The Point of View notion for Multiple Inheritance

Bernard Carré, Jean-Marc Geib

Laboratoire d'Informatique Fondamentale de Lille, UA C.N.R.S. 369
Université des Sciences et Techniques de Lille Flandres Artois
Bât. M3, 59655 Villeneuve d'Ascq Cedex, FRANCE
Tel: (33) 20 43 42 55, (33) 20 43 45 13
E-mail: {carre geib }@frcitl81.bitnet, {carre geib}@lifl.lifl.fr

Abstract: We examine several problems related to the preservation of the Independence Principle in Multiple Inheritance. This principle states that all the characteristics of independent superclasses must be inherited by subclasses, even if there are name conflicts. In this context, a conventional approach is to use explicit class selection. We show that this mechanism suffers from serious limitations, and leads to inhibition of refinement and genericity. Our experimental object-oriented language ROME introduces the "Point of View" notion (using an "as-expressions" mechanism) which solves these problems.

1. Introduction

Multiple Inheritance is a powerful paradigm in Object-Oriented Programming. It is of great interest in Artificial Intelligence for its classification power [Bra83] [Tou86] [Weg86]. Multiple Inheritance is also a very promising programming tool with good properties for modularity, reusability and incremental design [Cox86] [Mey88]. Nevertheless it is also a complex notion. Defining and using or reusing non trivial hierarchies of classes need precise principles and powerful new mechanisms. Our language ROME [Car88] [Car89ab] introduces the "Point of View" notion to deal, in a uniform way, with conflicts, refinement and genericity.

First we study the linear and graph-oriented multiple inheritance strategies in the scope of the Independence Principle which states that all the characteristics of independent superclasses must be inherited by subclasses, even if there are name conflicts.

We show that the "Explicit Class Selection Approach" (using compound selectors as "extended" Smalltalk does [Bor82]) suffers from some limitations which must be solved by redefinition or renaming techniques at the subclass design stage.

Moreover we show that this approach leads to two fundamental problems: The "Refinement Inhibition" and the "Genericity Inhibition".

For this, we introduce our experimental object-oriented language ROME, and its central "Point of View" notion. The ROME "Point of View" notion preserves the Independence Principle without the need for a renaming technique. It also preserves the Refinement Principle and the Generic Method aspect of inheritance. So it solves the problems exhibited in the first parts of this paper. This new notion is formally defined and is expressed in ROME through as-expressions.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1990 ACM 089791-411-2/90/0010-0312...$1.50

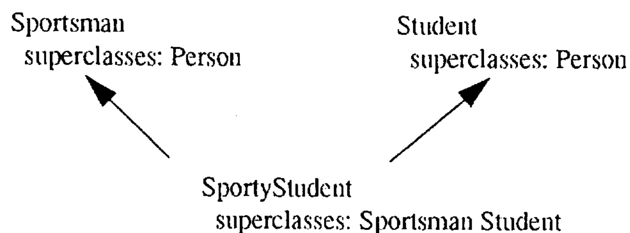
2. The Independence Principle

Consider the design in a Smalltalk-like language of the following two classes independently (i.e. by separate designers).

```
Sportsman:
 superclasses: Person
 instance variables: sportsmanNumber
 methods:
    cardNumber: x
        self validateCard: x.
        sportsmanNumber <- x
    validateCard: x
        "is x a valid Sportsman card number? "
    cardNumber
         ^ sportsmanNumber

Student:
 superclasses: Person
 instance variables: studentNumber
 methods:
    cardNumber: x
        self validateCard: x.
        studentNumber <- x
    validateCard: x
        "is x a valid student card number? "
    cardNumber
         ^ studentNumber
```

Now consider the incremental design of the class SportyStudent as a subclass of both the Sportsman class and the Student class. Such a class must inherit each of the characteristics defined by these two superclasses.



Such superclasses are called independent classes because they are not comparable relatively to the subclass/superclass partial order [Duc87]. So the Independence Principle can be stated as follows:

Independence Principle: There must be no conflict of characteristics inherited from independent classes (even if these characteristics have the same name).

There are two kinds of strategy for dealing with the conflicts problem of multiple inheritance (which are overviewed in [Sny87]): linear and graph-oriented strategies. We examine these two strategies in the scope of the Independence Principle

2.1 Linear Strategies

These strategies are used by languages like Flavors [Moon86], CommonLoops [Bob86],Yafool [Duc86]... Their common principle is: "*flatten the inheritance graph to a linear chain, without duplicates, and then treat the results as single inheritance*" [Sny87]. These strategies transform the partial ordering of classes into a total one by using the relative ordering of classes within the list of the direct superclasses of each class. As examples, the well-known depth-first and breadth-first technique are linear strategies.

The main problem with linear strategies is that they do not deal with the Independence Principle. Because the inheritance graph is transformed in a linear chain, there is always **a single characteristic** inherited by a subclass, even if several conflicting characteristics must be inherited, as is the case if the superclasses are independent (see the SportyStudent example). So in such strategies, you must name characteristics differently if you **potentially** want to inherit all of them in the design of further subclasses. We will refer to this approach as the "Unambiguous Naming Requirement".

This constraint is unacceptable according to the properties of Object-Oriented Programming for modularity (ability of designing a class independently of other classes which are not related to it, that is independent classes), reusability (ability of designing a class as a subclass of several other ones, with possible access to all characteristics defined in these classes, even if there are name conflicts), and incremental design (the design of a class must not affect the definition of its superclasses. It must neither reclaim modifications of these superclasses, nor presuppose the design of subclasses in the future). Linear multiple inheritance techniques do not guarantee these properties because of the Unambiguous Naming Requirement.

- **Reusability**: in the case of conflicts, you cannot simply reuse all the inherited characteristics because you have access to only one of the conflicting characteristics.
- **Incremental Design**: you must presuppose that a class will be able to be combined with other ones in the design of a common subclass, and preview the ambiguous naming problem at the design stage of the superclasses. Inversely, suppose that you design a class as a subclass of several other ones, you must solve the conflicts by renaming conflicting characteristics, that is modify the superclasses definition.
- **Modularity**: you cannot design a class independently of an unrelated one because of the non respect of the Independence Principle. More generally, you cannot design a class independently of potential other classes. You must have a global view on all the existing and future classes. It is easy to see how these problems are still more crucial when the independent classes are designed by distinct programmers.

2.2 Graph-Oriented Strategies

Such strategies deal directly with the inheritance graph, without transforming it. For example, Trellis/owl [Sch86] and "extended" Smalltalk [Bor82] use graph-oriented strategies. You have potentially access to every inherited characteristic. When a conflict arises, you must specify the superclass from which you want to inherit the characteristic. Note that this superclass is not necessarily the one which defines the characteristic but can be a superclass which inherits it without ambiguity.[1]

For example, "extended" Smalltalk extends the conventional message passing to the compound selector technique. A compound selector is a selector prefixed by a class name:

compound selector ::= className.selector

For the evaluation of a compound selector message, the associated method search (lookup) does not begin at the instantiation class of the receiver (as for simple selector lookup) but at the superclass named className specified in the compound selector.

Such a solution apparently solves the problems related to the "Unambiguous Naming Requirement" because it offers a mechanism to name conflicting characteristics

1. This is different from low-level tools like DoMethod, TryMethod or ApplyMethod of Loops [Bob81].

unambiguously. Each conflicting characteristic will be prefixed by the unambiguous name of its definition class. Assuming John, an instance of the class SportyStudent in the previous example. You can have access to the conflicting selector cardNumber respectively through the compound selector messages:

John Sportsman.cardNumber
John Student.cardNumber

Such messages can be used externally to communicate with the object or internally within the body of other methods as in the following method added to the SportyStudent class:

```
printAllCardNumbers
 self Sportsman.cardNumber print.
 self Student.cardNumber print
```

At this stage this technique appears to be better than the linear approach:
- Reusability: all the characteristics are potentially accessible.
- Modularity and Incremental Design: you do not have to care with the "Unambiguous Naming Requirement" yourself, because the technique offers you an unambiguous naming tool. As far as the incrementality is concerned, the classes do not need to be modified when designing a common subclass, because the conflicts, if any, will be solved **locally** at the subclass definition stage.

> Using the compound selector technique (we will call that the "Explicit Class Selection Approach") seems better than linear multiple inheritance for problem related to unambiguous naming. But this technique causes some other problems related to the respect of the Independence Principle. These problems are discussed in the following section.

3. Problems with the "Explicit Class Selection Approach"

The "Explicit Class Selection Approach" permits explicit selection of a class within conflicting superclasses. But this class selection is only used to search the method and then forgotten to solve the other conflicts which can arise in the application of this method. Characteristics which are referred to within the method are conventionally looked-up without taking into account the previous class selection, leading to potential other conflicts between independent classes.
In the previous example, assume that a valid Sportsman cardNumber begins with the two letters 'SP' and a valid Student cardNumber with the two letters 'ST'. Then what will happen with the following message (apparently valid)?:

John Student.cardNumber: 'ST515'

The lookup for the method named cardNumber:, beginning at the Student class level, does find the good one. This method sends the self message:

self validateCard: 'ST515'

which is not a compound selector one, so the conventional lookup finds some method "by default". In this case "extended" Smalltalk applies a depth-first search algorithm, which retrieves the method in the independent class Sportsman, so the result!:

error "wrong cardNumber"

The compound selector technique leads to an unexpected error for the initial message[1].

The simple solution of the "Explicit Class Selection Approach" is too superficial: the class selection constraint is not preserved to solve other potential and consequent conflicts due to intricate method applications.

There are two solutions for solving the exhibited problem. The first one is to solve or preview the potential conflicts in the superclasses, using the compound selector messages:

```
Sportsman:
 ......
   cardNumber: x
      self Sportsman.validateCard: x.
      sportsmanNumber <- x
 ......

Student:
 ......
   cardNumber: x
      self Student.validateCard: x.
      studentNumber <- x
 ......
```

This solution is quite similar to the one used in the case of linear strategies, leading to the same unacceptable constraints (see 2.1).
The second solution comes from the basic idea that "since the conflict appears at the subclass design stage, solve it **locally** at this stage". It consists in redefining in the subclass the methods which may provoke conflicting references.
Consider our example of the SportyStudent class, the

1. Note that the unexpected error does not happen for the message John Sportsman.cardNumber: 'SP222'. In this case the "by default' search finds the good method. This exhibits a dissymmetry similar to the linear strategies' one.

cardNumber: method is inherited twice and so must be invoked on SportyStudent instances using compound selectors. But the cardNumber: method refers to the validateCard: method which is also inherited twice. So the cardNumber: method may provoke unexpected errors and must be redefined in the SportyStudent subclass. The new cardNumber: method will replace the same-named method of the Student class (remember that the cardNumber: method of the Sportsman class may be normally inherited because of the "default" search).

```
SportyStudent:
  ......
    cardNumber: x
      "set studentNumber"
      "redefined, not inherited"
      self Student.validateCard: x.
      studentNumber <- x
  ......
```

Even if this solution preserves the incremental design property, programmers have to reimplement manually most of the methods in the subclass, which is quite cumbersome.

Another approach is the renaming mechanism (such as in Eiffel [Mey88]). Here you only have to "rename" all the conflicting characteristics ("features" in Eiffel) through "renaming clauses" which are implicitly taken into account to bind references within a method ("routine") call.
Such a solution is more flexible for the user because he only has to rename conflicting features, not to reimplement them. That this seems as if redefinitions were implicit and automatically done by the compiler.

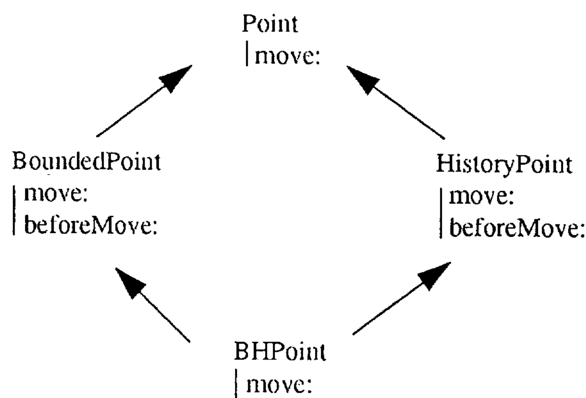
We will see that the "point of view" notion of our language ROME sets the programmer free from the renaming requirement, too. We will study first another fundamental problem due to the respect of the Independence Principle using the Explicit Class Selection Approach. We call it the "Refinement Inhibition Problem"

4. The Refinement Inhibition Problem

Let us state the Refinement Principle as:

Refinement Principle: The redefinition of a selector s in a subclass (that is the association of a new method definition to s) imposes that it must be this refined definition which will be bound to s, in any message applying s to objects (notably self) members of this subclass.

The Refinement Inhibition Problem refers to the fact that the previous principle can be easily bypassed, using the compound selector technique, since you can prefix the selector by any superclass, of any depth. The associated method definition will be searched starting from the specified class, despite any redefinition under this class[1].
We will not insist on the interest of the Refinement Principle, which will be too long here (see for example [Car89] for a discussion on this point). But consider one of the most important requirement for this Principle, which is the coherence guaranty.
Let us take the example of the four classes Point, BoundedPoint, HistoryPoint and BHPoint inspired from [Sny87]. A Point stands for a location object. A BoundedPoint is a Point which is bounded by min-max values for the location. A HistoryPoint is a Point which keeps a list of previous locations. And a BHPoint is a BoundedPoint and a HistoryPoint using multiple inheritance. The Point class implements a move: method. The BoundedPoint class and the HistoryPoint class redefine the move: method to use a beforeMove: method according to their particular tasks. The BHPoint class also redefines the move: method to use the two beforeMove: methods of its superclasses.



Now assuming P1 an instance of BHPoint (with max value set to 100), you can simply send the following messages to bypass the Refinement Principle and the subsequent constraints set by the subclasses:

P1 HistoryPoint.move:1000.
P1 BoundedPoint.move:1000.
P1 Point.move: 1000.

The first one violates the bounds of P1, the second one troubles the history list of P1 and the third one violates the bounds and troubles the history list of P1! So the compound selector technique does not guarantee the coherence requirements.
As a conclusion, this technique designed for reusability

1. more precisely along the path starting from the instantiation class of the receiver to the specified class.

according to the Independence Principle, leads to "over-reusability" according to the Refinement Principle. Redefined characteristics, which should not be accessible, are unfortunately visible as a side effect of a technique designed for another purpose.

> We will see that the "Point of View" notion of our language ROME guarantees the Independence Principle, with the preservation of the unrelated Refinement Principle. Before introducing this notion, let us consider another problem, the Generic Inhibition Problem, which is also solved by it.

5. The Genericity Inhibition Problem

Inheritance provides an implicit way for the definition of generic methods. Let us briefly expose the Generic Method Aspect of Inheritance (genericity versus inheritance is clearly studied in [Mey88]):

> *Generic Method Aspect of Inheritance: A method m is said "generically parametrized" by another method m' if m refers m' in a self message.*

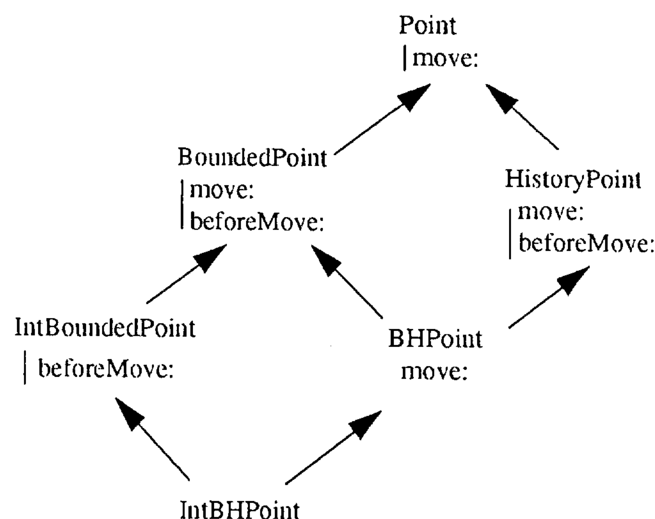
Consider the two methods m and m' in a class C, and a redefinition of m' in a subclass C' of C. Because of inheritance[1] the execution of the method m by an instance of C' implicitly refers to the redefined method m' of this class. This, exactly as if the redefinition of the method m' automatically implies the redefinition of the calling method m in the class C', i.e as if the method m were generically parametrized by the method m'[2].

Let us come back to the Point example and consider a new subclass of BoundedPoint, named IntBoundedPoint which constrains the point to have integer coordinates. This new class redefines the beforeMove: method to dynamically control the type of its parameter. At this stage a question remains: does the move: method need to be explicitly redefined in IntBoundedPoint?

With the Generic Method Aspect of Inheritance, redefinition of the move: method is not necessary because of the generic definition of the move: method. Implicitly, the move: method is "generically parametrized" by the beforeMove: method.

Unfortunately this implicit genericity is lost in the context of multiple inheritance, more precisely when using the Explicit Class Selection Approach to preserve the Independence Principle. The explicit reference to a method using the compound selector mechanism inhibits the dynamic binding of the method, and thus genericity.The fact that the method is looked-up starting at the specified class, disallows any further redefinition of the method under this class. This is the Genericity Inhibition Problem. As Bobrow and al. say: "This explicitness can cause problems because methods built in as constant information about the class hierarchy, which may change" [Bob86], we think that one of these problems is the present one.

To illustrate this problem, consider the next example as a natural extension of the initial hierarchy below the Point class. We want to refine the BHpoint class in a new IntBHPoint class, in order to inherit the new constraint on location coordinates. So we create the subclass IntBHPoint of the two classes IntBoundedPoint and BHPoint.



The fundamental remark is that the class IntBHPoint need not to reimplement the move: method because i) this method is correctly inherited from BHPoint and ii) this method is "generically parametrized" by the two beforeMove: methods, which are inherited from the two superclasses. We think that this is the good and natural design of this new class in respect of reusability and incrementality properties. Unfortunately this is not a solution when using the Explicit Class Selection Approach. Within this approach, the conflict of the two beforeMove: methods, inherited in BHPoint and used in the move: method of this class, has been solved with compound selectors. The move: method of BHPoint would appear like this:

```
BHPoint:
  ......
  Move: x
    self BoundedPoint.beforeMove: x.
    self HistoryPoint.beforeMove: x.
    .....
  ......
```

But these explicit class selections in BHPoint inhibit any redefinition of the beforeMove: methods for members of subclasses, such as IntBHpoint. So the move: method of IntBHPoint (inherited from BHPoint) does not take into

---

1. more precisely, because of the late binding mechanism.
2. as it would be the case in ADA for example.

account the redefinition of one of its generic parameters. Thus we lost the genericity as a side effect of the Explicit Class Selection Approach. In this example, the solution[1] is the redefinition of the move: method in the IntBHPoint class, like this:

```
IntBHPoint:
  ......
  Move: x
    self IntBoundedPoint.beforeMove: x.
    self HistoryPoint.beforeMove: x.
    .....
  ......
```

Of course the same problem would arise for any subclassification of the BoundedPoint and on the other side, HistoryPoint (consider for example, a ShortHistoryPoint subclass which binds the cardinality of the history list). Such a subclassification leads to subclasses potentially combinable with the BHPoint, in order to produce common subclasses. In each of these subclasses you might redefine every generic method which contains self compound selector messages.

In the previous section, we dealt with the Refinement Inhibition Problem by bypassing the most refined definition of a characteristic. Here the problem is addressed inversely, that is potential refinements of a method are not taken into account. We need a technique which allows the selection of conflicting superclasses to ensure the Independence Principle with the preservation of refinement and thus, genericity. We will see now how the Point of View notion of ROME addresses all these problems in a uniform manner.

6. The Point of View notion of ROME

ROME is a language whose purpose is the experimentation of new notions for Object-Oriented Design. It is defined by a lisp-based metacircular kernel, inspired from ObjVlisp [Coi86]. We will not present all the original features experimented in ROME, this paper focuses on one of its essential topics studied within our project, that is its multiple inheritance strategy based on the "Point of View" notion.

Let us briefly expose the structure of an object in ROME. A ROME object has a three-parts characterization: attributes, methods and selectors.

- Attributes and methods define its internal and encapsulated characterization. Methods are lisp-lambda expressions which can be applied only by the object through the application primitive "applymeth" or syntactically:

  {methodName arguments}

- Selectors define its external characterization, that is its interface for message sending. Each selector is associated to a method though an association clause:

  (selectorName methodName)

Message sending is the only way to communicate with an object through the primitive send, or syntactically:

[object selectorName arguments]

The object reaction is to apply ("to applymeth") its local method associated to selectorName, if both exist. The very distinction selector/method is a proper feature of ROME which amplifies the encapsulation properties.
Objects are described by classes, which are arranged within a multiple inheritance lattice. Classically OBJECT is the top of this lattice and defines the common characteristics of all the objects, notably local methods for reading (?) and writing (<-) their attributes:

{ ? attributeName }
{ <- attributeName value}

6.1 Point Of View definition

Rome associates a Point of View notion to the class notion as follows.
First we define:
- The *Dependent Classes* of a class C, Dep(C), as the set of all its related classes, that is the union of all its superclasses and its subclasses, including itself.
- The *Representation Classes* of an object O, Rep(O), as the set of all the superclasses of its instantiation class, including this class.
Then we define *the Point of View of a class C on an object O*, PoV(C,O), as the restriction of the whole inheritance lattice to the set of classes: Dep(C) $\cap$ Rep(O). See an illustration of the Point of View of BoundedPoint on the BHPoint object P1, i.e PoV(BoundedPoint, P1), on next page.

6.2 Using the Points of View: As-Expressions

The Point of View notion was introduced in ROME to respect the Independence Principle with the preservation of the refinement one, thus the implicit genericity.
Our multiple inheritance strategy is fundamentally graph-oriented. The "Explicit Class Selection Approach" through compound selectors is extended to the "Point of View Selection Approach" through characteristics "as-expressions". These expressions allow to refer to a

1. The renaming mechanism of Eiffel leads to another closed solution needing the redefinition of the beforeMove: method of IntBoundedPoint in IntBHPoint.

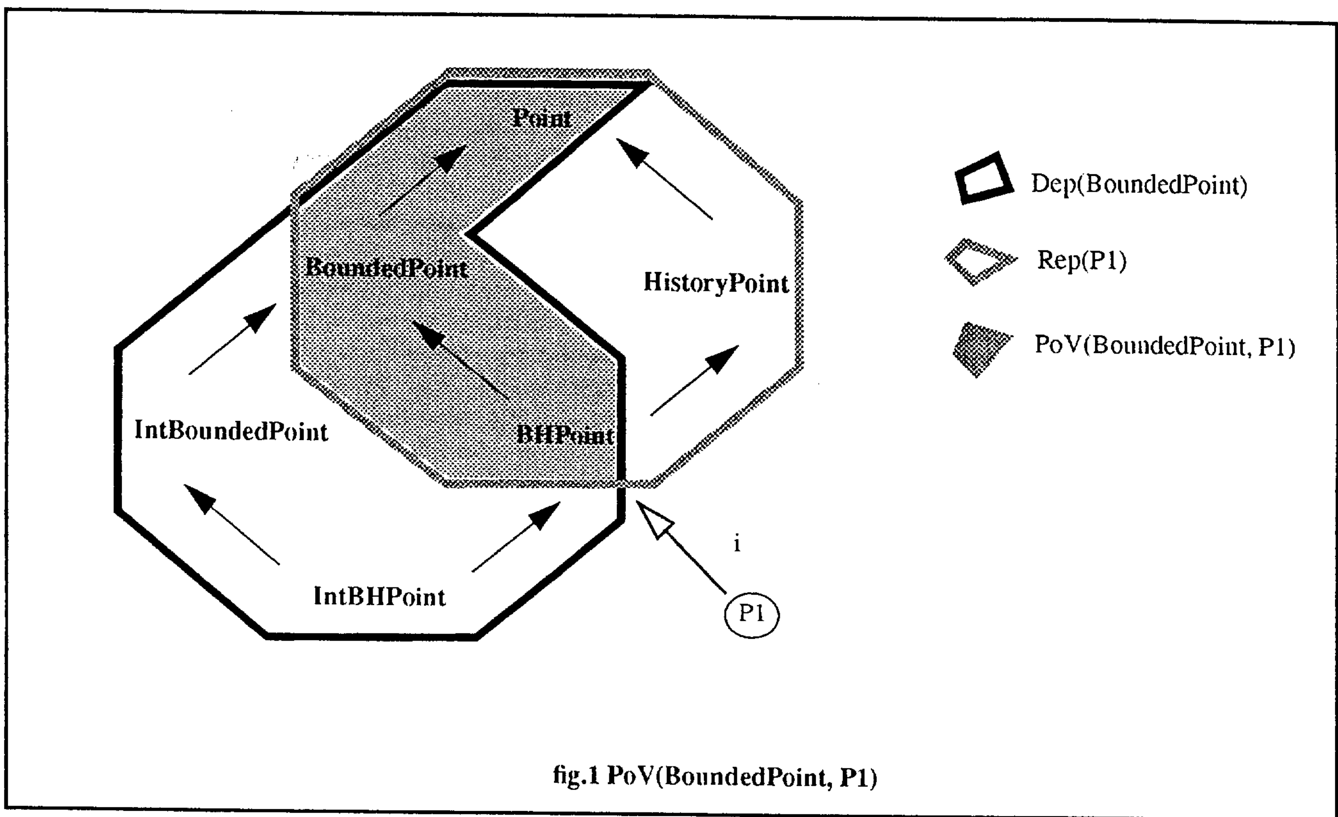


fig.1 PoV(BoundedPoint, P1)

characteristic (attribute, method or selector) as defined from a point of view.

The general form of an as-expression is:

(aCharacteristic as aClass)

The intuitive semantics of such an expression is:

Select the associated definition of aCharacteristic within the point of view of aClass on the current object.

Depending of the type of the characteristic, as-expressions appear in messages for selectors, method calls, association clauses of a method to a selector, and attribute references through the primitive methods for reading and writing.

These as-expressions are systematically used explicitly or implicitly. **Implicit Points of View** are automatically set by ROME when no explicit point of view is specified, according to the following rules:

- The implicit point of view in a message is that of the instantiation class of the receiver, that is the whole inheritance lattice of it.
- The implicit point of view associated to a method call, a reference to an attribute or a reference to a method in a selector association clause within a class is that of this class.

Consider for example the SportyStudent example implemented in ROME:

```
Sportsman
 superclasses (Person)
 attributes (cardNumber)
  methods
   (cardNumb (lambda (x)
        { validateCard x }
        {<- 'cardNumber x} )
    validateCard (lambda (x)
        ; is x a valid Sportsman
        ; card number?
      ))
 selectors
  (cardNumber: cardNumb)

Student
 superclasses (Person)
 attributes (cardNumber)
  methods
   (cardNumb (lambda (x)
        { validateCard x }
        {<- 'cardNumber x} )
    validateCard (lambda (x)
```

```
      ; is x a valid Student
      ; card number?
    ))
  selectors
   (cardNumber: cardNumb)

 SportyStudent
  superclasses (Sportsman Student)
```

Let us insist on the fact that the characteristics defined by each of the independent classes are exactly same named, so highly conflicting in the common subclass. As, in ROME, there is no attributes merging when independently inherited, the attributes named cardNumber are not merged[1].
Within the class Sportsman (resp. Student) the implicit point of view associated to each characteristic reference is that of this class, so that the body of cardNumb is implicitly interpreted as follows:

```
cardNumb (lambda (x)
  { (validateCard as Sportsman) x }
  {<- '(cardNumber as Sportsman) x} )
```

Similarly the definition clause of the selector cardNumber: is implicitly

```
(cardNumber: (cardNumb as Sportsman)).
```

Instead of adopting a completely theoretical presentation of the point of view notion, we will come back to the problems presented in the previous sections, and show how the present technique addresses them.

6.3 The Independence Principle

As-expressions of selectors correspond intuitively to an extension of external compound selectors. Assuming John is an instance of SportyStudent, the following messages allow the user to select one of the conflicting selectors named cardNumber:

```
[John '(cardNumber: as Student) "ST222"]
[John '(cardNumber: as Sportsman) "SP515"]
```

The first message, for example, specifies the point of view of Student which is:
SportyStudent -> Student -> Person -> Object
where the definition of the selector is:

```
(cardNumber: (cardNumb as Student))
```

1. Note that it was not the case in the Smalltalk-like program of section 2.

Consequently, this leads to the method application:

```
{(cardNumb as Student) "ST222"}
```

which calls the method:

```
{(validateCard as Student) "ST222"}
```

and affects the attribute:

```
{<- '(cardNumber as Student) "ST222"}
```

which is the desired effect.

We can see how the implicit points of view mechanism does allow local references within independent classes without borrowing potential conflicts. This respects the Independence Principle and solves the first problem exhibited in section 3.

Of course, points of view are used on conflicting inherited methods. For example the allCardNumbs method of SportyStudent would be defined like this:

```
allCardNumbs (lambda ()
  (print "Student:" {(cardNumb as Student)})
  (print "Sportsman:"
       {(cardNumb as Sportsman)}))
```

At this stage, points of view seem to be equivalent to explicit class selections, we will see now that the difference states in the refinement and genericity preservation properties.

6.4 Refinement Preservation

Points of view disallow to bypass the Refinement Principle. Their sole purpose is the Independence Principle guaranty. Potential refinements of a characteristic are respected because:

> The definition of a characteristic is not searched up from the specified class (as with the Explicit Class Selection Approach) but within the whole point of view of this class which includes its subclasses where the refinement may appear.

Consider the Point example as in the section 4 and one of the problematic messages:

```
[P1 '(move: as BoundedPoint) 1000]
```

Since P1 is precisely a BHPoint, which class defines a proper move: method, this one must be applied, and it is indeed the result. The point of view of BoundedPoint on P is (fig.1):
BHPoint -> BoundedPoint -> Point -> Object

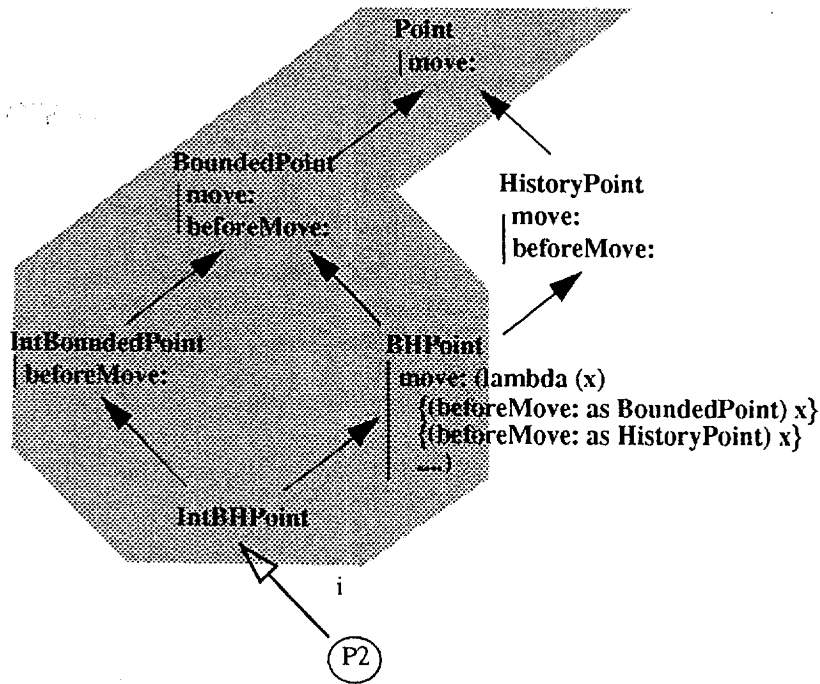


fig.2 PoV(BoundedPoint, P2)

and the lookup finds the good method.

As a property, an object is always considered according to its most refined characterization from some point of view. This can be interpreted as a natural extension of the same property in the context of simple inheritance. Points of view are only used to select one of the most refined independent characterizations, thus preserving the Refinement Principle.

6.5 Genericity Preservation

Remember that calling conflicting methods through compound selectors creates fixed references, so that any further refinement will not be taken into account, loosing the implicit genericity offered by inheritance. To the contrary, and as a consequence of the previous property of the point of view notion, genericity is preserved in the context of multiple inheritance.

Let us consider again the Point example as extended in the section 5, replacing the self compound selector messages by method as-expressions so that the move: method is now defined by BHPoint as shown in figure 2.

The Point of View notion ensures that the move: method is generically parameterized by the two beforeMove: methods "as BoundedPoint" and "as HistoryPoint". Consider the redefinition of the beforeMove: method "as BoundedPoint" in the subclass IntBoundedPoint and P2 an instance of IntBHPoint. The application of the move: method by P2 provokes the evaluation of the first as-expression which specifies the point of view of BoundedPoint.This point of view is pictured on the figure 2. The most refined beforeMove: method within this point of view is that of IntBoundedPoint, so the desired result.

Such a technique makes the programmer free of any added work, notably no redefinition, nor renaming task at the subclass design stage, as required by the conventional graph-oriented strategies. We claim that this flexibility is an essential property moreover when the potential refinements are numerous.

7. Conclusion

We have presented how the point of view notion of the ROME language guarantees modularity, incrementality, reusability and method genericity, respecting both the Independence and Refinement principles for multiple inheritance. This offers more flexibility to this promising feature, so encouraging its use. We could not expand all the details of this technique, hoping that this short introduction was intuitive enough to foresee its principle and appreciate its interest. More details can be found in [Car89], notably on the semantics of the point of view mechanism which is deeply coupled with the method and message evaluation one, a

problem which was not discussed here. Some other interesting features were not pointed on. The technique of the "super-as method" is an extension of the super-messages as offered by Smalltalk, with respect to the point of view properties. The "Multiple and Evolutive Representation" feature offers even more flexibility in designing and using classes lattices[Car90]. Through this technique, an object can be a direct member of several classes and this, dynamically. Coupled with the point of view notion, this simplifies the multiple subclassification task. These are steps towards advanced multiple classification of objects, which is one of the central objectives of the ROME project.

Bibliography

[Bob81] D.G. Bobrow and M.S. Stefik, *The LOOPS Manual*, Memo KB-VLSI-81-13, Xerox PARC, 1981.

[Bob86] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik and F. Zdybel, "CommonLoops: Merging Lisp and Object-Oriented Programming," *Proc. of the OOPSLA ACM Conference*, Portland, 1986.

[Bor82] A.H Borning and D.H Ingalls, "Multiple Inheritance in Smalltalk 80," *Proc. of the AAAI'82 conference*, Pittsburgh, 1982.

[Bra83] R.J. Brachman, "What IS-A is and Isn't: An Analysis of Taxonomic Links in Semantic Networks," *IEEE Computer*, vol 16, no. 10, 1983.

[Car88] B. Carré and G. Comyn, "On Multiple Classification, Points of View and Object Evolution," *Artificial Intelligence and Cognitive Sciences*, Manchester University Press, J. Demongeot, T. Herve, V. Rialle and C. Roche Eds., 1988.

[Car89a] B. Carré, *Méthodologie orientée objet pour la représentation des connaissances, concepts de point de vue, de représentation multiple et évolutive d'objet*, PhD thesis, University of Lille, France, 1989.

[Car89b] B. Carré and G. Comyn, "ROME, Multiple and Evolutive Object Representation," *Proc. of the Fifth AIICSR conference*, Czechoslovakia, North-Holland, 1989.

[Car90] B.Carré, L. Dekker and J.M. Geib, " Multiple and Evolutive Representation in the ROME language," *Proc. of the Second International Conference TOOLS*, J. Bezivin, B. Meyer, J.M. Nerson Eds., Paris, 1990.

[Coi86] P. Cointe, "The ObjVlisp Kernel: a Reflexive Lisp Architecture to Define a Uniform Object-Oriented System," *Proc. of the Workshop on Meta-Level Architecture and Reflection*, Alghero, 1986.

[Cox86] B.J. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading (Mass.), 1986.

[Duc86] R. Ducourneau and J. Quinqueton, *YAFOOL, encore un langage objet à base de frames*, Technical Report no. 72, INRIA, 1986.

[Duc87] R. Ducourneau and M. Habib, "On some Algorithms for Multiple Inheritance in Object Oriented Programming," *Proc. of the ECOOP conference*, Paris, 1987.

[Mey88] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.

[Moon86] D.A. Moon, "Object-Oriented Programming with Flavors," *Proc. of the OOPSLA ACM Conference*, Portland, 1986.

[Sny87] A. Snyder, "Inheritance and the Development of Encapsulated Software Components," *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner Eds., MIT Press, 1987.

[Tou86] D.S. Touretzky, *The Mathematics of Inheritance Systems*, Pitman, London, 1986.

[Weg86] P. Wegner "Classification in Object Oriented Systems". *ACM SIGPLAN Notice*, vol 21, no. 10, 1986.