloaded functions to define all possible combinations of data types and number of array dimensions is avoided.

2 Example with Complex numbers

The basic properties of operator overloading in Modelica 3.1 shall first be demonstrated by an example to introduce a user-defined data type Complex. In section 3, the formal rules are defined and design considerations are explained.

Assume a record "Complex" with overloaded scalar operators is available (see below). When using this definition in an interactive environment, e.g., in a Modelica script file that is executed by Dymola *(Dymola 2009)*, then in the command window of Dymola the output as shown in the right part of **Figure 1** appears.

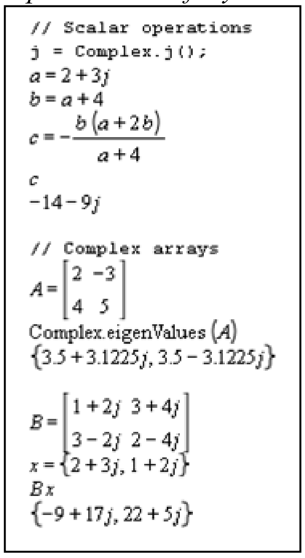
Script file

```
// Scalar operations
j = Complex.j();
a = 2 + 3*j
b = a + 4
c = -b*(a + 2*b)/(a+4)
c

// Complex arrays
A  = [2,-3; 4,5]
Complex.eigenValues(A)

B = [1+2*j, 3+4*j;
     3-2*j, 2-4*j]
x = {2+3*j, 1+2*j}
B*x
```

Output window of Dymola



```
// Scalar operations
j = Complex.j();
```

$a=2+3j$

$b=a+4$

$c=-\frac{b\left(a+2b\right)}{a+4}$

c

$-14-9j$

```
// Complex arrays
```

$A=\begin{bmatrix}2 & -3\\ 4 & 5\end{bmatrix}$

$\text{Complex.eigenValues}\left(A\right)$

$\{3.5+3.1225j, 3.5-3.1225j\}$

$B=\begin{bmatrix}1+2j & 3+4j\\ 3-2j & 2-4j\end{bmatrix}$

$x=\{2+3j, 1+2j\}$

Bx

$\{-9+17j, 22+5j\}$

Figure 1: Using the overloaded Complex data type in a script file (left) and the output in the command window of Dymola 7.3 (right).

From this example it can be seen that the user defined Complex type can hardly be distinguished from a built-in type like Real. In particular, standard array operations can be applied on Complex, although only the scalar operations are overloaded. Also type casts from Real or Integer to Complex are automatically performed, for example in "a = 2 + 3*j" where 2 is added to the Complex expression "3*j").

The "essential" difference to a built-in type is the name look-up: If a variable is declared as "Real a", then it is first determined whether "Real" is a built-in type before performing another lookup. If a variable is declared as "Complex c", then "Complex" is searched hierarchically from the current scope up to the global scope. For example, if a user introduces an own "Complex" type in the local scope, then this type is used and not the one from the global scope.

For the example above, the following definitions are needed:

```
record Complex
  Real re "Real part";
  Real im "Imaginary part";

  function j
    output Complex result;
  algorithm
    result := Complex(0,1);
  end j;

  operator 'constructor'
    function fromReal
      input  Real re;
      input  Real im=0;
      output Complex result;
    algorithm
      result = Complex(re=re,im=im);
    end fromReal;
  end 'constructor';

  operator '+'
    function add
      input  Complex c1;
      input  Complex c2;
      output Complex result;
    algorithm
      result := Complex(c1.re + c2.re,
                        c1.im + c2.im);
    end add;
  end '+';

  operator '-'
    function negate
      input  Complex c;
      output Complex result;
    algorithm
      result := Complex(- c2.re,
                        - c2.im);
    end negate;

    function subtract
      input  Complex c1;
      input  Complex c2;
      output Complex result;
     algorithm
       result := Complex(c1.re - c2.re,
                         c1.im - c2.im);
     end subtract;
   end '-';

   // also: '*', '/', '^', '==', '<>'

   operator 'String'
     function toString
       input  Complex c;
       input  String name="j";
       output String s;
     algorithm
       s := String(c.re);
       if c.im <> 0 then
          s := if c.im > 0 then
                  s + " + "
               else
                  s + " - ";
          s := s + String(abs(c.im))
                 + name;
       end if;
     end toString;
   end 'String';
end Complex;

function eigenValues
   input  Real    A [:,:];
   output Complex ev[size(A, 1)];
   import Modelica.Math.Matrices;
protected
   Integer nx=size(A, 1);
   Real    evr[nx,2];
   Integer i;
algorithm
   evr := Matrices.eigenValues(A);
   for i in 1:nx loop
     ev[i] := Complex(evr[i, 1],
                      evr[i, 2]);
   end for;
end eigenValues;
```

As can be seen, operator overloading is defined for functions that are defined in a record. The record definition holds a data structure in the usual way (here: two Real variables). Operators are defined in a record with the new construct

```
operator <name>
  …
end <name>
```

where <name> is the operator to be overloaded enclosed in apostrophes. This has the advantage that a valid, unique Modelica name is used which is very close to the operator that shall be overloaded.

Inside an "operator", one or more Modelica functions are defined. There are no particular requirements for these functions with the exception that every function must have exactly one output argument and that the number of arguments without a default value must be identical to the number of arguments required from the respective operator (e.g., function "add" inside operator '+' must have exactly two arguments without a default value. If there are more arguments, all must have a default value.

The special operator 'constructor' serves two purposes: First it gives different record constructors to provide various ways to generate an instance of the record. Second it is used to define automatic type casts. Examples:

```
// Default record constructor:
c1 = Complex(1,2);  // c1 = 1+2*j;

// Overloaded constructor "fromReal":
c2 = Complex(3);  // c2 = 3+0*j;

// Automatic type cast due to "fromReal":
c3 = c1 + 5;  // c3 = 6+2*j;
```

No overloaded operator is defined to add a Complex to a Real. However, a constructor is defined to generate a Complex number from the literal "5" and then there is an overloaded operator to add two Complex numbers.

3 Rules for Overloading

In this section the rules for the operator overloading are stated and design decisions are discussed.

3.1 Overloaded operators

A Modelica **record** can define the behavior for operations such as constructing, adding, multiplying etc. This is done using the specialized class **operator** (a restricted class similar to **package**) comprised of functions implementing different variants of the operation for the record class in which the respective **operator** definition resides. The overloading is defined in such a way that ambiguities are not allowed and give an error. Furthermore, it is sufficient to define overloading for scalars. Overloaded array operations are automatically deduced from the overloaded scalar operations, if an appropriately overloaded function for arrays is not present. The **operator** keyword is followed by the name of the operation which can be one of:

'constructor', '+', '-' (includes both subtraction and negation), '*', '/', '^', '==', '<>', '>', '<', '>=', '<=', 'and', 'or', 'not', 'String'.

The functions defined in the operator-class in the record must take at least one argument of this record type as input, except for the constructor-functions which instead must return one component of the record type. All of the functions shall return exactly one output.

The record may also contain additional functions, packages of functions, and declarations of components of the record. To avoid problems with slicing, it is not legal to extend from a record with operators.

The precedence and associativity of the overloaded operators is identical to built-in operators (e.g. '*' has always higher precedence as '+'). Definition of new operator symbols is not allowed. These restrictions simplify specification and implementation, and improve translation speed.

Only overloading of the most important operators is defined. In the future, this list might be extended, but the goal is to first get experience with a minimum set of overloaded operators.

3.2 Matching Functions

All functions defined inside the **operator** class must return one output and may include functions with optional arguments, i.e. functions of the form

```
function f
   input  A1 u1;
   …
   input  Am um = am;
   …
   input  An un;
   output B  y;
algorithm
   …
end f;
```

The vector P below indicates whether argument m of f has a default value (**true** for default value, **false** otherwise). A call f(a_1, a2,…, a_k, $b_1 = w_1$,…, $b_p= w_p$) with distinct names b_j is a valid match for the function f, provided (treating Integer and Real as the same type)

- A_i = typeOf(a_i) for $1 \leq i \leq k$,
- the names $b_j = u_{Qj}$, $Qj > k$, A_{Qj} = typeOf(w_i) for $1 \leq j \leq p$, and
- if the union of {i: $1 \leq i \leq k$ }, {Qj: $1 \leq j \leq p$}, and {m: P_m **true** and $1 \leq m \leq n$ } is the set {i: $1 \leq i \leq n$}.

This corresponds to the normal treatment of a function call with named arguments, requiring that all inputs have some value given by a positional argument, named argument, or a default value (and that positional and named arguments do not overlap). Note that this only defines a valid call, but does not explicitly define the set of domains.

3.3 Overloaded constructors and operators

As defined in detail in the Modelica language specification *(Modelica 2009)*, using an operator (such as '+') goes through a number of steps where a set of functions is found, and if one of them is a matching function it is used; multiple matches are seen as an error.

Array operations are defined in terms of the scalar operation, for multiplication assuming that the scalar element form a non-commutative ring that does not necessarily have a multiplicative identity (since the definition in the specification implicitly assumes that addition is associative and commutative); the operations vector*vector and vector*matrix are explicitly excluded, since there are cases where this does not give the "natural" interpretation, e.g., for complex vectors. For the future it will be possible to extend operations with complex conjugate (allowing a clean definition of vector*vector) and zero (allowing e.g. matrix multiplication with zero inner dimensions); without invalidating existing models.

The precise rules for binary operations will be now presented to show the flavor of the definition:

Let *op* denote a binary operator like '+'and consider an expression *a op b* where *a* is of type *A* and *b* is of type *B*. An example is "2.0 + j", where "2.0" is of type Real and "j" is of type "Complex.

1. If *A* and *B* are basic types or arrays of such, then the corresponding built-in operation is performed (e.g., for "2 + 3", the built-in operation for two Integer numbers is performed).
2. Otherwise, if there exists exactly one function *f* in the union of *A.op* and *B.op* such that f(*a*,*b*) is a valid match for the function *f* , then *a op b* is evaluated using this function. It is an error, if multiple functions match. If A is not a record type, A.op is seen as the empty set, and similarly for B. Note, Having a union of the operators ensures that if A and B are the same, each function only appears once. In our example, "2.0 + j" has only a match in the Complex record after converting 2.0 to Complex: Complex.'+' and therefore a matching function was found.
3. Otherwise, consider the set given by *f* in *A.op* and a record type *C* (different from B) with a constructor, g, such that C.′constructor′.g(b) is a valid match, and f(a, C.′constructor′.g(b)) is a valid match; and another set given by *f* in *B.op* and a record type *D* (different from A) with a constructor, h, such that D.′constructor′.h(a) is a valid match and f(D.′constructor′.h(a), b) is a valid match. If the sum of the sizes of these sets is one this gives the unique match. If the sum of the sizes is larger than one there is an ambiguity which is an error.
   Informally, this means: If there is no direct match of "a op b", then it is tried to find a direct match by automatic type casts of "a" or "b", by converting either "a" or "b" to the needed type using an appropriate constructor function from one of the record types used as arguments of the overloaded "op" functions. Example using the Complex-definition from above:
```
Real    a;
Complex b;
Complex c = a+b;
// interpreted as:
Complex.′+′(
 Complex.′constructor′.fromReal(a),b);
```
4. If A or B is an array type, then the expression is conceptually evaluated according to the rules for arrays (Modelica 2009, section 10.6). The resulting scalar operations are then treated with 1-3. Example:
```
Complex A[2,2], x[2];
Complex b[2] = A*x;
// interpreted as:
b[1] = A[1,1]*x[1] + A[1,2]*x[2];
b[2] = A[2,1]*x[2] + A[2,2]*x[2];
// The scalar operations can now be
// treated with the rules for scalar
// operations
```
5. Otherwise the expression is erroneous.

3.4 Syntactical simplification

In many cases there is only one function in the operator; either because only one makes sense or because another is not yet added. This is handled by stating that

```
operator function '*'
       …
end '*';
```

is treated in the same way as

```
operator '*'
  function multiply
     …
  end multiply
end '*';
```

The advantage of the shorter form is that it reads nicer, and avoids introducing an arbitrary name of a function.

However, by stating that they are equivalent, no loss of functionality is introduced; and one can always later add additional overloaded variants in a safe way.

4 Design and future considerations

The overall design is intended as a first step, and intended to allow future extensions in a backward compatible way.

4.1 Operator as a "semi-package" in record

In the current design an operator defines a hierarchical level; grouping together the variants.

The alternative of having multiple overloaded functions with identical names and different signatures (as in C++) was considered; but rejected for several reasons including the fact that it would no longer be possible to uniquely reference a function by name. However, the syntactic simplification introduced avoids redundant levels.

Another alternative would be to have the operators defined in the enclosing scope of the record - similarly to Ada. This would have required a modification of the function call lookup to include some form of argument-dependent name lookup ("Koenig lookup") as in C++ (ISO 1998, Section 3.4.2). This would be complex to implement, and possibly influence existing function calls (note that in Modelica function calls normally use hierarchical names in contrast to many other languages). Furthermore it was found that it often leads to a two-step hierarchy where a record 'complex' was defined in a package 'complexPackage' merely containing the record and its operations (cf. "header files"); and this was not deemed attractive.

One of the drawbacks of this design is that new operations on existing types cannot be added without modification of classes, which may not be possible for protection or licensing reasons.

Stroustrup (1994, Chapter 11) describes several related design issues and tradeoffs for C++.

4.2 Symmetric

Binary operators are defined so that operations can either be found in left or right operands. This is needed in order to handle combinations with built-in types in a clean way.

4.3 Few priority levels

For function matching there are only a few levels defined; whereas, e.g. C++ has a much more detailed set of priorities between functions in order to handle type conversions and many arguments for general functions.

A number of such detailed rules were considered in the Modelica design group, but due to limited resources they could not be investigated. Thus such cases currently lead to ambiguities, these cases could in the future be disambiguated with more detailed rules – but the intent is that everything that is currently unambiguous will stay that way.

4.4 Fewer operators

It is common to define only a few operators and define others in terms of these. This is here done for array operations, but not for e.g. relational operators (usually everything is defined in terms of '<' and/or '=='). It was not clear how common overloaded relational operators will be in Modelica and for what purpose, and thus this was deemed as an issue that will be handled in the future.

An important consideration is whether relational operators will be used for general routines such as sorting as in the Standard Template Library of C++ (where '<' is more used as a sorting order than a mathematical total order); or for more general mathematical routines, e.g. computations for IEEE floating numbers including NaN where such rules do not hold.

4.5 Zero values and complex numbers

As indicated above matrix multiplication is currently undefined if the inner dimension is zero. A simple solution would be to introduce an operator 'zero' having no inputs and returning the additional identity of the class. An important consideration will be whether this operator should be required for matrix multiplication in general; and whether it should be used for other purposes.

Similarly vector*vector could be defined if there existed an operator 'conjugate' in the class.

4.6 Hierarchy of conversions

In the future it might be necessary to add another 'constructor'-operator containing only explicit constructors – i.e. constructors that, as in C++, only will be called if the constructor is explicitly invoked and not for implicit conversions.

Without this care must be taken when designing multiple records such that conversions form an ordered hierarchy.

At one point in the design it was considered to have conversions in both directions and instead introduce additional operators to disambiguate calls; e.g. have Complex and ComplexPolar that both can be converted automatically from the other one and instead define operations such as addition to disambiguate the results:

```
record Complex
  …
  operator '+'
    function addComplex
      input Complex a;
      input Complex b;
      output Complex c;
    …
    function addPolar "Example only"
      input Complex a;
      input ComplexPolar b;
      output Complex c;
    …
end Complex;
```

The problem with this approach is that c+2 is ambiguous since it is not clear if 2 should be converted to polar or Cartesian form before being added. It would be possible to handle this by having an additional operation for addition with Real; but it was deemed that the resulting number of functions grew too much and a cleaner design was to remove addPolar.

5 Conclusion

Modelica 3.1 was released in May 2009. The operator overloading as introduced in this new version was discussed and examples are given to demonstrate the usage. The introduced operator overloading is seen as a first step, to gain experience with it in Modelica. Especially, it is clear that function overloading is missing and has to be introduced.

With respect to other languages, the design is restrictive, but has the advantage that it usually suffices to define overloaded scalar operations between the same types. Array operations and operations between different types can then be automatically deduced by a Modelica tool.

6 Acknowledgements

Partial financial support of DLR by BMBF (BMBF Förderkennzeichen: 01IS07022F) for this work within the ITEA project EUROSYSLIB (http://www.itea2.org/public/project_leaflets/EUROSYSLIB_profile_oct-07.pdf) is highly appreciated.

Furthermore, we would like to thank Marcus Baur (DLR) for fruitful discussions.

References

ANSI (1983): **Ada Language Reference Manual.** ANSI/MIL-STD 1815A.

Baur M., Otter M., and Thiele B. (2009): **Modelica Libraries for Analysis and Design of Linear Control Systems**. In F. Casella (editor): Proc. of the 7th Int. Modelica Conference, Como, Italy. www.modelica.org/events/modelica2009

Dymola (2009). **Dymola Version 7.3**. Dassault Systèmes, Lund, Sweden (Dynasim). Homepage: www.dymola.com.

ISO (1998): **International Standard, Programming Languages – C++.** ISO/IEC 14882:1998.

Modelica (2009). **Modelica Language Specification 3.1**. www.modelica.org/documents/ModelicaSpec31.pdf

Stroustrup B. (1994): **The Design and Evolution of C++.** Addison-Wesley.