Color profile: Generic CMYK printer profile
Composite Default screen

Chapter Seven

Bringing Programming by Demonstration to CAD Users

Patrick Girard

Laboratoire d'Informatique Scientifique et Industrielle
École Nationale Supérieure de Mécanique et d'Aérotechnique

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

Abstract

This chapter presents a suite of systems we developed that constitute a solution for bringing programming to end users in the field of computer-aided design (CAD), by using programming by demonstration (PBD). This suite includes the LIKE system, which laid the foundations of our method, and the EBP system (Example Based Programming in Parametrics), which is intended to enable CAD system users to generate every program that describes the geometric shapes of a collection of parts through the interactive graphic design of one example of this collection. From a PBD point of view, they prove that, at least in some application area where system users have particular skills, complete PBD environments may be developed. From a CAD systems point of view, this approach proves that parametric CAD systems, which are already very successful for sequential (or simple repetitive, pattern-based) parametric design, may be extended to support the parametric design of every conditional or repetitive shape aspect. From a user interface viewpoint, it also proves that very powerful macro-with-example recorders may be developed.

7.1 Introduction

Over the last few years, lots of advances have been achieved to reduce the programming skills and the abstraction level that are required for computer use and programming. Visual programming (Glinert 1990) permits users to select graphically both the functions and the variables that constitute programs. Programming by demonstration, or PBD (Cypher 1993), allows direct interaction with example values that represent the program variables instead of their abstract names, or iconic presentations. Many experimental systems have proved both the usability and the interest of the latter approach. Nevertheless, no PBD system, to our knowledge, has reached the same expressive power as conventional programming in its application area.

The goal of this chapter is to present a suite of systems we developed that constitute a solution for bringing programming to end users in the field of computer-aided design (CAD), by using PBD. This suite includes the LIKE system, which laid the foundations of our method, and the EBP system (Example Based Programming in Parametrics), which is intended to enable CAD system users to generate every program that describes the geometric

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

shapes of a collection of parts through the interactive graphic design of one example of this collection ("variant programming"; Roller 1990).

7.2 PBD and CAD

In her book *A Small Matter of Programming: Perspectives on End-user Computing,* Nardi (1993) has identified CAD as a natural candidate for end-user programming, because these systems "allow end users to create useful applications with no more than a few hours of instruction." We will see in this section how this can be extended to complete PBD features. First, we describe the application field of PBD in CAD, called *parametrics programming.* Then, we detail two approaches, *variational* and *parametric,* and relate them to PBD terminology. Last, we give the requirements for PBD we adopted.

7.2.1 CAD: A suitable Area for PBD

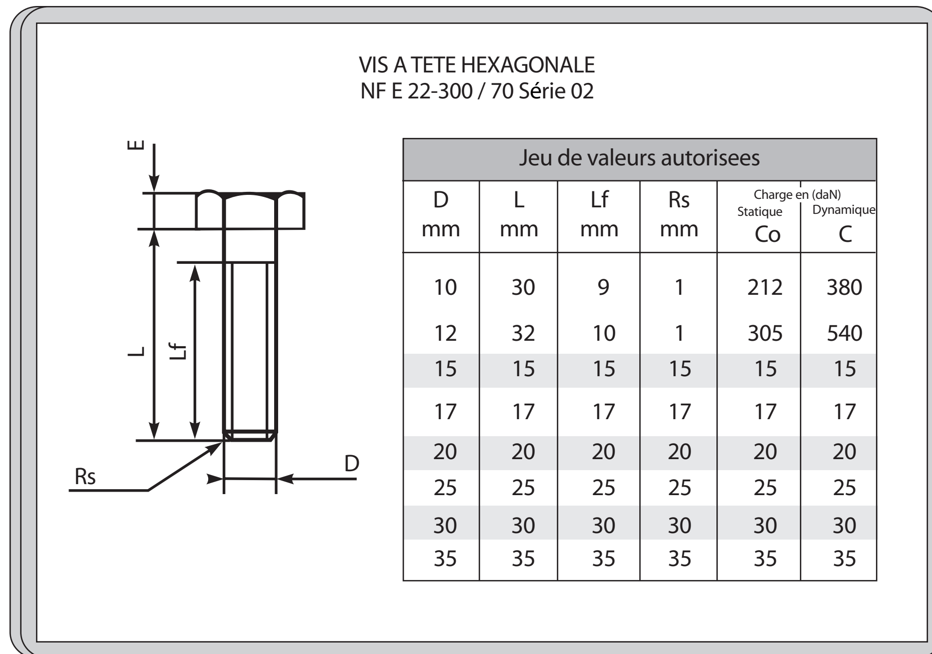
PBD opens the door to new generations of programming environments. In the fields where some visual appearance may be assigned to variable values, direct manipulation of these values allows implicit programs design. SmallStar (Halbert 1984) for iconic desktop programming, Peridot (Myers 1993), Garnet (Myers et al. 1990], Macros by Example (Olsen and Dance 1988) for UIMS programming, KidSim (Cypher and Smith 1995) for simulation, WYSIWYC Spreadsheet (Wilde 1993), Geometer's Sketchpad (Jackiw and Finzer 1993), and ProDeGE+ (Sassin 1994) in drawing systems have proven in different fields the validity of this approach. Despite this success, most systems seem to be mainly at a prototype stage. In the CAD area, PBD, under the name of "parametrics," has really found some commercial market.

Designing new products often entails assembling preexisting components intended to be used in different products. These components, named "standard parts," are gathered into families described by a part family model (see Fig. 7.1). According to Shah and Mäntylä (1995), "a part family model represents a collection of parts exhibiting some variation in dimensions, tolerances, and overall shape that nevertheless are considered similar from the viewpoint of a certain application." The context of some part family corresponds to some product standard (e.g., the family of ISO 1014 hexagonal screws), to some supplier's environment family, or some family of

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

Figure 7.1



Jeu de valeurs autorisees					
D mm	L mm	Lf mm	Rs mm	Charge en (daN) Statique Co	Dynamique C
10	30	9	1	212	380
12	32	10	1	305	540
15	15	15	15	15	15
17	17	17	17	17	17
20	20	20	20	20	20
25	25	25	25	25	25
30	30	30	30	30	30
35	35	35	35	35	35

Example of part family.

firm-specific components, which are described by end users for internal use. Because of the rather huge number of members of these collections, some unique part family shall describe the whole collection of corresponding shapes.

In the first generation of CAD systems, part family models were described as parametric programs. In these conventional CAD systems, such programs were textually described, often in fortran or in the C language. When triggered, they create geometric entities by means of application programming interface (API). A lot of these systems have been developed on end-user sites where draughtsmen were trained on CAD modeling. So, a strong requirement exists in the CAD area for an end-user-oriented programming paradigm.

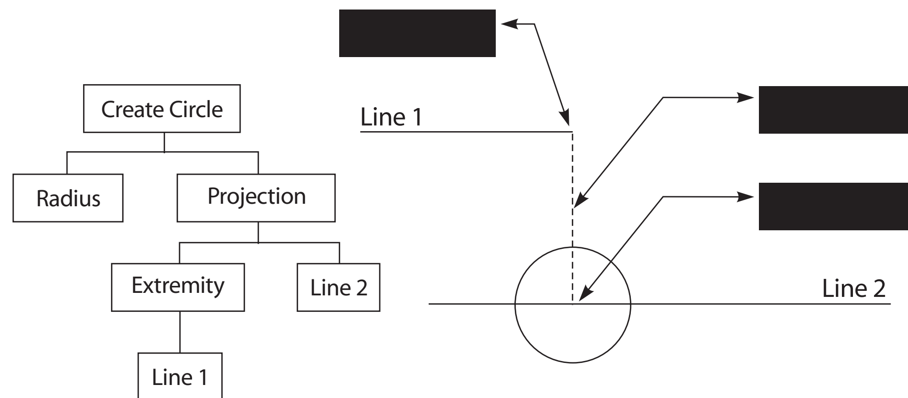
The second reason that the PBD approach may be easily implemented in the CAD area is the kind of dialogue language CAD systems support.

S
R
L

V:\002564\002564.VP
Monday, December 18, 2000 2:00:49 PM
TNT Job Number: [002564] • Author: [Lieberman] • Page: 138

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE **7.2**



Example of a structured task in CAD.

CAD models, or technical drawings, are very different from pictures or artistic drawings: they must conform to strong rules depending on the application area (mechanical design, architectural area, etc.). When designing such drawings, draughtsmen perfectly know the relationships that must exist between the entities of their model, and they want to have the capability of expressing these constraints in their design process. Since the early beginning of CAD, every CAD system provides commands that enable the expression of such constraints. Geometric constraints are so specified by means of geometric operators (e.g., *middle_of, starting_point, projection_of . . . onto . . .*). Numerical constraints are specified by *display calculators* that provide both algebraic operators (e.g., +, -, *, /) and geometric functions. These functions (e.g., *distance_of, angle_between, radius_of*) refer to model entities as parameters and return numerical values that, in turn, may be involved in numerical expressions. Therefore, CAD system interfaces enable users to specify explicitly every constraint that shall hold between objects, and CAD users are accustomed to specifying such constraints. Just recording these constraints builds the basis of sequential imperative program recording.

Figure 7.2 shows a typical CAD task: assume we want to create a circle whose center is geometrically constructed as a projection of a line extremity on another line (the two lines are assumed to be already drawn). The goal/subgoal hierarchy, as shown on the left part of the figure, is often broken in CAD systems to allow users to make appropriate constructs. For example,

S
R
L

V:\002564\002564.VP
Monday, December 18, 2000 2:00:54 PM
TNT Job Number: [**002564**] • *Author:* [**Lieberman**] • *Page:* **139**

Color profile: Generic CMYK printer profile
Composite Default screen

this typical task may be achieved by the following sequence of CAD system primitives (italicized words denote CAD primitives):

- *Create a point* at the end of line 1 (creating point 1).
- *Create a vertical line* through this point (creating line 3).
- *Create a point* at the intersection of this line and line 2 (creating point 2).
- *Create a circle* whose center is point 2 and radius is 10.

This explicit constraint-based definition of geometrical entities is very natural in CAD systems. Rather than direct manipulation, it involves command-operand dialogue style.

7.2.2 Variational and Parametric solutions

While every modern CAD system supports this kind of constraint-based definition of entities, constraints recording appeared much more recently. Beside the MEDUSA system (Newell, Parden, and Parden 1983), which provided for constraint-recording capabilities in 1983, the generalization of this feature appeared in the late 1980s. At this time, a new generation of systems appeared on the market; they were able to record these constraints, to change the numerical values involved in these constraints, and to compute the new model resulting from these values. These systems, often called *dimension-driven* systems (Roller 1990), have a twofold data structure and a twofold behavior. On the one hand, users may build (or change or compute) the displayed model. On the other hand, users may ask for visualization of the constraints and the numeric values that are involved in the example design. This information, which stands for the program in the PBD terminology, is displayed in some conventional symbolic way—for instance, through dimensioning. Then, users select the values they want to change and enter new values, and the system automatically computes the new model that corresponds to the same constructive process, or to the new solution of the same set of constraints.

In fact, dimension-driven systems proceed from two different approaches: declarative and imperative (Pierra, Potier, and Girard 1994). *Variational systems* hide the declarative program. Users build the example and specify the constraints, either explicitly or implicitly. These constraints are recorded as a set of equations, and some solver derives the solution. Variational geometry lies in geometric problem solving. The solution may

S
R
L

V:\002564\002564.VP
Monday, December 18, 2000 2:00:54 PM
TNT Job Number: **[002564]** • *Author:* **[Lieberman]** • *Page:* **140**

Color profile: Generic CMYK printer profile
Composite Default screen

be unknown from the user. Once constraints are stated, with some approximate geometric description, the solver tries to compute a solution. This approach corresponds to the popular sketchers that are available on most recent CAD systems: users draw some freehand sketch of a 2D model, and the solver computes the exact model after constraints are stated. The user interface is very friendly. Lots of methods have been used to solve these constraints. Most efficient methods are based on graph reduction (Bouma et al. 1995 ; Owen 1991).

Despite an actual progress, this approach suffers from three intrinsic weaknesses. First, the set of equations has generally many solutions (exponential number), and only system specific heuristics have been defined so far to guess the "user intent." Very simple examples have been published (Bouma et al. 1995) in which the computed solutions were obviously not intended to be the right ones. Second, because of the heuristic-driven nature of solving processes, different systems should provide different solution for the same model. Finally, Pure variational systems are unable to capture the purely procedural constructs, such as Boolean regularized operations or sweeping. Therefore, every so-called "variational system" is in fact a hybrid system that is variational in 2D and mainly procedural in 3D.

Procedural systems, often called *parametrics,* address 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 every instance, characterized by its parameter values, to be generated automatically in a deterministic way." *Parametric systems* hide an imperative program. This program is often captured using the example design process: the CAD system "spies" on draughtsmen while they are designing their example. As long as the example model grows, the constructive logic of draughtsmen is captured. Afterward, the CAD system is able to replay the constructive logic, possibly with new input values. The internal representation of programs may be textual, but it is more generally based on data structures (Pierra et al.; 1994; Solano & Brunet 1994) such as directed acyclic graphs (Cugini, Folini, and Vicini 1988). The Pro-Engineer system (Parametric Technology Inc.) is the most popular example of this approach.

In the CAD area, the dimension-driven approach is so attractive that presently every competitive CAD system must provide such capabilities. This large diffusion proves the practical interest of the approach. It also proves that draughtsmen, end users, are able to generate parametrized shapes (i.e., real visual programs) without programming knowledge—that is not to say without any modification of their working process. Effectively, drawing shapes is slightly different from drawing *families* of shapes. Nevertheless, this activity does not stand at the abstract level of conventional

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

programming activity. Dimension-driven systems, or parametrics for short, largely facilitate the design of part family models. For collection or single shapes, just designing one shape provides for generating every family's shape.

7.2.3 Requirements for PBD in CAD

Most choices that have been made for our systems are governed by the domain area (CAD), especially users' habits and needs (particularly their explicit way for describing relations between objects), and by the goals of the PLUS (Parts Library Usage and Supply) project, in which our work took place.

The portability of parts libraries between different CAD systems is a major economic concern for CAD system users, component manufacturers, and CAD systems vendors. This portability would drastically increase the number of available part families on the different CAD systems and therefore would increase the quality and the productivity of the design process for assembly modeling. To allow such a portability, a whole set of concerns, known as the CAD-LIB approach, has been developed (Pierra and Aït Ameur 1994). They constitute the agreed basis of European and International standardization works (CEN/TC310-pr ENV 40004 and ISO/TC184/SC4-ISO 13584 P-LIB).

Besides an object-oriented data model for the exchange of parts library data, the PLUS project had to develop an approach for the exchange of part family geometric models. When the project started (in 1993), the parametric technology did not appear mature enough to be able to commit to the development of a standard exchange format for parametric data models. Therefore, the selected approach has been rather conservative. It consisted in developing a standard API (now available as ISO DIS 13584–31) associated with a Fortran binding. Every CAD system that supports some implementation of this standard API would be able to execute Fortran programs referring to this API. However, this approach was in fact less conservative than it might appear at first glance: the project also included the development of a PBD system that was intended to be able to generate these variant programs through pure graphical interactions.

This context defines the requirements that governed the EBP system design. First, the generation process should be deterministic and fully controlled by the designer: one and only one shape should be generated for every allowed value of the input parameters, and precisely the shape that corresponds to the part. Second, every kind of shape family that might be

V:\002564\002564.VP
Monday, December 18, 2000 2:00:55 PM
TNT Job Number: [002564] • Author: [Lieberman] • Page: 142

Color profile: Generic CMYK printer profile
Composite Default screen

described using some conventional way of programming should be able to design using this system. Finally, the system should be able to generate an external representation of its internal data structure in the format of a Fortran program conforming to the standard API.

Note that if the first requirement to follow a procedural approach without any implicit inference or heuristic mechanism is met, neither of the two last requirements were fulfilled by either the existing parametric systems or the existing prototype of PBD systems. While several systems support predefined repetitive pattern structures, none of them, as far as we know, supports general-purpose program/subprogram structure with graphical parameter passing mechanisms and recurrence-based iterations where each loop is defined through explicit recurrence relationships within the previous branch.

7.3 Toward a Complete Solution

In this section, we describe the solutions we adopted to build actual PBD systems in the CAD area. First, we briefly describe the system with a standard CAD point of view. Then, we focus onto the naming problem. Last, we expose how we reach a complete expressiveness into constructed programs. In this part, we focus on PBD control structure definition.

7.3.1 Classical 2D CAD systems

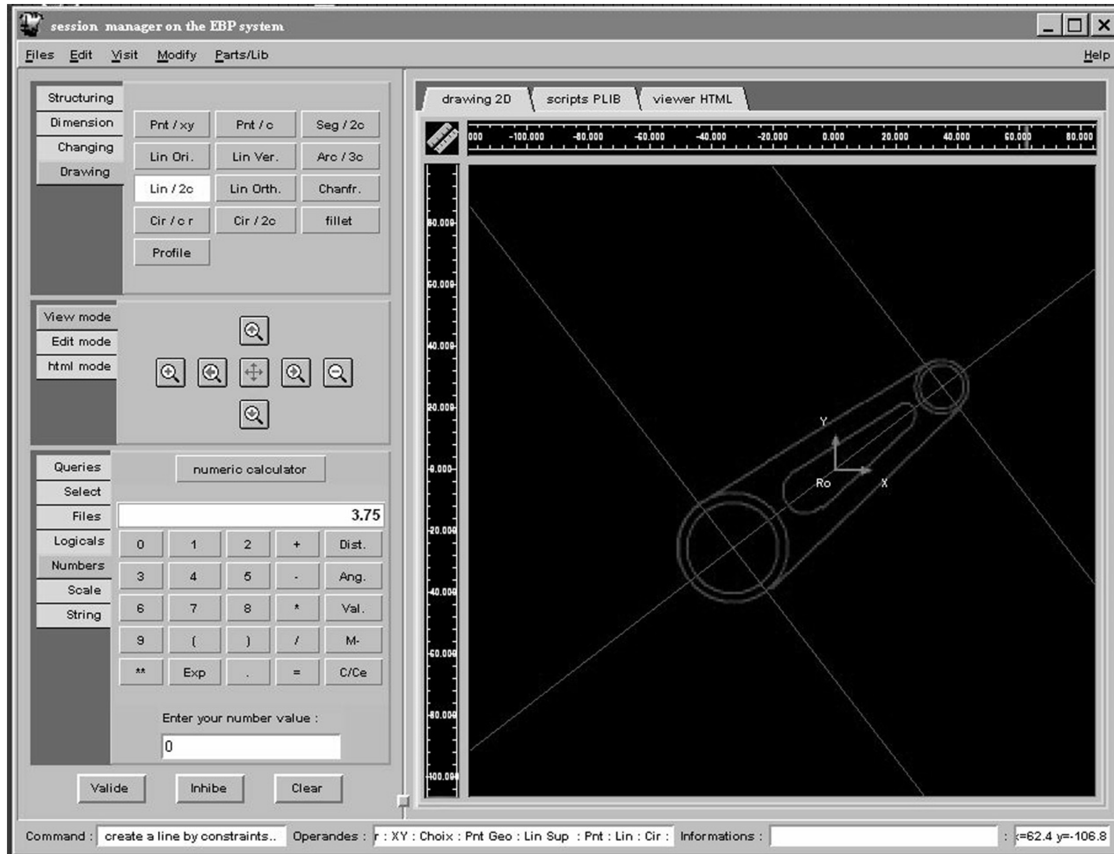
As mentioned in the introduction, we developed two CAD PBD systems, named LIKE and EBP. LIKE was built in late 1980s, to study the possibilities of including PBD in CAD. Written over the GKS system, it has been replaced in 1992 by EBP, which is more complete. In the following, we only describe EBP, because its present features include the results of the LIKE study.

The EBP system (Potier 1995) is a 2D CAD system. It manipulates simple geometric entities (points, unbounded lines, trimmed lines, circles, curves, etc.) and structured entities (composite curves, planar surfaces, and structured sets). Most constraints that result from technical drawing rules are supported. EBP provides a powerful display calculator that enables graphical inputs of both numerical and graphical expressions. Last, EBP allows model definitions through the use of menus and graphical interactions. The first version of EBP used the X-MOTIF interface and ran on Sun-Solaris (Sun Inc.) and DEC-Alpha (Digital) platforms. A new industrial version has been

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

Figure 7.3



Snapshot of the EBP system.

developed under Tcl-Tk, and a complete distribution that runs on PC Windows platforms is available at the URL http://www.lisi.ensma.fr.

Figure 7. 3 shows a snapshot from the “classic” CAD system EBP. On the right, we can see the drawing area, which displays the CAD model. The left part of the window is divided in three command areas. On the top, 2D classic commands are available, while in the middle, viewing commands are given. The third area is devoted to low-level entries, such as string or numerical entries. On this snapshot we can see a classical feature on a CAD system: the display calculator, which allows expressions including graphical functions (e.g., *distance between two entities*). EBP owns another calculator, the logical calculator. This feature, which shares some commonalties with a

S
R
L

V:\002564\002564.VP
Monday, December 18, 2000 2:00:56 PM
TNT Job Number: **[002564]** • *Author:* **[Lieberman]** • *Page:* **144**

Color profile: Generic CMYK printer profile
Composite Default screen

mechanism presented in Smith's (1977) Pygmalion system (bottom, left), enables CAD users, trained in using the numerical display calculator, to specify graphically the control predicate of their alternative or repetitive shape aspects. For instance, a fillet may be defined as dependent onto the constraint

val (line_1) > 2 x distance (point_1, point_2),

where *val(line_1)* means the length of line_1, and where *line_1, point_1* and *point_2* are objects that can be graphically selected on the example.

7.3.2 Naming and Its specificity in CAD

The major difference between simple scripts that record user interactions and real programs concerns the identification of the status of the values involved in interactions. For example, the integer value 3, which is an input in interactive mode, may have, in programs, three different statuses: (1) a constant, (2) a parameter (i.e., a value that must be provided again every time the program is triggered), or (3) an "internal" variable (i.e., a variable whose value results from a previous program statement). Many works in PBD addressed this problem, and the first of them, SmallStar (Halbert 1984, 1993), proposed an *a posteriori* explicit differentiation. The specificity of CAD allows us to choose a different solution, but it also highlights the problem of dynamic references to variables. We illustrate our arguments with the LIKE system, in which we first implemented our solutions. EBP has a similar object management.

The Problem of Naming

It is clear that, when it receives the integer value 3, the system cannot decide what should be the status of this value for the implicit program being constructed. The real challenge for PBD systems is to define conventional dialogue protocols that appear natural enough to users to enable a nearly implicit specification of the status of every object.

The great particularity of CAD systems that belongs to a general class of interactive applications we can classify as "editors" is that their main goal is to create new objects from user inputs. The example of standard parts is more precise: the goal of programs to be constructed is to draw (in fact, create into the CAD model) graphical entities from mainly numerical parameters. This fact implies, in terms of programming, that a program recorded during the use of such a system would essentially create new objects

V:\002564\002564.VP
Monday, December 18, 2000 2:00:56 PM
TNT Job Number: [**002564**] • *Author:* [**Lieberman**] • *Page:* **145**

Color profile: Generic CMYK printer profile
Composite Default screen

at any step. Its context (the set of objects it manipulates) would increase at any instruction. To respect this particularity, the LIKE system applies different rules for managing the three classes of objects.

Despite their nature (string, real numbers, or graphical entities), parameters of a program must be explicitly defined by the user, who is supposed to give a name, a question prompt, and a specific example value to each parameter. This constitutes a "parameter context." Furthermore, parameters may be selected for using menus. The way parameters can be defined is not very important: some informal experiences on users that we managed show us that our class of users (CAD expert users) was not afraid to have to explicitly define the parameters they used. Giving two examples differing by values of parameters (as in Tinker; Lieberman 1993), to let the system "infer" that these values were varying, was not considered to be efficient. They preferred doing it themselves.

Because we chose this explicit solution for parameters, its appeared natural to the one hand to consider "simple" objects, as numbers or strings, to be implicitly constants when the user enters them and, on the other hand, to consider graphical entities, which may be visually selected on the display, to be implicitly internal variables.

During the recording phase, the LIKE system performs two tasks. First, it manages the dynamic context: objects are automatically introduced in the context as soon as they are created by the CAD system. Therefore, the CAD system notifies the PBD manager for each created object. It also provides the database reference of this object. Second, it ensures value/reference substitution. As soon as the user selects them, the PBD manager must identify the objects by comparing them to the values contained in its context. The corresponding variable references are stored in the program. This filtering process also permits the PBD manager to forbid any access to objects that are not considered as visible (i.e., that are neither in the dynamic context nor in the parameter context). Obviously, this automatic naming mechanism is based on numbering (later we will see the consequences of this point).

During the running phase, symmetrical actions are done: dynamic context management is performed in the same way, allowing the inverse reference/value substitution needed by the running process in the CAD system. Each time an object is created, its reference is stored in the right variable. Then its value is sent to the CAD system, for each program reference, to the corresponding variable.

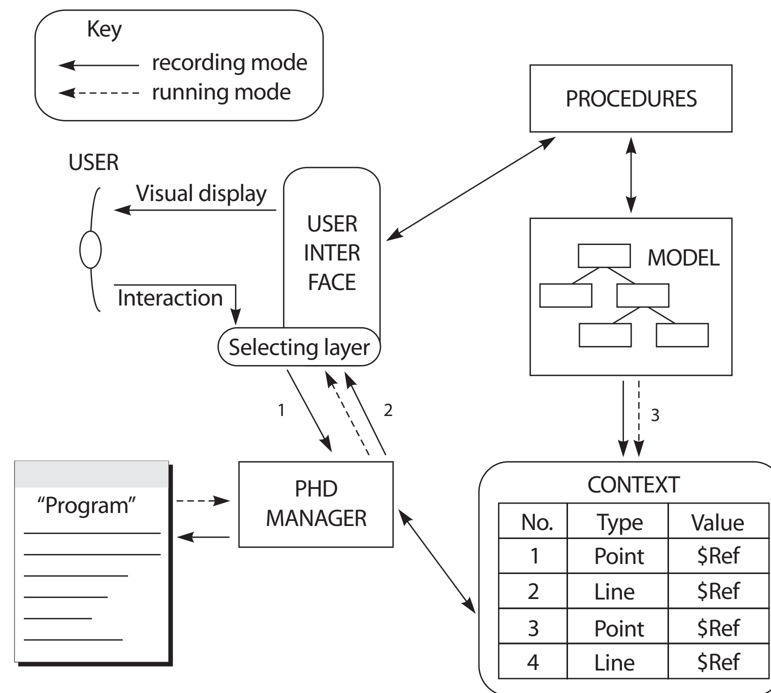
In recording mode, each input (command or value) from the user goes through the PBD manager (arrows 1 and 2 in Fig. 7.4). The values that correspond to references on model entities are captured after the identification layer (selecting layer) of the user interface, where locator positions are

S
R
L

V:\002564\002564.VP
Monday, December 18, 2000 2:00:57 PM
TNT Job Number: [002564] • Author: [Lieberman] • Page: 146

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE **7.4**



Architecture of the LIKE system.

replaced by entity identifications. Commands are recorded in the program, together with operators of the display calculator that are considered commands. Each entity identifier is replaced by the corresponding variable, by means of looking to the dynamic context. The dynamic context manager is notified for every entity creation (arrow 3), and new internal variables are created and assigned to entity identifiers.

The context is dynamically extended during the whole process of the example design. It enables, during the program construction, the substitution of values (system identifiers) by names (numbers) in the recorded program. It permits, while the program is running, the substitution of names (number) by values (system identifier) that are to be sent to the system. The consistency of these rules is obviously based on the assumption that every running of the program will generate the same types of objects in the same order as in the example. In particular, when debugging programs, each change that results in a different number of *object creations shifts* the "

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

naming" of further objects. Consequently, each reference on these shifted variables has to be modified. Moreover, the debugging suppression of actions that generate objects requires the checking and the possible invalidation of further actions that use these particular objects. All this management must be performed by the PBD manager.

Another consequence is more important. The validity of example-based programming is based on one prerequisite: it is assumed that running will *repeat* the example. Values may change, but actions and control flow have to be the same. This condition, quite natural for purely sequential programs, no longer has meaning in the structured programming context. When conditionals are used, the two branches cannot be assumed to create the same number of objects. For loop structures, the number of iterations varies from any execution. Hence, creation number of variables may vary. We will see in the following section how this problem may be solved.

Ambiguity Removal and System Determinism

Another specificity of CAD systems has consequences on deducing and translating what the user means. It relates to the ambiguity of geometric constructs. This ambiguity is not specific to variational systems; it is, in fact, intrinsic to geometry, in which every constraint that involves a circle or a distance corresponds, in general, to two different solutions.

For example, building a line that starts on a given point and ends tangential to a circle leads to two possible solutions, as pointed out in Figure 7.5(a). This problem is perfectly solved in interactive geometric design. Most CAD systems use the mouse-click position of each input object to discriminate the possible constructs. They assume that the designer approximately knows the expected solution. For example, in the case of Figure 7.5(b), the pointing click position results in the choice of the upper line solution.

If this user-friendly dialogue convention shall be maintained at the user interface level; that is, to design the example/program, it cannot be stored in the parametric program where, for different values of the parameters, the position might correspond to a different solution. This problem of constraint-based geometric constructs is also well known in variant programming (Roller 1990), in which programming languages were used for defining part family models.

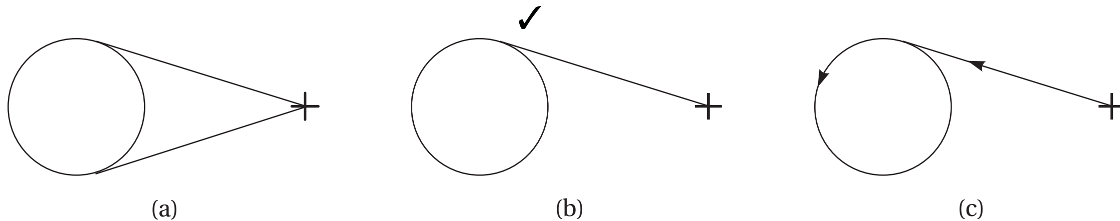
In such parametrics programs, context-free ambiguity removers are defined. In the target API, ambiguity removal is defined by topological informations: geometry entity orientation and in/out for circles. For example, Figure 7.5(c) shows the unique solution of the function *Line_by_Point_and_Circle* from Figure 7.5(b), which is defined by coherent

S
R
L

V:\002564\002564.VP
Monday, December 18, 2000 2:01:01 PM
TNT Job Number: **[002564]** • *Author:* **[Lieberman]** • *Page:* **148**

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE **7.5**



An example of the ambiguity of geometric constructs: (a) the two possible solutions, (b) interactive solving with a click, and (c) a good solution for programs.

entities orientation. EBP ensures the translation from the context-sensitive information captured at the user interface level (the position of the mouse click) into a context-free information recorded in the program.

In EBP, every entity is oriented according to the way it was constructed. Lines are oriented from their origin (first point) to their extremity (last point), circles are oriented according to the given points (when defined by three points) or counterclockwise (when created by center and radius), and so on. During program recording, the system translates the proximity disambiguity mechanism into the orientation mechanism as follows:

- With the proximity mechanism, the system calculates the right construction.
- Then, the system checks for the circle orientation.
- If this orientation is consistent with the solution (as in Fig. 7.5[c]), the system records the drawing without modification.
- If not, the system records the following sequence: change circle orientation, draw the line, and change the circle orientation again.

This mechanism, which remains unknown from users, is perfectly determinist.

7.3.3 Expressiveness

In Cypher, Kosbie, and Maulsby (1993), characterizing PBD systems is done by quoting whether they support either kind of structure as loop,

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

alternative, and so on. We considered the problem from the opposite point of view: what mechanisms give a full expressiveness of programs?

Following the structured programming paradigm, we decided to include in our systems every classic control structure: sequence (implicitly), alternative, iteration (general loop structure), and subroutines allowing recursion. Two kinds of feature must be quoted: internal support of these structures by the program mechanisms (essentially by the context manager) and interactive definition of the structures by users.

Full Control Structure Support

Thanks to our implicit context management (Girard and Pierra 1990, 1995), our systems may include a full-control structure support. More precisely, conditionals, iterations and subroutines are fully supported.

Applying the encapsulation principle to the context of subroutines is straightforward: each subroutine has its own dynamic context (internal variables) and parameter context. Subroutines are stored together with the example values for their parameters. This allows running them during program construction, as soon as the user selects parameters, for example, through a menu.

In our present implementation, subroutines have unique output parameters that consist in the set of every graphical entity created by each subroutine. These sets are inserted as unique variables within the dynamic context of the calling program. The content of this variable is a pointer toward the dynamic context of the embedded program. When the embedding program is run, the expressions that define the current parameters are again evaluated, and the PBD manager performs parameter matching before triggering the embedded program.

The language is considered as block structured, so each block may access its embedding context. Each control structure is considered as a block and is associated with its own context. Yet, this context is considered as a whole, and is inserted as a unique (pointer) variable in the embedding context. Nevertheless, this context is structured. Each branch corresponds to one context, which is defined during the example phase. In running mode, this context is created again before performing the control structure. The context of an *If Then Else End_If* structure consists of two branch contexts. Both exist in example definition, but only one is used when the program is run (according to the value of the control predicate).

In a loop structure, the loop context consists of a list of contexts, each one associated with one turn. Specific associations between objects of each context allow the definition of recurrence relationships and, in so doing, general loops (see the next section).

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

This block structure of the dynamic context allows the use of a stack mechanism during example phase or running phase. While executing in either mode, the visible context is the stack of the dynamic contexts in use at the present time. So, implicit references may be searched through the whole stack. In terms of user dialogue, this means that the user is able to refer to the objects of the current branch but also to those of all the including branches.

PBD Control Structure Definition

Purely interactive definition of control structure by PBD is a rather different problem than supporting any structure in PBD systems. As for other choices, we always have privileged deterministic solutions and explicit mechanisms triggered by users. Among our systems, EBP is the most complete for control structure definition.

Conditionals and iterations require Boolean expression definition, which is made explicitly through both numerical and logical calculators. The two alternate branches of conditionals may be defined either in a consistent way (running again the instance with alternate parameter values) or in some inconsistent way (drawing the two solutions with the same parameter values).

Several iteration features are provided: set iterations over rubber-band rectangle selections and multiple geometric transformations are straightforward in CAD systems and supported by EBP. As for ambiguity removal, context-dependent information (the two corners of the rubber-band rectangle) are translated into context-free information (the set of entity names that were referenced by this shortcut). But much more general features, such as *Repeat n times, While loops* and *Repeat . . . until loops* are also provided. They allow recurrence-based definitions, in a purely interactive way.

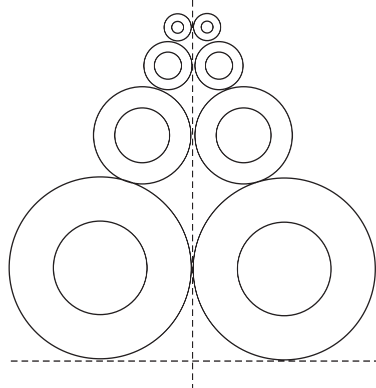
Let us illustrate an interactive REPEAT UNTIL definition. Assume we want to design the drawing shown in Figure 7.6. It is made of circles decreasing by a rate of one-half radius at each loop, to reach a given minimum. In classical CAD systems, the constructing process for this drawing is very tedious: while every CAD system owns geometrical transformation that may, for example, duplicate the left part of this figure from the right part, none of them owns the specific geometric transformation that allows building this suite of circles. Each step must be done independently from the previous one, with the repetition of actions such as “creating a circle tangential to a line and a circle, with a radius being half of the one of another circle.”

Nevertheless, the definition of a general algorithm to build this drawing is possible. The program to be constructed might be defined as an iteration

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE 7.6



A (rather complex) drawing.

(to obtain one column of circles) and a symmetry (to obtain the other one). We only need to know the value for the biggest radius and the value for the minimum radius (to stop the construct). Any loop but the first might be defined as follows: *build a first circle, tangential to the corresponding circle in the previous loop, tangential to the central vertical axis, and with a radius half that of corresponding circle in the previous loop; then create a second circle whose center is the same as the first circle, and whose radius is half the previous circle.* As shown in Figure 7.6, the first loop has a slightly different specification because the constraints that define the first circle refer to entities of the embedded context (the horizontal line). But, in fact, the only difference is in the nature of the manipulated objects.

If we analyze this problem, three points are to be highlighted: (1) in one loop, objects are defined from objects created during the previous loop and during the current loop; (2) actions performed during the first loop are the same as actions performed during the next ones—the only difference concerns the reference to the objects belonging to the previous loop, which must be found in the embedding context (in the example, the first circle is tangential to both existing lines); and (3) recurrence relationships may be fully defined during the second execution of the first loop actions, just by asking the user for the actual object that shall be used for every referenced object in the first loop. it may be the same object (the vertical line in our example) or any object that has been created during the first loop (in our example, the horizontal line must be replaced by the first circle of the first loop).

S
R
L

V:\002564\002564.VP
Monday, December 18, 2000 2:01:10 PM
TNT Job Number: **[002564]** • *Author:* **[Lieberman]** • *Page:* **152**

Color profile: Generic CMYK printer profile
Composite Default screen

These three remarks are the basis of the user interface and the dialogue conventions of the EBP system. The definition process is the following. Two commands are provided: REPEAT and UNTIL. The first one initiates the loop description and allows the user to record the actions of each loop. The second one stops the loop definition and starts the condition definition. Because of the PBD process, another step is necessary, which consists of defining the recurrence relationships between one loop and the following. After some experience with users, it seemed more usable to do this recurrence definition prior to the condition definition.

Figure 7.7 illustrates the different steps of this definition. Before the loop, the user has to define the two required parameters (Radius and Minimum, which are displayed as menus options by the system, rectangles in the figure) to create the two lines and then select the REPEAT command (Fig. 7.7[a]).

At this point, the first loop can be demonstrated, with full access to the embedding context. So the user selects the create circle command (the rounded rectangle "cercle"), points out the two lines, and clicks on the menu item that denotes the radius. The first circle is created (Fig. 7.7[b]). Then, he or she constructs the second circle: the same command is needed, and the user must point out the first circle (in so doing, they are assumed to have the same center). To give the right radius, the user must define a formula with the calculator. So, he or she activates the calculator command ("calcul"), gives the radius of the circle by clicking on the radius command (rayon) and pointing out the circle, and clicks on the different buttons "/", "2," and "=." The system can build the second circle (Fig. 7.7[c]).

The first loop is now complete. To follow the definition of the iteration, the user selects the UNTIL command. The system automatically switches to running mode, to perform the second loop and to help the user define the recurrence relationships. Every command performed during the first loop is run, and the system echoes the embedding context objects. To define the relationships, it asks the user for either picking the same (which defines a constant reference during the whole iteration) or picking another object that has been created during the previous loop (which defines a recurrence relationship, reevaluated for any loop). Every other entity selection is refused by the system. The same mechanism is provided for any expression, allowing the definition of new expressions (in our example, the expression for the radius of the first circle). Figure 7.7(d) illustrates this interactive phase.

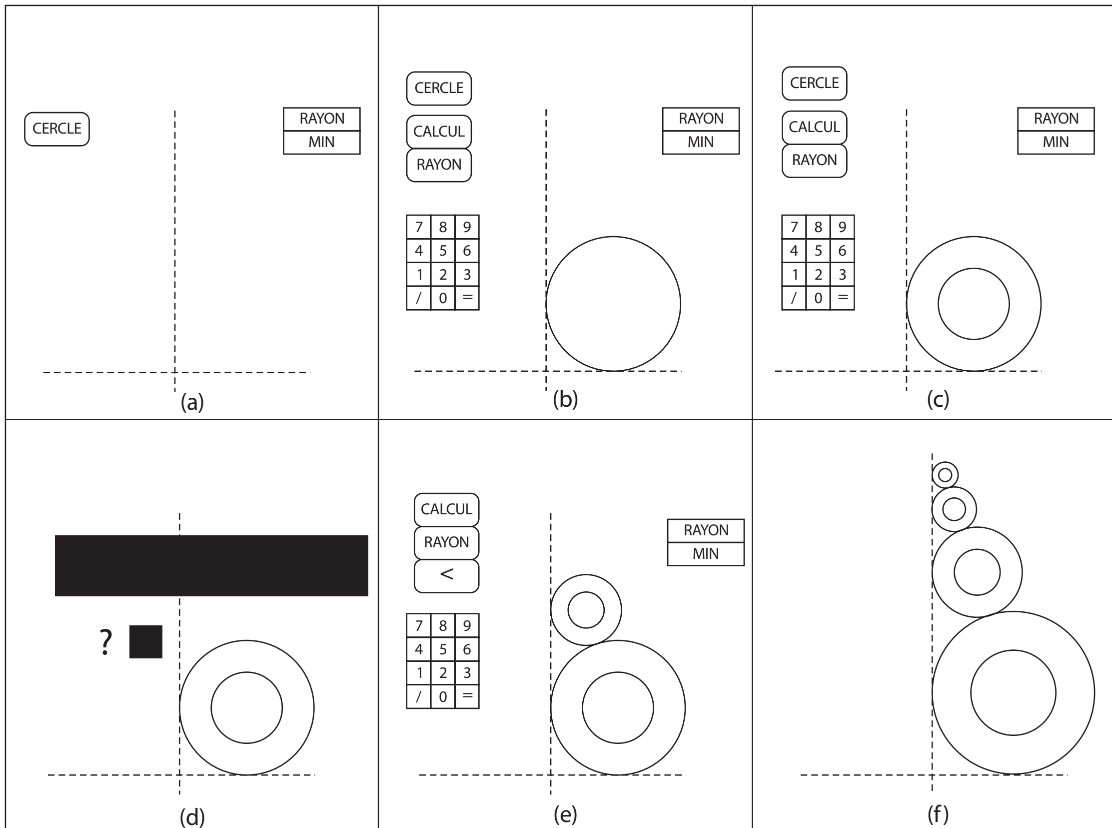
Once each object reference has been confirmed or changed by some reference to entities from the first or the current loop, the system asks the user for the definition of the control expression. This solution implicitly defines

S
R
L

V:\002564\002564.VP
Monday, December 18, 2000 2:01:10 PM

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE 7.7



Interactive definition of a loop.

a recurrence-based relation that is consistent for the whole iteration. Using the display calculators (Fig. 7.7[e]), the user can access every object from the iteration and the embedding context. In our case, he or she must activate the logical calculator ("calcul" command), and demonstrate the expression: the radius ("rayon" command) of the last circle (the user points the circle) is lower ("<" command) than the minimum (click on the menu item that denotes the minimum, "Min"). Clicking on the "=" box fires the calculus and implicitly asks EBP to handle the execution of the loop (the iteration definition is complete).

S
R
L

V:\002564\002564.VP
Monday, December 18, 2000 2:01:15 PM

Color profile: Generic CMYK printer profile
Composite Default screen

After that, the system runs the remaining loops until the controlling expression fires. Each recurrence relation is evaluated from any loop. The result is given in Figure 7.7(f). Finally, the user must define a symmetry of every created circle (a native operation in CAD systems), which involves the whole context of the iteration (any circle).

7.4 True Explicit PBD Solutions

EBP provides for true explicit PBD solutions. In this section, we detail the features of PBD. In the first subsection, we explain the specific commands that have been added to the CAD system to make PBD, focusing on recording and running programs. In the second subsection, we detail the features that make EBP an actual programming environment.

7.4.1 Fully Integrated PBD Systems

Compared to CAD systems, the EBP system is able to describe every collection of shapes that might be described by conventional programs, even if they contain repetitive or alternative shape aspects. Compared with the existing PBD systems, (1) users may specify on the example every kind of conditional or recurrence-based loop structure without any textual manipulation, (2) the system does not use any inference mechanism but explicit dialogue conventions that are fully integrated within the usual dialogue of the CAD system, and (3) the system generates neutral forms of programs that may be run, later on, on every other CAD system that supports a standard application program interface (API).

Visual Programming by Demonstration is achieved by "command recording" mode. This means that, unlike some parametrics systems in which "programs" are directly related to example values (e.g., the function *line_2_points* directly refers to the example point), in EBP, programs (which are called *instances*) are separated from examples. Relationships between example values and program variables are given through the dynamic context of the program. This mechanism, usual in programming languages, ensures an indirect reference from the program variables to their current values (in the example). It provides more independence between the PBD manager that deals with variables and the CAD system in which example values are CAD database pointers. When the program is rerun (e.g.,

V:\002564\002564.VP
Monday, December 18, 2000 2:01:15 PM
TNT Job Number: **[002564]** • *Author:* **[Lieberman]** • *Page:* **155**

Color profile: Generic CMYK printer profile
Composite Default screen

during modification), the EBP variables are not changed, but their addresses, stored in the dynamic context, are updated.

After recording mode activation *(Record Instance),* the EBP system "spies" on the user and builds an instance. Switching to running mode allows the system to run this instance *(Apply Instance).*

The only additional commands from traditional CAD systems are designed to *RECORD* / *NAME* / *LOAD* / *APPLY* instances and to *DEFINE* / *READ* / *WRITE* / *ENTER* parameters. Adding control structures requires other specific commands (IF / THEN / ELSE, REPEAT / UNTIL), as stated in the previous section.

A typical session of EBP would be as follows: after piece analysis (What are the parameters? Where are the dependencies?), the user begins PBD recording. He or she defines the parameters and then draws an example, using the parameters instead of "direct values." The DEFINE command opens a window, where a name is given (it is displayed every time the program is run). The ENTER command enables entering the values of the parameters for the example. These values are entered through the CAD system interface, and their types define the parameter's types. The WRITE/READ command enables recording/getting values on/from a file that will be linked to the program instance. This is used for recording the allowed sets of parameter values for part families. As soon as parameters are defined, they are displayed in a menu where the user can pick them up, for example, when defining expressions using the display calculators.

When the example is complete, the user can save the resulting instance, change some parameters, and try a new run. Recording in files values for parameters is very easy; this allows rapid testing. Recorded instances are included in a pop-up menu and are usable with minimal effort.

The LOAD command selects an instance, and the APPLY command runs it. Note that these commands may be selected both outside and inside the recording mode. In the first case, a model will be created into the CAD system database. EBP appears as a macro-by-example facility. In the second case, the APPLY command is recorded as a call routine in the embedding instance, and EBP ensures parameter passing. In both cases, after the *Apply* command has been selected, EBP displays each parameter name and waits for a value. This value is defined using the whole CAD system user interface. This means that, when applying the instance in recording mode, parameter value definition consists of every expression that involves entities or parameter values of the embedding instance. These expressions are stored in the embedding instance, and the expression value stands for the actual parameter value in the embedded instance.

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

7.4.2 An Actual Programming Environment, but for users...

EBP is a complete programming environment that provides for every usual debugging facility, in a programming-by-example style. Every interaction with programs is done through example interaction. Generated programs are shown in some specific window that is only displayed upon user request. A special menu, the *Visit menu,* allows rerunning the instance.

Intelligent UNDO/REDO and Program Modification

During both program recording and debugging, every modification may be done into the program. Successive UNDOs enable returning to previous steps. Then, some additional constructs may be done, and some steps may be modified or deleted. EBP manages the program dynamic context to ensure that addition/deletion does not change references to variables. When REDOing some command that references some deleted entity, EBP asks the user for a new entity to replace the previous one.

Note that UNDOing is quite surprising at first use: the system does not record every user interaction, such as pulling down a menu to choose a user command or pointing at a graphical object. Instead, it records CAD actions as a whole, such as “creating a circle . . . .” So, UNDOing and REDOing are done in terms of CAD actions.

UNDOing and REDOing control structures are even more difficult. Because of control expressions, the definition of these structures is rather different from their simple execution. So, *Undo*ing such a structure “undoes” the structure as one step, and REDOing “redoes” it as a whole, too. Stepping into a control structure is a debugging facility, as stated in the next section.

Debugging on Example

To have a complete programming environment, debugging facilities must be provided to the user. As in every debugging environment, the EBP manager enables programs to be run step by step or until their end or to be reinitialized. But specific needs result from PBD.

The first one relates to dynamic context management. At any point of a rerun, inserting or deleting actions is possible. In the first case, it may result in adding new variables in the context of the program. This is perfectly assumed by the context manager, which adjusts every implicit variable reference. In the second case, consequences may be more dangerous: while debugging a whole program, deleting actions often result in not creating

S
R
L

V:\002564\002564.VP
Monday, December 18, 2000 2:01:16 PM
TNT Job Number: [**002564**] • *Author:* [**Lieberman**] • *Page:* **157**

Color profile: Generic CMYK printer profile
Composite Default screen

objects that may be used later in the program. After analyzing the user tasks and habits, we decided to provide two mechanisms to help good debugging: the user can *replace* or *delete* an action. When the user replaces an action, he or she is assumed to give a new action whose resulting object has the same semantics as the first one. This is the case when a wrong parameter was first used. For example, in our example of Figure 7.7., the second circle has a radius that is either half the radius parameter or half the first circle's radius (the same value, in fact). But, in the logic of the needed algorithm, only the second interpretation is right; the first would result in a surprising construct.

When the user deletes an action (assuming this action really builds an object), the system looks in the program and searches for every object whose construction depends on the result of the deleted action. Then, it can help the user modify by deleting invalidated actions or replacing the nonexisting object by new one. In the same way, control structures can be inspected and modified, as can actions for control or recurrence relationship definition.

Another specific feature of EBP must be explained. It may be called "debugging on example." Because the textual representation of programs is never supposed to be displayed, debugging "virtual" programs might appear difficult. Fortunately, the example always give an input/output interface with the program: It is possible to visit the program *until one entity is drawn.* The user graphically selects this entity, and the program is run until it is drawn. The user may then make every modification on the example/program, before rerunning the remainder of the program. This point is very important, because it states that concrete representation of programs is not needed to achieve a complete programming environment.

Program Generation and New Features

EBP was designed to produce standard parts-portable program libraries (ISO13584-compliant Fortran programs reference the ISO 13584–31 API). Figure 7.8 illustrates this code generation.

Within the PLUS project, EBP is already used to generate the library of a bearing and linear system supplier. Some other exchange formats have been generated, including AutoLISP, Java classes, and a STEP-compliant parametric exchange format that was recently proposed (Pierra et al. 1996). Future work includes the development of an industrial product for an ISO 13584-compliant file generator and 3D extensions.

Another point under study in our laboratory is to split from structured PBD to object-oriented PBD. Nontrivial problems, such as new PBD classes'

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE 7.8

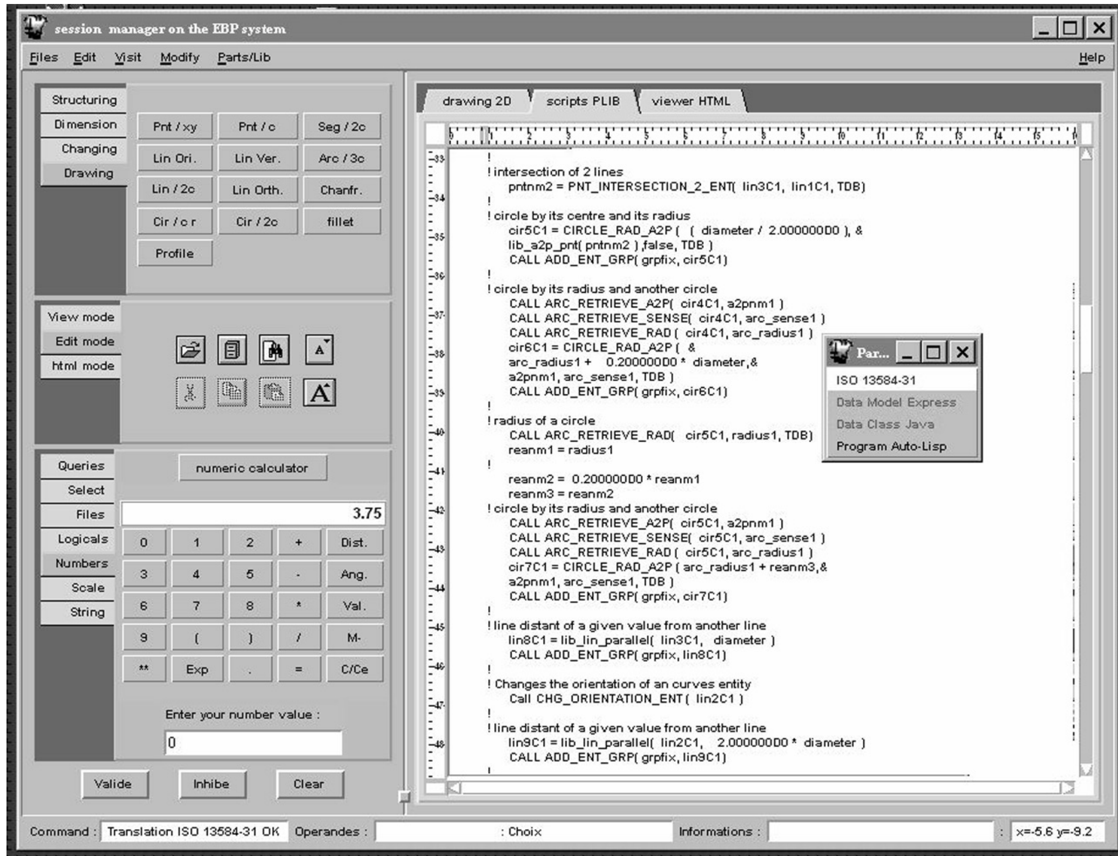


Illustration of program generation.

definitions, and “PBD in the large” have to be addressed. Recent advances have been made for the first issue (Texier and Guittet 1999), and need to be extended.

7.5 Conclusion

In this chapter, we have presented a suite of PBD environments for CAD systems. These systems

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

- do not use any inference mechanism to ensure full user control onto the (implicit) program,
- support every control structure of imperative programming without any direct interaction with the program,
- are able to generate conventional programs that may be used on different CAD systems, and
- constitute a full PBD environment.

From the PBD point of view, the EBP system proves that, at least in some application areas in which system users have particular skills, complete PBD environments may be developed. “Complete PBD environment” means both computational completeness of generated programs and real debugging with example facilities.

From the CAD system’s point of view, this approach proves that parametric CAD systems, which are already very successful for sequential (or simple repetitive, pattern-based) parametric design, may be extended to support the parametric design of every conditional or repetitive shape aspect.

From a user interface viewpoint, usual interactive systems are generally only sequential systems. The EBP system suggests extending the dialogue command language toward recurrence-based repetitive command constructs. It also proves that very powerful macro-with-example recorders may be developed.

References

Bouma, W., I. Fudos, C. Hoffmann, J. Cai, and R. Paige. 1995. Geometric constraint solver. *Computer Aided Design* 27, no. 6: 487–501.

Cugini, U., F. Folini, and I. Vicini. 1988. A procedural system for definition and storage of technical drawings in parametric form. In *Eurographics* ’88. Eurographics.

Cypher, A., ed. 1993. *Watch what I do: Programming by demonstration.* Cambridge, Mass.: MIT Press.

Cypher, A., D. S. Kosbie, and D. Maulsby. 1993. Characterizing PBD systems. In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

Cypher, A., and Smith, D. C. 1995. KidSim: End user programming of simulations. In *Human factors in computing systems (CHI’95),* Denver, May 7–10). New York: ACM/SIGCHI.

S
R
L

V:\002564\002564.VP
Monday, December 18, 2000 2:01:17 PM
TNT Job Number: **[002564]** • *Author:* **[Lieberman]** • *Page:* **160**

Color profile: Generic CMYK printer profile
Composite Default screen

Girard, P., and Pierra, G. 1990. End user programming environments: Interactive programming-on-example in CAD parametric design. In *EUROGRAPHICS'90,* (Montreux, Sept 3–7). Cambridge: Eurographics.

———. 1995. Structures de contrôle générales en programmation par démonstration. In *Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'95),* Toulouse, October 11–13, ed. P. Palanque. Cépaduès.

Glinert, E., ed. 1990. *Visual programming environments.* IEEE Computer. Los Alamitos, California.

Halbert, D. 1984. Programming by example. Ph.D. diss. University of California, Berkeley.

———. 1993. SmallStar: Programming by demonstration in the desktop metaphor. In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

Jackiw, R. N., and W. F. Finzer. 1993. The Geometer's Sketchpad: Programming by geometry. In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

Lieberman, H. 1993. Tinker: A programming by demonstration system for beginning programmers. In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

Myers, B. A. 1993. Peridot: Creating user interfaces by demonstration. In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

Myers, B. A., D. Giuse, R. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal. 1990. GARNET: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer* 23, no. 11: 71–85.

Nardi, B. A. 1993. *A small matter of programming: Perspectives on end-user computing.* Cambridge, Mass.: MIT Press.

Newell R., G. Parden, and P. Parden. 1983. Parametric design in MEDUSA System. Paper presented at CAPE'83, Amsterdam, April 25–28.

Olsen, D. R., and J. R. Dance. 1988. Macros by example in a graphical UIMS. *IEEE Computer Graphics and Applications* 12, no. 1: 68–78.

Owen, J. 1991. Algebraic solution for geometry from dimensional constraints. In *ACM symposium on foundations of solid modeling,* (Austin, Tex., May 8–10). New York: ACM/SIGGRAPH.

Pierra G., and Y. Aït Ameur. 1994. Logical model for parts Libraries. ISO-CD 13584-20.

Pierra, G., Y. Aït Ameur, F. Besnard, P. Girard, and J.-C. Potier. 1996. A general framework for parametric product model within STEP and parts library. In *European PDT Days* (London, April 18–19). London: PDTAG-AM.

Pierra, G., J.-C. Potier, and P. Girard. 1994. Design and exchange of parametric models for parts library. In *27th International Symposium on Advanced Transportation Applications, ISATA'94* (Aachen, Germany, October 31–November 4).

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

Potier J.-C., 1995. Conception sur exemple, mise au point et génération de programmes portables de géométrie paramétrée dans le système EBP. (Ph.D. diss.): LISI/ENSMA, Université de Poitiers.

Roller, D. 1990. Dimension-driven geometry in CAD: S survey. In *Theory and practice of geometric modeling*. Berlin: Springer.

Sassin, M., 1994. Creating user-intended programs with programming by demonstration. In *IEEE symposium on visual languages,* (St. Louis, Mo., October 4–7), ed. A. L. Ambler and T. D. Kimura. : IEEE.

Shah, J. J., and M. Mäntylä. Parametric and feature-based CAD/CAM: Concepts, techniques and applications. New York: Wiley.

Smith, D. C. 1977. *A computer program to model and stimulate creative thought*. Basel: Birkhauser.

Solano, L., and P. Brunet. 1994. Constructive constraint-based model for parametric CAD systems. *Computer Aided Design* 26, no. 8: 614–621.

Texier, G., and L. Guitter. User defined objects are first class citizens. In *Third Conference on Computer-Aided Design of User Interfaces (CADUI/99),* (Louvain-la-Neuve, Belgium, October 21–23), ed. J. Vanderdonkt and A. Puerta. The Hague: Kluwer Academic.

Wilde, N. 1993. WYSIWYC (What You See Is What You Compute) spreadsheet. In *IEEE Symposium on Visual Languages, August 24–27, 1993*. Bergen, Norway: IEEE.

S
R
L