Representation of Complex Objects : Multiple Facets with Part-Whole Hierarchies

Francis Wolinski & Jean-François Perrot
LAFORIA, Institut Blaise Pascal,
Université Paris VI & CNRS, Paris
Boîte 169, 4 Place Jussieu, 75252 Paris Cedex 05, France

1- INTRODUCTION

1.1- General

We study a problem in object-oriented knowledge representation. The traditional class/instance and inheritance paradigms form a sound basis for a computer simulation of real-world objects "as they are described", thereby giving a tool for expressing knowledge about the world in a specific way which is neither procedural nor declarative, and which we propose to call "object-oriented". In our opinion, knowledge representation differs from programming in that it has to be practised by experts of the domain and not by professional programmers. Hence, the programming tools it uses must be designed to fit the intellectual processes of the domain expert rather than to suit the needs of the implementer. In this respect, we consider that the class/instance mechanism is a very satisfactory machine realization of the "general concept"/"specific instance" way of thinking, whereas inheritance (simple or multiple) is far less acceptable as a classification scheme. As a consequence, we shall concentrate on improving the instanciation process and use inheritance in a standard way purely as a programming tool.

Anyhow, this well-known, well-implemented and well-understood paradigm must be extended in at least two ways in order to become a really usable tool for representing substantial amounts of knowledge about complex objects. Namely, it has to deal with two dimensions of structural complexity. First, objects usually must be considered from various points of view. Second, many objects are thought of as being composed of various parts that are themselves considered as sub-objects (and not as attributes). These two dimensions have been repeatedly explored in the past (Points of view : Goldstein & Bobrow in PIE [9], Bobrow & Stefik in LOOPS [2] and lately Carré in ROME [5] [6]. Part-whole : the LOOPS and Thinglab [3] systems, lately Blake & Cook [1]). But nowhere have they been treated together: for instance the LOOPS primitives dealing with points of view (classes `Node` and `Perspective`) are not easily combined with metaclass `Template` catering for part-whole hierarchies.

1.2- Aim of paper

In this paper, we propose to bring these two dimensions together in an analysis and an implementation based on a restricted application domain, that of robot representation. We claim that by restricting the field we are able to formulate some proposals about the intellectual processes used by domain expersts and thus to motivate our solutions.

More precisely, the aim of this paper is to study both dimensions together and to propose a set of tools which insures their harmonious cooperation. Our proposal extends the approach of Goldstein & Bobrow in PIE [9] on the multi-facet aspect by integrating the part-whole hierarchy aspect.

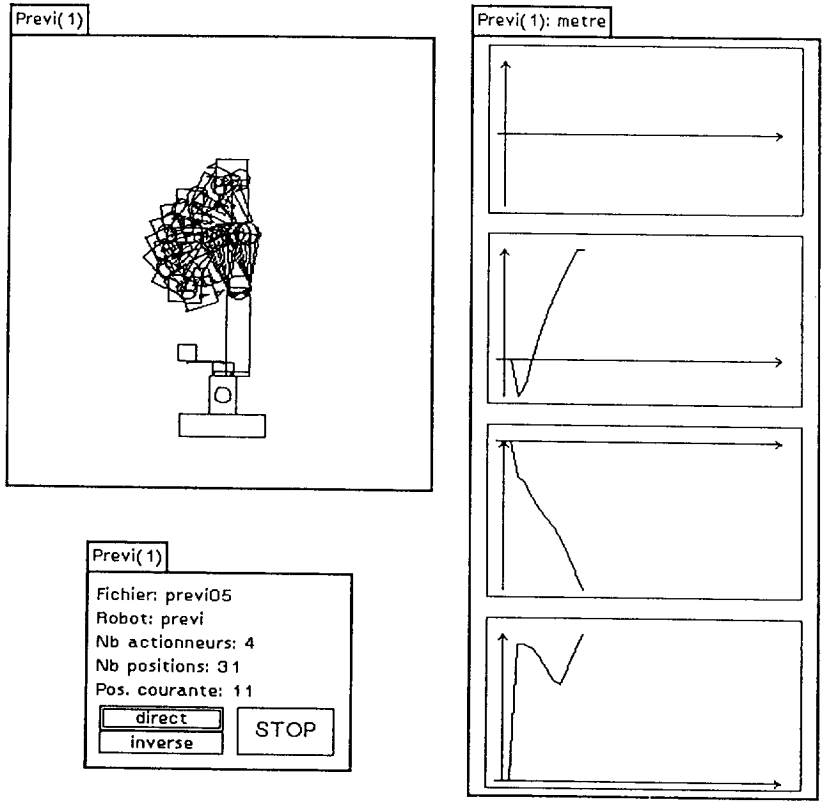


Fig. 1 : A view of manipulator Previ as represented in *Systalk*

2- CONTEXT OF WORK

2.1- Robotics at EDF

The work partially reported here was done in the Robotics Group of Electricité de France (EDF). Its aim was to provide the various specialised teams dealing with robots at EDF (mechanics, control, sensing, CAD, trajectory generation) with a computer system where their various approaches would be housed in a uniform way. The requirements included an interface with CAD tools for reading in robot specifications as well as input of trajectories computed off line and their execution. We chose Smalltalk-80 as a base and developed the *Systalk* system [15].

About 12 different operational domains totalling 40 classes or so were covered, with 50 predefined robotic components. The system was used in two main robotic fields :

- manipulators, with execution of off-line computed trajectories and force and torque calculation (see [14]). All manipulators used or studied by EDF were modelled in *Systalk*.
- mobile robots, with simulation of sensors and qualitative control (see [7]).

2.2- User specification

Our application caters for 4 levels of user competence :

- (a) the *system developer* : he has full knowledge of the architecture of *Systalk* at its various levels of implementation, he is able e.g. to improve on implementation efficiency.
- (b) the *programming specialist* : he is a Smalltalk programmer, but only knows the outward description of *Systalk*, he is therefore unable to modify structural implementation choices. His task is to manage and extend the set of specialised classes that represent the various facets of robots (see under § 3 for more details). Typically, he would have to improve the speed of computation of forces and torques, and to program a new facet "robot compliance under external contraints". He also may have to extend the library of predefined robotic components (such as various sorts of primitive geometric bodies and more sophisticated joints).
- (c) the *robot conceptor* : he knows nothing of Smalltalk, but has a complete knowledge of *Systalk'* s functionalities. His task is to build classes modelling actual robots, using the representation tools we propose via the user interface.
- (d) the *basic user* : he knows neither Smalltalk nor the full structure of *Systalk*, but is well acquainted with *Systalk'* s user interface. His job is to instantiate one or several robot classes and to use the instances for simulation purposes and thereby derive informations of interest for his project.

Systalk was designed mainly to accomodate user (c). We assume that our robot conceptor thinks of the robot he has to represent as a hierarchy of sub-systems ultimately made up of standard robot components to be picked off the shelf. The various facets of the complete robot (visible form to be drawn on screen, articulated motion, constraints that are applied at various points of the structure etc.) are correspondingly built up from those of the sub-systems **in an automatic way**, provided the communication schemes that link the various sub-system facets are specified. Note this simulation of the mounting of a robot results in a kind of multiple reuse of software components, since robotic components are re-used whereas each of them does reuse more elementary software components corresponding to its facets.

2.3- Example : A construction of a "simple" robot : the *Previ* manipulator (figs. 1 and 2)

2.3.1- *Facets*

Robots in our application context have a number of facets, or activities, which will be implemented as independent objects (see infra § 3.1). Here are the most important :

- *Cartesian frame* : The position of the robot in space is given by a three dimensional coordinate system bound to the robot, expressed in the coordinates of the supersystem of the robot. Accordingly, a move of the robot relative to its supersystem will be appear as a transformation of its cartesian frame. Class `cartesianFrame` defines 4 instance variables O, X, Y, Z with values in $\mathbf{R}^3$, as well as methods for algebraic computation of translations, rotations and transpositions.

- *Shape* : The visible shape of the robot, to be drawn on the screen. The actual drawing is obtained by top-down activation of the `Shape` facet objects attached to the subsystems that make up the robot (see § 4). Eventually, each elementary component has its own shape, described by some subclass e.g. `Cylinder`, `Cone`, `Parallelepiped`, which defines the necessary instance variables (e.g. for `Cylinder`, variables `height` and `radius`) as well as the display methods (which of course will make use of the `cartesianFrame` facet).

- *Solid* : Mass and centre of gravity, the last being recomputed at every instant in function of the state of the (articulated) robot.

- *Kinematics* : Expresses the role of each part of the robot as a motion transformer : when a move is applied to it, it executes the move (transforming its `cartesianFrame`) and transmits it to the objects with which it is bound.

- *Force (and Torque)* : Computes the forces from which elastic deformation may be deduced.

- *Control* : Logical organization of motion operators.

- *Measurements* : *idem* for motion, speed and stress sensors.

2.3.2- *Subsystems*

Robots are usually analysed in 3 levels : the robot is composed of a number of articulated bodies, each of which is made up of a few geometric solids and joints, which are all predefined and available in store. Constructing a robot is thus accomplished in two steps, first the building of the individual parts, second the assembly of the parts.

In our example the following standard elements will be used :

- Geometric solids : bars, boxes and fingers
- Joints : static and revolute

Our *Previ* robot is made up of five articulated bodies :

- a *base*, comprising 3 bars, one static joint and one revolute joint.
- one link of kind *link1*, of 4 bars, 2 boxes with one static joint and one revolute joint.
- two links of kind *link2*, of 2 bars, one static joint and one revolute joint.
- a *gripper*, made up of 2 boxes, 2 fingers and one static joint.

According to this analysis, the *Systalk* user of type (c) wishing to represent a *Previ* manipulator involves:

- making sure that classes `Bar`, `Box` and `Finger`, as well as `SJoint` and `RJoint` are predefined in the system ;
- defining 4 new classes `PreviBase`, `PreviLink1`, `PreviLink2` and `PreviGripper` ;
- defining a new class `Previ`.

Then any number of instances of Previ may be created by instantiating class `Previ` (the work of a *Systalk* user of type (d), and used *ad libitum* via the various facets (executing trajectories, displaying motion etc).

The essence of *Systalk* is to give means to express that class `Previ` is obtained by composing classes `PreviBase`, `PreviLink1` etc. in an intelligible way, and that in turn `PreviBase` is composed of classes `Bar`, `SJoint` and `RJoint` in an analogous way etc. To formulate the structural and functional relationships that constitute the definition of these entities, we introduce three categories of discourse :

- a system (robot) has several *facets*, or activities ;
- a system is composed of several *subsystems* ;
- facets of the same kind in different subsystems are linked by *communication schemes*.

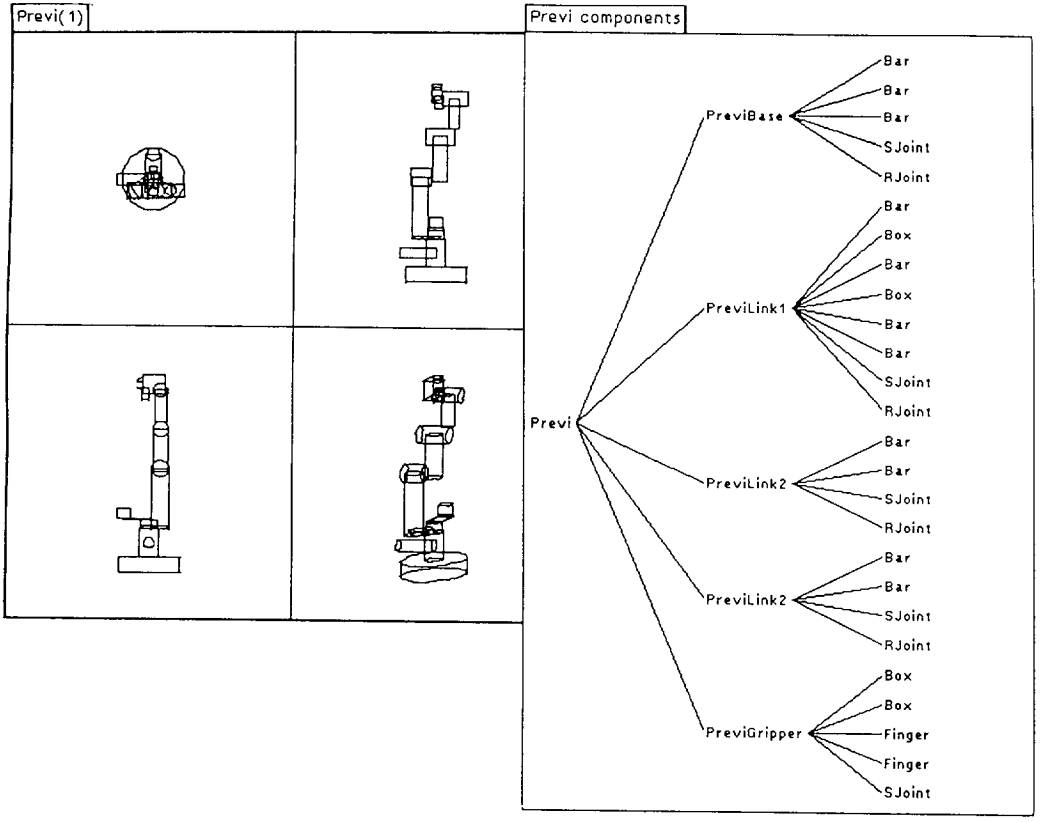
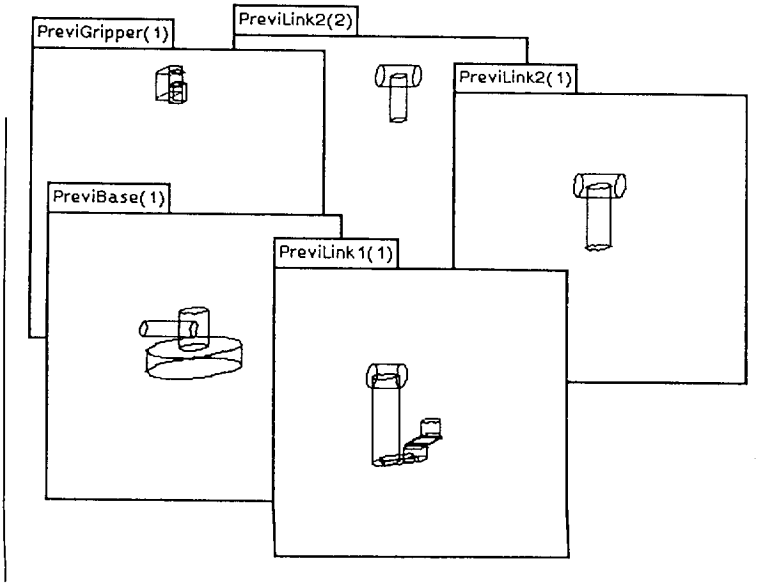


Fig. 2 : Manipulator Previ : its Shape facet (top left), its decomposition tree (top right), and (the Shape facets of) its five components (below)



As we said before, the first two have been considered many times, whereas the third is (to our knowledge) new. It is needed to link facets and hierarchy together. The originality of our work is to present an integrated architecture with all three aspects working smoothly together.

One might choose to translate this analysis directly at the level of the implementation language (supposing it powerful enough). That is, part-whole hierarchy and multi-facets could be rendered using the standard object-oriented techniques of aggregation of attributes and (multiple) inheritance. See B. Carré [5] for a thorough discussion.

Clearly, the programming tools at the disposal of users (c) and (d) must be distinct from the general-purpose Smalltalk primitives accessible only to (a) and (b).

3- MULTI-FACETS

3.1- Explicit delegation

3.1.1- *Our choice*

The traditional choice for implementing multi-faceted objects is between multiple inheritance and explicit delegation. Rather than endowing Smalltalk with a refined multiple inheritance scheme such as the point-of-view approach of Carré [6] , we chose to materialize each facet by an independent object, to which the messages corresponding to its activities are delegated, and which in turn is able to delegate parts of its work to other facet-objects. Apart from its comparative easy implementation in Smalltalk, it has the advantage of allowing dynamic modification of the facets. In this we follow the approach of PIE, whose conclusion we adopt
"In most cases, we have found that the sender knows the point of view that the recipient should employ to understand the message" [9, p. 77]

Our syntax for delegation uses a method `function:`, to send message `msg` to a given facet of a certain `system`, write :

```
(system function: facetName) msg
```

3.1.2- *"Contractual backing" between facets*

Facets possess a backward pointer to the system to which they belong (an instance variable called `system`), thus they have indirect access to the other facets of the same system. For instance, facets of the `Shape` kind have recourse to the facet `cartesianFrame` of the system for display. We call this kind of cooperation between facets *contractual backing*.

Although it was virtually present in PIE, its authors don't seem to have made use of it, since they defined functionally independent facets such as resistor and plane location. Here is a simple example for facet `Solid` which defines the instance variable `mass` :

```
!Solid methodsFor: 'access' !

density
     ^ mass/(system function: #Shape) volume !!
```

Communication between facets using contractual backing accurately reflects the laws of physics and in the same time allows a high degree of code sharing.

3.2- Prototypes

3.2.1- *Problem statement*

The problem is then to write only one class hierarchy per facet (e.g. for the "visible shape" facet `Shape`, classes `Cylinder`, `Parallelepiped`, `Cone`, `Sphere` etc.), which will be "used" by all the robot classes that possess the facet. This raises a well-known problem which we summarize as follows.

Creating a multi-faceted object is a rather heavy process, since it demands the creation of all facet-objects, hence the creation method must provide all their relevant instance variable values for due initialization. This is admissible when the object is created for the first time, much less so when it is to be reused. Hence, one would wish to adopt another approach and be able to duplicate complex object instead of buiding them *ex nihilo*. However, physical copying (`deepCopy`) carries with it more information than is actually needed (irrelevant details about the actual state of the object). So we are led to defining equivalence relations between complex objects that are intermediate between complete identity (equal values of attributes) and belonging to the same class. For instance, the general notion of a cylinder (class `Cylinder`, with attributes `height` and `radius`), may be refined as cylinder-with-radius-10, cylinder-with-height-50, cylinder-with-radius-10-and-height-50, this last notion being clearly distinct from any given instance of a cylinder having those values in its instance variables. Whereas the general notion is adequately represented by class `Cylinder`, the other three don't seem to require independent classes to represent them, since they do share most of the information they carry with the said class. Therefore a new implementation concept is needed, different from `class` as well as from `instance`. In PIE, this concept was called *contextualization* and implemented as a pair `(class, dictionary)`.

3.2.2- *Multi-parameter classes*

We follow here the same line with a different terminology : PIE contextualizations are called here **prototypes**, and seen as named contexts, i.e. pairs (`name`, `dictionary`). To use them we introduce **multi-parameter classes**, the metaclasses of which define an instance variable (for this technique, see Cointe [4, 8]) called `prototypes`, pointing toward a dictionary of contexts (dictionaries). In these classes the instantiation method `new:` takes as an argument a `contextName` and yields as a result an instance of the class initialized according to the context which is the value of `contextName` in the dictionary `prototypes`. They are defined as subclasses of the abstract class `MultiParaObject`, and all facet classes of *Systalk* belong to this hierarchy.

```
Object subclass: MultiParaObject
     instanceVariableNames: ''
     classVariableNames: '' !

!MultiParaObject methodsFor: 'initialization'!

init:  aPrototype
     aPrototype associationsDo:
          [:a | self perform: (a key, ':') asSymbol
                with: a value]!!

MultiParaObject class
     instanceVariableNames: 'prototypes'!

!MultiParaObject class methodsFor: 'creation'!

new:  contextName
     ^super new init: (prototypes at: contextName) !!
```

For instance, class `Cylinder` will inherit from `MultiParaObject`, and the value of its variable `prototypes` is the dictionary containing all the cylinder prototypes corresponding to all the various robot parts of cylindrical shape that are currently available in the system. The names to be used as keys for this dictionary are generated automatically as the path (see section 4.2.2) referencing the part in the part-whole hierarchy of the robot being built. Actually the dictionary itself will have a hierarchical structure (see section 4.2.3).

For the Previ manipulator only there will be no less than 11 such prototypes, corresponding to `PreviBase` (3 parts), `PreviLink1` (4 parts) and `PreviLink2` (2 parts, used twice). When an instance of Previ is created, class `Cylinder` is accordingly instanciated 11 times:

```
Cylindre new: 'Previ.1.1' asSymbol    (first part of the base)
Cylindre new: 'Previ.1.2' asSymbol    (second part of the base)
Cylindre new: 'Previ.2.2' asSymbol    (second part of first link)
etc ...
```

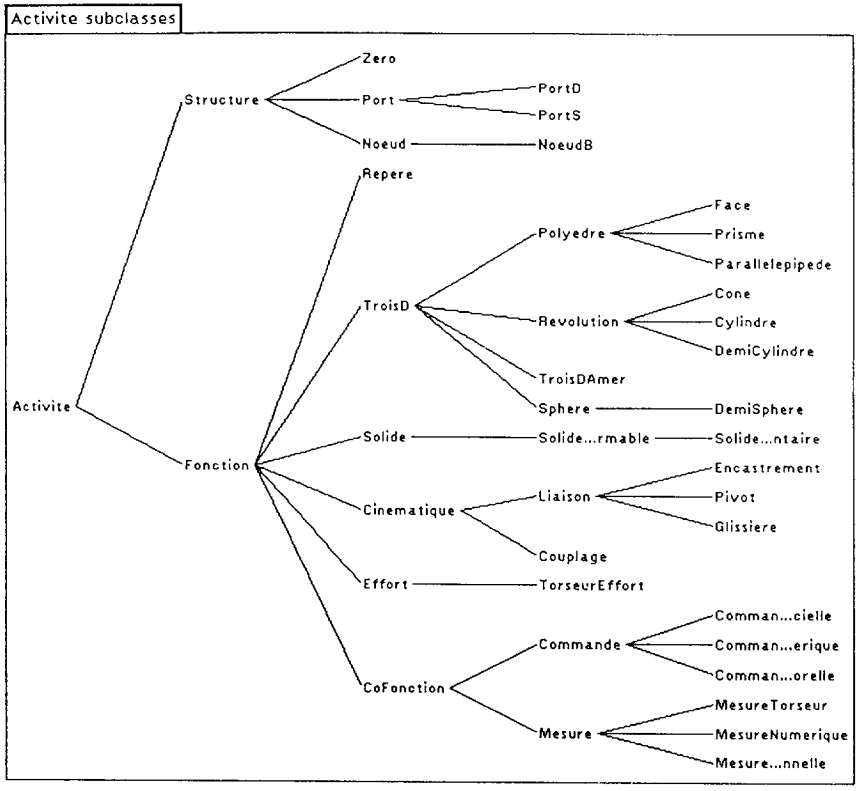


Fig. 3: The class hierarchy of *Systalk* facets
(note that *3D* is French for *Shape*, *Effort* idem for *Force&Torque*, and *Repere* for *CartesianFrame*)

4- PART-WHOLE HIERARCHIES

4.1- Problem statement

In some sense, giving the set of facets of a system (robot) amounts to defining its type. Once this type is fixed, the structure of the system is defined by its decomposition in subsystems. It is therefore natural to try and describe a system in terms of its subsystems and relations between them, without explicitly mentioning the facets. Such a description will be formalized as the writing of a class (e.g. class `Previ`).

At the class level, only the classes of the subsystems will appear. This implies a certain loss of information that must be compensated by an equivalent injection of knowledge in those classes on the one hand and into the composition methods on the other hand. Our purpose is to give some techniques for doing so (preliminary report in [12]). Typically, we must be able to specify that a certain subsystem not only belongs to a certain class, but also has some fixed parameters, or that it must satisfy some constraints. Since most of the

properties of our systems are attached to their facets, our first task will be to get together part-whole hierarchies and our multi-facet technique.

The hierarchical decomposition of a system induces a corresponding hierarchical structure for each of its facets. The corresponding trees are identical to the decomposition tree of the system where a label `nil` at a node denotes the absence of the facet for the corresponding part (compare fig. 4 with fig. 2 top right).

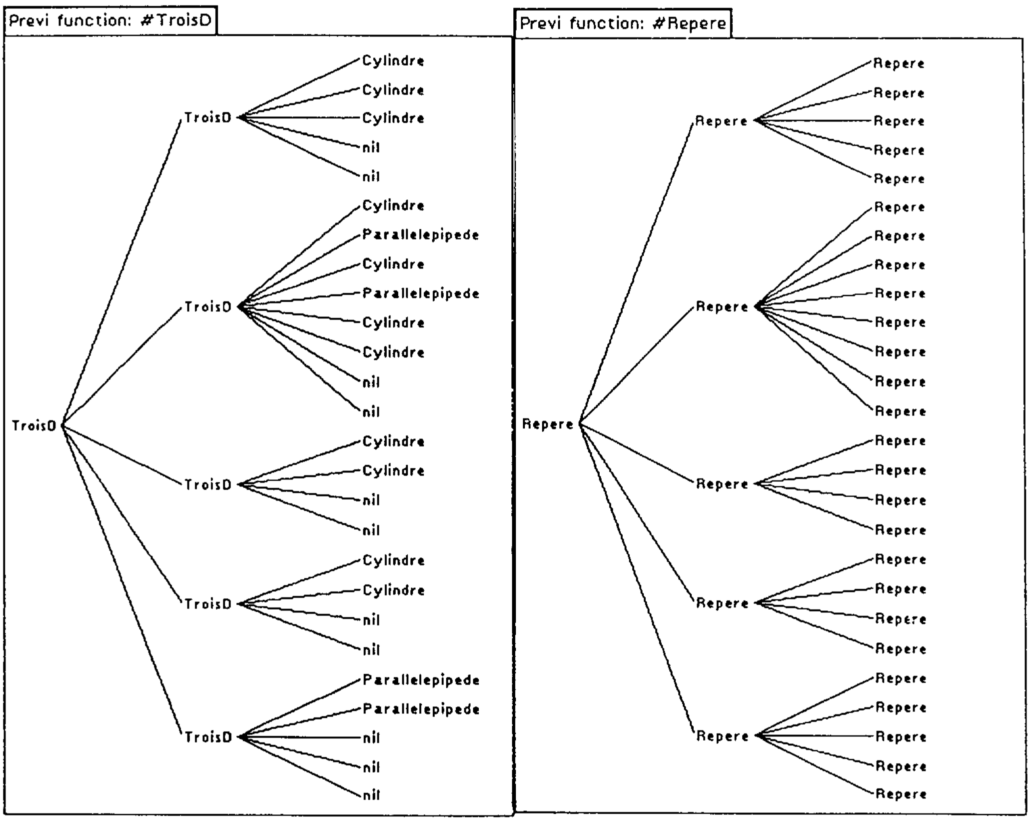


Fig. 4 : The decomposition trees of *Previ'* s facets `Shape`(= TroisD) and `CartesianFrame`(= Repere)

As an example of the hierarchical structure of facets consider the Previ manipulator, defined as the composition of 5 articulated bodies (one base, one gripper, three links):

- its `Shape` facet (visible shape) is clearly composed of the `Shape` facets of the 5 bodies.

- its `Solid` facet is composed in a way that is less visible, but not less real, via an algebraic summation of the masses and the centres of gravity of the subsystems (i.e. from their `Solid` facets).

- the strucure of its facet `cartesianFrame`, on the contrary, is not to be seen as a decomposition, but in the expression of the cartesian frames of the 5 subsystems with respect to that of the system.

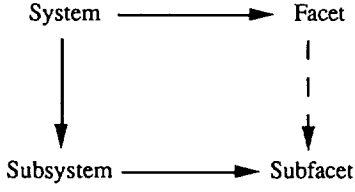
This example shows that the induced hierarchical structure is not the same for every facet. Actually, the way this structure is derived from that of the system depends on the facet class, but only on it. It is an integral part of the definition of the facet.

At the implementation level, the hierarchical structure of a facet is not explicitly represented. It is computed on first demand from the structure of the system, then (to save computation time) it is stored in its associated communication link (see § 5.1.). We must now find means to automate the definition of subfacets from the definition of subsystems.

4.2- Hierarchical prototypes

4.2.1- *Lazy instanciation*

We start again from the instanciation problem. Creating a system involves creating all its subsystems, hence all the facets of its subsystems. The facets of a subsystem are also subobjects of the facets of the system, so that there are two ways to create them :



In order to save memory and speed up the instanciation process, we use a kind of lazy instanciation, where all subsystems are instantiated, whereas facets are created only when required. Therefore, subfacets will be reached through the subsystem they belong to and not from their "superfacet".

4.2.2- *Paths*

Now subsystems are referred to as paths (following Thinglab) in the decomposition tree. If *S* is the name of the system, its subsystem n° *i* will be denoted by *S.i*, and subsystem n° *j* of *S.i* will be denoted by *S.i.j* etc. Such a path (e.g. `'Previ.3.1'`) is the only name of the subsystem known to *Systalk*. Hence we must deduce from it all the prototypes that we need for the different facets of the subsystem it refers to. Recall that, for each facet, these prototypes are to be found inside the class of the facet. Thus we must set up an interpretation scheme for subsystem paths that works with every such (multi-parameter) class.

Now, we have to take into account the fact that a component may be considered at various levels of integration. In the same way as we refined (in § 3.2.1) the general notion of a cylinder as cylinder-with-radius-10 etc, we want to consider `Bar` (1) in itself, (2) as the first element of `PreviLink2` in itself, (3) as the first element of `PreviLink2` taken as the third component of `Previ` in itself and (4) should `Previ` be used as n^{th} component of a supersystem `X`, as the first element of the third component of the n^{th} component of `X` in itself, etc. These different notions of a bar are naturally denoted by paths `'Bar'`, `'PreviLink2.1'`, `'Previ.3.1'` and `'X.n.3.1'`, etc.

We want to provide an explicit representation of those various notions in order to allow the full reuse of them. Clearly, they have to be represented as prototypes attached to class `Bar`. Moreover, each of these prototypes is included (as a dictionary) in the next one in the above integration order. This order corresponds to an incremental specification of the prototypes:

- path `'Bar'` says nothing more than "it is a bar", leaving all parameters undefined (generic bar).
- path `'PreviLink2.1'` says that this bar is the first element of a super-system that is of kind `PreviLink2`. This specifies the relative orientation of the bar, which is reflected in its facet `cartesianFrame`, but not its geometric dimensions.
- path `'Previ.3.1'` says that the abovesaid supersystem is indeed the third subsystem of `Previ` and which completes the specification of the bar.
- other paths (like `'X.n.3.1'`) won't add any specifications to the bar.

4.2.3- *Utilization trees*

So, when referring to a bar, path `'Previ.3.1'` can be interpreted as a sequence of increasingly constrained contexts, starting with the empty context and ending with a fully specified one. We call each of these contexts a **utilization** of the bar. We further observe that all such paths concerning a given system can be assembled as a tree, which we call the

utilization tree of the bar in the system. In some sense, the family of those trees for the various subsystems is dual to the decomposition tree of the system. At the level of class `Bar`, the utilization trees corresponding to the various robots available in *Systalk* are collected as a single tree, with the empty context as root, which we call the **utilization tree** of subsystem `Bar`.

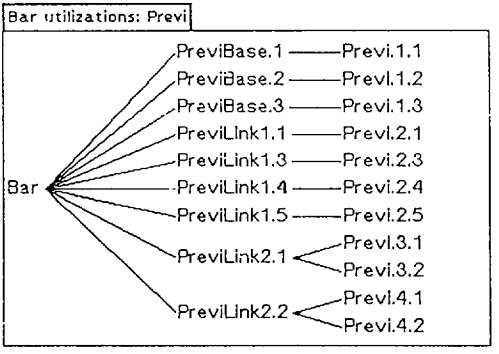


Fig. 5 : The utilization tree of `Bar` in `Previ`

This solves our problem : suppose our subsystem `Bar` owns facet `F`, the utilization tree is readily converted into a tree of prototypes to be given as a value to the variable prototypes of class `F`. We only have to make a slight change to the mechanism of multi-parameter classes in order to accomodate a tree of prototypes instead of a dictionary of contexts.

This requires the introduction of a class `Prototype`, subclass of `Dictionary`, which will define the hierarchical tree structure by means of an instance variable `superPrototype`, and to modify method `init:` of class `MultiParaObject` as follows :

```
!MultiParaObject methodsFor: 'initialization'!

init:  aPrototype
     aPrototype associationsDo:
          [:a | self perform: (a key, ':') asSymbol
                with: a value].
     self init: aPrototype superPrototype  !!
```

5- COMMUNICATION SCHEMES

5.1- The idea and the three classes

So far, our implementation captures the structural relationships between subsystems. There remain to express the functional ones. For instance, facet `Kinematics` works with a number of contacts between homologous facets of static joints, revolute joints, links etc. These communications between facets of the same kind are organized in *communication schemes* which we represent by specific objects (which we call *communication links*) having as attributes several *communication channels* (symbolized by mere numbers) and an *interconnection matrix* [13]. They fall into three main classes which we call `Zero` (vertical communication of a father with all of its sons collectively), `Port` (vertical communication of a father with each of its sons individually) and `Node` (horizontal communication between brothers).

Communication links of classes `Zero` and `Port` answer to messages :

```
propagate: aSelector
propagate: aSelector arguments: anArray
```

those of class `Node` to messages (one for emission and one for reception) :

```
propagate: aSelector arguments: anArray target: aChannel
propagate: aSelector arguments: anArray source: aChannel
```

To each type of facet corresponds one type of communication scheme. For instance, facet `Shape` communicates in mode `Zero`, `Kinematics` in mode `Node`. This correspondence is expressed by an instance variable called `structure` in all the facet classes (defined in the metaclass), with value the associated class of communication schemes. Every facet instance accesses its own communication link through a method called `comLink` : here is a simple example :

```
!Measurement methodsFor: 'work' !

getValue
     ^self comLink propagate: #getValue !!
```

where it is understood that `Measurement` facets of elementary components have specific `getValue` methods.

Conversely, every communication link accesses the facet object with which it is associated through a method called `function` (with no argument). It owns an instance variable `subComLinks` with value the set of the communication links of the subfacets, thus representing in an indirect way the hierarchical structure of the facet as indicated in § 4.1.

5.2- An example of communication

Propagation of the `display` message to facet `Shape.` : displaying a shape needs a cartesian frame which has to be transformed down the communication line according to the relative positions of the subparts. This transform belongs to facet `Shape` but is operated by the communication scheme thanks to a compound selector (**`trans.`**<propagated selector>). As usual, `Shape` subclasses will have their own specialized `display` methods.

```
!Shape methodsFor: 'display' !

displayAt: aPoint frame: aCartFrame scale: aNumber
     self comLink propagate: displayAt:frame:scale:
               arguments: (Array with: aPoint
                                with: aCartFrame
                                with: aNumber)!!

!Shape methodsFor: 'transform'!

trans.displayAt: aPoint frame: aCartFrame scale: aNumber
     ^Array with: aPoint
          with: ((system function: #CartesianFrame)
                                  transform: aCartFrame)
          with: aNumber!!
```

Method `trans.display` of `Shape` is operated by method `transform:arguments:` of `Zero` as shown in the code for `propagate:`

```
!Zero methodsFor: 'propagation'!

propagate: aSelector arguments: anArray
     self subComLinks isEmpty
          ifTrue: "rock-bottom facet : execute"
                    [self function perform: aSelector
                                withArguments: anArray ]
          ifFalse: "intermediary facet : transform then propagate"
                    [self subComLinks do:
                      [:cl| cl propagate: aSelector
                         arguments: (cl transform: aSelector
                                          arguments: anArray ) ]] !!
```

5.3- Prototypes with translation

In the hierarchical contruction of a system S, if no coupling occurs, the channels of the communication links of the various facets of the subsystems of S become automatically channels of the links of the facets of S. For instance, assembling two robots with p resp. q degrees of freedom (i.e. p resp. q input-output channels for facet `kinematicControl`) yields a robot with $p+q$ degrees of freedom, i.e. $p+q$ channels in the same facet. Coupling between subsystems represents physical bindings and reduces the total amount of channels in the supersystem. For instance, a rigid binding between subsystems A and B results in

identifying a channel of *A* with a channel of *B*, submitted to compatibility relationships (e.g. male/female parts, see [14] for details).

Hence we have to set up some sort of algebra for interconnection matrices, which is part of the definition of our communication scheme classes :

- Schemes of class `Zero` have no channel at all, hence no matrix. They address directly the subfacets of the same kind and establish a collective communication with them (as demonstrated by the code for propagation above).

- In schemes of class `Port`, each channel of the subsystem corresponds to a channel of the supersystem, through an automatic renumbering process. The interconnection matrix is trivial.

- In schemes of class `Node`, the interconnection matrix may be arbitrary, and must be explicitly given by the user.

Here is the matrix for the `Kinematics` facet of our Previ example. Each line corresponds to one of the 5 subsystems (see § 2.3.2 and fig. 2), all of which happen to have the same number of channels, namely 2. Each channel *c* appears as a pair of integers *(x y)*, indicating that channel *c* is connected to channel number *y* of subsystem number *x* (*0* meaning the supersystem).

PreviBase :	[(0 1) (2 1)]	the 1st channel of the base is the input channel of Previ, the 2nd one is connected to the 1st channel of PreviLink1
PreviLink1:	[(1 2) (3 1)]	
PreviLink2:	[(2 2) (4 1)]	
PreviLink2:	[(3 2) (5 1)]	
PreviGripper:	[(4 2) (0 2)]	the 2nd channel of the gripper is the output channel of Previ.

As a consequence, Previ has two kinematics channels left instead of the 10 channels provided by its 5 subsystems.

Communication scheme classes are submitted to the same process of incomplete instanciation via prototypes as facet classes. However, what is practically needed are prototypes that specify the total number of channels as well as the interconnection matrix, but leave open the exact identity of the subsystems that they are going to interrelate. These subsystems are represented by *ad hoc* placeholders (their numbers). The symbolic matrices (such as the one above) are stored in instances of a special class, a variant (but not a subclass) of `Prototype`. This class also supports the translation process of the placeholders into the actual subsystems during instanciation.

In our example, in the course of instantiating class `Previ`, numbers *0* to *5* appearing in the symbolic connection matrix will be replaced by the actual subsystems of the instance of `Previ` that is created.

6- CONCLUSION

We have presented a working system that integrates in a homogeneous way multi-facets and part-whole hierarchies. Of course many improvements are in order, notably some form of compilation to gain speed.
The main direction to be explored, in our opinion, is the meta-knowledge needed to implement reasoning about the system. Our first attempt was to couple *Systalk* with our version of OPUS [10], a Smalltalk-80 interpretation of OPS-5 (see Pachet [11]) and to have OPUS production rules control a *Systalk* robot. The next will be to integrate a powerful semantic network. Work is going on in this way.

ACKNOWLEDGEMENTS

We wish to thank Mr Michel Delbos, formerly head of the Robotics Group of the Sureillance, Diagnostics and Maintenance Dept, Electricité de France, for his help and support while developing *Systalk*.

REFERENCES

[1] Blake, E. and Cook, S : On including part hierarchies in object-oriented languages, with an implementation in Smaltalk, ECOOP '87, p. 45-54.

[2] Bobrow, D. and Stefik, M. : The LOOPS Manual, Xerox Corp. (1983).

[3] Borning, A. : THINGLAB - A Constraint-Oriented Simulation Laboratory, Ph.D. thesis, Stanford 1979.

[4] Briot, J.-P. and Cointe, P. : A Uniform Model for Object-Oriented Languages Using the Class Abstraction, IJCAI '87, vol.1, p. 40-43.

[5] Carré, B. : Une méthodologie orientée objet pour la représentation des connaissances - concepts de point de vue, de représentation multiple et évolutive d'objets, Thèse, Université de Lille, 1989.

[6] Carré, B. and Geib, J.-M. : The Point of View notion for Multiple Inheritance, OOPSLA-ECOOP '90, p. 312-321.

[7] Coiffet, Ph., Zhao, J., Zhou, J., Wolinski, F., Novikoff, P., Schmit, D. : About qualitative robot control, Nato Workshop on Expert Systems and Robotics, Corfu 1990.

[8] Cointe, P. : Metaclasses are First Class : the ObjVlisp Model, OOPSLA '87, p. 156-167.

[9] Goldstein, I. and Bobrow, D. : Extending Object Oriented Programming in Smalltalk, First Lisp Conference, Stanford 1980, p. 75-81.

[10] Laursen, J. and Atkinson, R. : OPUS : a Smalltalk Production System, OOPSLA '87, p. 377-387.

[11] Pachet, F. : Mixing Rules and Objects : an Experiment in the World of Euclidean Geometry, 5th International Symposium on Computer and Information Sciences, Nevsehir (Turkey) 1990, p. 797-805.

[12] Wolinski, F. : Gestion des contraintes induites dans la structuration des objets en sous-objets, Reconnaissance des Formes et Intelligence Artificielle (RFIA), Paris 1989, p. 163-171.

[13] Wolinski, F. : Modeling and simulation of robotic systems using the Smalltalk-80 environment, TOOLS '89, p. 141-149.

[14] Wolinski, F. : Représentation de systèmes robotiques en Smalltalk-80, Convention IA 1990, Paris (Hermes publ.) p. 685-699.

[15] Wolinski, F. : Etude des capacités de modélisation systémique des langages à objets appliquées à la représentation de robots, Thèse, Université Paris VI, 1990.