Classification of actions
or Inheritance also for methods

Bent Bruun Kristensen,
Institute of Electronic Systems, Aalborg University Centre, Aalborg, Denmark

Ole Lehrmann Madsen,
Computer Science Department, Aarhus University, Aarhus, Denmark

Birger Møller-Pedersen,
Norwegian Computing Center, Oslo, Norway.

Kristen Nygaard,
Institute of Informatics, University of Oslo, Oslo, Norway

Abstract

The main thing with the sub-class mechanism as found in languages like C++, SIMULA and Smalltalk is its possibility to express *specializations*. A general class, covering a wide range of objects, may be specialized to cover more specific objects. This is obtained by three properties of sub-classing: An object of a sub-class inherits the attributes of the super-class, virtual procedure/method attributes (of the super-class) may be specialized in the sub-class, and (in SIMULA only) it inherits the actions of the super-class.

In the languages mentioned above, virtual procedures/methods of a super-class are specialized in sub-classes in a very primitive manner: they are simply *re-defined* and need not bear any resemblance of the virtual in the super-class. In BETA, a new object-oriented language, classes and methods are unified into one concept, and by an extension of the virtual concept, virtual procedures/methods in sub-classes are defined as *specializations of the virtuals* in the super-class. The virtual procedures/methods of the sub-classes thus inherits the attributes (e.g. parameters) and actions from the "super-procedure/method".

In the languages mentioned above only procedures/methods may be virtual. As classes and procedures/methods are unified in BETA this gives also *virtual classes*. The paper demonstrates, how this may be used to parameterize types and enforce constraints on types.

1. Introduction

One of the key language concepts associated with object-oriented programming is the notion of classes and sub-classes.

The essence of sub-classing is that objects of any sub-class of a class C to some degree have the properties of C-objects. To some degree, because sub-classes may introduce properties, that are "in contradiction" with the intention of the super-class. Languages supporting classes/sub-classes do not prevent programmers from making such sub-classes, but in various ways they support, that objects of sub-classes behave like objects of the super-class.

One such support is that an object of a sub-class *inherits* the attributes (in terms of variables and procedures/methods) of the super-class. If MessageWindow is a sub-class of Window, it is known that a

MessageWindow object has at least the same attributes as Window-objects. For languages with qualified references this is essential for the possibility of checking uses of references. A reference qualified by Window may only denote objects of class Window and of sub-classes to Window. When denoting an object by a Window qualified reference, then we are sure that the object has the attributes of Window, even if the object is a MessageWindow-object.

Most object-oriented languages support *virtual* attributes. A procedure attribute in e.g. SIMULA [SIMULA67] and C++ [Stroustrup] has to be declared explicitly as a virtual (in order to be a virtual attribute), while all methods in Smalltalk [Smalltalk] and Flavors [Cannon83] are virtual attributes.

Attributes declared in sub-classes are attributes, that objects of the sub-classes have *in addition to* those of the super-class, while virtual procedures/methods are both procedures/methods of the super-class and of the sub-classes. If an object is known to be a C-object or an object belonging to *some* sub-class of C, it will have at least the C-attributes, but no assumptions are made about the extra attributes of sub-classes. But it is assumed that a virtual procedure/method of C may be specialized in a sub-class so that it reflects the intention of this, and that this specialization is used when the virtual procedure/method is executed, even when it is used within the super-class.

Some object-oriented languages support objects with individual action sequences in addition to attributes. In SIMULA and BETA ([BETA83a], [BETA85], [BETA87]) objects have, in addition to attributes, an action sequence. In case of inheritance in these languages, the sub-class objects inherits the actions of the super-class. The details of how this is done will be given below.

When defining classes, sub-classes and especially virtuals, an element of abstraction called *generalization/specialization* is used. A super-class is a generalization, that is common for a set of classes, and sub-classes are specializations of this class to cover different special situations. One way of specializing is to add attributes in the sub-classes, but the most powerful mechanism is virtuals. By means of virtuals it is possible to specialize procedures/methods, that are attributes of the general class, so that they cover situations appropriate for the special sub-classes.

Even though this is the most common use of virtuals, most object-oriented languages with virtuals do not support, that definitions of virtuals in sub-classes may be specializations of the virtual definitions in the super-class. A sub-class method in Smalltalk and Flavors need not have more than the identifier in common with the corresponding method in the super-class. A recent improvement of SIMULA has made it possible to specify the parameters that all procedure definitions in sub-classes shall have. But apart from that, a definition of a virtual in a sub-class consist of a pure *re-definition* of the virtual definition in the super-class.

In the use of these languages it has, however, been recognized that a pure re-definition of the virtual is not always what is wanted. When (re)defining a virtual in a sub-class, the definition (or effect) of the virtual in the super-class is also wanted. In Smalltalk this is obtained by simply sending the message to the super-class (from within the method of the sub-class), so that the method of the super-class will be performed. Flavors provides a more sophisticated scheme (e.g. before and after methods). In SIMULA the virtual definition in the super-class is simply not accessible as part of or after a binding of the virtual.

In this paper it is demonstrated that by an extension of the virtual concept and by a unification of classes and methods, it is possible to use the sub-class mechanism when specializing virtual procedures/methods in sub-classes. This implies, that virtual procedures/methods in sub-classes are defined as *sub-procedures/methods* of the procedure/method in the super-class.

Section 2 describes the unification of classes and methods. This is a result of a more general unification covered by the notion of *pattern* in the object-oriented language BETA. Section 3 describes the virtual concept based on the general notion of pattern, and section 4 gives a series of examples on the use of virtual patterns.

2. The notion of pattern and sub-pattern in BETA

BETA is a language based on one abstraction mechanism, the *pattern*, covering concepts like classes, methods and types found in other languages.

A BETA program execution consists of a collection of *objects*. An object consists of an *attribute-part* and an *action-part*. Objects are described by *object descriptors*.

BETA supports objects of different *kinds*. *System* objects execute their actions in *concurrency* with other system objects, *component* objects execute their actions *alternating* (that is at most one at a time and interleaved at well-defined points) with other components, while *item* objects are executed as part of systems, components or other items. For the purpose of this paper only item objects are covered. For a description of concurrency and alternation in BETA see [BETA85, BETA87].

Even though objects may be of different kinds, there is only one form of object descriptor.

```
(#
    Decl₁; Decl₂; ...; Decl_n
enter In
do Imp
exit Out
#)
```

where $Decl_1; Decl_2; ... Decl_n$ are declarations of the attributes. The action part of an object consists of an enter-part, a do-part and an exit-part. The do-part, *Imp*, is an imperative that describes the actions of the object. Prior to the execution of an object, a value may be entered into the enter-part, *In*. After the execution of an object, a value may be delivered from the exit-part, *Out*. The enter-part corresponds to a value-parameter and the exit-part to a result-parameter.

A *pattern* is described by a *name* and an object descriptor:

```
P:   (#
        Decl₁; Decl₂; ...; Decl_n
    enter In
    do Imp
    exit Out
    #)
```

Objects generated according to P will have the *structure*, that is the same set of attributes, the same enter-part, the same do-part and the same exit-part. The kind of an object is not specified as part of its descriptor, but as part of its generation. We will say, that an object generated according to P is a P-object or an object of P.

A pattern may be a *specialization* (or a *sub-pattern*) of another pattern. This is specified by *prefixing* the object descriptor of the sub-pattern with the name of the super-pattern:

```
P1: P (#
        Decl'₁; Decl'₂; ...; Decl'_m
    enter In'
    do Imp'
    exit Out'
    #)
```

A P1-object will have the attributes declared in P and those declared in P1. Execution of a P1-object will consist of executing the actions of P, specified by *Imp*. Execution of the special imperative **inner** in the actions of P imply the execution of the actions of P1, specified by *Imp´*.

The **inner** construct is the mechansim, that is provided for specialization of actions. According to [Thomsen] specialization of actions may be obtained in two ways: as a specialization of the effect of the general action or as a specialization of the partial ordered sequence of part-actions, that constitue the general action. The **inner** mechanism supports the last form of specializations. Examples will be given below.

Attributes of objects in BETA may be part objects, references to other objects and patterns. As an example consider a general pattern Window in a window system :

```
Window: (# UpperLeft,LowerRight:@ Point;
            Next: ^ Window;
            Display: (# ... #);
         #)
```

A Window object will have the part objects (@) UpperLeft and LowerRight (of the pattern Point), a reference (^) Next to someWindow object and a pattern attribute Display.

In case pattern attributes are used to generate and execute item objects, they correspond to local procedures in SIMULA and to methods in Smalltalk and Flavors. As BETA has a general sub-pattern concept, also covering item objects acting as local operations, we have in Smalltalk terms "classes and sub-classes of methods".

It is not a new idea to use the class/sub-class mechanism for methods. In 1975 Vaucher proposed prefixed procedures as a structuring mechanism for operations [Vaucher]. In BETA this is a consequence of a more general notion of pattern/sub-pattern. As Vaucher has demonstrated, a class/sub-class mechanism for methods is useful in itself. In the next section the notion is used in the specification of virtuals.

As an example of the usefulness of specialization of action, consider a pattern defining a small set of integers. The pattern attribute ForAll defines a general scanning of the elements of the set:

```
SmallIntSet:
    (# A:[100] @ Integer;
       Top:@ Integer;
       ...
       ForAll:
           (# Index:@ Integer;
              Current: (# exit S[Index] #)
           do 1 → Index;
              Loop:
              (if (Index ≤ Top)
              // true then
                   inner;
                   Index + 1 → Index;
                   restart Loop
              if)
           #);
    #);
```

Given a SmallIntSet-object S, a specialization of ForAll of S is obtained by prefixing with S.ForAll:

```
do . . .; S.ForAll(# do Current → DoSomething #); . . .
```

For each execution of **inner** in the general S.ForAll, the specialization, that is Current → DoSomething, is executed, doing something for each element in the set.

3. Virtual patterns

A pattern attribute may be specified *virtual*. This implies almost the same thing as for methods in Smalltalk and Flavors: Any generation and execution of objects according to a virtual pattern implies the generation and execution of objects according to a possible definition (or *binding*) of the virtual in the actual sub-pattern.

BETA has an extension of this known scheme: *a virtual pattern is qualified with a pattern*, and any binding of the virtual pattern (in sub-patterns of the pattern with the virtual pattern attribute) must be a *sub-pattern of the qualifying pattern*. This has the consequence that objects of a virtual pattern are known to be objects of at least the qualifying pattern. This makes it easier to preserve the intention of a virtual attribute in sub-patterns. It may also be used for checking purposes and for generating more efficient code for generation and execution of objects according to the virtual pattern.

A pattern attribute V of pattern P is declared as virtual by:

```
P: (# V:< Q #)
```

Q is the name of a pattern and is the qualification of V. In sub-patterns of P, the virtual V may only be bound to a sub-pattern of Q. So even though V may be defined differently in different sub-patterns of P, it is still in P known to be at least a Q. The bindings of a virtual pattern in different sub-patterns are thus enforced to be specializations of the definition in the super-pattern.

In P-objects and in objects of sub-patterns of P with no binding of V, the qualifying pattern Q is the definition of V. So the qualifying pattern is also *default-binding*.

A *further* binding of V in sub-patterns of P is specified by:

```
P1: P  (#   V::< Q1   #)
```

where Q1 is a sub-pattern of Q. Q1 does not have to be an immediate sub-pattern of Q. Q1 is both the binding of V and the the qualification of the (still) virtual V.

A shorthand is provided so the qualification does not have to be a separate pattern:

```
P: (#   V:< (# X,Y:@ Real#)   #)
```

The binding of V in

```
P1: P  (#   V::< (# Z:@ Real #)   #)
```

is a shorthand, that binds V to a sub-pattern of the qualification, so in P1 a V-object will have the attributes X,Y and Z.

A further binding as specified above, that is V::< Q1, specifies that V is also virtual in the sub-pattern P1. If V is bound by a *final binding*, of the form

```
P1: P  (#   V:: Q1   #)
```

then V is not virtual in P1 and it is not possible to bind V in sub-patterns of P1.

4   Examples on specialization of actions and virtual patterns

4.1   Why specialization of virtuals

As a very general pattern in a window system we may have the pattern Window:

```
Window:
(#
  UpperLeft,LowerRight:@ Point;

  Display:(#  { display the border of the  Window } #);
#)
```

As described here, a Window object may be displayed by executing an object of the local pattern attribute Display, and it will only display the border of the window.

Suppose that we have the following specializations of Window:

```
WindowWithLabel: Window (# Label:@ LabelType  #)

MessageWindow: Window (# Message:@ Text #)
```

We would like windows and their Display attribute to have two properties:

1. All objects of sub-patterns of Windows may be displayed, independently of which sub-pattern and independently of how a sub-pattern will be displayed. For example a WindowWithLabel will display the

label at a special position in the window, while a MessageWindow will display the message in some form. Given an object of pattern Window or of any sub-pattern, it shall be assured, that it has a Display attribute, and that this is the one defined especially for the actual sub-pattern of Window.

```
SomeWindow.Display
```

shall lead to the Display defined in the pattern of the object currently denoted by SomeWindow. SomeWindow is an object reference qualified by Window, which means that it may denote Window-objects or objects of sub-patterns to Window.

2. Display of sub-patterns of Window shall, in addition to displaying the information special for the sub-pattern, also display the border of the window. That is the Display attributes of sub-patterns shall be specializations of the Display of Window.

The former is obtained by declaring the Display attribute of Window as a virtual pattern. Then any generation and execution of Display objects will imply the generation and execution of Display objects according to the definition of Display in the actual sub-pattern. This is the same as for methods in Smalltalk.

The latter is obtained by declaring Display in Window not only virtual, but also give it a qualification, in this case a descriptor as part of the Display, describing the displaying of the border:

```
Window:
   (#
      UpperLeft,LowerRight:@ Point;

      Display:< (# do  { display the border }; inner  #);
   #)
```

By this declaration Display is specified as a virtual that at least displays the border. This means that in sub-patterns of Window, the Display may be given a descriptor which is appropiate for the actual sub-pattern of Window, but this descriptor will be a specialization of Display in Window.

```
WindowWithLabel:
      Window (# Label:@ LabelType;

                  Display::< (# do { display Label }; inner  #)
               #);

MessageWindow:
      Window (# Message:@ Text;

                  Display ::< (# do { display Message }; inner  #)
               #)
```

When e.g. Display of a MessageWindow is executed then the action of its super-pattern is executed, displaying the border of the window. Execution of **inner** will lead to the actions of the Display for MessageWindow and will display the message. MessageWindow and WindowWithLabel may be used as super-patterns and the Display may then be further specialized.

In Smalltalk the same would be done by sending the message Display to the super-class (from Display in the sub-class). The responsibility for preserving the property of Display defined in the super-class also in sub-classes is left to the sub-classes, while in BETA the super-pattern defines what is to be preserved of its virtual attributes in all sub-classes.

4.2   Specialization of Initialization

A well-known problem in Smalltalk-like languages is the "initialization procedure/method problem". Consider a class C with a virtual procedure/method Init. In sub-classes of C it is desirable that Init may be defined, so that it incorporates the Init of C. In Smalltalk it is necessary to send Init explicitly to the super-class.

In BETA the Init of the sub-patterns will be sub-patterns of the Init in C, and they will thus have as part of them (as super-pattern) the Init in C.

Consider a pattern defining point objects:

```
Point: (# X,Y:@ Integer;
          Init:< (# do 0 → X; 0 → Y; inner #);
      #)
```

The specification of Init indicates that it is a virtual pattern. A sub-pattern of Point may bind Init to a descriptor that is a sub-descriptor of the Init in Point:

```
ThreeDPoint: Point (#  Z:@ Integer;
                       Init::< (# do 0 → Z; inner  #);
                   #)
```

When executing the Init of ThreeDPoint, the actions of its super-pattern are performed, assigning 0 to both X and Y. Execution of the **inner** in Init of Point implies the execution of the actions in a possibly sub-pattern; in this case the assignment of 0 to Z. In case of a ThreeDPoint-object, the **inner** following 0 → z is an empty action, but in case ThreeDPoint is used further to define e.g. FourDPoint, then the **inner** would imply execution of initialization special for FourDPoint.

4.3   Type parameters

In the following example the SetType pattern local to SmallWindowSet does not define a method local to SmallWindowSet, but define (data) objects constituting a set of Windows.

```
SmallWindowSet:
   (#
      SetType:< Window;
      A:[100]@ SetType;
      Top:@ Integer;

      DisplaySet:
         (# do (for Inx:Top repeat A[Inx].Display  for)  #);
      ...
   #)
```

The type of the elements in the set is the virtual pattern SetType. The qualifying pattern Window says that the elements will at least be Window-objects. SetType may be bound to different sub-patterns of Window in different sub-patterns of SmallWindowSet, thereby obtaining specialized sets.

DisplaySet displays the whole set. As the elements of the set are known to be at least Window-objects, they will have a Display attribute.

A WindowWithLabel set is obtained by binding SetType to WindowWithLabel:

```
SmallWindowWithLabelSet:
   SmallWindowSet (# SetType::< WindowWithLabel  #)
```

All windows objects in this set will be WindowWithLabel objects. Note that while the bindings of Init in the previous example were to descriptors local to the sub-pattern (as it also are for re-definitions of methods in Smalltalk), SetType is here bound to a non-local pattern WindowWithLabel.

As Display of Window is a virtual pattern, then the DisplaySet as described in SmallWindowSet will also work for SmallWindowWithLabelSet. Execution of Display in

```
(for Inx:Top repeat A[Inx].Display  for)
```

will now be the execution of Display of WindowWithLabel.

4.4   Generics, polymorphism

A thorough comparison of generics (as found in e.g. Ada) and inheritance is found in [Meyer]. The following example demonstrates how constrainted genericity as defined in [Meyer] may be obtained by means of virtual patterns.

Consider the problem of writing a function Minimum, that is to work for two objects of any type. The constraint on the types in this situation is that a LessThan operation is defined. By use of virtuals this may be obtained by defining LessThan as a virtual pattern attribute of a pattern Type, and enforce the parameters of Minimum to be objects of at least the pattern Type. Type is assumed to be a general pattern. Objects of pattern Type wil e.g. be assignable. For the purpose of this example only the property that a LessThan operation is defined on Type objects is covered.

```
Type: (# OpType:< Type;
         ...
         LessThan:< (# Parameter:@ OpType;
                        Result:@ Boolean;
                     enter Parameter
                     do inner
                     exit Result
                     #);
      #);


Minimum:
    (#
       T:< Type;
       A,B,Result:@ T;
    enter (A,B)
    do
       (if A → B.LessThan
       //True   then A → Result
       //False  then B → Result
       if)
    exit Result
    #);
```

Notice, that as B is at least a Type object, it has a LessThan attribute, and A may be entered as a parameter. This is valid for for all bindings of T, as it may only be bound to sub-patterns of Type. The general Minimum, as described here, may therefore be checked independently of the binding of T.

All patterns that are to be assignable (and in this case comparable by LessThan) will be sub-patterns of Type. The binding of the virtual LessThan will for each sub-pattern give the implementation of LessThan for the appropriate type. As an example, Integer will be a sub-pattern of Type:

```
Integer: Type (# OpType:: Integer;
                 LessThan::
                     (# {implementation of LessThan for integers } #);
              #)
```

A Minimum function for integers is obtained by binding the virtual T of Minimum to Integer. The following specifies the execution of an object that has a descriptor prefixed with Minimum. The only thing specified in the sub-descriptor is the binding of the virtual T to Integer:

```
(I,J) → Minimum(# T::Integer #) → K;
```

The values of I and J are entered into the enter-part of the object prior to execution and the result is assigned to K after execution of the object.

4.5   Procedures/methods as parameters

Many languages allow parameters to procedures to be procedures. A procedure P with a formal procedure Fp is specified by

```
procedure P(Fp); procedure Fp;
begin
      ... ; Fp(1,2); ... ; Fp(3,4); ...
end;
```

When calling P with a procedure ActualFp

```
P(ActualFp)
```

the procedure ActualFp is passed, and all calls of Fp in the execution of P will be calls on ActualFp. ActualFp must then be a procedure with formal parameters that corresponds to the actual parameters (1,2) and (3,4), e.g.

```
procedure ActualFp(x,y); real x,y; begin ... end;
```

The actual procedure may be different in different calls of P. In simple cases a compiler may check that all possible actual parameters to P satify the requirements on the formal parameter, but in general this mechanis implies that each call of the formal parameter must be checked at execution time for correspondance between formal and actual parameters. An alternative is to specify as part of Fp the requirements on the actual procedure parameters.

In BETA the effect of a procedure as parameter is obtained by a virtual pattern, that is used to generate and execute item objects.

```
P: (#
      Fp:< Fpar
   do
      ...; (1,2) → Fp; ... ; (3,4) → Fp; ...
   #)
```

The qualification of Fp is a pattern, that defines the parameters:

```
Fpar:(# X,Y:@ Real enter(X,Y) #);
```

By qualifying with a pattern, that defines the parameters, it is assured, that all bindings of the virtual pattern Fp are to sub-patterns of Fpar. All bindings will thus have these parameters, and this may be checked at compile-time.

The calling of P with an actual procedure parameter is obtained by executing an item object according to a descriptor prefixed by P and with a binding of the virtual Fp:

```
do  ...; P (# Fp:: ActualFp #); ...
```

where ActualFp is a sub-pattern of Fpar and therefore has the parameters X,Y:

```
ActualFp: Fpar (# ... do ... #)
```

5. Conclusion

It has been demonstrated that by a slight extension of the virtual concept known from virtual procedures in SIMULA and methods in Smalltalk-like languages, a language may offer better support for specialization, in the sense that a method in a sub-class may be a specialization of the method in the super-class.

In BETA this is obtained by a generalization of langauge concepts like class, procedure, method, function and types into one concept, the pattern. Specialization of patterns in general thus implies specialization for patterns that act as methods. A pattern attribute, that is specified virtual and defined as an operation on the object, of which it is an attribute, works like a Smalltalk method. By requirering that a virtual pattern only may be bound to sub-patterns of a pattern qualifying the virtual pattern, it is enforced that all bindings in different sub-patterns are specializations of the qualifying pattern.

This mechanism may also be applied to languages, that do not have a generalization like the pattern concept. In e.g. Smalltalk the only requirement would be, that a method may be a specialization of a more general method - that is the sub-class mechanism should be applied also to methods and not only to classes.

Acknowledgements

We would like to thank Knut Barra, Axel Hagemann and Claus Nørgård for commenting the paper.

References

[BETA83a] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: *Abstraction Mechanisms in the BETA Programming Language.* Proceedings of the Tenth ACM Symposium on Principles of Programming Languages, 1983.

[BETA83 b] O.L. Madsen, B. Møller-Pedersen, K. Nygaard: *From SIMULA 67 to BETA.* Proceedings of the Eleventh SIMULA User's Conference, 1983

[BETA85] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: *Multi-sequential Execution in the BETA Programming Language.* Sigplan Notices, Vol. 20, No. 4 April 1985

[BETA87] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: *The BETA Programming Language.* To appear in: *Research Directions in Object Orienetd Programming.* Edited by B. Shriver and P. Wegner. MIT Press Spring 1987.

[Cannon83] H. Cannon: Flavors, A Non-Hierarchical Approach to Object-Oriented Programming. Draft 1982

[DELTA75] E. Holbæk Hansen, P. Haandlykken, K. Nygaard: *System Description and the DELTA Language.* Norwegian Computing Center, Oslo 1975

[Meyer] B. Meyer: *Generisity versus Inheritance.* In *OOPSLA, Object-Oriented Programming Systems, Languages and Applications.* Conference Proceedings, Sigplan Notices Vol.21 No.11 November 1986.

[SIMULA67] O.J. Dahl, B. Myhrhaug & K. Nygaard: *SIMULA 67 Common Base Language,* Norwegian Computing Center, 1968, 1970, 1972, 1984

[Smalltalk] A. Goldberg, D. Robson: *Smalltalk 80: The Language and its Implementation.* Addison Wesley 1983.

[Stroustrup] B. Stroustrup: *The C++ Programming Language.* Addison Wesley 1986.

[Thomsen] K.S. Thomsen: *Multiple Inheritance, a Structuring Mechanism for Data,Processes and Procedures.* DAIMI PB-209, Aarhus University, April 1986.

[Vaucher] J. Vaucher: Prefixed procedures: *A structuring Concept for Operations.* Infor, Vol 13, no.3, October 1975.