

Formal Specification, Modelling and Exchange of Classes of Components according to PLib. A case study.

Eric Sardet, Guy Pierra, Yamine Ait-Ameur

LISI / ENSMA

Teleport 2 - Avenue 1

Site du Futuroscope - BP 109

86960 FUTUROSCOPE CEDEX

FRANCE

e-mail: sardet@ensma.univ-poitiers.fr

pierra@ensma.univ-poitiers.fr

yamine@rabelais.univ-poitiers.fr

Abstract : Component catalogues embed various kinds of component knowledge that are to be captured in intelligent electronic catalogues. Thus, the PLib specification, officially ISO 13584, encompasses a lot of aspects and includes a number of EXPRESS constructs. The size of this specification makes it difficult to apprehend how its different building blocks (in EXPRESS schema) fit together.

This paper presents, through a simple, precise and complete example, a case study of the use of PLib for intelligent electronic catalogues modelling and exchange. This example consists of a simple screw family. The following points are addressed. First, the definition of a data dictionary that describes, for the catalogue user, all the concepts involved in the component family. This data dictionary constitutes, in PLib, the dictionary level. Second, the description of all the different screws of the screw family is realised. This set of possible instances is implicitly (intentionally) defined using table and expression. This description constitutes, in PLib, the library specification level. Finally, the description of the representations that are provided for our simple parts family is done. This description includes both the characterisation of the particular provided representations (i.e., the simplified 2D geometry) and the information needed to generate these representations (i.e., external files that contain parametric programs).

The case study shows how the different elements of ISO 13584 fit together, and enables to model the different kinds of component knowledge embedded in paper catalogues.

Introduction

In several engineering areas, such as electronic engineering or mechanical design, a product is often defined from pre-existing components. The availability of all the pre-existing

components in digital libraries, and their exchange or remote access through the network would drastically increase the efficiency and the quality of the design process of such products.

The goal of the PLib standard (officially ISO 13584, Parts Library) is to define a neutral format for the representation of pre-existing components, thus providing for exchanging parts libraries between heterogeneous systems.

The PLib architecture involves several mechanisms that are intended to solve the different difficulties involved in parts library modelling and exchange. The first one is the computer-sensible identification of all the concepts involved in a digital library. A semantic data dictionary information model has been developed (together with IEC where it is published as [IEC1360-2]) in order to solve this problem. The second problem is the multiplicity of the discipline-specific representations of a part. An architecture that consists of two kinds of parallel class hierarchies (the general model class hierarchy, that models the component classes, and the different functional model class hierarchies that model the different discipline-specific representations) has been defined in order to offer this multi-representation capability. The third one is the parametric nature of the representation. Indeed, to avoid data blow up, parametric descriptions are mandatory. Thus, two approaches have been developed to model parametric geometry. The first one is based on classic variant programming. It led to the definition of a standard geometric interface (documented in [ISO13584-31]). A second approach, more recent, is a STEP compliant parametric data model.

One of the major difficulty of the PLib information model is that its formal specification is huge, and therefore complex and not handily to understand.

The goal of this paper is to present, through a simple and precise example (a screw), a case study of the use of the PLib resource constructs for parts library modelling and exchange.

In the first section of this paper, we briefly outline the main features of the EXPRESS language. Our focus is on the graphical notations for EXPRESS, known as EXPRESS-G, and on the exchange format of a population compliant with an EXPRESS information model, known as its physical file structure, that are used in the remaining part of the paper.

In the second section, we present the target example in terms of what and how it should be described. Our example is a (small) family of hexagonal screws that we want to describe together with their geometry. The data dictionary description and the library specification of this component family are introduced in the third section. The main concepts of PLib are thus developed and illustrated. Section 4 overviews the multi-representation capabilities of PLib. They are illustrated on our case study through the definition of 2D geometric representations for our screw family. As a conclusion, the capabilities of the PLib specification are discussed.

1. EXPRESS-G notations and physical file structure

EXPRESS is a specification language which has been designed in the context of the STEP (STandard for the Exchange of Product model data, officially ISO 10303) project. Its main objective is the description of models for exchanging product data and product data models [ISO10303-11]. This language can be used for the specification of several applications in the computer science area, and it has been proven to be well suited even for the specification of the dynamic aspect of digital libraries [APS 95a].

This section introduces the main features of the EXPRESS language. It gives a global overview of this language and focuses on the constructs that will be used in the remainder of this paper. Following [ISO 10303-11], an EXPRESS specification is defined by a set of entities (ENTITY) which represent the objects to be modelled. Each entity is defined by a set of characteristics, namely the attributes. Each attribute has a domain (TYPE) where it takes its value, and EXPRESS allows to constraint this domain thanks to the domain constraint rule (WHERE clauses). These entities have a hierarchical structure allowing multiple inheritance as in several object oriented languages. This part of the specification defines the structure of the data model.

Unlike most of the data modelling formalisms that mainly capture cardinality or set-oriented constraints on the data conforming to the data model, EXPRESS enables to model any kinds of constraints. Thanks to several built-in functions, and to a PASCAL-like procedural language, functions may be defined. These functions in turn may be used to define constraints, either in local WHERE clauses, on the data described in an entity type, or in global rules (RULE clauses) that constraint the complete data model. This outstanding capability enables for instance to specify all the constraints that shall hold in the description of the entities that define a set of (object oriented) classes to ensure that this description fulfils the usual constraints of an object oriented language (e.g., visibility through inheritance, class encapsulation, ...). At last, entities and functions are gathered in a common structure named SCHEMA that provides for modularity.

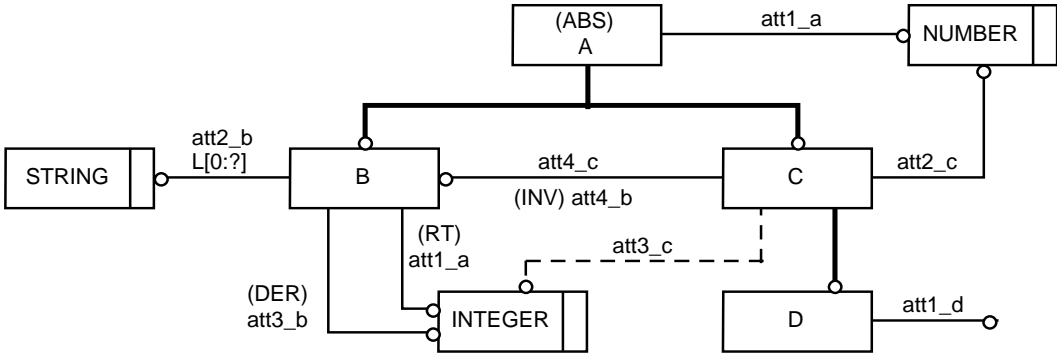


Figure 1: Example of EXPRESS-G representation

The previous schema example (see **Figure 1**) introduces three entities A, B, C. A is the parent of B and C; both of them inherit all the characteristics from A. The keyword (ABS) that stands for ABSTRACT SUPERTYPE indicates that A is an abstract class and thus does not have instances. The following graphical notations have been used:

- All the lines are oriented toward their rounded extremity.
- A thick line stands for the "is_a" relationship (inheritance).
- A normal line represents an attribute.
- A dashed line represents an optional attribute.

The data type of an attribute is represented at the end of an attribute relationship. It may be another ENTITY (B for att4_c) or a simple type represented in a box with a double right side (e.g., STRING).

The data type of an attribute may be redefined by a more restrictive one (e.g., *att1_a* defined as NUMBER in *A*, is redefined as INTEGER in *B*). It is identified by (RT).

When the precise data type of an attribute is not relevant for the purpose of some discussion (or when it is not still defined), an attribute may be represented as a simple line with a rounded end (*att1_d*).

When the value of an attribute may be computed by evaluating an expression whose domain consists of other attributes (or entity instances), this attribute is said to be derived (DER). Its value is not represented in an exchange context.

(INV) introduces an inverse attribute. In our example, the entity *C* has established a relationship with the entity *B* by way of the explicit attribute *att4_c*; so, the inverse attribute *att4_b* may be used to describe this relationship in the context of the entity *B*.

Instances of a model defined in EXPRESS can be described and exchanged through *physical files*. Their format is defined in [ISO10303-21] which specifies an exchange structure using a clear text encoding for instances of EXPRESS defined models. The file format is suitable for the transfer of instance data among computer systems.

Instances of the previous schema example would be written as follows (the attribute value order results from the list order of the attribute definition in the textual version of this schema):

```
#1 = B ( 33,          /* the first attribute of entity B, inherited
                    from A, is redefined as an integer */
        *,          /* the asterisk represents a redefined
                    attribute (att1_a) */
        ('abc', 'xyz')); /* the aggregates are written into
                        parentheses; here a list of two strings */

#2 = C ( 1.2,        /* the real attribute 1.2 (att1_a) inherited
                    from parent entity A */
        (0.75, 0.45), /* set of two reals */
        $,          /* the dollar character represent an optional
                    non-evaluated attribute */
        #1 );      /* the reference of an instance of entity B
*/
```

Figure 2: An example of instances representation

The values of attributes of simple types (INTEGER, STRING, LIST, ...) are directly represented into instances of entities. The values of entity data type attributes are represented by entity names (for example #1 in instance of entity *C*). Note that neither INVERSE attribute nor DERIVE attribute are represented in an instance.

Next sections make a large use of the EXPRESS-G notations and of physical file. This section has presented the kernel of EXPRESS-G and physical file which is enough to understand the basic constructs presented in those sections.

2. Example presentation

2.1. The example

In this section, we present the example that will be treated. **Figure 3** illustrates the parts family intended to be described. It is a family of screws, denoted SCREW, provided by a screw supplier and intended to be used in some mechanical context.

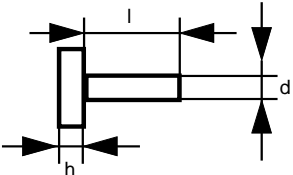


Figure 3: The case study

Our intend is to completely describe the supplier catalogue, without any reference to a dictionary that might be defined by some standard (note that such a dictionary does not yet exist for mechanical products). Therefore, the screw family shall be modelled at both the dictionary definition and the library specification levels. The allowed instances will be those where the values of the properties describing the part belong to the list of authorised tuples of values defined in the library specification of the part (see **Figure 4**).

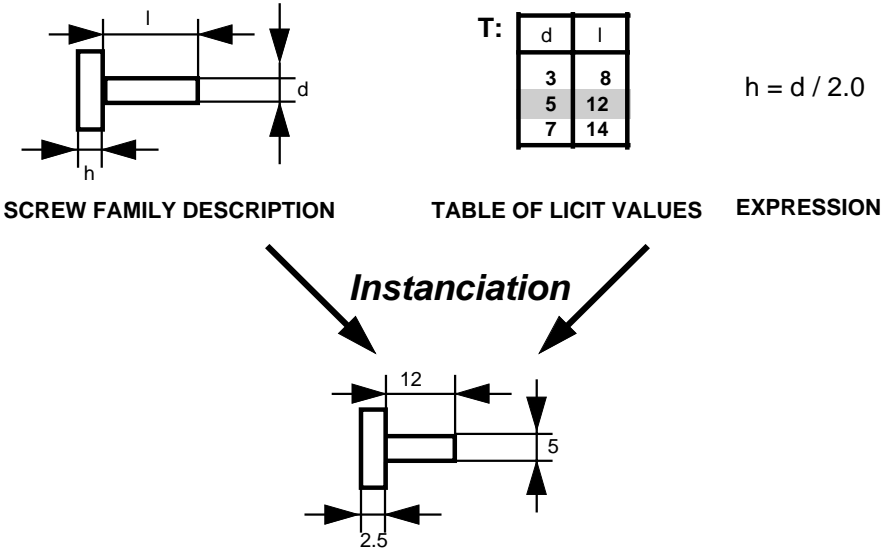


Figure 4: Complete description of the family to be described

The general model description (what is the family of parts), the functional view description and the functional model description (the class that provides the previous kind of representations, i.e., geometry, for this particular family of parts) will be described in the different modelling steps.

Thus, we should introduce the definition of three classes at the dictionary level (one for each kind of classes), and two classes at the library level (one for the general model class and one for the functional model class). Note that a functional view class is like an "interface" in Java, or a "specification" in ADA: it has no "body". This means that in PLib such a class is only associated with a dictionary definition.

The description of three part properties will also be done as part of the general model description:

the nominal diameter d , is the identification characteristics of the screw (this only property is sufficient to identify one particular instance within its family),

the length l computed from a table named T and from the nominal diameter d , and

the thickness of the head screw, named h , derived from the nominal diameter d through the evaluation of an expression ($h = d / 2.0$).

The functional view class description enables to define what kind of representation(s) are provided by the digital library. Its dictionary definition includes the name of the representation ("geometry") and the particular properties that characterise in detail one representation of this kind, i.e.:

what is the *geometry_level*: 2D, wireframe or solid;

what is the level of detail (*detail_level*): simplified, standard or extended;

what are the *sides* to be displayed: front, rear, right, left, top, bottom.

Note that such functional view class definitions are intended to be standardised in the view exchange protocol series of ISO 13584. When a digital library provides only such "standardised" representations, the corresponding functional view classes need not to be defined: they should only be *referenced* using the BSU mechanism out-lined below.

Finally, the functional model class, that is required to provide a 2D, simplified representation for the six different views of the three different screws of the SCREW family will be completely defined. Thus, another property will be introduced in order to reference the parametric program corresponding to the chosen screw side representation.

3. Definition of a parts family

A general model class is the PLib model of one particular family of components, here the SCREW family.

3.1. General model dictionary definition

The dictionary definition level provides a computer sensible identification and a human-readable definition of the concepts involved in a particular family of components.

3.1.1. The Basic Semantic Units

The description of a data dictionary according to PLib requires to specify what are the identifiers of the different concepts (these identifiers are called Basic Semantic Unit: BSU) involved in the parts family definition. These identifiers define unambiguously and universally each concept within a PLib compliant data dictionary (see **Figure 5**).

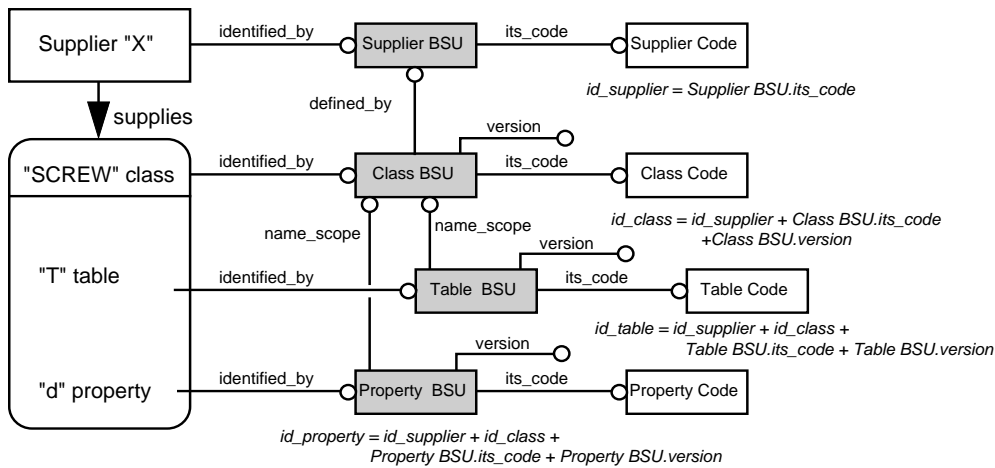


Figure 5: The BSU mechanism

Therefore, a BSU has to be defined for each element described in the data dictionary (the supplier *X*, the class "SCREW", the table *T*, and each one of the properties that describes the SCREW parts family (*d*, *l*, *h*)).

The following figure (see **Figure 6**) shows different possible instantiation of the BSU concept (the codes are fixed character length).

```
#20 = supplier_BSU ('X_Supplier_____', *);
#50 = class_BSU ('Screw_____', '001', *, #20);
#90 = property_BSU ('d_____', '001', *, #50);
#120 = table_BSU ('T_____', '001', #50, *);
```

Figure 6: Some BSU instances

Following the PLib specification, it can be noticed that the string '001' is corresponding to the version number of the described BSU.

3.1.2. The dictionary elements

A BSU identifies a concept and enables to reference it. A dictionary element provides a computer-sensible and human-readable definition of the concept. This relationship between these two levels is presented in **Figure 7**.

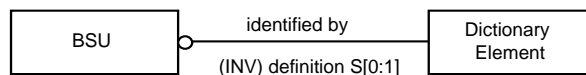


Figure 7: The BSU / Dictionary element relationship

The dictionary element entity is subtyped in order to define a particular dictionary definition structure for each items that can be involved in a dictionary definition. In our example, dictionary elements are associated to :

supplier: definition of the organisation he belongs to, address, ...:

```

#31 = supplier_element (#20, /*reference to its supplier BSU*/
                        $, /*an optional value (date of
creation)*/
                        #21, /*organisation: not represented here*/
                        #22); /*address: street, city, ...: not
                               represented here*/

```

Figure 8: A supplier dictionary definition

properties: classification of the property (part characteristics, context parameter, ...), definitions of the name (in a human readable format with possible translations), definition of the data type and of the unit:

```

#91 = non_dependent_P_det (#90, /*reference to its property BSU*/
                          $, '001', /*revision*/
                          #92, /*the human readable name of the
                               concept, with possible translations*/
                          'nominal diameter', /*definition*/
                          $, $, $, *, $, (), $,
                          'TO3', /*classification of the measure
                               according to ISO 31*/
                          #93, /*data type definition:
                               length_measure*/
                          $);

```

Figure 9: A property dictionary definition

class: name (in a human readable format with possible translations), definition, list of properties that may be used to describe an instance of this class (applicability of the properties):

```

#71 = component_class (#50, /*reference to its class BSU*/
                      $, '001',
                      #72, /*the human readable name of the
                           concept, with possible translations*/
                      'Family of hexagonal screws ...',
                           /*definition */
                      $, $, $, *, $,
                      (#90, #100), /*applicable properties:
                                   reference by means of their BSU*/
                      $, $, $, $, $);

```

Figure 10: A class dictionary definition

table: definitions of the columns, and among them, the key of the table:

```

#121 = RDB_table_element (#120, /*reference to its table BSU*/
                        $, '001', *,
                        #122, /*the human readable name of the
                               concept, with possible translations*/
                        'This table...', $, $, *,
                        (#96, #116), /*the columns meaning:
                               variable semantics entities that refer to
                               the property_BSU of d and l
                               (not represented here)*/
                        (#96), /*the key of the table: d*/
                        *, *, *);

```

Figure 11: A table dictionary definition

3.2. General model library specification: class extension definition

Only some instances of the SCREW family are allowed (in our example: three different instances). This set of instances is defined through the:

choice of the identification characteristics of the family (the properties that enable to completely and unambiguously characterise an instance of the parts family): the *d* property in this example,

definition of the allowed set of values for this identification characteristics by means of a domain definition, and

definition of the derivation functions that specify the values of the other part characteristics (here: *h* is specified by means of an algebraic expression and *l* by means of a table whose key corresponds to the identification characteristics).

3.2.1. Overall Architecture

The relationship between a dictionary element and an associated library specification is outlined in **Figure 12**. A uniqueness constraint exists, in ISO 13584, for the cardinality of the *referenced_by* attribute. Therefore, every reference to a concept (e.g., a class) is done through its BSU. The same reference mechanism is used for a class whose dictionary element and/or library specification belongs to the digital library and for a class assumed to be available on the receiving system (e.g., a standardised dictionary). This reference through BSU is named "the BSU mechanism".

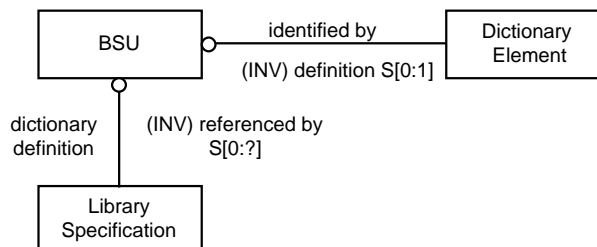


Figure 12: The Dictionary Element / Library Specification relationship

Each specification realised at the library level must be consistent with the corresponding dictionary element definition (type / value consistence).

3.2.2. Variables representation

The definition of an expression (in order to compute the value of the head size h from the nominal diameter of the screw d) requires to model the variables that are involved as operands. The following example shows how to represent the variable associated to the nominal diameter d .

```
#97 = real_numeric_variable ();
#96 = self_property_value_semantics (#90, $); /*#90 is the BSU for d*/
#98 = environment (#97, #96);
```

Figure 13: Variable representation in a physical file

This example shows that a variable is defined by a three-fold modelling structure:

a syntactical level: definition of the name (entity name) and of the type of the variable (in **Figure 13**, the "name" of the real variable is #97; this "name" may be referenced in expressions),

a semantic level: definition of the meaning of the variable, and of the mechanism that associates it with a value (in **Figure 13** the variable stands for the diameter d of the "SELF" instance, d is referenced by its BSU: #90),

the association between syntax and semantics is done through the environment relationship (#98 in the example) using an entity/relationship model.

3.2.3. Optional and displayable natures of properties

The dictionary definition of the properties includes its data type definition. When used in the context of the library specification of a particular class, a property may be allowed to have no value. For instance, when modelling a "bolt+nut+optional washer" assembly class, some assembly instance have no washer.

The library specification includes the definition of the optional or mandatory nature of the data dictionary properties, and the possible display ability of these same properties. The following figure (see **Figure 14**) defines the nominal diameter d as being not optional (every screw *has* a diameter) and displayable.

```
#900 = opt_or_mand_property_BSU (
    #90, /*reference to its property BSU*/
    .F., /*is not optional*/
    .T.); /*is displayable*/
```

Figure 14: Optionality and displayability definition

3.2.4. Tables contents

The library specification of a table enables to define the actual content of this table. It shall be consistent with the data types defined in the data dictionary.

```

#910 = table_content (#120, /*reference to its table BSU*/
*,
(#911, #912), /*the list of the referenced
columns*/
'001', '09-27-96', *);
#911 = real_column (
(3.0, 5.0, 7.0), /*the values of d the column
associated to the nominal diameter*/
*,
'NR2..3.3', /*the format for displaying the
values of the column (a real
like 123.456)*/
*, *);

```

Figure 15: Table library specification

A table content references the BSU that identifies it, and defines the set of values for each of its columns (see **Figure 15** above).

3.2.5. Domains specification

The identification characteristics of a screw (its nominal diameter d in the example) must belong to some well defined value domains. Such a domain constraint is specified through a domain restriction whose structure is outlined in **Figure 16**.

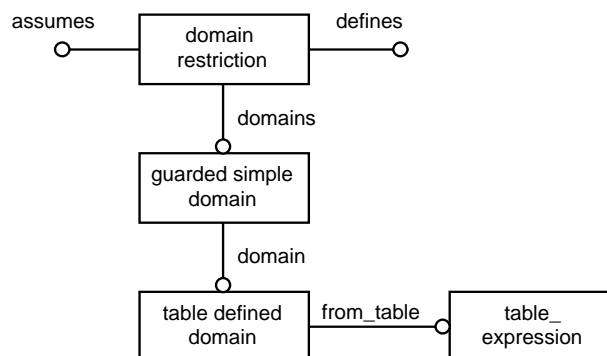


Figure 16: Domain restriction definition

Note that such domain restriction may be defined through different structures (e.g., ranges, relational algebra expressions on tables, ...).

In the example (see **Figure 17**), the value domain of the nominal diameter d is a column (identified through its semantics #96 that refers to the BSU of d (see **Figure 13**)) that belongs to the previously defined table (#120). This domain is not dependent on the value of some other properties, therefore its guard (a Boolean expression), has the special value "others".

```

#901 = domain_restriction ((#96),      /*the property domain that is
                                     defined (defines)*/
                          (),          /*the required properties for
evaluating the property domain (assumes)*/
                          (#902),     /*the guarded domain */
                          $);
#902 = guarded_simple_domain (#903,   /*the guard*/
                              #904);  /*the domain*/
#903 = others ();                    /*this guard means 'in every
                                     cases' */
#904 = table_defined_domain (#905);  /*the referred table*/
#905 = table_literal (#120);         /*refers to the table BSU*/

```

Figure 17: Value domain specification

Note that the table might be an implicit table that results, e.g., from a cartesian product of two other tables.

3.2.6. Derivation functions

The d property being the unique identification characteristics of a SCREW instance, there exists a functional dependency between d and l , and between d and h . A derivation function expresses how l or h should be derived from d .

Two kinds of derivation functions appear in our example: an algebraic derivation and a table derivation.

3.2.6.1. Algebraic derivation function

For the h property, the derivation function is defined by a specialisation of the simple functional domain (that defines a domain for variables as a singleton of which the value is specified as a function) which is a general expression defined value (in the example, the function is an expression $h = d / 2.0$) as presented in **Figure 18**.

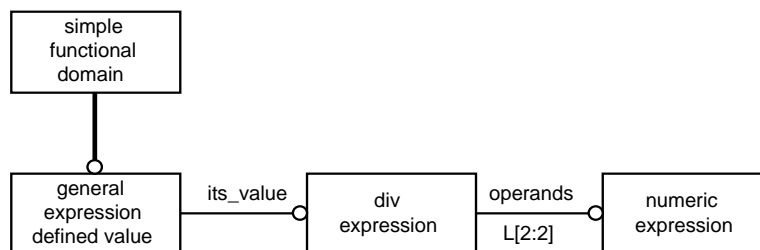


Figure 18: Derivation by algebraic expression

In the example, the derivation function is described as a division between a real variable (associated with the nominal diameter d) and a real constant (2.0), both playing the role of the operands. The following physical file (see **Figure 19**) shows how it is instantiated.

```

#1001 = functional_domain_restriction (
    (#106),      /*the property of which the value
                 is defined, here h*/
    (#96),      /*the self properties that define
                 the parameters of the derivation
                 function, here d, see figure 13*/
    (#1002),    /*domains*/
    $);
#1002 = guarded_functional_domain (
    #1003,      /*the guard: always the same
                 function*/
    #1004,      /*the derivation function*/
    *);
#1003 = others();
#1004 = general_expression_defined_value (#1005);
#1005 = div_expression ((#1006, #97), /*the operands: #97 is the variable
                                     associated with d, see figure 11*/
    *);
#1006 = real_literal (2.0);          /*a literal value*/

```

Figure 19: Algebraic derivation specification: $h = d / 2.0$

This expression specifies the functional dependency between d and the head size h that shall hold for every instance of the class.

3.2.6.2. Table defined derivation function

A derivation table is also defined as a specialisation of the simple functional domain that is a table defined value (there exists in this table a unique tuple (line) that clearly identifies the searched value) as presented in **Figure 20**.

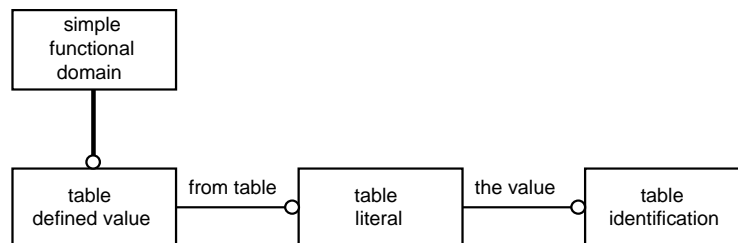


Figure 20: Derivation table

In our example, the value of the length l may be derived from the nominal diameter d using the table T . This functional dependency description is presented in the following physical file (see **Figure 21**).

```

#1101 = functional_domain_restriction (
    (#116),      /*the property 'l' that
                is defined and that belongs to the table
                (defines)*/
    (#96),      /*the parameters of the
                derivation function (here:d) that belong to
                the table (assumes): see figure 13*/
    (#1102), $); /* the derivation function*/
#1102 = guarded_functional_domain (#1103, #1104, *);
#1103 = others();
#1104 = table_defined_value (#1105);
#1105 = table_literal (#120);          /*the referenced table
                                       (the table_BSU)*/

```

Figure 21: Table derivation: $d \rightarrow l$

The *assumes* attribute is the identification characteristics d (the nominal diameter), and the *defines* attribute is the length l to be computed. The referenced table (this reference is realised through the BSU of the table) is the table named T (#120).

3.2.7. Class extension

The general model class extension (see **Figure 22**) of the SCREW parts family gathers all the library specification defined previously.

```

#5000 = item_class_extension
    (#50,          /*reference to its BSU*/
    *, *, *, *,
    (),           /*new derived properties*/
    (),           /*new mandatory properties*/
    (#901),       /*the domain specification
                  for d*/
    (#1001, #1101), /*the derivations*/
    (), (), (), (), (), '001', '001',
    (),
    (#900),       /*identification characteristics
                  (d): see figure 14*/
    (#1000, #1100), /*derived charact. (l, h)*/
    (),           /*context dependent characteristics*/
    .F., $, $, $, $, (), $);

```

Figure 22: The general model class extension

4. Representation of a parts family

The digital library shall not only define the different screws of the SCREW family. It should also provide some representation (e.g., geometry, symbolic, ...) for each of these screws.

4.1. Overview

In PLib, the definition of one representation category (e.g., geometry, schematics, behavioural simulation model, ...) is done through a functional view class. A functional view class just includes a dictionary definition. It specifies the structure of what is to be (dynamically) created by the LMS when the corresponding view is requested by the user for whatever part in the library.

A functional model class is the (object oriented) structure that enables the (dynamic) creation of one particular functional view (e.g., the geometry) for all the different parts described in one general model class. The data structure of a functional model class is similar to the one of a general model class. It includes a class definition and a class extension (library specification).

The relationship between these three kinds of classes is outlined in **Figure 23**

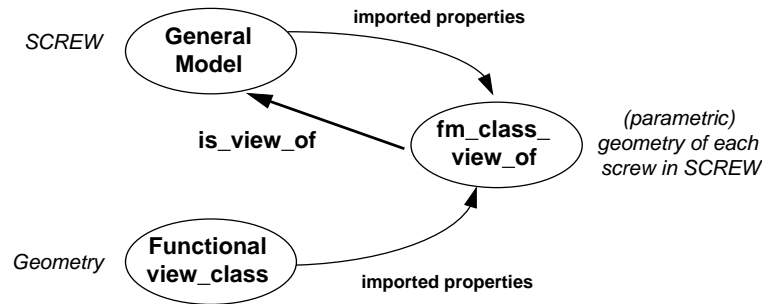


Figure 23: Roles and relationships between classes in PLib

In our example, we will define a functional model class representing some kind of geometry specified by a functional view class for the SCREW parts family.

4.2. The functional view class

A functional view class is identified by a BSU, and contains some properties called the view control variables.

In general, the functional view class and its associated view control variables are assumed to be known by the receiving system, and they are referenced by their BSU. To clarify the structure of the functional view class, we assume that it is also exchanged as part of the digital library.

The "geometry" view is defined in ISO WD 13584-101. It includes three view control variables that characterise respectively:

what is the geometry level: 2D, wireframe or solid;

what is the level of detail: simplified, standard or extended;

what are the sides to be displayed: front, rear, right, left, top, bottom.

For each item (class and properties) referenced, an identifier (the BSU) shall appear in the library (see **Figure 24**).

```

#40 = supplier_BSU ('ISO_____', '001');
#140 = class_BSU ('Geometry_____', '001', *, #40);
#150 = property_BSU ('geometry_level', '001', *, #140);
#160 = property_BSU ('detail_level__', '001', *, #140);
#170 = property_BSU ('side_____', '001', *, #140);
  
```

Figure 24: BSU definition of the functional model items

The dictionary definitions are similar to those of a general model class. However, an interesting point should be noticed. The view control variables are defined as having an

ordered discrete type (as specified previously). In fact, values of these types are coded by some integers associated with a human-readable meaning, possibly provided in different languages ("non_quantitative_integer_type") (see **Figure 25**).

Geometry Level:		Detail Level:		Side:	
2D	->1	simplified	->1	top	->1
wireframe	->2	standard	->2	bottom	->2
solid	->3	extended	->3	right	->3
				left	->4
				front	->5
				rear	->6

Figure 25: Association View Control Variables / Integers

For instance, the *geometry_level* view control variable is defined as a non quantitative integer type (an integer without any unit) taking its value in the integer subset (1, 2, 3). The first value of this discrete type corresponds to a 2D geometry level, ...(see **Figure 26**).

```
#1500 = non_quantitative_int_type ('N..1', #1501);
#1501 = value_domain ((#1502, #1504, #1506), $, $, $);
#1502 = dic_value (1, #1503, $);      /* value=1, translated meaning in
                                     #1503*/
#1503 = item_names ('2D', (), '', $, $);
#1504 = dic_value (2, #1505, $);
#1505 = item_names ('wireframe', (), '', $, $);
...
```

Figure 26: Representation of a non_quantitative_in_type

Note that, normally, functional view classes are intended to be specified as standard data in the view exchange protocol series of parts of PLib and to be stored in the user system. Therefore, a digital library referencing such a functional view class contains the BSU defined in **Figure 24**.

4.3. Functional model class dictionary definition

4.3.1. Overview

The functional model class to be described shall be able to create *geometry* views for all the screws in SCREW. But, such a functional model class is not required to create *all* the geometry views (i.e., 3 x 6 in 2D + 3 in wireframe + 3 in solid = 24 different geometry views). It may create only *some* of the geometry views. This is achieved, in PLib, by the view control variable range which defines the bounds of the supported sub domain for the different view control variables.

In the example, the supported view control variable ranges are:

level [1:1] : a 2D representation (#155 in **Figure 27**),

detail [1:1]: a simplified representation (#165 in **Figure 27**),

side [1:6]: a representation for each side of the part (#165 in **Figure 27**).

The physical file representation is then the following:

```
#155 = view_control_variable_range (#150, 1, 1);
#165 = view_control_variable_range (#160, 1, 1);
#175 = view_control_variable_range (#170, 1, 6);
```

Figure 27: View Control Variable ranges definition

The (parametric) geometry is intended to be represented by ISO 13584-31 compliant FORTRAN programs provided as library external files (files delivered together with the ISO 10303-21 files that define the library structure). In order to trigger the program associated to the view that must be displayed, a new representation variable, called *prg*, has to be defined (at the BSU and the dictionary level). Its value will be computed from a table and from the *side* view control variable (see **Figure 28**).

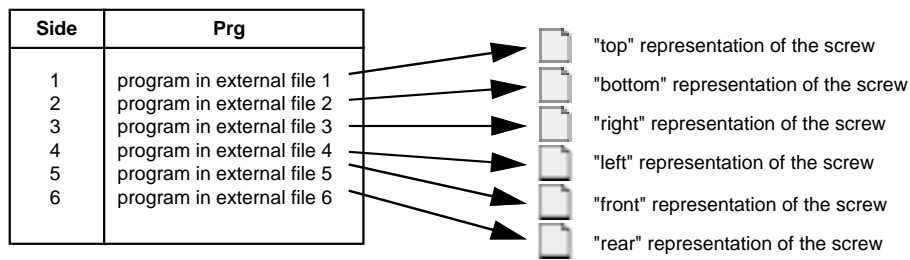


Figure 28: Association Side / Program

Thus, if the *side* value is the front one (*side* coded by the integer 5), the display program *prg* corresponding to this value will be triggered ("front" representation of the screw).

4.3.2. The BSU and dictionary elements of the functional model class

A basic semantic unit has to be created for the *prg* property. Its dictionary element, in terms of PLib, shall be *representation_P_DET* (a property that is defined in a functional model or a functional view class)

Then, at the dictionary definition level, the functional model class will:

- refer to its class BSU,

- reference the functional view class it is able to create,

- specify the view control variable range it supports,

- reference the general model class it represents,

- define the applicability of the *prg* representation property, and

- import some properties from the general model class in order to be able to compute the complete representation (in the example, the three properties will be imported to be used as parametric program parameters).

The final functional model class dictionary definition is presented in the following figure:

```

#71 = fm_class_view_of (#130,          /*reference to its BSU*/
                        $, '001',
                        #72,          /*item_names*/
                        'Creates geometry for...', /*Definition*/
                        $, $, $, *, $,
                        (#180),      /*BSU of the prg property: not
                                      represented here*/
                        $,
                        #140, /*the created view (reference to the
                              BSU of the functional_view_class
                              see Figure 24*/
                        (#155, #165, #175), /*the v_c_v ranges
                                              see figure 27*/
                        (#150, #160, #170), /*the v_c_v BSU
                                              imported from the functional_view_class */
                        (),
                        #60, /*      the      is_view_of      semantic
relationship:
                                the reference to the BSU of the described
                                class */
                        (#90, #100, #110), /* imported properties
                                              from the general model class */
                        ());

```

Figure 29: The functional model class dictionary definition

4.4. Functional model library specification: class extension

4.4.1. The view creation mechanism

The functional model library specification defines the template and the behaviour of the instances of a functional model class. It defines the properties that need to be valued in the context of an instance of such a class, and the methods associated with such an instance (the main role of such methods is to call the different programs or representations associated, as external files, to such a class). Its content is shown in **Figure 34**.

Figure 30 outlines the different steps performed by the library management system when the user requires a particular view for some SCREW instance already selected from the SCREW family.

These steps are the following.

- 1 - The end user asks for a particular geometry (2D, standard, front view) of a particular screw of the SCREW family.
- 2 - The two following processes are performed in parallel:
 - 2a - a message is sent to the LMS in order to determine what is the functional model class that must be instantiated, according to the previous determined instance of the general model together with the view required by the user, and
 - 2b - an instance of the functional view class is created, according to the view definition and view control variable values defined by the end user. Note that the view instance contains a representation attribute (an instance of a functional view is defined as a subtype of an ISO 10303 representation) intended to be "filled" by the functional model

instance through an implementation resource (here, the FORTRAN interface defined in [ISO 13584-31]).

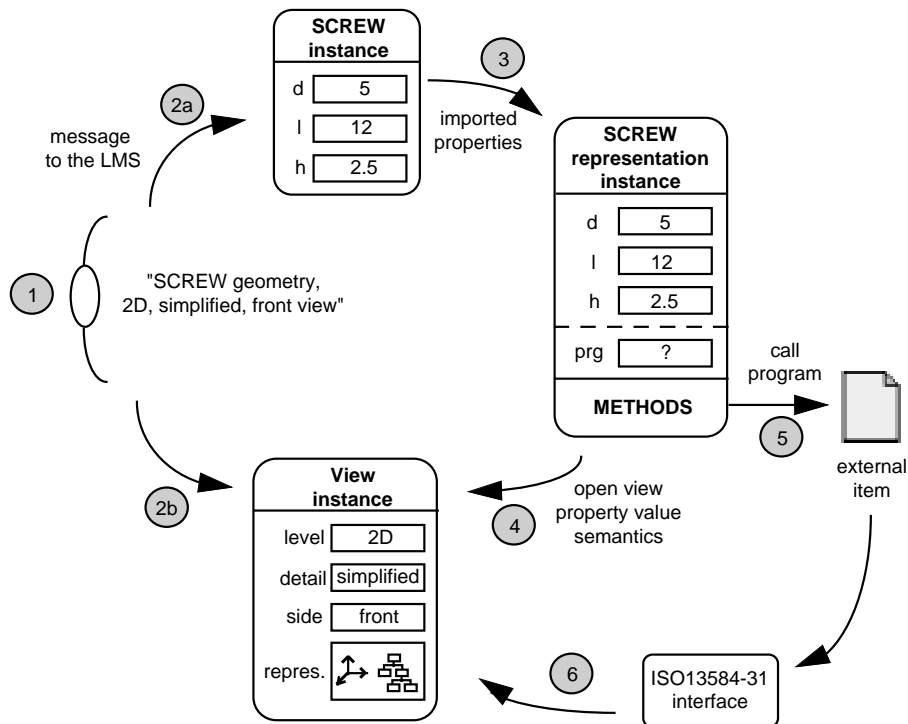


Figure 30: The view creation mechanism

3 - The functional model class instance is initialised by the LMS that assigns values for the properties declared as required properties (the imported ones) from the SCREW screw (the nominal diameter d , the length l , the head size h). This instance contains as slots only those properties declared in the functional model class extension (we assume here, that only d , l , h , and prg are so-declared, see Figure 34). Then, the relevant instance method is triggered. Note that during the running of the method, the functional model class instance is associated with the instance of the functional view class. Then, the prg property needs to be evaluated, through a derivation table, from the $side$ property. Therefore, the functional model class instance needs to know the value of the $side$ view control variable defined in the instance of the functional view class in order to compute the right prg program (through the methods).

4 - Through an open view property value semantics, the method (described in the next section) accesses to the geometry view to get the $side$ value of this functional view class instance.

5 - The prg attribute is computed (through a derivation table whose key is the $side$ property) and permits to call the relevant program stored as an external file.

6 - Through the ISO 13584-31 implementation resource, the (external) programs fill the representation attribute of the open view.

4.4.2. Methods description

In PLib, methods describe the dynamic behaviour of functional model class instances. In our example, a method is needed to get the value of the *side* defined in the context of the functional view class (the view required by the end-user, step 4 of **Figure 30**) and to trigger the right (externally defined) program (step 5 in **Figure 30**).

A method is defined by its specifications that specify the view created by the method and the functional model class instance properties used in the method (the imported properties from the general model class for instance), and by its body that specifies in a procedural way the list of the statements that shall be performed.

```
#3000 = method(  
    #3001,      /* the specification of the method */  
    #3002,      /* the body of the method */  
    #7); /* the used program interface: here, ISO13584-31*/
```

Figure 31: The method definition

A method also defines the interface referenced for filling the representation attribute of the current open view (the instance of the functional view class) that will be used (e.g., the ISO DIS 13584-31 Geometric Programming Interface, a CAD system specific interface, an Application Protocol from ISO 10303, ...).

The method specification contains the following attributes: the reference to a particular functional view class, the supported view control variables ranges and the model needed properties.

```
#3001 = method_specif (  
    #140, /* reference to the BSU of the created  
           functional_view_class */  
    (#155, #165, #175), /* the supported v_c_v ranges:  
                           sames as Figure 27*/  
    (#90, #100, #110), /* the properties from the  
                           functional model whose values are needed:  
                           d, l, h*/  
    ());
```

Figure 32: The method specification

The method body declares the variables accessed from the method (context) and describes the different (guarded) statements that have to be performed: in our example, a program call (the program corresponding to the selected side), this program being stored in an external file. It is represented by the following physical file:

```

#3002 = method_body (
    (#97, #107, #117, #177, #187),    /*the variables
        of the method context: d, l, h, prg, side
        (see Figure 13*/
    (#3020)); /*the statements of the method to be
        performed*/
#3020 = guarded_statement ((#3021, #3023));
#3021 = boolean_literal (.T.);
#3023 = call_program_statement (#187, /*the method variable (prg)*/
    #2401, /* the referenced program to be
executed
        and defined by a table derivation
        (see Figure 17)*/
    (#97, #107, #117), /*the input parameters*/
    (), ());

```

Figure 33: The method body

4.4.3. The functional model class specification

The functional model class extension defines the template and the behaviour of the functional model class instances. It references the external files used (the referenced programs), the required interface (ISO 13584-31), the part characteristics (*d, l, h*), method variables (the *side* and *prg* properties) and the methods it supports (defined above).

```

#1300 = functional_model_class_extension (
    #130, /* reference to the BSU */
    *, *, *, *, (), (), (),
    (#2401), /* the derivation table that computes
        the program from the side */
    (),
    (#2303,..., #2308), /* the referenced programs */
    (),
    (#7), /* the used program interface */
    (#11, #12), /* the used view exchange protocol */
    '001', '001', $,
    (#900, #1000, #1100), /* the required item
        characteristics: d, l, h*/
    (), (),
    (#1800), /* the method variable (prg) */
    (),
    (#3000), /* the associated method */
    $, $, (), $);

```

Figure 34: The functional model class library description

This last figure achieves the description content of a digital library describing our SCREW family with its geometric representation.

Conclusion

This case study has permitted to introduce and to illustrate the main concepts and resource constructs of the PLib information model. It has shown how the information is organised and outlined, and how this information may be accessed on the end-user site to support a user-friendly dialogue. More precisely, the following concepts have been introduced:

- the capability to separate the definition of the concepts (dictionary definitions that may exist alone, e.g., for standard dictionaries),

- the capability to reference any concept using its only BSU, whether or not its dictionary definition and/or library specification are present in the same exchange context ("BSU mechanism"),

- the complete but implicit description of all the parts of a parts family together with their intrinsic and permanent characteristics (general model class),

- the complete description of parts representations,

- the difference between a discipline-specific representation of any parts (functional view class) and a particular discipline-specific representation of a particular family (functional model class),

- the capability for multi-view descriptions,

- and last, the sharing of information between the general and the functional models.

This case study has therefore shown how it was possible to capture the knowledge on parts, e.g., coming from a paper catalogue, and to supply them in an intelligent way.

The modelling aspect of parts library still a little bit complex. However, the methodology that has been developed in the PLUS project gives some rules, guidelines and a representation formalism of a high abstraction level, that enables a parts supplier to model its own parts library.

A number of other concepts were not illustrated in this case study. They include:

- the representation of context parameter and context dependent characteristics that enable to select a part from the design context and to model the part behaviour,

- the assembly modelling,

- the use of STEP-compliant representation for modelling (as external files) one discipline-specific representation of each part family.

Nevertheless, the use of these concepts, and their representation through the PLib resource constructs, would not change the global structure of a PLib compliant digital library as presented in this case study.

On the other hand, the present case study was mainly oriented towards computer experts and software developers. For parts supplier, a completely different kind of representation proves necessary. Indeed, suppliers don't care about EXPRESS resource constructs or physical file instances. But they need to be able to design their own digital library to ensure its user-

friendliness. Such a representation using different notation [APS 95a] was also developed in the ESPRIT PLUS project. It is available on the Web [EPS+ 96].

References

- [APS 95a] AIT-AMEUR, Y., PIERRA, G., SARDET E., "Extending the Modelling Power of the EXPRESS language by Formal Annotations", Proc. of the 3rd Interna. Conf. of EXPRESS User Group EUG'95, Grenoble 21-22 Oct. 1995.
Available on our anonymous ftp server [ftp.lisi.univ-poitiers.fr](ftp://ftp.lisi.univ-poitiers.fr/pub/documents/papers/95/Formal_Annotations.ps.gz) ,
[/pub/documents/papers/95/Formal_Annotations.ps.gz](ftp://ftp.lisi.univ-poitiers.fr/pub/documents/papers/95/Formal_Annotations.ps.gz) .
- [APS 95b] AIT-AMEUR, Y.,PIERRA, G., SARDET E., "Using the EXPRESS language for Metaprogramming", Proc. of the 3rd Interna. Conf. of EXPRESS User Group EUG'95, Grenoble21-22 Oct. 1995.
Available on our anonymous ftp server [ftp.lisi.univ-poitiers.fr](ftp://ftp.lisi.univ-poitiers.fr/pub/documents/papers/95/MetaProgramming_in_EXPRESS.ps.gz),
[/pub/documents/papers/95/MetaProgramming_in_EXPRESS.ps.gz](ftp://ftp.lisi.univ-poitiers.fr/pub/documents/papers/95/MetaProgramming_in_EXPRESS.ps.gz) .
- [EPS+ 96] Elu, P., Pierra, G., Sardet, E., Kasan, D., Neumaier, A., "Design of a PLib-compliant Intelligent Electronic Catalogue - Supplier Oriented Methodology", part of WP3, Task 31 of ESPRIT #8984 PLUS, 1996.
Available on our anonymous ftp server [ftp.lisi.univ-poitiers.fr](ftp://ftp.lisi.univ-poitiers.fr/pub/PLUS/D13_22_PLUS_PLIB_METHOD/) ,
[/pub/PLUS/D13_22_PLUS_PLIB_METHOD/](ftp://ftp.lisi.univ-poitiers.fr/pub/PLUS/D13_22_PLUS_PLIB_METHOD/) directory.
- [IEC 1360-2] IEC FDIS 1360-2: Standard data element types with associated classification scheme for electric components, Express dictionary schema, F. Th. A., van Noesel, 1996.
- [ISO10303-11] ISO DIS 13584-11: Industrial Automation Systems and Integration, Product Data Representation and Exchange, Spiby, P., Schenck, D., The EXPRESS Language Reference Manual, 1995.
- [ISO130303-21] ISO IS 10303-21: Industrial Automation Systems and Integration, Product Data Representation and Exchange, Implementation methods: clear text encoding of the exchange structure.
- [ISO13584-20] ISO DIS 13584-20: Industrial Automation Systems and Integration, Parts Library, General resources, Ait-Ameur, Y, Wiedmer Hans U., Eds, ISO, Geneve, 1996.
- [ISO13584-24] ISO CD 13584-24: Industrial Automation Systems and Integration, Parts Library, Logical Model of Supplier Library, PIERRA, G., AIT-AMEUR, Y., SARDET, E, Eds, ISO, Geneve, 1996.
- [ISO13584-31] ISO DIS 13584-31: Industrial Automation Systems and Integration, Parts Library, .Geometric Programming Interface, Heine, Lutz-R, Eds, ISO, Geneve, 1996.

- [ISO13584-42] ISO DIS 13584-42: Industrial Automation Systems and Integration, Parts Library, Methodology for Structuring Parts Families, PIERRA, G., WIEDMER, H.U., Eds, ISO, Geneve, 1996.
- [Pierra 95] PIERRA, G., "Modelling classes of pre-existing components in a CIM perspective: the ISO13584/ENV 40014 Approach", *Revue internationale de CFAO et d'Infographie*, vol. 9, n°3, 1994 , pp. 435-454.
- [Pierra 93] PIERRA, G., "A Multiple Perspective Object Oriented Model for Engineering Design" , in: *New Advances in Computer Aided Design & Computer Graphics*, X. Zhang, Ed., International Academic Publishers, Beijing, China, 1993, pp. 368-373.
Available on our anonymous ftp server [ftp.lisi.univ-poitiers.fr](ftp://ftp.lisi.univ-poitiers.fr/pub/documents/papers/93/Object_Model_for_Engin.ps.gz) ,
[/pub/documents/papers/93/Object_Model_for_Engin.ps.gz](ftp://ftp.lisi.univ-poitiers.fr/pub/documents/papers/93/Object_Model_for_Engin.ps.gz)
- [PPG 94] PIERRA, G., Potier J.C., Girard P., "The EBP system: Example Based Programming for parametric design", Workshop on Graphic and Modelling in Science and Technology, Coimbra, 27-28 june 1994 , in: Springer Verlag Series, 1996.
Available on our anonymous ftp server [ftp.lisi.univ-poitiers.fr](ftp://ftp.lisi.univ-poitiers.fr/pub/documents/papers/94/Coimbra94.ps.gz) ,
[/pub/documents/papers/94/Coimbra94.ps.gz](ftp://ftp.lisi.univ-poitiers.fr/pub/documents/papers/94/Coimbra94.ps.gz)