Pierra G., Potier J.-C., & Girard P. The EBP system : Example Based Programming for Parametric Design. in Modelling and Graphics in Science and Technology, Eds. J. Teixeira & J. Rix, Springer-Verlag, 1996, pp. 124-140.

The EBP System : Example Based Programming System for Parametric Design[1]

Guy Pierra, Jean-Claude Potier and Patrick Girard
Laboratoire d'Informatique Scientifique et Industrielle - ENSMA
Site du Futuroscope - 86960 FUTUROSCOPE - FRANCE
e-mail pierra@ensma.univ-poitiers.fr

Abstract

This paper investigates the problem of parametrics within the perspective of generating and exchanging families of cognate products. Two ways are discussed. A parametric representation gathers together within the same structure the parametric definition and one specific instance: the current instance. An EXPRESS model that follows this approach is presented. Its integration within the STEP Standard would enable the exchange of (simple) parametric designs. The use of a parametric program, based on a standard API, is a more conventional approach. We present the EBP system which enables such a program to be generated through purely graphical interactions. This system provides for all the constructs required in the target program: variables, expressions, functions, and control structures. If the API itself supports all these constructs, the parametric representation may be restored on the receiving system.

---

[1] Workshop on Graphics and Modelling in Science and Technology, Coimbra, Portugal, 27-28 June 1994

The EBP System : Example Based Programming System for Parametric Design[1]

Guy Pierra, Jean-Claude Potier and Patrick Girard
Laboratoire d'Informatique Scientifique et Industrielle - ENSMA
Site du Futuroscope - 86960 FUTUROSCOPE - FRANCE
e-mail pierra@ensma.univ-poitiers.fr

Abstract

This paper investigates the problem of parametrics within the perspective of generating and exchanging families of cognate products. Two ways are discussed. A parametric representation gathers together within the same structure the parametric definition and one specific instance: the current instance. An EXPRESS model that follows this approach is presented. Its integration within the STEP Standard would enable the exchange of (simple) parametric designs. The use of a parametric program, based on a standard API, is a more conventional approach. We present the EBP system which enables such a program to be generated through purely graphical interactions. This system provides for all the constructs required in the target program: variables, expressions, functions, and control structures. If the API itself supports all these constructs, the parametric representation may be restored on the receiving system.

1 Introduction

Portability of parts libraries is a major economic concern for Computer Aided Design (CAD) System users, for component manufacturers, and for CAD system vendors. Such a portability would enable the level of granularity of the data model of future CAD Systems to be changed. Besides dealing with points, curves, surfaces and solids, the data model of these systems will consist of technical objects like bearings, capacitors or stairs according to their field of application. To allow such a portability, a whole set of concepts, known as the CAD/LIB approach, has been developed in Europe [1]. This set of concepts is based on the

[1] The research described in this paper was funded partially by EU under project ESPRIT III # 8984 (PLUS), and partially by the French Ministry of Industry under grant 93.4.930080.

experience gained using proprietary solutions and/or national standards that only partially fulfilled the end-user needs. These concepts constitute the agreed basis of both the European and the International standardization work in this area (CEN/TC310-PR ENV 40004 and ISO/TC184/SC4-ISO 13584).

The question of parametrics is crucial for such a topic, and a lot of effort has been deployed in order to clarify the concept of parametrization, and its relationship to the concept of a product. For ISO 13584, parametrization is related to the concept of product (or part) classes. A product class is a set of products which are described together, which are assigned a common name, and whose instances may be distinguished from each other by the values of some numeric-valued, string-valued or Boolean-valued parameters called product (or part) *identification_attributes*. Fixing the identification_attributes within the class fully identifies the product *instance.* A representation class is a set of representations which are described together and whose instances may be distinguished from each other by the values of some numeric-valued, string-valued or Boolean-valued parameters called *representation_attributes*. Fixing the representation_attributes within the class fully specifies the representation instance. A shape class, also called a (geometric) parametric model, is only a special case of a representation class. Each instance is fully defined by its parameter values. Exchanging libraries of parts requires a capability for generating, for exchanging, and for interpreting parametric models. The goal of this paper is to present the two innovative approaches developed within the ESPRIT PLUS project, whose role is to validate and to improve the CAD/LIB concepts, for parametric model generation and exchange.

In the next section of this paper, we define the requirements for parametric modelling in the context of parts library definition and exchange. In the third section, we investigate the concept of parametrics, and we relate it to the concept of a program and to the example-based programming paradigm. In the fourth section, we outline the general framework developed to allow parametric representation exchange within the context of the emerging Standard ISO 10303 (STEP). This recent proposal [2] is under consideration for insertion within STEP. In the fifth section, we present the EBP system. Its role is to allow an end-user to generate a parametric model in the format of a program. The use of such a format is not quite as old-fashioned as it may appear at first glance. Firstly, as shown in this paper, this parametric description may be generated by purely graphic interactions, even for those programs that contain control structures, which is not usual in parametric or variational systems. Secondly, the interpretation of such programs may generate a parametric representation on the receiving system, if this system provides such a facility.

2 Parametrics in the Context of Parts Library

Over the last few years a lot of effort has been made in order to ensure more flexibility of a designed shape. The different approaches, sometimes grouped under the name dimension-driven geometry [3], address two very different problems.

2.1 Declarative Model

Declarative approaches [4], and in particular variational geometry, consist of geometry problem solving: "given a model with a sufficient number of geometric constraints and a topological or approximate geometric description, we want the precise model to be evaluated automatically" [3]. The solution may be unknown to the user. He or she states the constraints. The role of the system is to compute the (possible) solution(s).

Mathematically, let P be the set of parameters defined over a domain $D \quad P$ (often $P = \mathbb{R}^n$), and let S (shape) be the set of values (points, curves, numeric values,...) that describe an explicit instance. S belongs to the set S of all the possible shapes. A *declarative model* is an equation :

$$A(P,S) = 0;\ P \in D,\ S \in S$$

where A is an operator which is, generally, neither linear nor convex.

A lot of methods have been used to solve this problem: algebraic approaches, often based on the Newton-Raphson iterative method [5] [6] [7], inference engine [4] [8], Buchberger's Gröbner Bases Method [9] [10], constraint graphs [11] [12], and many others [13]. The popular 2D "sketcher", available on various CAD systems, corresponds to this approach. Unfortunately, there exist only partial solutions that address specific problems [14]. There is no declarative modeller, to our knowledge, able to generate in a deterministic way all the shape instances that belong to a shape class which represents any families of parts.

2.2 Parametric Model

The imperative approach, discussed in this paper, addresses a very different problem: "given a class of shapes whose design process is well known and may be supported by the interface of some CAD systems, we want any instance, characterized by its parameter values to be generated automatically in a deterministic way". We call such a structure a *parametric model*.

Mathematically, using the above notations, a parametric model is a function:

$$F: D \mapsto S;\ S = F(P)$$

Where F is the function that defines the instance from its parameter values.

Since the domain *D* is often non-specified, the function may "fail". This means that the parameter values do not belong to the domain *D*. Nevertheless, for all sets of parameter values that belong to *D*, the parametric model defines exactly one instance. In the context of Parts Library we are only interested in the parametric model.

2.3 Structure of a Parametric Model

The function F is always expressed as a composition of functions (which may consist of a singleton):

$F = f_n \circ f_{n-1} \ldots \circ f_1$

we call the f_i functions the *parametric functions*.

To provide for parametric modelling and exchange requires a capability for designing, for exchanging and for restoring this composition of functions.

3 Parametric Representations, Imperative Programs and Example-Based Programming

The characteristic of the parametric approach is that the class level, in fact the function , is always designed and represented together with one instance. We call this instance the *current instance*. Figure 1 shows a very simple example based on numeric expression. The values which appear on the tree are the *current instance* values. In fact these values play two roles. (1) They explicitly represent the *current instance*. (2) These values stand for variables whose types may be deduced from the values.

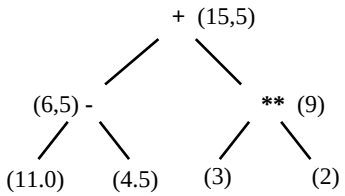

Fig. 1. A (parametric) expression

Provided that the user builds this expression on the display computer of the CAD system, and provided that the expression itself is recorded, the (implicit) parametric program is perfectly defined:

```
real: x, y, z, result
integer: i,l,k
z := x - y;
k := i**l;
result := z + k;
```

Fig. 2. The corresponding program

One difficulty remains. When the user reintroduces, in a subsequent expression, the value "15.5" on the display computer, the system is incapable of knowing whether it is a new "parameter", or whether it is the previously computed value. Fortunately, this problem does not exist in geometry design. Since the entities are graphical entities stored in the CAD system database, the values for the *current instance* are database pointers. Picking up the same entity refers to the same database pointer (in the *current instance*), and therefore to the same variable in the (implicit) program (Fig. 3).

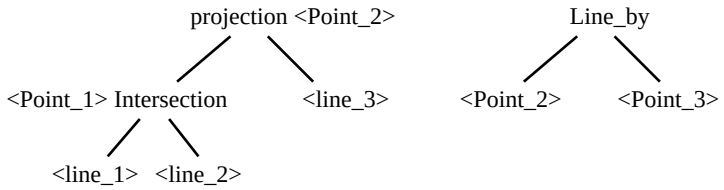

Fig. 3. A sequence of parametric constructs

When, in the above example, <Point_2> is referred to within the second construct, the implicit variable is clearly identified.

Since numeric values are largely used in geometric design, the same solution shall be applied for the numeric values of the *current instance*. In parametric design (and in the EBP system), numeric values are considered as database entities. They are stored in the database and they may be accessed through a menu. Picking up such a value clearly identifies the (implicitly referenced) variable.

In the parametric approach the *current instance* is embedded within the program (or inversely). Example-Based programming allows us to split it up, should we want to do so.

The concept of example-based programming shares a lot of commonality with this approach to parametric design. First analysed in HALBERT's PhD Dissertation [15], and formalised in MYERS' works [16] [17], the main idea of example-based environments for program design is to avoid the abstraction level

of variables by permitting the user to deal with a specific value of each variable (the "*example*"). In an example-based visual environment, instead of selecting the functions and picking up the variables to which each function shall apply, the user (programmer) *does the function* on *values* that stand for the program variables. The difference with parametric design is that, in example-based programming systems, the (implicit) program is separated from the *example* (the *current instance*). All created values (in the example) stand for implicit variable declaration (in the program context). All references to values (in the example) stand for reference to variables (in the implicit program). One main role of these systems is to manage the context of the program which ensures the indirect link between example values and program variables [18]. Fig. 4 shows the dynamic context management of the EBP system presented in section 5.

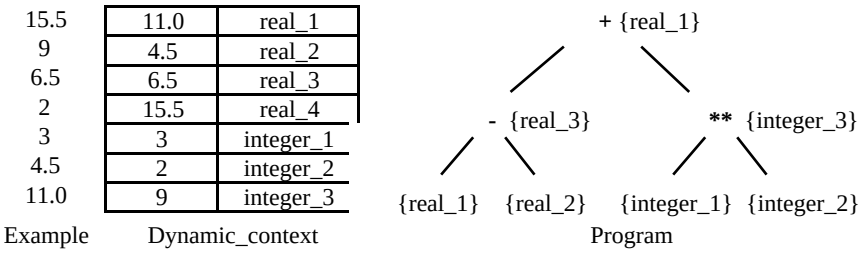

Fig. 4. The dynamic context management of example-based programming systems

4 A Neutral STEP-Compatible Data Model of Parametric Representations

STEP is an International Standard for the computer-interpretable representation and exchange of product data. Its objective is to provide a neutral mechanism capable of describing product data throughout the life cycle of a product, independent from any particular system. STEP, which is now being published, contains a lot of innovative features such as:
– the use of formal methods for data specification,
– the definition of mappings from data models to implementation forms,
– the definition of powerful resources for product modelling, and, in particular, geometric modelling.

Nevertheless, in its development process STEP focused on explicit product modelling, and, in particular, on explicit geometry. In STEP a *representation* is defined as a set of (explicit) *representation_items* (e.g., in geometry: points, curves, surfaces or solids) associated with a *representation_context*.

```
ENTITY representation
   items: SET [1:?] OF representation_item;
   context_of_items: representation_context;
END_ENTITY;
```

Fig. 5. The STEP data model for representation in EXPRESS

Since the goal of the programs generated by the EBP system presented in section 5 is to create geometry within the database of any target CAD-systems (supporting the standardized API), and since more and more CAD-systems provide proprietary parametric modelling facilities, it appeared necessary to try to develop a formal model of the database of such systems. The main objective of this model was to investigate whether or not it was possible, from a parametric program created by the EBP system, to generate a parametric representation if the receiving system provided for such modelling facilities. The result of this investigation, outlined in this section was twofold. (1) It showed that the answer to this question was positive provided that the numeric entities and the numeric expressions might be defined by interface functions. (2) It provides a global framework for the integration of parametric product modelling facilities within the context of STEP [2].

As discussed in section 3, the basis of this model is to propose to link systematically a parametric model with one specific instance (the *current instance*). We call such a parametric model a *parametric representation.* This approach enables us to consider the current instance of a parametric representation, and therefore the parametric representation itself as a special case (a SUBTYPE) of a STEP representation. This is shown in Figure 6 using the EXPRESS-G graphic formalism .

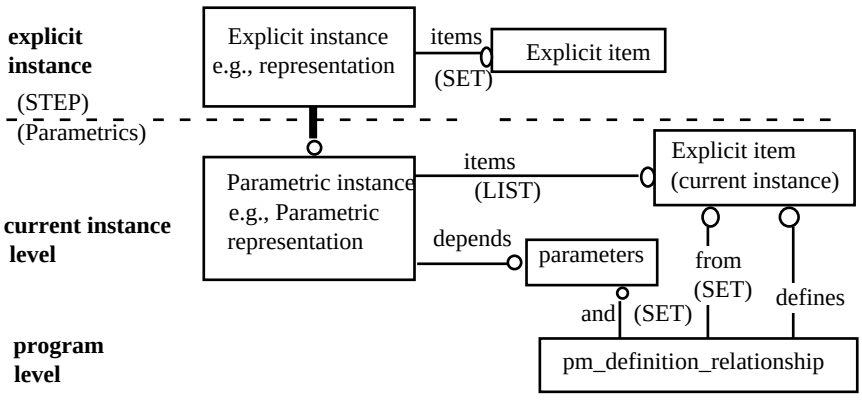

Fig. 6. The (simplified) data model of a parametric representation

The particularities of this SUBTYPE are two fold. (1) Each *representation_item* is referred to by one entity, called a *pm_definition_relationship*, which provides its parametric definition and corresponds to the functions f_i of clause 2.3. (2)The set of *representation_items* is now ordered. The parametric definition of each entity defines this entity on the basis of some representation parameter and/or some other representation items but these latter shall precede the referenced entity in the entity list.

Moreover, a (parametric) representation instance may itself be dependent on the context in which the instance is inserted. For example, the extended length of a pneumatic cylinder depends on the other parts of the global design. When an instance is inserted in a higher-level assembly, it becomes an *occurrence*. And an occurrence representation depends on the value of some other parameters called *context_parameters*. Modifying *context_parameters* (e.g., compressed length of a spring) only modifies the occurrence representation, it does not change the product (instance) itself. Generally speaking, the values of the occurrence context parameters result, directly or indirectly, from the representations attributes or context parameters of the (parametric) representation in which the occurrence is inserted.

Finally, *context_integrity_constraints* and *global_integrity_constraints* enables the modelling of the domain *D* (see clause 2.2), and an output interface specifies the *representation_items* (often called the "datums") that shall be instantiated in any occurrence representation.

```
ENTITY pm_representation
   SUBTYPE OF (representation);
   representation_attributes     :LIST[1:?] OF UNIQUE
                                    pm_representation_parameter;
   context_parameters            :LIST[0:?] OF UNIQUE
                                    pm_representation_parameter;
   parametric_definition         :LIST[1:?] OF UNIQUE
                                    representation_item;
   context_integrity_constraints  :OPTIONAL SET[1:?] OF
                                    Boolean_expression;
   global_integrity_constraints  :OPTIONAL SET[1:?] OF
                                    Boolean_expression;
   output_interface              :LIST[0:?] OF UNIQUE
                                       pm_named_representation_item;
DERIVE
   SELF\representation.items     :SET[1:?] OF representation_item
                                 :SELF.representation_attributes
                                     +SELF.context_parameters
                                     +SELF.parametric_definition;
WHERE
...
END-ENTITY;
```

Fig. 7. The proposed EXPRESS model for parametric representation

Two categories (i.e., SUBTYPES) of parametric functions exist, modelled as *pm_definition_relationship*.

A *pm_canonical_definition* is a special case of *pm_definition_relationship* which is associated one to one to any *representation_item* and which:
– associates to each numeric-valued, string-valued or Boolean-valued attribute of this *representation_item* one *expression* that enables its computation, and,
– collects, in the *used_items* derived attribute:
(1) the *representation_items* directly or indirectly referred to by these *expressions*, (through the *used_representation_item* expression tree traversal function) and,
(2) the *representation_items* directly referred to by the *representation_item* itself.
This entity shall be subtyped for each *representation_item*.

The role of the *used_items* attribute is to assert that the composition of parametric functions (see clause 2.3) is consistent.

```
ENTITY cartesian_point
   SUBTYPE OF (point);
   coordinates: LIST[1:3] OF
    length_measure;
END_ENTITY;
```

```
ENTITY pm_canonical_cartesian_point
   SUBTYPE OF
      (pm_canonical_definition);
   SELF\pm_definition_relationship.de
      fines : cartesian_point;
   coordinates: LIST [1:3] OF
      numeric_expressions;
DERIVE
   SELF\pm_definition_relationship.us
      ed_item:
   SET [0:?] OF representation_item
      :=used_representation_item
            (SELF.coordinates);
END_ENTITY;
```

Fig. 8. The STEP definition of a cartesian_point and its proposed pm_canonical_definition

A *pm_constraint_based_definition* is a parametric function that defines a new *representation_item* through constraints with pre-existing *representation_items*. No canonical definition for such constraint-based parametric functions exists: it depends on the system (or standard format) designer choice

```
ENTITY
   SUBTYPE OF pm_constraint_based_definition
   item_1: line;
   item_2: line_or_plane
DERIVE
   SELF\pm_definition_relationship.used_item
      :SET[0:?] OF representation_item
      :=[SELF•item_1] +[SELF•item_2];
END-ENTITY;
```

Fig. 9. A (simplified) example of a pm_constraint_based_definition for cartesian_point

Besides its interest for exchanging directly parametric models in the format of pm_representations, this data model points out the role of expressions, and, in particular, of numeric expressions. In program-oriented parametric model exchange, these expressions shall be:

- captured by the EBP system during interactive program construction, and
- restored (through the interface) on the receiving system in order to be able to restore not only the *current instance* but also the parametric representation itself.

It should be emphasised that these expressions shall contain not only the usual operators and functions (+,*,/..., SIN, ...), but also grapho-numeric functions which enables a value which results from some other *representation_items* (e.g., *distance_of*(.,.); radius_of (.),...) to be involved in a numeric expression.

5 The EBP System

Parametric programs constitute the other method of exchanging parametric models. The EBP system is a programming environment for designing such programs. Based on the visual example-based-programming paradigm, EBP is intended to reduce the programming tasks required for parametric program generation to a minimum. During an interactive session, a draftsman only designs one instance (the *example*) on a CAD system. As long as the draftsman acts, the EBP system records his work, and translates it into a real program.

The CAD system builds up the example representation by means of system *procedures* involving *parameters*. These procedures and parameters are implicitly controlled, in interactive mode, by the user manipulating *commands* and (values) *operands*. To record a program requires the explicit description of the (implicitly) involved procedures and parameters. Is this translation easy?

Obviously, this translation is not straightforward. The first difficulty concerns the management of objects, what we have called the "dynamic context" in the third section of this paper. This object management allows direct translation of operands (values) in references (variable names). The second difficulty split into the two classical problems known in compilation theory as interpretation and code generation. We will describe how these different problems are addressed in the EBP system.

5.1 Object Management

The first difference between designing an explicit representation and specifying a program lies at the level of the manipulated objects. Command languages refer to values. Programs refer to variables. If we consider the designed representation as an example of the (implicit) program, the problem is to identify what values stand for parameters, constants and internal variables.

As far as the geometric entities are concerned we have already pointed out that the (example) values may be captured as CAD database pointers. The condition, applied in the EBP system, is to capture their values above the designation layer of the dialogue interface (see Fig. 10). Therefore, the only problem is to create

and to manage the (dynamic) context of the program. Each entity creation has to generate an internal variable (implicit) declaration. Recording does not only consist of noticing interactions: it requires the object manager (named "context manager") to be notified for each object creation or deletion. This is done through a specific link between the CAD System database and EBP context manager (see fig.10).

As far as numeric values are concerned, all the parameters shall be declared by a specific command and are considered as database entities. When an explicit numeric value is involved in an expression, it is implicitly considered as a constant.

5.2 Parsing the Command Script

In compilation theory, program interpretation is based on parsing. The syntactic abstract tree is the basis of semantic analysis and code generation. Unfortunately, this approach may not be used for the interpretation of interactive command languages.

Firstly, command languages have in fact no (or an extremely limited) syntax. All the sequences of interactions are allowed, even if they are useless. No fatal error is raised after any interaction inputs (i.e., a command language never crashes with an "error # 347" message !). The user is intended to react to the interactive command messages. Therefore, interpretation of such incomplete or incoherent sequences of commands and operands would lead to incorrect syntactic abstract trees.

Secondly, command languages are heavily context-sensitive. Dialogue interfaces contain a lot of dialogue context variables that enable command remanence, multiplicity of "threads of task accomplishment"[19] or inhibition of previous commands. Command remanence means that some procedure may be invoked twice, without repetition of the interactive command. Multiple threads of task accomplishment allow the user to stop an unfinished sequence of interactions in order to carry out realise another one, and to terminate the first sequence after the end of the second one. So, the operands of one command are embedded within the operands of another.

Such dialogue threads act in fact as non context-free languages, which are known to be hardly translatable. These two aspects show the non-feasibility of syntax-driven translation.

5.3 From Syntax- to Semantic-Driven Translation

Even if, for ergonomic reasons, dialogue interfaces enable any dialogue (or near any dialogue) sequences, some representation is in fact built as a result. The CAD system procedures which are really triggered constitute the semantics of the command sequence. Capturing these semantics enables the semantic tree of the

parametric program to be directly built. In the EBP system, the user commands are captured only when a system procedure is triggered, and not *within* (or below) the dialogue interface.

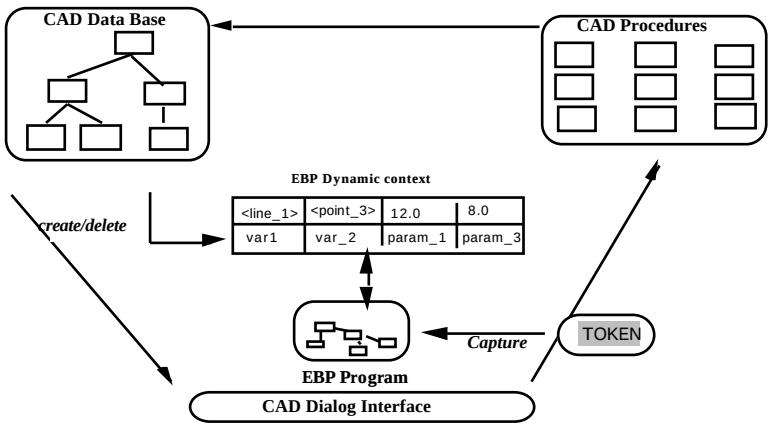

Fig. 10. Architecture of the EBP system

5.4 The Problem of Expressions

We emphasised, in section four, the need to capture expressions. These expressions may be numeric (e.g., <Param_2> * <Param_2>), grapho-numeric (e.g., *distance_of* <Point_1>, <Point_2>) or graphic (e.g., *projection_of* <Point_1> *onto* <Line_1>). When a procedure is triggered, only the result of such expressions is transferred. Recording only this result would be a loss of the real parametric function. When constants are the unique objects involved in an expression, there is no real loss. On the other hand, when parameters or representation_items are involved, such a loss is unacceptable.

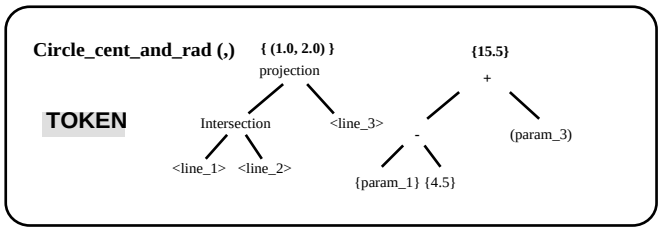

Fig. 11. Content of the captured tokens

The solution used in the EBP system (in fact, in the dialogue interface of the CAD system) consists of gathering within the same token the expression value and its syntactic abstract tree. When an expression is not used (the user changes his "thread of task accomplishment", or his mind) the expression remains in the dialogue interface and does not appear within the program. When the result of an expression is sent to a CAD system procedure, the EBP system captures the name of the procedure together with the expression abstract trees.

5.5 Ambiguity in Constraint-Based Geometry

The other major difficulty is the ambiguity of constraint-based geometry. In a graphical command language, such an ambiguity is removed by the pointing device coordinates. To capture such coordinates is useless: changing the program parameters would change the meaning of the position coordinates. In EBP, this problem is solved by duplicating all the ambiguous constraint-based construction. The version used during program (implicit) construction obtains the pointing device coordinates as input parameters, and returns some specific value (based on entity orientations) as output parameter. The version used during program execution (and code generation) uses this specific value as input parameter to remove the ambiguity of the construction. It should be noted that this problem is a general one in parametric design. All geometric programming interfaces must address it, or be restricted to non-ambiguous (very poor indeed) constructs. For example, in 2D geometry, most of the constructs that involve a line extremity, or a tangency to a circle are ambiguous.

5.6 Introducing Control Structure in Command Languages

Unlike the usual parametric or variational systems, and unlike most command languages, EBP provides purely interactive graphic facilities for control structure design.

We only outline here the solution used which was developed for the LIKE system [20] [21].The predicates of these control structures are defined using the display computer which is part of the interface provided to the user. The recurrence definition of a loop is specified by the interactive user by designing successively (1) the control predicate, (2) the first (example) step of the loop and (3) the second step of the loop. EBP may then infer (algorithmically) the recurrence relationship. Therefore, it may generate the other steps and store, in the semantic tree, the recurrence relationship.

The reason for the lack of control structures in most example-based programming systems is to be found in the dynamic context management. Identifying objects during the interaction process requires, as we have previously shown, an object management, with an automatic naming process of the created objects. This automatic process is in all cases some kind of numbering. Control

structures are very inappropriate for this automatic naming. Alternative structures create different objects for each branch. Iterations create inherently varying numbers of objects. What are the consequences for the subsequent naming?

In the EBP system, we defined strong rules to manage the dynamic context. The idea is based on the concept of local variables in subroutines. The exchange between the calling process (the embedding context of the program) and the called the subroutines (contexts that are local to control structure branches) are governed by this metaphor. For example, an object which was created during a loop structure is not individually attainable in the embedding program (after the end of the loop structure). The set of entities that were created during the same loop structure are attainable as a whole. These rules, which seem to be quite natural for end-users, solve all the context management problems that result from the introduction of control structures in example-based programming systems [21].

6 Conclusion

Parametrics is an emerging technology that encompasses several different problems. In this paper we have defined a parametric model as a function that depends on an explicit set of parameters and that may generate each representation instance by composition of internal functions. There are two ways of representing such a parametric model: either as a data structure that we called a parametric representation, or as a parametric program referring to an API. As far as the parametric representation is concerned, we represented the global function by embedding in its description one specific instance: the current instance. The values contained in this instance stand for the functions variables and are referred to by the functions that constitute the global function. We have presented the overall architecture of an EXPRESS model that follows this approach and that might be integrated into the STEP standard to provide for parametric product modelling facilities exchange.

As far as the second approach is concerned, which corresponds to the parametric modelling facilities provided in the first release of the Parts Library Standard (ISO 13584), we have used the Example Based Programming paradigm to enable the design of such a program by an end-user. In the EBP system, the user designs an example. The values of the example are captured by the EBP system which creates corresponding variables. The recorded functions refer to these variables, and EBP manages, both in recording and in running mode, a dynamic context where program variables are associated with example values. The functions are captured on the surface of the user interface, and each token contains not only the operand values, but also their whole expression tree if these values result from expressions. These expressions are stored within the program, and, when running such a program on a parametric system supporting the API ,

the parametric representation may be restored within its database provided that the API supports both the parametric functions and the expressions, and provided that the parametric program is only sequential. Finally EBP also enables to generation of parametric programs that contain control structures. Their expressive power is therefore greater than the usual parametric representations.

References

[1] Pierra, G.: Modelling classes of pre-existing components in a CIM perspective: the ISO 13584/ENV 40004 approach, Revue Intern. CFAO et infographie,9,3, 1994, p 435-454.

[2] Pierra, G.: A general framework for parametric product modelling, ISO-STEP meeting, Davos, Mai 1994, ISO/TC 184/SC4/WG2 N183, 51 p.

[3] Roller, D., Schonek, F., Verroust, A.: Dimension-driven geometry in CAD : a survey in : Theory and Practice on Geometric Modeling, Springer Verlag, 1989, pp. 509-523.

[4] Sunde, G.: A CAD system with declarative Specification of Shape. Proc. of EuroGraphigs Workshop on Intelligent CAD Systems, Noodwijkerhout, The Nederlands, 21-24 April, 1987.

[5] Hillyard R., Braid, I.: Analysis of dimensions and tolerances in computer-aided mechanical design. Computer Aided Design, 10, 3, 1978, pp. 161-166.

[6] Light, R., Gossard, D.: Modification of geometric models through variational geometry. Computer Aided Design, 14, 4, 1982, pp. 209-214.

[7] Lee, K., Andrews, G.: Inference of the positions of components in an assembly : Part 2. Computer Aided Design, 17, 1, 1985, pp. 20-24.

[8] Aldefeld, B.: Variation of geometries based on a geometric-reasoning method. Computer Aided Design, 20, 3, 1988, pp. 65-72.

[9] Chou, S.-C.: Proving-Elementary Geometry Theorems Using Wu's Algorithm, Contemporary Mathematics, 29, AMS, 1984, 243-286.

[10] Chou, S.-C., Schelter, W.F. & Yang, J.-G.: Characteristic Sets and Gröbner Bases in Geometry Theorem Proving, Workshop on. Computer-Aided Geometric Reasoning, INRIA, Sophia-Antipolis, 1987, pp.29-56.

[11] Sutherland, I. E.: A Man-Machine Graphical Communication System, Proc. of AFIPS Spring Joint Comp. Conf., 23, 1963, pp. 329-346.

[12] Kin, N.: PictureEditor : A 2D Picture Editing System Based on Geometric Constructions and Constraints. Proc. Comp. Graphics Int.'89, Leeds, Springer-Verlag, 1989, pp.193-208.

[13] Dufour, J.F.: Programmation et résolution de problèmes de construction géométrique, BIGRE ,67, Jan. 1990, pp..136-147.

[14] Verroust, A.: Construction d'objets géométriques définis par des contraintes. BIGRE, 67, Jan. 1990, pp. 62-74

[15] Halbert, D.: Programming by example. PhD. Thesis, Berkeley Univ., California, 1984, pp.121.

[16] Myers, B., A.: Visual Programming, Programming by Examples, and Program Visualization : A Taxonomy. Proc. of SIGCHI 86, Human Factors in Computer Systems, New-York, 1986, pp. 59-66.

[17] Myers, B.: Taxonomies of Visual Programming and Program Visualization. J. of Visual Lang. and Comp., 1, 1990, 97-123.

[18] Girard, P., Pierra, G.: Command Recording versus Parametric and Variational Systems, and old/new third way of parametrizing CAD models by End Users. COMPEURO'93 (IEEE-SEE), 1993, pp. 194-200.

[19] Bass, L., Coutaz, J.: Developing Software for the User Interface. SEI Series in Software Engineering, Addison-Wesley, 1991, 251 p.

[20] Girard, P.: Environnement de programmation pour non-programmeurs et parametrage en conception assistée par ordinateur: le système Like, PhD Thesis, Univ. of Poitiers, 1992, 195 p.

[21] Girard, P., Pierra, G.: One more step towards end-user programming environments: introducing control structures in visual example-based programming. (to appear).