ISO/IEC JTC1/SC22/WG5 N1541

To: WG5/J3

From: Lawrie Schonfelder

Subject: Comments on draft Submodule TR J3/03-123

Date: July 2003

Subsequent to writing the previous version of this paper I have spent some time looking carefully at the issues related to the functionality needed for this enhancement. My conclusion is that the previous paper, J3/03-143, was broadly correct in its conclusions but not in its emphasis nor in its presentation.

Having decided that a submodule approach to supporting the separation of the design of user interface and facility implementation is the right one, the key issue is what is the relationship between a submodule and its parent module. In Fortran terms what is the nature of the association between entities in the parent and entities of the same name in a submodule. It should be noted that what we are defining as a submodule is an additional non-executable program unit similar to a module and which like a module constitutes a separate scoping unit.

A submodule is delimited by statements of the form

```
SUBMODULE (<parent-name>) <submodule-name>
…
ENDSUBMODULE <submodule-name>
```

The <parent-name> identifies the parent that will have declared a number of named entities and possibly accessed a number of others by use association or by a previous level of parent/child association. All of these entities will be visible in the submodule. The question is what are the association rules that apply to these entities within the submodule? Host association was suggested in 03-123. This I contend is very much an inappropriate choice.

Host association currently has two essential propertiies:

1. the host is a containing program unit at the source code level, and
2. if in the contained scope a name available from the host via host association is redeclared then the local name refers to a new local entity and access to the host entity is masked

Both of these are inappropriate for the submodule/parent relationship. By definition a submodule is not contained within its parent module. To have a submodule redeclaration of a parent entity create a new local entity that masks access to the parent entity is likely to cause an error. For example, with the 03-123 the following code structure would be legal.

```
MODULE POP
  INTEGER,PARAMETER::N=10
  FORWARD INTERFACE
    FUNCTION FUN(a)
      REAL::a(N),FUN
    ENDFUNCTION FUN
  ENDINTERFACE
ENDMODULE POP

SUBMODULE(POP)::SON
  INTEGER,PARAMETER::N=50  ! new local N masking the host associated N
  CONTAINS
  IMPLEMENTATION FUN
    FUNCTION FUN()   ! the interface is not redeclared
       ! the dummy argument will therefore appear with name a and size 20
       ! it will be a cause of some surprise then if the programmer writes
      a(1:N)=0.0
     ! and finds an array overflow
      ! body of function
      ....
    ENDFUNCTION FUN
  ENDIMPLEMENTATION FUN
  FUNCTION SUBFUN(a)
    REAL::a(N),SUBFUN  ! no bracketing means this N is the local one accessed normally
    ! body of function
  ENDFUNCTION SUBFUN
ENDSUBMODULE SON
```

A further complication arises since it would appear to be legal under the proposed host association rule that SUBFUN could have been named FUN. In this case we have the confusion of which FUN would be invoked by a reference to FUN in other procedures within the submodule.

An interface body declared in the parent can only refer to a parent entity. If this interface is redeclared in the submodule but a local entity of a similar name is also declared that masks the parent entity the characteristics of the procedure could be different and hence in error. Even if language is added to say that in this case the parent entity is accessed and not the local submodule entity there is much scope for confusion. Host association allows a whole variety of horrible codes to be produced that sometimes will be caught by the compiler and sometimes they will not but will just produce wrong results. Fundamentally any reference within a submodule to a name inherited from the parent should be a reference to the parent entity. This will greatly reduce the scope for inadvertent obscure and wrong code.

Use association is closer to what is required but is not totally appropriate either. As currently defined use association applies from one named program unit to another identified by name, which is essentially what applies for a submodule/parent. However, use association at present applies only via a USE statement that must name a module. The entities that are made accessible from this module are controlled first by the accessibility attributes declared for them in the module and secondly by the controls that are applied locally on the USE statement. A submodule of necessity must have access to all accessible entities from its named parent. There is no local control in the parent/child inheritance and the accessibility attributes in the parent do not apply to this association. Finally use association deals with redeclaration by the simple expedient of banning it. Any redeclaration of a name made visible by use association is currently defined as an error. Part of the point of the submodule concept is to subdivide the development of large facility packages so that possibly different programmers might be responsible for interface design (module) and implementation (submodule). It is therefore highly desirable that the essential parts of the module declarations be duplicated by redeclaration in the submodule as essentially processor checkable documentation for the implementor.

I contend a new parent/child submodule association is required. The required properties of this submodule association are:

1. all entities visible in the parent are accessible in the submodule,
2. redeclaration in part or in full of a parent entity is permitted but such declaration must confirm attributes and characteristics of the parent entity and are a reference to the parent entity, a new entity is not created,
3. redeclaration of a data entity may not change the definition status nor the initialisation value of a parent entity, and
4. a procedure may be defined by a procedure body at most once in any chain of descendants.

It should be noted that with this form of association between submodule and parent, host association still applies between the contained scope of a procedure and its containing host. In the case of the parent declared interface body it accesses the data environment of the parent by host association and for the implementation defined in a descendent, it accesses the data environment of its containing submodule by host association. In this case this includes the data environment of the parent inherited into the submodule by the association rules defined above, plus any new data environment declared within the submodule. This latter by definition must be additional to and different from the parent data. I contend that this is precisely the desired behaviour.

The only remaining language syntax that is needed is a keyword to indicate that a specific interface body declaration applies to a descendent procedure and not an external. This I contend is logically a qualification of a specific procedure not of an interface block. To qualify an interface block rather than the individual interface bodies complicates the construction of generic procedure sets and provides no compensating advantage. I would propose we spell this keyword `FORWARD` and that it be used as a prefix to the `FUNCTION` or `SUBROUTINE` header statement on an interface body.

With this definition of association no other language syntax is needed; the clumsy verbose `IMPLEMENTATION <proc-name>...ENDIMPLEMENTATION` bracketing is not required. Such bracketing would be needed if host association were used since within the brackets the effects must be partially counteracted. With submodule association as defined above the occurrence of the parent procedure name is sufficient to provide the reference to the relevant ancestor declaration. However, it may be desirable to locally distinguish a procedure body that is providing a definition for a "forward" procedure and one that is defining a "local" module/submodule procedure. To this end a companion prefix keyword could be usefully included that is attached to the header of the relevant procedure body definition. I have suggested that `SEPARATE` would be a suitable spelling for such a prefix.

The following is an example of the sort of program structure that is possible with this proposal. The basic package is one providing facilities for variable precision arithmetic (drawn from my VPA module). The interface declarations are included in a parent module and the implementation definitions are given in two submodules, one defines the arithmetic operations the other the logical comparison procedures.

```
MODULE VARIABLE_PRECISION_ARITHMETIC

PRIVATE
INTEGER,PARAMETER :: radd=8
INTEGER,PARAMETER :: rad=100000000
TYPE NUMBER
  PRIVATE
  INTEGER         :: exp=rad+2     ! holds the base rad exponent
  INTEGER,POINTER :: sig(:)=>NULL()! holds the significand
ENDTYPE NUMBER
INTEGER :: ndig=14   ! controls the current accuracy
                     ! initially set to provide at least 104D

INTERFACE ASSIGNMENT(=)
  FORWARD ELEMENTAL SUBROUTINE num_ass_num(var,expr)
    type(NUMBER),INTENT(IN) :: expr
    type(NUMBER),INTENT(INOUT) :: var
  ENDSUBROUTINE num_ass_num
  FORWARD ELEMENTAL SUBROUTINE num_ass_int(var,expr)
    INTEGER,INTENT(IN) :: expr
    type(NUMBER),INTENT(INOUT) :: var
  ENDSUBROUTINE num_ass_int
ENDINTERFACE  ASSIGNMENT(=)

INTERFACE OPERATOR(+)
  FORWARD ELEMENTAL FUNCTION num_plus_num(l,r)
    type(NUMBER),INTENT(IN) :: l,r
    type(NUMBER) :: num_plus_num
  ENDFUNCTION num_plus_num
FORWARD ELEMENTAL FUNCTION num_plus_int(l,r)
```

```
    type(NUMBER),INTENT(IN) :: l
    INTEGER, INTENT(IN) :: r
    type(NUMBER) :: num_plus_int
  ENDFUNCTION num_plus_int
FORWARD ELEMENTAL FUNCTION int_plus_num(l,r)
    INTEGER, INTENT(IN) :: l
    type(NUMBER),INTENT(IN) :: r
    type(NUMBER) :: int_plus_num
  ENDFUNCTION num_plus_num
  FORWARD ELEMENTAL FUNCTION plus_num(r)
    type(NUMBER),INTENT(IN) :: r
    type(NUMBER) :: plus_num
  ENDFUNCTION plus_num
ENDINTERFACE  OPERATOR(+)

INTERFACE OPERATOR(<)
  FORWARD ELEMENTAL FUNCTION num_lt_num(l,r) ! OPERATOR(<)
    type(NUMBER),INTENT(IN) :: l,r
    LOGICAL :: num_lt_num
  ENDFUNCTION num_lt_num
ENDINTERFACE  OPERATOR(<)

PUBLIC :: NUMBER,PRECISION,ASSIGNMENT(=),OPERATOR(+),OPERATOR(<)

ENDMODULE VARIABLE_PRECISION_ARITHMETIC
```

The first submodule will define assignment and the arithmetic operators,

```
SUBMODULE(VARIABLE_PRECISION_ARITHMETRIC)::VPA_ARITH_PROCS
CONTAINS

  SEPARATE ELEMENTAL SUBROUTINE num_ass_num(var,expr)
! redeclares and refers to interface from parent
    type(NUMBER),INTENT(IN) :: expr
    type(NUMBER),INTENT(INOUT) :: var
    ! implements assignment between NUMBER values
    ! body of procedure
  ENDSUBROUTINE num_ass_num

  SEPARATE ELEMENTAL SUBROUTINE num_ass_int(var,expr)
! redeclares and refers to interface from parent
    INTEGER,INTENT(IN) :: expr
    type(NUMBER),INTENT(INOUT) :: var
    ! implements assignment of an INTEGER to a NUMBER performing the required conversion
    ! body of procedure
  ENDSUBROUTINE num_ass_int

  SEPARATE ELEMENTAL FUNCTION num_plus_num(l,r)
! redeclares and refers to interface from parent
    type(NUMBER),INTENT(IN) :: l,r
    type(NUMBER) :: num_plus_num
    ! implements addition between a NUMBER and a NUMBER
    ! body of procedure
  ENDFUNCTION num_plus_num

  SEPARATE ELEMENTAL FUNCTION num_plus_int(l,r)
! redeclares and refers to interface from parent
    type(NUMBER),INTENT(IN) :: l
    INTEGER, INTENT(IN) :: r
    type(NUMBER) :: num_plus_int
    ! implements addition between a NUMBER and an INTEGER
    ! body of procedure
  ENDFUNCTION num_plus_int

  SEPARATE ELEMENTAL FUNCTION int_plus_num(l,r)
! redeclares and refers to interface from parent
    INTEGER, INTENT(IN) :: l
    type(NUMBER),INTENT(IN) :: r
    type(NUMBER) :: int_plus_num
    ! implements addition between an INTEGER and a NUMBER
    ! body of procedure
  ENDFUNCTION int_plus_num

  SEPARATE ELEMENTAL FUNCTION plus_num(r)
! redeclares and refers to interface from parent
    type(NUMBER),INTENT(IN) :: r
    type(NUMBER) :: plus_num
    ! implements monadic addition for a NUMBER
    ! body of procedure
  ENDFUNCTION plus_num

END SUBMODULE VPA_ARITH_PROCS
```

Note the redeclarations in the submodule reconfirm the attributes and characteristics of entities accessed from the parent. Also note the `SEPARATE` keyword is not strictly necessary since the names like `plus_num` are sufficient to provide the reference to the inherited `FORWARD` interface. The keyword does however provide additional "documentation" for the human reader that could be useful when, as is not uncommon, the submodule contains a number of auxiliary procedures that do not define procedure bodies for forward procedures.

The following submodule would independently implement the logical comparison operators for VPA

```
SUBMODULE(VARIABLE_PRECISION_ARITHMETRIC)::VPA_COMP_PROCS
CONTAINS
  SEPARATE ELEMENTAL FUNCTION num_lt_num(l,r)
! OPERATOR(<) the interfaces here are simple so are not fully redeclared
    ! implements the logical < comparison between NUMBER values
    ! body of procedure
  ENDFUNCTION num_lt_num
END SUBMODULE VPA_COMP_PROCS
```

This time the whole parent declarations are not repeated merely referenced from the parent declaration via the interface name `num_lt_num`.

As an example of what this proposal would permit, it would be possible for the module designer to present an implementation programmer with a "stub" submodule that contained the relevant declarations from the module. This would provide checkable exact "documentation" of the interface and relevant semantic information of the required implementation e.g. a stub like

```
SUBMODULE(VARIABLE_PRECISION_ARITHMETRIC)::VPA_ASSGN_PROCS


INTEGER,PARAMETER :: radd=8
INTEGER,PARAMETER :: rad=100000000
TYPE NUMBER
  PRIVATE
  INTEGER         :: exp=rad+2     ! holds the base rad exponent
  INTEGER,POINTER :: sig(:)=>NULL()! holds the significand
ENDTYPE NUMBER
INTEGER :: ndig=14   ! controls the current accuracy
```

```
                     ! initially set to provide at least 104D
CONTAINS

  SEPARATE ELEMENTAL SUBROUTINE num_ass_num(var,expr)
    type(NUMBER),INTENT(IN) :: expr
    type(NUMBER),INTENT(INOUT) :: var
    ! implements assignment between NUMBER values

  ENDSUBROUTINE num_ass_num

  SEPARATE ELEMENTAL SUBROUTINE num_ass_int(var,expr)
    INTEGER,INTENT(IN) :: expr
    type(NUMBER),INTENT(INOUT) :: var
    ! implements assignment of INTEGER to NUMBER value

  ENDSUBROUTINE num_ass_int

END SUBMODULE VPA_ASSGN_PROCS
```

The implementor has all the necessary information to perform the implementation task. If either the module designer or the submodule implementer makes a change in the redeclared entities there will be a compilation failure indicating this. Checkable redeclaration is a powerful aid to large project development and code maintenance.