Journal for Geometry and Graphics
Volume 3 (1999), No. 1, 17–38

A Data Model Architecture for Parametrics

Dago Agbodan, David Marcheix, Guy Pierra

Laboratory of Applied Computer Science (LISI),
National School of Engineers in Mechanics and Aeronautics (ENSMA)
Téléport 2 — Avenue 1, BP 109 Chasseneuil du Poitou
86960 Futuroscope cedex, France
emails: agbodan, marcheix, pierra@ensma.fr

Abstract. In recent years, history-based, constraint-based and feature-based CAD systems (often gathered under the generic name of parametrics), appeared as a major progress both to express and to capture conceptual designs and design intents. This deployment raise two major issues. The first one is to define a data model that provides for exchange capabilities between heterogeneous CAD systems and for archiving. The second one is the well known "topological naming" problem.

The goal of this paper is to propose an unified modeling framework for parametric data, that addresses these two issues. This framework, defined in the object-flavored EXPRESS data specification language, involves a three layers architecture. Gathering the complete definition of a parametric object in the same data model permits both to simplify the data-management, and to define a neutral description of parametrics, enabling exchange between heterogeneous CAD systems.

Key Words: CAD/CAM, geometric modeling, parametrics, constraint-based model, formal specification

MSC 1994: 68U07

1. Introduction

In recent years, history-based, constraint-based and feature-based modelers (often gathered under the generic name of parametrics), appeared as a major progress both to express and to capture conceptual designs and design intents. Nowadays, most commercial CAD systems support some of these capabilities and several research prototypes contribute to extend these capabilities or to improve their reliability.

The very specific feature of the parametrics technology is that their data structure is two-fold. On the one hand, they record, as a snapshot, the geometric shape of the currently designed product, often as a B-rep model. We call this representation the *current instance*.

ISSN 1433-8157/\$ 2.50

On the other hand, they record the conceptual design of which the current instance results, its *parametric specification*, which consists of constraints represented as expressions that reference the current instance and involve various kinds of operators. The current instance and the parametric specification being expressed apart from each other, an issue is to handle a complex data-storage management.

Another major issue consists in modeling the relationships between the references as they appear in the parametric specifications and the values as they appear in the current instance. Parametric specifications always include some ordered processes where geometric and topological entities have been modified. The current instance only contains a snapshot of the result of this process. Therefore the parametric references cannot be mapped one-to-one with the current instance. The reference mechanism shall therefore abstract from the low-level geometric or topological entities (that may, or not, exist) to only reference the "invariants" of the parametric object.

The goal of this paper is to propose a data model architecture that may be used to address these two issues. This architecture is exemplified using the EXPRESS data specification language developed by Schenck and Wilson [18]. But the same approach may be used in any object oriented data base environment. To represent in the same framework the values (of the current instance), the variables to which they correspond and the expressions where these variables are involved, we propose to capture in the data model the abstract syntax tree of these expressions and to use the entity relationship approach to model the usual interpretation function that associates values to variables. To separate the references intended to be used in the parametric specification and the physical entities that belong to the current instance, we propose a three layers architecture where a reference layer, called the *dynamic context*, enables to capture and to reference the abstract entities involved in the parametric specification whatever be their physical representation in the snapshot of the current instance. In this model, the current instance is represented using the EXPRESS resources defined in ISO 10303 (STEP) for modeling explicit geometry. Our architecture may therefore be used to extend the expressive power of ISO 10303 towards parametric geometry modeling.

This paper is structured as follows. In section 2, we expose the major issues about parametric data model exchange: data and expression management, entity naming and name matching. The third section discusses some pre-existing works addressing those issues and the fourth section enumerates the principles of our data model. In section five, we briefly outline the main features of the EXPRESS specification language that is used in the remaining part of the paper. In section six, we first refer to an already proposed taxonomy of parametric data models [14] and we propose a structure that may support the different kinds of parametric geometry, whether they follow a functional, a variational, or an hybrid approach. Then, we discuss the relationships between a parametric reference, as it appears in a parametric specification, and the geometric entity(ies) to which it corresponds. Borrowing to programming language compiling theory, we propose to associate to a parametric specification a context that contains all the references used in this specification. But, unlike in compiling, where such a context is built once and for all at compile time, this context is intended to be dynamically created during the design process. The role of this context is to model the names which abstract the geometry as it is at any stage of this process and therefore provide for referencing entities that disappear in a latter stage of the design process. Section seven, shows how this context may be used both to structure the names of all the geometric entities that may result from one unique constructive gesture and to provide an unambiguous name for entities resulting from some collisions. In the last section, a model for representing variables and expressions is proposed. Throughout this paper, we mainly use a simple 2D example to illustrate the different parts of the data model and we represent the internal data structure of the data base using the exchange format associated to the EXPRESS language. The complete example is presented in appendix A.

2. Major issues

Exchanging the two-fold structure of a parametric model raise three major issues. The first one is to define a data model that provides for the simultaneous exchange of expressions, of variables involved in these expressions and of values of these variable. The two other issues, known as the topological naming problem [9], are to assign persistent names to geometric and topological entities that may not exist or that may cut into several pieces in the geometry of the current instance. This topological naming issue hides in fact two different problems: the entity naming problem (at design process) and the name matching problem (after re-evaluation).

2.1. Expressions, variables and values

The parametric specification of a parametric data model contain algebraic expressions that involve variables and various kinds of algebraic operators (equational systems for 2D, boolean operations, arithmetic operations, ...). The current instance involves values of these variables. Therefore, exchanging a parametric data model needs to represent in the same framework the values (of the current instance), the variables to which they correspond and the expressions where these variables are involved. HOFFMANN and JUAN [4] suggest to represent the expressions as a string whose syntax may be "adopted from FORTRAN or some other programming language". In this paper, we propose to use the meta-programming approach [1]. We capture in the data model the abstract syntax tree of expressions and we use the entity relationship approach to model the usual interpretation function that associates values to variables. Having the same information modeling structure for modeling numeric expression and for other parts of the model (constraints, geometry, ...) allows to assert and to check the consistency properties of the global exchange context.

2.2. Naming problems

The complete structure of a parametric data model gathers the geometric representation of the current instance, and a composition of constructive functions (or a set of constraints in an equality-based model). The simplest way to connect these two layers is to enable the constraints to directly reference the geometric entities that constitute the current instance. This approach has already been proposed [14] for functional parametric models where only CSG entities were referenced. Unfortunately, this approach may no longer be followed when the constructive gesture involve references to B-Rep entities as shown in Fig. 1.

In the example illustrated in Fig. 1 the initial model is designed by means of four successive constructive gestures. The fourth one consists of rounding edge "e". If the initial model is exchanged after this fourth step, the current instance no longer contain edge "e": it was removed by the rounding function. Thus the function "round (e)" which has the edge "e" as input parameter cannot any longer be represented in the parametric specification part of the model. Therefore "names" are needed to represent the entities referenced in the parametric

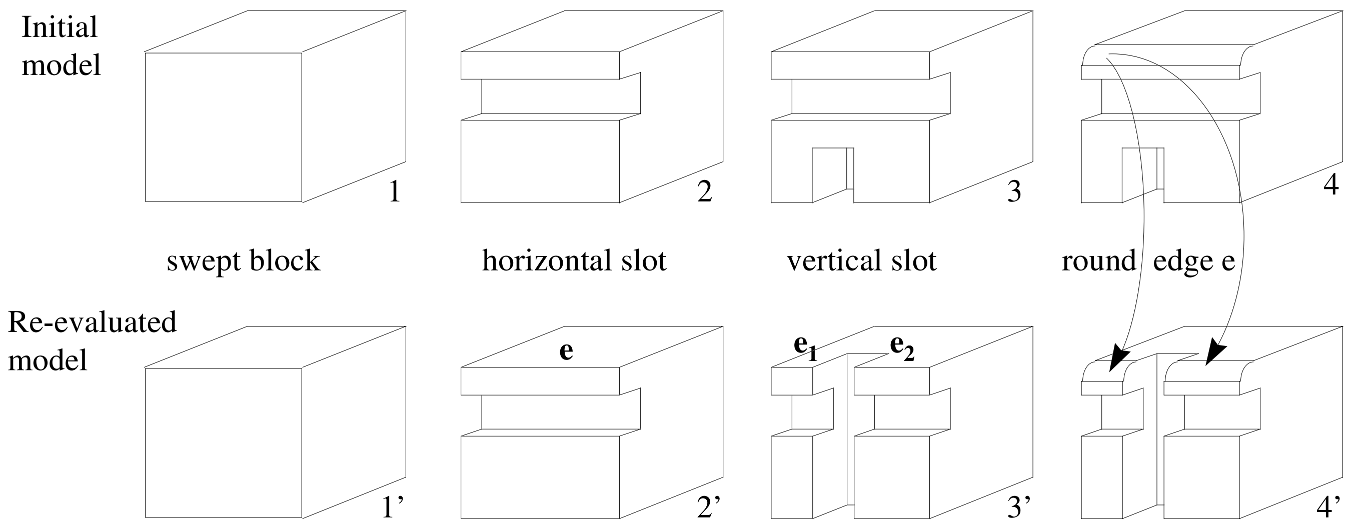


Figure 1: Naming and name matching problems.

specification whether or not they exist in the model snapshot. Moreover each constructive gesture creates several entities which have to be distinguished and therefore named, even if all entities exist in the model snapshot

These names shall be defined in such a way that, when the parametric specification is used to generate a new geometric model, the geometric entity referenced by a name in the re-evaluated model is "the same" as the one referenced by the same name in the initial model. To define such robust names, two different kinds of geometric and topological entities may be distinguished.

2.2.1. Invariant entities

An invariant entity is a geometric or topological entity which can be, completely and unambiguously, characterized by the structure of a constructive gesture and its input parameters, independently of involved values.

In Fig. 1, invariant entities includes the end face of the swept block, the lateral shell of the horizontal slot with its begin and end faces (that may, or not exist), the face resulting from the rounding gesture, etc.. To characterize, i.e., to "name", such entities, information models are to be defined that relate these entities to constructive gestures and to their input parameters.

2.2.2. Contingent entities

Beside those invariant entities, there exist entities that depend on the context of a constructive gesture. We call contingent entity a geometric or topological entity that results from an interaction between the pre-existing geometric model and invariant entities resulting from a particular constructive gesture. For example, in Fig. 1, the number of lateral faces of the vertical slot in the initial model (step 3) and in the re-evaluated model (step 3') is not identical. A naming mechanism is also required to define how to name these contingent entities.

2.3. Name matching

If the topology of the current instance do not change when the parametric specification is re-evaluated, the only issue is name robustness: to identify which entity in the re-evaluated model correspond to every entity in the initial model. When re-evaluation leads to topology changes a new issue is to match two different structures. For example let us come back to the model presented in Fig. 1. At step 3' edge "e" has been split into edges "e1" and "e2". Thus at step 4' the problem is to determine which edge(s) has(ve) to be rounded. The problem is to identify, i.e., to match, edge "e" with edges "e1" and "e2" despite the different topology.

Note that the matching mechanism is system specific, but the data model shall contain enough information enabling to run a matching algorithm.

3. Related work

Following the pioneer work of Hoffmann and Juan [4], over the last few years several authors have analyzed the internal structure of parametric data models, proposing some editable representations [4, 14, 19, 15, 10], discussing their underlying mathematical structures [14] and proposing some naming scheme or mechanisms [9, 2]. Recently, several mechanisms for persistent naming have been proposed. Most of the formats proposed to capture the parametric specification are based on the programming language paradigm: either through specific languages [4, 19], or through existing ones [10].

Our intend being to use a data model oriented approach for both the current instance and the parametric specification, we just outline below two previous works that addressed the naming issues and that we re-used in our own approach. The first one focuses on naming, the second one focuses on name matching.

3.1. Chen

Chen [2] uses an editable representation, called Erep [4], which is an unevaluated, high-level, generative, textual representation, independent of any underlying core modeler, to abstract the design operation and to name all entities. Chen defines a precise structure for invariant entity naming, particularly, for sweep operation. Every entity in a sweep is named by reference to the corresponding source entity of the swept 2D contour and the constructive gesture. He also proposes a fine identification technique for contingent entities based on topological adjacencies and feature orientation. In the Chen approach every contingent entity is named. Unambiguous names are generated by composition of topological adjacencies. No mechanism is defined to handle name matching.

In our approach only the contingent entities that are referenced by some latter operations are associated with a name. We use a similar approach to Chen's one for naming the invariant faces that result from a sweep.

3.2. Kripac

Kripac [9] focuses on the name matching. He proposes an interesting structure for identification of contingent entities based on face history (creations, splits, merges and deletions of faces) and a complex name matching algorithm.

Kripac's Topological ID System consists of 3 parts. First a face graph structure allowing both naming of all entities (edges and vertices are named in terms of their adjacent faces) and

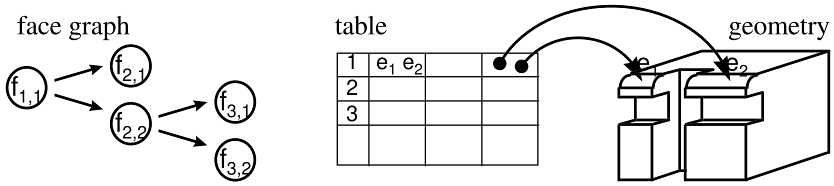


Figure 2: KRIPAC's Topological ID System.

name matching after re-evaluation. Second a table recording names for the only contingent entities that are referenced together with pointers on the geometry more some informations for the matching algorithm. Third the geometry of the designed object. At each re-evaluation the old entities are matched with the new ones.

3.3. Current limitation

A first limitation, addressed in this paper, is that KRIPAC describes a prototype system, not an exchange format. A second limitation is that the KRIPAC matching algorithm includes a lot of heuristics and its domain of correctness is not clearly defined. An issue is to formalize entity naming and name matching mechanisms. RAGHOTAMA and SHAPIRO [16] have given a first answer to this issue.

4. Principle of the proposed data model

We summarize below the design principles used in the exchange model we propose.

1. Use of EXPRESS [18] for information modeling.
2. Any reference between parametric specification and geometric model is done by a name level (three levels model).
3. Names are strongly typed: the name involved either as input or as output of each parametric specification stand for entity whose data type are strictly defined.
4. Only those names that correspond to entities referenced in the parametric specification shall be represented in the name level, other may or not be represented.
5. Invariant entities are identified from the structure of constructive gesture and the structure of its parameters independently of their values.
6. Contingent entities are identified trough their direct or indirect relationship with invariant entities.
7. Variables and numeric expressions are represented in the data model by meta-programming.
8. The re-evaluation of geometry in case of change in topology is not specified : it is the specificity of each CAD system.

5. The EXPRESS language

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.

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 [18]. 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 for a meta-programming purpose [1].

Following [5], an EXPRESS specification is defined by a set of entities (ENTITY) which represent the objects to be modeled. 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 object oriented languages. This part of the specification defines the structure of the data model.

Unlike most of the data modeling 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 on our parametric data model to ensure that such a model is consistent. At last, entities and functions are gathered in a common structure named SCHEMA that provides for modularity. The following example shows the main features that can be found in an EXPRESS specification.

```
SCHEMA Example ;

     ENTITY A
         ABSTRACT SUPERTYPE OF (B, C) ;
             att1_a : REAL ;
     END_ENTITY ;

     ENTITY B
         SUBTYPE  OF (A) ;
             att2_b : LIST [0:?] OF STRING ;
     DERIVE
         SELF\A.att1_a : INTEGER := 0 ;
         att3_b : INTEGER := SIZEOF (SELF.att2_b) ;
     INVERSE
         att4_b : C FOR att4_c ;
     END_ENTITY ;

     ENTITY C
         SUBTYPE OF (A) ;
             att2_c : SET [0:?] OF REAL ;
             att3_c : OPTIONAL INTEGER ;
             att4_c : B ;
         WHERE
             WR1 : f (SELF) ;
             WR2 : att4_c.att3_b = SIZEOF (SELF.att2_c) ;
     END_ENTITY ;
```

```
    FUNCTION f (x : C) : BOOLEAN ;
        LOCAL res : REAL := 0.0 ; END_LOCAL ;
        REPEAT i := 1 TO SIZEOF (x. att2_c) ;
            res := res + x.att2_c[i] ;
        END_REPEAT ;
        RETURN (res = x\A.att1_a) ;
    END_FUNCTION ;

END_SCHEMA ;
```

Figure 3: An EXPRESS schema example.

The previous schema example 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 ABSTRACT SUPERTYPE indicates that A is an abstract class and thus does not have instances. The following notations have been used:

- . is the dot notation allowing to access the entity attributes.
- \ character allows to unambiguously reference an inherited attribute (it is used here for illustration purpose).
- DERIVE indicates that the attribute value is computed by evaluating an expression whose domain generally consists of other attribute values (some built-in EXPRESS function enable to reference the complete set of entity instances that belongs to the data model). Some inherited attributes may be derived (example of att1_a in B).
- INVERSE 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 that relationship in the context of the entity B.
- OPTIONAL keyword indicates that, in a given instance, the attribute needs not to have a value.
- SIZEOF is one of the many EXPRESS built-in functions. It gives the length of any aggregate data type (list, set, bag, ...).
- SELF is an EXPRESS keyword for a variable representing the current entity.
- WHERE introduces the WHERE clause which constraints the data corresponding to each instance of an entity data type. It can be built by EXPRESS expressions as in the WR2, or by an externally defined function as in WR1 where the function f is defined as a boolean function.

To make easier the understanding or the design of the structure of such a schema, it can be described (as shown in Fig. 4) graphically using the EXPRESS-G symbolism [5]. EXPRESS-G is a graphical notation for the display of a data specifications defined in the EXPRESS language. This notation only supports the structural part of an EXPRESS model. The constraint part needs to be stated textually.

Instances of a model defined in EXPRESS can be described and exchanged through physical files. Their format is defined in [6] 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 (as an abstract supertype, A has no instance):

```
#1 = B ( 33,              /* the first attribute of entity B, inherited from A,
                          is redefined as an integer */
         *,               /* the asterisk represents a derived attribute as
                          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 */
```

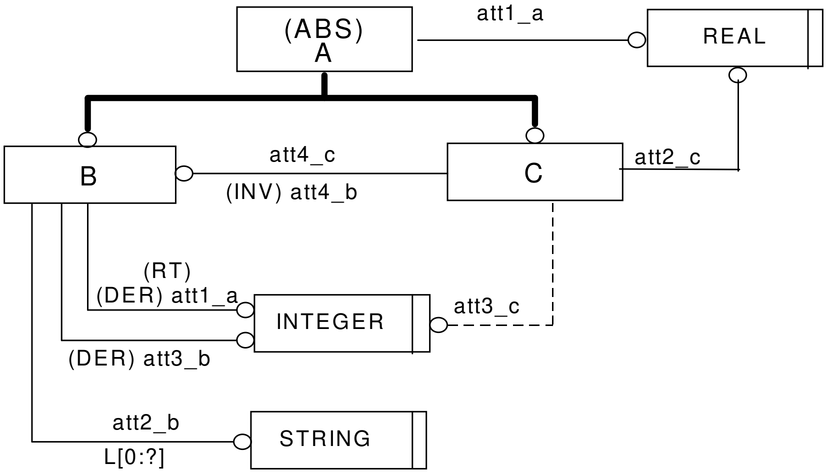


Figure 4: EXPRESS-G representation of the schema example (Fig. 3).

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.

The goal of this paper is to propose a data model architecture. The definition of a data model requires to use a data modeling language. EXPRESS is such a language, that we chose for several reasons. They can be summarized in the following six points:

- it is an international standard [5], extensively used in the CAD area (STEP),
- it has a complete specification, allowing, with appropriate tools, nearly a one-to-one map to an object oriented language like C++ or Java,
- it can be represented either textually or graphically (EXPRESS-G); this capability provides two levels of abstraction for designing data model,
- through ISO 10303-21 [6], an EXPRESS model automatically defines an exchange format for instances of such a model,
- thanks to its modularity, several standard resources schemas (for geometry, for expressions, etc.) that can be re-used already exists and,
- last but not least, it is used in STEP for the exchange of explicit geometry; an exchange format for parametric geometry should be compatible with this standard.

Next sections make a large use of the EXPRESS language. This section has presented the kernel of EXPRESS which is enough to understand the basic constructs presented in those sections.

6. Data model structure

As discussed in [14], parametric models can be classified according to their underlying mathematical structure, into two major approaches. First, the equality-based approach, also referred to as variational geometry [12], captures a parametric specification as a set of non-oriented constraints between geometric entities. This set of constraints is translated into a set of equations. Second, the functional approach, also referred to as constructive approach [17], captures each constraint as a function, and the whole model as a composition of functions. Due to the intrinsic weakness of each approach [14], few commercial products are restricted to only one of these approaches. Although all the known 3D systems follow a functional approach for 3D shape design, most of them support equality-based parametric definitions for 2D-contours, and some of them support equality-based parametric positioning of 3D shapes [11].

6.1. Parametric specification

From a data modeling point of view, this means that no parametric systems use *only* non-oriented constraints. Therefore, the generic structure of a parametric data model may be defined as a set of constraints, each constraint requiring that some already existing entities are available (modeled in Fig. 5 through the *assumed* attribute of the *constraint* entity) and constraining a set of other entities (*defined* attribute). If different entities are involved in the *assumed* or *defined* attribute of some constraints, their roles are, in general, non identical. Therefore the *assumed* and the *defined* attributes shall correspond to a list (that may be empty for the *assumed* list).

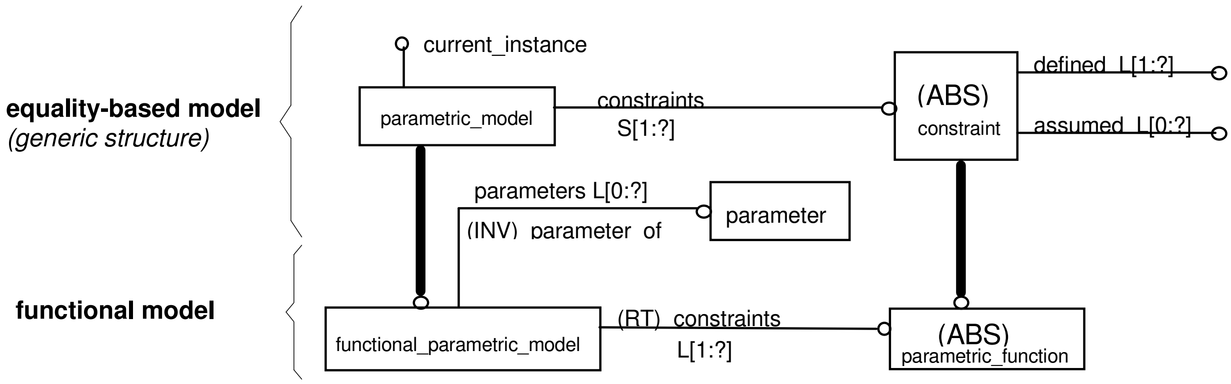


Figure 5: EXPRESS data model of a parametric specification.

As shown in the Fig. 5 using the EXPRESS-G symbolism, a functional parametric data model is a subtype of such a generic structure. For this subtype:

1. each constraint is a function, and
2. the constraints are ordered (composition of functions), and
3. some variables, clearly identified, define the domain (*parameters*) of the global parametric function.

The model shown in Fig. 5, thanks to its double-subtyping, allows to model the three approaches to parametrics (pure equality-based approach, functional approach or hybrid models as they exist in commercial systems). First a geometric representation can be equality-based by stating its model to be a *parametric_model* whose attribute *constraints* is restricted to be a set composed only of *constraint*(s) (i.e., no *parametric_function*). This model may be used, for example, for 2D variational contours. The second possibility for a parametric object is to be functional. This is done by specifying that its model is a *functional_parametric_model* where the *constraints* attribute is derived to be a list of *parametric_function*(s). Such a functional model can be used both in 2D or in 3D for, e.g., recording a design process. The last possibility is to have an hybrid geometric representation which is obtained by specifying its model to be a *parametric_model* and its *constraints* to be a set of *constraint*(s), some of them being *parametric_function*(s). For example, equality-based 2D profiles and (functional) 3D extrusion of these profiles may be described with such an hybrid model. In our proposed data model, both *constraint*(s) and *parametric_function*(s), which in fact constitute the parametric specification layer, are defined as abstract supertypes. They shall be specialized for each specific constraint or parametric function.

This data model only proposes a structure for the parametric specification part of a parametric model. The relationship between this parametric specification and the current instance is discussed in the next section.

6.2. Naming level

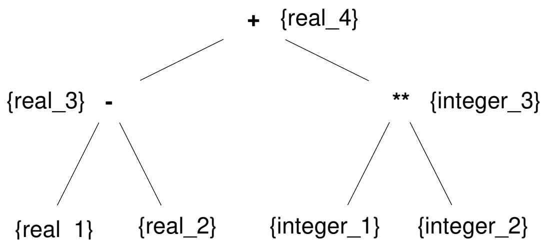
As illustrated in section 2.2, the direct link between parametric specification (constraints) and current instance (geometric shape) compromise the support of entities modification (round in this example). Therefore no direct reference shall take place.

6.2.1. Concept of dynamic context

Such a problem is well known in programming language from which we propose to borrow the concept of context. In traditional programming, a program does not directly reference variables by their values. It contains the names of the variables. The association name/value is done by a symbol table, called a *context*. The role of the context, which is built at compile time, is to ensure the indirect link between the (variant) example values and the (invariant) program variables. Such a name/context/value association allows a program to unambiguously reference a variable whatever be its current value.

Unfortunately, a parametric model has no declarative clause that may be used, like in programming language, to create persistent names of the geometric or topological entities involved in the design. This problem is similar with the one encountered in example-based programming [13], also called programming-by-demonstration [3], where the user build an example of a program, and the programming-by-demonstration manager is supposed to abstract variable names from the example values. In such environment, the concept of context may also be used as proposed in [14]. The only difference is that this context is not built once and for all, at compile time, but is built *dynamically* throughout the design of the example. Fig. 6 shows an example of the use of a dynamic context to abstract from an (example) expression onto a procedural program. Assume that a user uses a display calculator to input the expression ((11.0 - 4.5) + (3**2)). For each input value, a new entry is created in the dynamic context of which the data type is defined by the example. The program (tree) references these entries.

15.5	11.0	real_1
9	4.5	real_2
6.5	6.5	real_3
2	15.5	real_4
3	3	integer_1
4.5	2	integer_2
11.0	9	integer_3



Example Dynamic context Program

Figure 6: Dynamic context management in programming-by-demonstration [14].

This *dynamic context* mechanism already proposed for programming-by-demonstration systems seems to be well suited for recording a parametric data structure.

6.2.2. Parametric_reference

In the previous example (Fig. 6), the invariants to be represented in the program were numeric variable names. In parametric geometry, these invariants mainly consist of persistent names for geometric and topological entities. We still call this layer the *dynamic context* layer. This dynamic context, modeled by a *parametric_reference* entity, is an abstraction mechanism of the basic geometric entities. It is in fact a naming mechanism which characterizes a parametric definition independently of its value. As presented in Fig. 7 the *parametric_reference* entity shall be subtyped for each *representation_items* (i.e., points, curves, vertices, ...) intended to be referenced by a constraint.

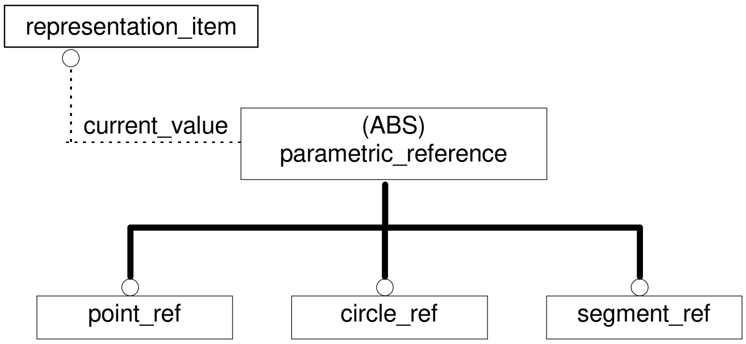


Figure 7: Simplified representation of the *parametric_reference*.

In the data model architecture we propose, all the constraints refer to *parametric_reference*(s) that (possibly) refer to geometric entities. So, if we come back to our rounded block example (Fig. 1), the constraint would be "round (e_ref)". This constraint is always valid because, even if the geometric entity (edge e) is deleted, its reference (e_ref) still exist. Now, our data model architecture for parametrics, as shown in Fig. 8, is three layered:

- the current instance (explicit geometry, STEP-compliant),
- the parametric specification (constraints),
- the dynamic context (link between both).

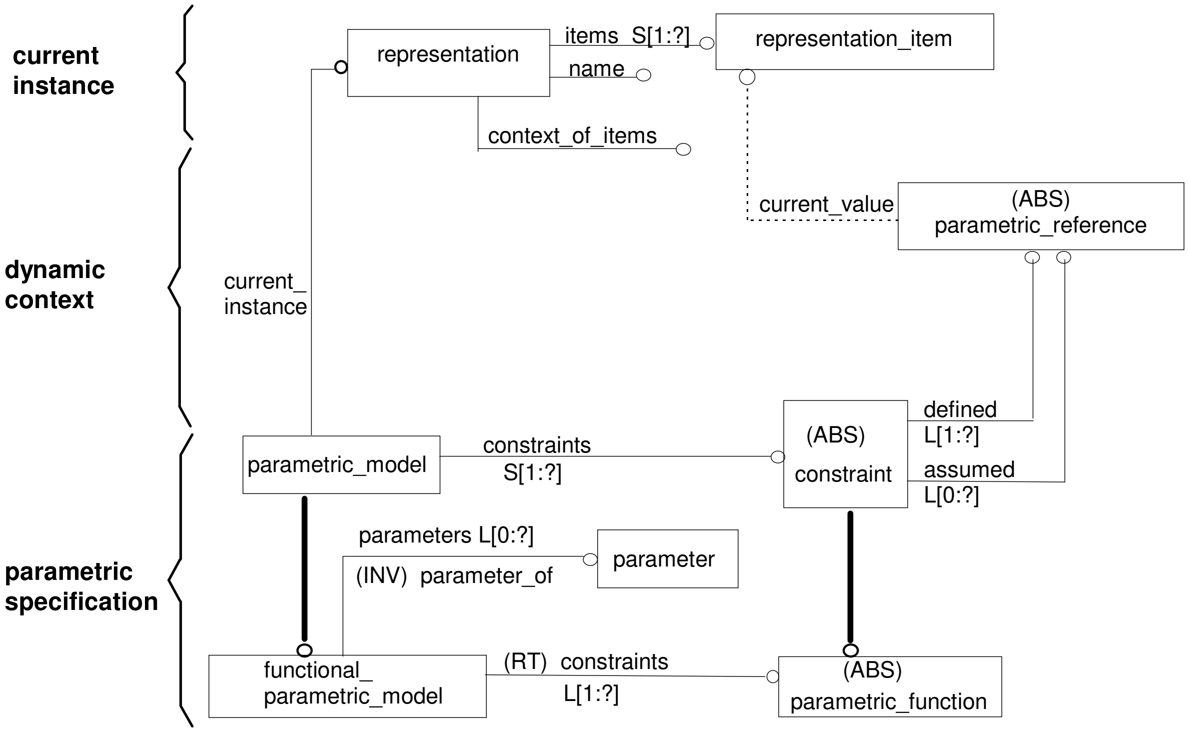


Figure 8: Simplified representation of a parametric data model.

An EXPRESS model being slightly abstract, in the following section we use a simple 2D example. This example is shown in Fig. 9, to illustrate the concepts of our data model. The corresponding example of physical file compliant with our parametric data model is presented in Fig. 10 (the whole physical file is presented in Appendix A). It shows clearly the three layers and how the link is made between the geometric representation items and the constraints (here functions) in which they are involved. The constraints refer to the name layer (dynamic context) which in turn refer to the geometric items. The geometric items are compliant with the STEP geometry. Note that canonical functions, are particular subtypes of *parametric_function*, where each attribute value is specified by means of an expression or an existing *parametric_reference*. As an example, a *canonical_cartesian_point_function* defines a *cartesian_point_ref* (DEFINED attribute) by means of two (in 2D) expressions. These expressions, in turn, may involve various *parametric_reference*(s). The *assumed* attribute is therefore re-defined as a derived attribute whose value is computed by an EXPRESS function. A *canonical_axis2_placement_2d_function* defines an *axis2_placement_2d_ref* (DEFINED attribute), from a *cartesian_point_ref* that constitutes its origin (ASSUMED attribute). Note that the *polyline*s indexed by #60 (D1) and #62 (D2) no more exist in the current model but their name indexed by #600 and #620 still exist.

In this section, the concept of dynamic context was just presented as a means to isolate the parametric specification from the low-level geometric entities referenced in the constraints. We will show in the next section how a *parametric_reference* may be used represent structured names.

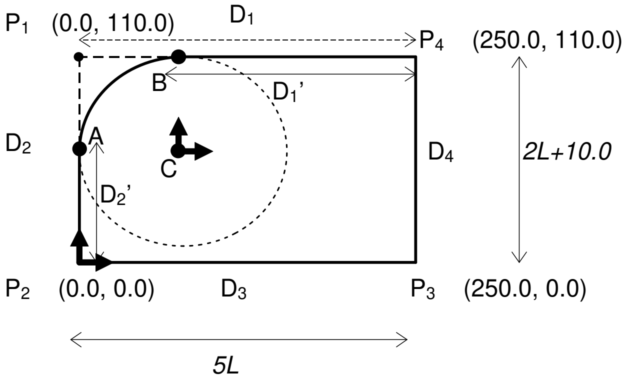


Figure 9: 2D example.

```
/* current instance layer */
    /* dynamic context layer */
        /* parametric specification layer */

#20 = cartesian_point ( 'P1', (0.0,110.0) );
    #200 = cartesian_point_ref ( #20, ( ) );
        #2000 = canonical_cartesian_point_function ( (#200), *, (#210,#220) );
    #210 = real_literal ( 0.0, * );
    #220 = ...     /* the expression 2L+10.0 will be described in
    section 9 */

#30 = cartesian_point ( 'P2', (0.0,0.0) );
    #300 = cartesian_point_ref ( #30, ( ) );
        #3000 = canonical_cartesian_point_function ( (#300), *,
        (#210,#210) );
...
#28 = axis2_placement_2d ( 'rectangle\_axis', #30, $, $ );
    #280 = axis2_placement_2d_ref ( #28, ( ), $, $ );
        #2800 = canonical_axis2_placement_2d_function ( (#280), *, (#300) );

#50 = cartesian_point ( 'P4', (250.0,110.0) );
    ...
#60 = polyline ( 'D1', (#20,#50) );                       /* does not exist */
    #600 = polyline_ref ( $, ( ), $ );
        #6000 = canonical_polyline_function ( #600, *, (#200,#500) );
#62 = polyline ( 'D2', (#20,#30) );                       /* does not exist */
    ...
```

Figure 10: Beginning of the exchange file.

7. Naming invariant entities

When a constructive gesture recorded as a parametric function, creates a single geometric entity (for instance in Fig. 10, function #2000 that created #20), this geometric entity is unambiguously characterized by a *parametric_reference* (#200) entity defined as the output of the parametric function. In fact, most of the parametric functions create an highly structured set of entities. For instance, inserting a slot in a B-rep block creates a structured set of new faces, edges and vertices.

7.1. Structuring the name

A *parametric_reference* may be used as a structuring mechanism and therefore allows functions to create more than one simple geometric object. Even in 2D, geometric items are, most of the time, structured entities, so that their creation involves other geometric or topological entities which are, if they don't already exist, automatically created. For example in a STEP-compliant 2D representation, a *circular_arc* relies on the construction of a basis circle and two trimming points. The circle, in turn, requires a reference *axis2_placement*, which, in turn, requires an origin *cartesian_point*. Note that, in such cases, one construction gesture created a structured entity of which every item is well known. Therefore its structure can be hard coded. This is a first case of the naming mechanism, where several created geometric or topological entities may be related unambiguously to an unique construction function.

An example where one function (*arc_fillet_2entities*) creates several geometric objects is presented below in Fig. 11. In this example, the function (#7900), rounds the corner between edges D1 (#600) and D2 (#620) from Fig. 9. The created *circular_arc* (#79), directly creates the references of the trimming points A and B (#730 and #740) and the reference of the basis circle (#720) which in turn creates the placement coordinate system reference (#710) which in turn creates the point reference (#700).

```
#70 = cartesian_point ( 'C', 40.0, 70.0 );
    #700 = cartesian_point_ref ( #70, ( ) );
#71 = axis2_placement_2d ( 'circle_axis', #70, $ );
    #710 = axis2_placement_2d_ref ( #71, ( ), #700, $ );

#72 = circle ( 'basis_circle', #71, 40.0 );
    #720 = circle_ref (#72, ( ), #710 );
#73 = cartesian_point ( 'A', 0.0, 70.0 );
    #730 = cartesian_point_ref ( #73, ( ) );
#74 = cartesian_point ( 'B', 40.0, 110.0 );
    #740 = cartesian_point_ref ( #74, ( ) );

#79 = circular_arc ( 'round', #72, (#73), (#74) );
    #790 = circular_arc_ref ( #79, ( ... ), #720, #730, #740 );
    /* the collision ( ... ) is discussed in section 8 */
        #7900 = arc_fillet_2entities ( #790, (#600, #620), *, #800);
    #800 = real_literal ( 40.0, * );
```

Figure 11: Illustration of the *parametric_reference* mechanism for the creation of structured invariants.

7.2. Referencing input parameters of constructive functions

The second mechanism to generate name (i.e., *parametric_reference*) of invariant entities consists in referencing input parameters of a constructive function. For example, in a sweep operation, large number of topological and geometrical entities are created. These entities may be categorized in two sets. First the invariant entities that exist for any sweep. These invariant entities are: an initial face, a final face and a lateral shell and they may be identified as defined in section 7.1. Second the invariant entities specific of this particular sweep that result from sweeping a particular item of the swept contour.

In our model we just use this capability to identify the different invariant faces of the lateral shell. Then, unlike in the CHEN approach, these faces are used to identify all the edge and vertex. For instance sweeping one edge of the swept contour defines a face (part of the lateral shell). Such an implicitly created element may be defined by (see Fig. 12 for an example) :

- the sweep constructive gesture,
- the "input" item of the sweep,
- the type of the result which may be either geometric (e.g. a surface) or topological (e.g. a face).

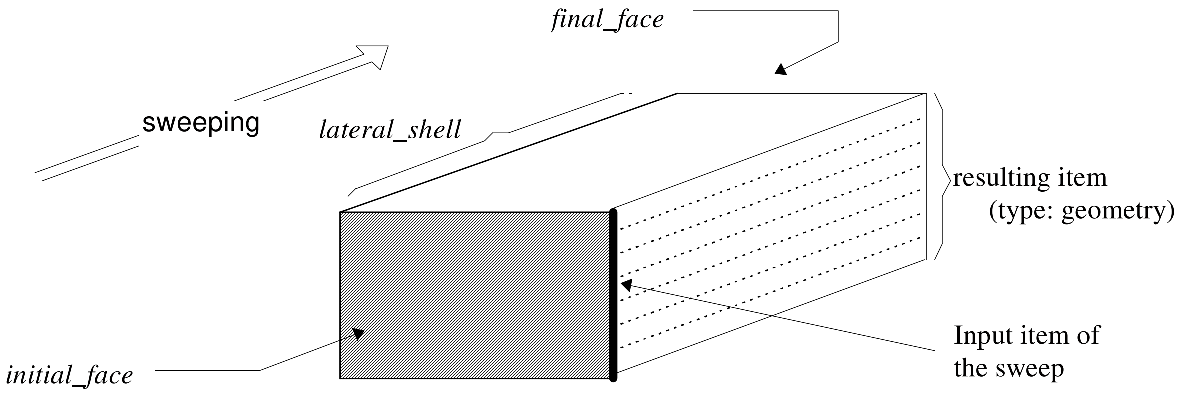


Figure 12: Characterization of a sweep.

In the previous examples, we have shown how *parametric_reference*(s) may be used to unambiguously identify those entities that systematically result from the creation of another entity. But there exist another kind of geometric entities in the current instance of a parametric data model, namely contingent entities which are entities that result from the interaction between a new invariant entity and the pre-existing geometric model where this entity is inserted.

8. Naming contingent entities

The identification structure of the entities that result form collisions shall reference both the new invariant entity and the existing entities that were involved in the collision. This link identification structure is represented in our dynamic context layer in which we implement KRIPAC's face graph. Each *split* or *merge* of faces reference both the old face(s) and the new one(s) and the faces adjacent to the new one. This mechanism is exemplified in this paper in our 2D example by the *collision* entity. The data model shown in Fig. 13 enables to capture this local history for 2D

Using this data model, each new edge (D'1 and D'2 see Fig. 9) is now unambiguously named by referring to the pre-existing entity it modified. The physical file of our rounded rectangle can be completed as below with the new lines (#75 and #76) and the collision mechanism (#770 and #780).

```
#75 = polyline ( 'new_D1', (#74,#50) );            /* called D1' in the paper */
    #750 = polyline_ref ( #75, ( ), $ );
#76 = polyline ( 'new_D2', (#73,#30) );            /* called D2' in the paper */
    #760 = polyline_ref ( #76, ( ), $ );

    #770 = collision ( #600, #750 );
```

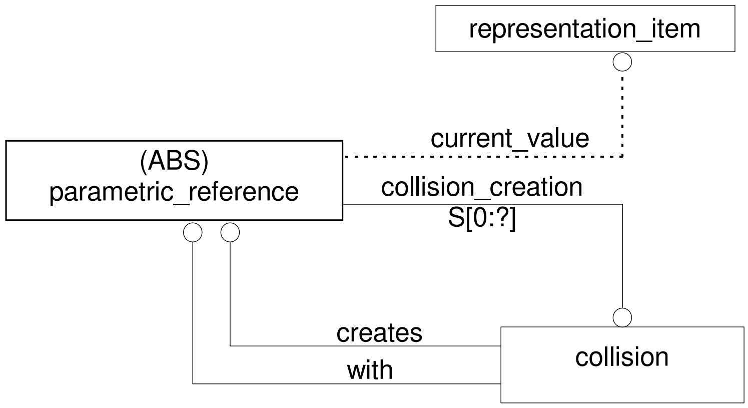


Figure 13: Complete model of a *parametric_reference*.

```
        #780 = collision ( #620, #760 );

    #79 = circular_arc ( 'round', #72, (#73), (#74) );
        #790 = circular_arc_ref ( #79, (#770,#780), #720, #730, #740 );
            #7900 = arc_fillet_2entities ( #790, (#600, #620),* , #800);
        #800 = real_literal ( 40.0, * );
```

To complete the description of the physical file (except the expression which are described in next section) all the *representation_items* (in our example geometric items: #20, ..., #80) that constitute the current instance shall be recorded in a *representation* (#1) associated with a 2D *representation_context* (#3). This current instance is referenced by a functional parametric model (#2) that records the set of *constraints* (#2000, ..., #7900) and the list of *parameters* (here the length L: #1000).

```
#1 = representation ( '2D_example', (#20,#28,#30, ... ,#77,#78,#79,#80), #3 );
#2 = functional_parametric_model ('model1', #1, (#2000,#2802,#3000, ..., #7900), (#1000));
#3 = representation_context ( 'local_context', '2d', 2 );
```

In the next section we show briefly how the expressions can be modeled in our proposed data model.

9. Modeling expressions

As shown in Fig. 9, a parametric data model shall contain algebraic expressions that involve variables and various kinds of algebraic operator (equational systems for 2D, boolean operations, arithmetic operations, ...). The first problem is to model a variable. A variable consists of three parts:

(i) a syntactical representation that provides a name that enables the variable to be referenced and that specifies its type of allowed values,

(ii) a mechanism, usually termed a context (in imperative programming) or an environment (in functional programming) that generates by some means (it may be, e.g., stored) the value of this syntactical representation,

(iii) a function, usually called interpretation function, which bounds the value mechanism to the syntactical representation.

Following ISO IS 13584-20 [8], this threefold concept may be modeled by a threefold data model presented in Fig. 14.

(i) A *variable* entity captures the syntactical representation of a variable and defines, by subtyping, its allowed type of value. This entity is referenced by e.g., expression, functions, etc..

(ii) A *variable_semantics* entity captures the mechanism that generates a value.

(iii) A relationship, termed environment, associates a *variable_semantics* with a *variable*.

In parametric data models, we want to capture two semantics:

1. the concept of the internal_variable; the value of such a variable results from the constraints (or parametric functions) that contain its syntactical representation in their defined attribute, and
2. the concept of the formal parameter of a functional parametric model; such a variable should not belong to the defined attribute of any (internal) constraint; its value results from an external mechanism that assigns an actual value to a formal parameter when and where the (functional) parametric model is involved.

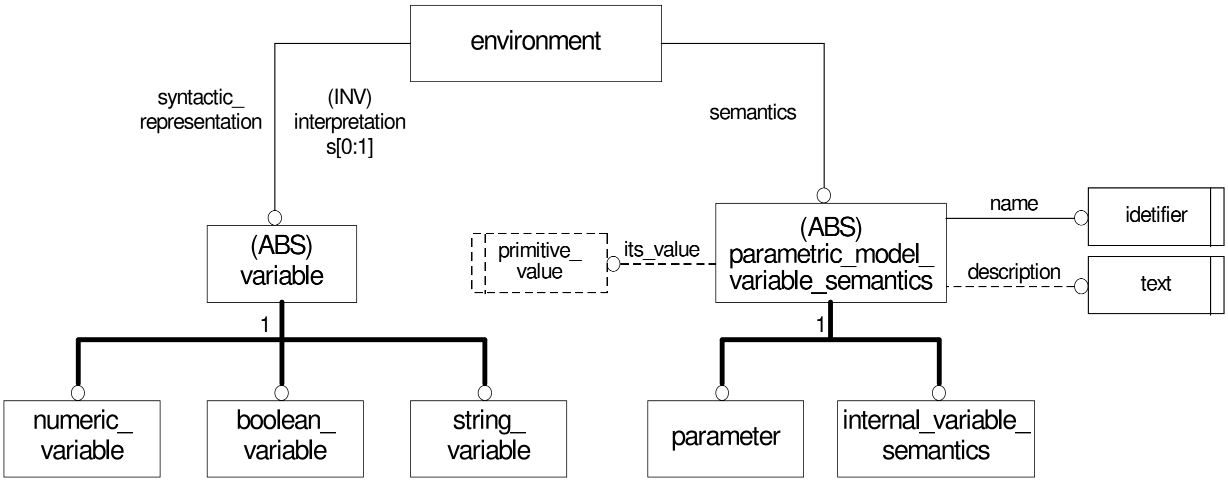


Figure 14: Simplified model of variables in parametric modeling

Such a mechanism enable to capture complex expressions by representing their abstract syntax tree. As an example, the expression 2L+10.0 (needed in Fig. 9) may be represented as defined in Fig. 15. The *environment* entity (#1001) is used to associate the generic variable *real_numeric_variable* (#100) to its value carried by the *parameter* (#1000).

```
#220 = plus_expression ( #230, #260 );
#230 = mult_expression ( #240, #100 );
#240 = int_literal ( 2, * );
#260 = real_literal ( 10.0, * );

#100 = real_numeric_variable ( );
    #1000 = parameter ( 'L', 50.0, $ );
    #1001 = environment ( #100, #1000 );
```

Figure 15: Modeling expression

The complete description of the example defined in Fig. 9 is described in appendix A.

10. Conclusion

With the development of commercial CAD systems that support various parametrics capabilities, a strong requirement is emerging from the CAD user community to be able to exchange or to archive parametrics in a neutral way. Such a parametric data model should be able to represent both the current instance of the parametric object and the parametric specification from which this current instance results. It should support a robust naming mechanism allowing to record constraints on entities that are no longer present in the current instance and providing for name matching during re-evaluation.

The current instance is an explicit geometric shape. Therefore, following ISO 10303, it may be modeled in EXPRESS and exchanged as a file compliant with ISO 10303-21. In the paper we have proposed to use the same data specification language, and the same exchange format, for the parametric specification.

The architecture we have proposed involves three layers :

- the current instance is modeled according to the specifications defined for explicit geometry by ISO 10303,
- the parametric specification is modeled by capturing, in the data model, the abstract syntax tree that defines the recorded constraints, and by modeling explicitly values, variables and the interpretation function that assign values to variables,
- the dynamic context is an in-between layer, that provides persistent names for constrained geometric entities whatever or not these geometric or topological entities still exist in the current instance.

The naming scheme we proposed distinguishes invariant entities and contingent entities. Invariant entities associated with a particular constructive gesture need to be defined for each category of constructive gesture. Invariant entities may be structured and we have shown how to access to their internal structure by structuring the names or by referencing input parameters of the constructive gesture. Various kinds of topological naming may be used for contingent entities. We have shown how an approach similar to the one defined by Kripac might be used within the context of our model.

Our data model has already been partially implemented. We plan to use it to experiment the exchange between two different parametric CAD systems.

References

[1] Y. Aït-Ameur, F. Besnard, P. Girard, G. Pierra, J-C. Potier: *Formal Specification and Metaprogramming in the EXPRESS Language.* International Conference on Software Engineering and Knowledge Engineering SEKE'95 (IEEE — ACM Sigsoft), Rockville, USA, 181–189 (1995).

[2] X. Chen: *Representation, Evaluation and Editing of Feature-Based and Constraint-Based design.* Ph.D. thesis, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, 1995.

[3] A. Cypher: *Watch what I do, Programming by demonstration.* MIT Press. 1993.

[4] C.M. Hoffmann, R. Juan: *EREP: an editable high-level representation for geometric design and analysis*. Technical Report CER-92-24, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, 1993.

[5] ISO 10303-11: *Industrial Automation Systems and Integration, Product Data Representation and Exchange, "The EXPRESS language reference manual"*. ISO, Geneva 1994.

[6] ISO 10303-21: *Industrial Automation Systems and Integration, Product Data Representation and Exchange, "Clear text encoding of the exchange structure"*. ISO, Geneva 1994.

[7] ISO 10303-42: *Industrial Automation Systems and Integration, Product Data Representation and Exchange, "Integrated generic resources: Geometric and topological representation"*. ISO, Geneva 1994.

[8] ISO IS 13584-20:1998: *Industrial Automation Systems and Integration, Parts library, Part 20: "Logical ressource: Logical model of expressions"*. ISO, Geneva 1998.

[9] J. Kripac: *A mechanism for persistently naming topological entities in history-based parametric solid models (Topological ID System)*. Proceedings of Solid Modeling '95, Salt Lake City, Utha, USA, 21–30 (1995).

[10] T. Laakko, M. Mäntylä: *Incremental constraint modelling in a feature modelling system*. Computer Graphics forum **15**, no. 3, EUROGRAPHICS'96, Poitiers, France, 366–376 (1996).

[11] K. Lee, G. Andrews: *Inference of the positions of components in an assembly: Part 2*. Computer Aided Design **17**, 1, 20–24 (1985).

[12] R. Light, D. Gossard: *Modification of geometric models through variational geometry*. Computer Aided Design **14**, 4, 209–214 (1982).

[13] B. Myers: *Taxonomies of Visual Programming and Program Visualization*. J. Visual Lang. and Comp. **1**, 97–123 (1990).

[14] G. Pierra, J-C. Potier, P. Girard: *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, 1994.

[15] G. Pierra, Y. Aït-Ameur, F. Besnard, P. Girard, J-C. Potier: *A general framework for parametric product model within STEP and Part Library*. European Conference Product Data Technology, London, 18-19 April, 1996.

[16] S. Raghotama, V. Shapiro: Boundary Representation Variance in Parametric Solid Modeling. Report SAL 1997-1, Spatial Automation Laboratory, University of Wisconsin-Madison 1997.

[17] D. Roller, F. Schonek, A. Verroust: *Dimension-driven geometry in CAD: a survey*. In Theory and practice on Geometric Modelling, Springer Verlag, 509–523 (1989).

[18] D. Schenck, P. Wilson: *Information Modelling The EXPRESS Way*. Oxford University Press, 1994.

[19] L. Solano, P. Brunet: *Constructive Constraint-based model for parametric CAD systems*. Computer-Aided Design **26**, no. 8, 614–621 (1994).

Appendix A

In this section we present the whole physical file of the rounded rectangle example (Fig. 9) discussed in this paper.

```
#1 = representation ( '2D_example', (#20,#28,#40,#50,#60,#62,#64,#66,#68,#70,#71,#72,#73,
                      #74,#75,#76,#77,#78,#79,#80), #3 );
#2 = functional_parametric_model ( 'model1', #1,(#2000,#2800, ..., #7900), (#1000) );
#3 = representation_context ( 'local_context', '2d', 2 );

    #100 = real_numeric_variable ( );
        #1000 = parameter ( 'L', 50.0, $ );
        #1001 = environment ( #100, #1000 );


/* current instance layer */
    /* dynamic context layer */
        /* parametric specification layer */

                      /* POINTS */
#20 = cartesian_point ( 'P1', (0.0,110.0) );
    #200 = cartesian_point_ref ( #20, ( ) );
        #2000 = canonical_cartesian_point_function ( (#200), *, (#210,#220) );

    #210 = real_literal ( 0.0, * );
    #220 = plus_expression ( #230, #260 );
    #230 = mult_expression ( #240, #100 );
    #240 = int_literal ( 2, * );
    #260 = real_literal ( 10.0, * );

#28 = axis2_placement_2d ( 'rectangle_axis', #30, $ );
    #280 = axis2_placement_2d_ref ( #28, ( ), $, $ );
        #2800 = canonical_axis2_placement_2d_function ( (#280), *, (#300) );

#30 = cartesian_point ( 'P2', (0.0,0.0) );
    #300 = cartesian_point_ref ( #30, ( ) );
        #3000 = canonical_cartesian_point_function ( (#300), *, (#210,#210) );

#40 = cartesian_point ( 'P3', (0.0,250.0) );
    ...
#50 = cartesian_point ( 'P4', (250.0,110.0) );
    ...

                      /* RECTANGLE */
#60 = polyline ( 'D1', (#20,#50) );                /* does not exist */
    #600 = polyline_ref ( $, *, ( ), $ );
        #6000 = canonical_polyline_function ( #600, *, (#200,#500) );
#62 = polyline ( 'D2', (#20,#30) );                /* does not exist */
    ...
#64 = polyline ( 'D3', (#30,#40) );
    ...
#66 = polyline ( 'D4', (#40,#50) );
    ...

                      /* ROUND */
#70 = cartesian_point ( 'C', 40.0, 70.0 );
    #700 = cartesian_point_ref ( #70,  ( ) );
```

```
#71 = axis2_placement_2d ( 'axis', #70, $ );
    #710 = axis2_placement_2d_ref ( #71, ( ), #700, $ );
#72 = cartesian_point ( 'A', 0.0, 70.0 );
    #720 = cartesian_point_ref ( #72, ( ) );
#73 = cartesian_point ( 'B', 40.0, 110.0 );
    #730 = cartesian_point_ref ( #73, ( ) );
#74 = circle ( 'basis_circle', #71, 40.0 );
    #740 = circle_ref (#74, ( ), #710 );

                      /* NEW LINES */
#75 = polyline ( 'new_D1', (#73,#50) );           /* called D1' in the paper */
    #750 = polyline_ref ( #75, ( ), $ );
#76 = polyline ( 'new_D2', (#72,#30) );           /* called D2' in the paper */
    #760 = polyline_ref ( #76, ( ), $ );

                      /* COLLISIONS */
    #770 = collision ( #600, #750 );
    #780 = collision ( #620, #760 );

                      /* ROUNDING FUNCTION */
#79 = circular_arc ( 'round', #74, (#72), (#73) );
    #790 = circular_arc_ref ( #79, (#770,#780), #740, #720, #730 );
        #7900 = arc_fillet_2entities ( #790, *, (#600, #620), *, #800);
    #800 = real_literal ( 40.0, * );
```

Received August 14, 1998