Programming by Demonstration, a user-oriented programming paradigm for graphic systems[1]

P. Girard, G. Pierra

Laboratoire d'Informatique Scientifique et Industrielle
Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Site du Futuroscope - B.P. 109 - 86960 FUTUROSCOPE Cedex - France
e-mail : {girard,pierra}@ensma.univ-poitiers.fr

[1] *paru dans* Girard P. & Pierra G. Programming by Demonstration, a user-oriented programming paradigm for graphic systems. *Eurographics Workshop on Object-Oriented Programming Paradigm, Maastricht*, 1-2 Sep. 1995. p. 217-232..

Programming by Demonstration, a user-oriented programming paradigm for graphic systems

P. Girard, G. Pierra

Laboratoire d'Informatique Scientifique et Industrielle
Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Site du Futuroscope - B.P. 109 - 86960 FUTUROSCOPE Cedex - France
e-mail : {girard,pierra}@ensma.univ-poitiers.fr

Abstract

Computer graphics development does not only need efficient approaches and programming paradigms for software designers. It also requires end-users to be able to program their own system. The goal of this paper is to show that the Programming by Demonstration (PbD) approach, where system end-users specify programs by designing examples of their behaviour, is now mature enough to constitute an attractive paradigm for graphic programming environments.

Programming by Demonstration involves two aspects. First, it requires the capability for the system to associate each value, in the example, with one object (constant, internal variable or parameter) in the program. This constitutes the *dynamic context management* of the program. Second, it must assume the definition of dialogue conventions (*dialogue protocols*), which enable users to (implicitly) specify programs, with as few as possible changes to the user skills and to the interactive command language.

We show in this paper how structuring the dynamic context provides for structured program context management, and how some "natural" dialogue conventions may be defined in order to enable the graphical specification of every control structure, including the general purpose recurrence-based loop structure, which is not supported, as far as we know, by any other PbD system.

The system we have developed proves that the PbD approach is now a mature enough technology, enabling graphic systems end-users to really program their system.

Keywords

Programming by Demonstration (PbD), Example-Based Programming (EBP), Human-Computer Interaction (HCI), Computer Aided Design (CAD), Parametrics.

INTRODUCTION

During the past few years, lots of advances have been achieved to reduce the programming skills and the abstraction level which are required for computer use and programming. Visual programming [8] permits users to graphically select both the functions and the variables which constitute programs. Programming by Demonstration, or PbD [5], allows direct interaction with values that represent the program variables instead of their abstract names, or icons. Nevertheless, one major weakness of these systems remains: they are not computationally complete, i.e., none of them support the general purpose control structures such as recurrence loops that have been proved to be necessary [1] [16] to express any computable function.

The goal of this paper is twofold. First, it is to introduce an application area of PbD, namely Computer Aided Design (CAD), where the PbD approach has proved to be particularly useful; every modern CAD system supports some PbD facilities. Second, it is to present a PbD system that achieves, in this application area, the computational completeness of the produced programs.

The structure of this paper is as follows. Section 1 presents the related works in the area of PbD. In section 2, we discuss the application of the PbD approach in the context of CAD. Parametric systems follow an imperative approach: they record the user’s intent. Variational systems follow a declarative approach: they only record the constructs and use an inference mechanism to interpret the user intent. None of them supports the general control structures we present in this paper. Section 3 briefly presents the LIKE system. Section 4 investigates the problem of dynamic context management. We show how, in the LIKE system, context structuring enables consistant context management for both subroutines and control structures. Section 5 presents the dialogue protocol we have implemented for subroutines, control structures, and namely general purpose recurrence-based loop structures.

The last section presents some evaluation of the results we get and present the new developments we have initiated on the basis of the achieved results. We show in conclusion that PbD provides a mature paradigm for the development of end-user oriented programming facilities in computer graphics.

1. Related works

Programs are made of structured sets of functions that involve objects (constants or variables). In conventional programming, programs are defined as texts. Functions are known by their names, and objects are known either by their literal values, when they are constant objects, or by their identifiers, when they are variable objects. Programming consists of two steps. (1) Identifying and naming objects to be involved. And, (2) designing structured sets of functions that perform the target task.

Both steps require abstraction skills: function names stand for dynamic process that shall be mentally simulated by the programmer, and variable identifiers stand for values that may change from any runnings. Mastering such abstractions of dynamic processes is all but natural for human beings: it results from experience and extensive training. This is why learning conventional programming requires so much time.

1.1. From Visual programming to Programming by Demonstration (PbD)

In the 1980s, several works have laid foundations for visual programming (reprinted in [8]). The main idea of this new approach is to replace words by pictures. While textual programming offers no more help for understanding programs than meanings of words, visual programming uses pictures as a much more intuitive representation of this meaning. Based on greater expressive power of well selected drawings, compared to words, visual programming has introduced, in different fields, pictorial representations for both variables and program functions.

Using examples to design programs makes up the second step towards end-user programming environments. Instead of *selecting functions* and *picking up variables* on which each function shall apply, users (programmers) *do* functions on *values* which stand for program variables. This programming paradigm was born in Pygmalion [37]. It has been largely analysed in Halbert's PhD Dissertation [9], and has been formalised in Myers' works [21] under the name of Example-Based Programming. Then, various workshops took place, and have lead to state of the art reports on end-user programming [20][5]. The Cypher’s compilation introduces the term of Programming by Demonstration (PbD), which seems to be largely accepted today[2] [5]. The main idea is to avoid the abstraction level of variables by enabling the user to deal only with specific values of these variables. On example design, PbD systems analyse inputs, and build the program able to generate the example, and some variants of this example. This analysis may be done by immediate recording of the sequence of actions (**programming-with-example** in Myers terminology) or by a differred global inferencing mechanism over the whole example value (**programming-by-example**).

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 design of programs. So, Halbert’s SMALLSTAR [9] for iconic desktop programming, Myers’ PERIDOT [19] and GARNET [22], Olsen’s MACROS BY EXAMPLE [25] and Miyashita’s IMAGE [18] for UIMS programming, Stasko’s DANCE [39] for algorithm animation, Wilde’s WYSIWYC

[2] Following Cypher, we will consider to be equivalent the two expressions ‘Example-Based Programming’ and ‘Programming by Demonstration’

Spreadsheet [43] or Geometer's Sketchpad [11] and Sassin's ProDeGE+ [35] in drawing systems, have proved in several different fields the efficiency of this approach.

1.2. PbD and Program constructs

Two points are largely quoted in Cypher's compilation [5]: **inference** and **program constructs**. Cypher argues that *how to infer the user's intent* is the main challenge confronting PbD. The system needs to determine the user's *intent* in performing the action, in order to convert a recorded action into a program to perform that action. Most of PbD approaches try to increase their capabilities for inferring user's intent. As we will see in section 2, maybe as a result of the application domain we are dealing with, our approach is slightly different. Though, instead of inference (which is quite 'magic' for users, and often ambiguous on the basis of only one simple example), we prefer using precise and perfectly predictable **dialogue conventions** which allow users to fully express their intent. Section 5 presents some of these conventions.

Program constructs, and more precisely subroutines, conditionals and loops, are the second challenge for PbD systems. In fact, they are largely restricted in the existing systems. In 1990 [21], this fact was considered as a major drawback for PbD systems. Cypher's 'Watch what I do' has a good look on these features in existing PbD systems, that we will summarise below.

In current PbD systems, subroutines with parameters are proposed in Chimera [12], Geometer's Sketchpad [11], Macros-by-example [15] and AIDE [28]. Nevertheless, parameters are limited to input parameters; so doing, we should consider them as 'macros', instead of real subroutines. Recursivity may only be found in Geometer's Sketchpad [11], but the lack of control expressions in that system makes the recursive definition incomplete: the user is required to give the recursion depth he/she wants the system to apply at any runnings.

Conditionals are frequently supported (Peridot [19], Turvy [14], Metamouse [15], TELS [17]), but are restricted to exception handling (object existence test, geometrical constraint satisfaction, ...). Last, iterations are mainly *set iterations* [9] which consist of repeating actions on object lists (Peridot [19], Eager [4]) or in texts (TELS [17]). Some systems search for repetitive patterns in user actions, and infer repetitive tasks (Eager[4], Turvy [14], Chimera [12], Peridot [19]), or allow a predefined number of object creations (Peridot [19], Metamouse [15]). General recurrence-based loops are not supported. The only system that introduces restricted forms of recurrence relations (chosen from predefined classes of recurrence patterns) is Peridot: '*the constraint in all items after the first is made to refer to the corresponding item in the previous cycle* [5].

Finally, the systems which may be considered to be the most complete drop out the major characteristic of PbD, that is directly interacting with the example: they ask users for textual modifications when introducing control structures (Tinker [13], Smallstar [9], Geonode [41]).

In the next section, we discuss the slightly different state of the art in the CAD area.

2. PbD IN COMPUTER AIDED DESIGN

As Nardi quotes, Computer Aided Design (CAD) systems *'allow end users to create useful applications with no more than a few hours of instruction'* [23]. We will see in this section that the same assertion holds to use basic PbD capabilities.

2.1 From conventional CAD systems to parametrics, that are implicit PbD systems

In previous CAD systems, parametric programs, able to generate different cognate shapes according to the values of some parameters, were largely used. They enabled, for instance, factorisation of the shape descriptions of whole families of standard parts (screws, bearings, ...). In conventional CAD systems, such programs were textually described, often in FORTRAN or in the C language. When triggered, these programs created geometric entities by means of Application Programming Interfaces (API). This approach has drastically changed in the new generation of CAD systems: the user interactively creates some particular shape, and the system records the corresponding program.

In parametric systems, the CAD system "spies" the draughtsman while he/she is designing his/her example. As long as the example model grows, the constructive logic of the draughtsman is captured. Afterwards, 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 [30][38] such as direct acyclic graphs [3]. The Pro-Engineer® system[3] is the most popular example of this approach.

In the CAD area, the parametric approach is so attractive that, at the present time, all competitive CAD systems shall 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. In fact, drawing **shapes** is slightly different from drawing **families of shapes**. Nevertheless, this activity does not stand at the abstract level of conventional programming activity.

2.2. CAD, a suitable area for PbD

The reasons why PbD is so mature in the CAD area are twofold. CAD systems are intended to allow users, generally draughtsmen, to design geometric models. Therefore, thanks to this geometric nature, the objects which are managed by the system have an intrinsic visual representation. This fact is a well-known criterion for efficiency in visual programming techniques [2][36].

The other reason is the kind of dialogue language that is supported by CAD systems. CAD models, or technical drawings, are very different from pictures or artistic drawings: they shall comply with strict rules depending on the application area (mechanical design, architectural area, and so on). 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 to express these constraints in their design process. Therefore, each modern CAD system provides commands which enable the expression of such constraints. Geometric constraints are specified by means of geometric operators (e.g., *middle_of*, *starting_point*, *projection_of ... onto...*). Numerical constraints are specified by *display calculators* which provide both algebraic operators (e.g., +, -, *, /) and grapho-numerical functions. These functions (e.g. *distance_of*, *angle_between*, *radius_of*, ...) take as parameters references to model entities, and return numerical values which, in turn, may be involved in numerical expressions. Therefore, CAD system interfaces enable users to explicitly specify all the constraints that shall exist between objects, and CAD users are used to specifying such constraints. Just recording these constraints builds the basis of sequential imperative programs.

Figure 2 (on next page) shows a typical example of technical drawings where two points were firstly created by literal coordinates issued from the keyboard, and where a circle was then created, centred on the middle point of the two points and which radius was equal to 1/4 of the distance that separated these two points (commands are underlined, references to model entities are between <>, literal value inputs from the keyboard are in quotation marks, and arrows show the entities newly created inside the model). Note that we don't make any difference between any command (e.g., <u>create</u> ...) and values selected in menus or obtained through expressions (e.g., <u>4</u>).

This example points out two important characteristics of the CAD area. The first one is that, thanks to the geometric nature of the manipulated entities, these entities have an *intrinsic visual appearance.* The second one is that CAD end-users naturally express relationships between objects, either by selecting them upon the display or by using display calculators. These two features explain why programming by demonstration may be so successful in this specific area.

While every modern CAD system support this kind of constraint-based definition of entities, constraints recording appeared much more recently. In the late 90's, a new generation of systems appeared on the market, which were able to record these constraints. These systems, often called *Dimension-driven* [34], have a twofold data structure and a twofold behaviour. On the one hand, users may build (or may change, or may compute) the displayed model, in fact, in the PdD terminology, the example. On the other hand, users may ask for visualization of the constraints and the numeric values which are involved in the example design. This information is displayed in some conventional way (for

3 Parametric Technology Inc. USA.

instance through dimensionning). Then, users select the values they want to change, enter new values, and the system automatically computes the model which corresponds to the same constructive process, or to the new solution of the same set of contraints.

• User interactions

create point coordinate
'12.5'
'0' → P1
create point coordinate
'32.5'
'0' → P2

create circle center radius
middle point
<P1>
<P2>
calculator
1
/
4
*

distance
<P1>
<P2>
= → C1

• Output

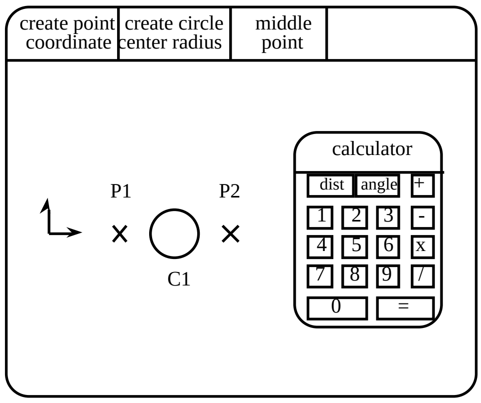

Figure 2 : A typical input sequence on CAD systems, and its result

In fact, dimension-driven systems proceed from two different approaches (declarative approach and imperative approach) [31], which may be formalized as follows. Declarative approaches [40], 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" [34]. 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 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 [6]. The popular 2D "sketcher", available on various CAD systems, corresponds to this approach. Unfortunately, there exist only partial solutions concerning specific problems [42].

The imperative approach 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", which defines 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.

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

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

we call the f_i functions the *parametric functions*.

It shall be noticed that, even if the set of constraints is in fact a program (either imperative or declarative, this program is never displayed. This is why the parametric approach is so popular: interacting with examples, users implicitly build programs.

This approach has also its weakness: the "program" is basically sequential. The systems based on the declarative approach do not accept any change in topology. The systems based on the imperative approach only support the basic ones.The only control structures are the implicit ones that already existed in the previous CAD system generations: repetitive geometric transformations (e.g., *rotate* 3 *trimes by* 10°), and set iterations where the same function ios applied to sets of entities, identified by rubber rectangles. Particularly, none of the systems which are available on the market supports the reccurrence loop construct that we have introduced in our system.

3. The LIKE system

The LIKE system is an experimental 2D CAD system, designed to investigate PbD in the field of CAD. It allows model definitions through the use of menus and graphic interactions. It provides a powerful display calculator that enables graphical inputs of both numerical and Boolean expressions.

Visual Programming by Demonstration is achieved by “command recording” mode. This means that, unlike some parametrics systems where "programs" are directly related to examples, for instance using constraint, in our approach, programs are separated from examples. Relationships between example values and program variables are represented through the dynamic context of the program.

As in other dimension-driven systems, programs may remain completely implicit: debugging facilities enable introduction of breakpoints by selecting entities where users want the program to stop. Then, changes may be done, e.g., creating new entities. The following variables are then renumbered, by the context manager in the context, and by the PbD manager in the progam. Finally, the remaining part of the program may be run. A program visualization is also provided in the format of sets of lines where each line corresponds to one user input (see figure 2).

The LIKE system does not use any inference. The user explicitely states its intent, and the system records it. Therefore, additional commands have been added to the host CAD system, in order to create/name/load/run programs, to create/name parameters, to define control structures (IF, THEN, ELSE, END_IF, and FOR, WHILE, END_LOOP) and to call and run subroutines (CALL_ROUTINE, RUN_ROUTINE).

4. Object management in the LIKE system

The major difference between simple scripts, that record user interactions, and real programs concerns the identification of the status of the values which are involved in interactions. For example, the integer value “3”, which is an input in interactive mode, may have, in programs, three different status. It may be (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 which value results from a previous program statement.

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. The real challenge for PbD systems is to define conventional dialogue protocols which appear natural enough to users to enable a nearly implicit specification of the status of every object. In order to achieve this goal, the LIKE system applies different rules for managing these three classes of objects.

Despite their nature (string, real numbers or graphical entities), parameters must explicitely be defined by the user, who is supposed to give a name, a question prompt, and a specific example value to each parameter. Furthermore, parameters may be selected for using from menus.

"Simple" objects, as numbers and strings, are implicitely considered as constants when the user enters them.

On the other hand, graphical entities, which may be visually selected on the display, are implicitely considered as internal variables. During the recording phase, the LIKE system performs two tasks. Firstly, 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 objet. Secondly, it ensures value/reference substitution. As soon as they are selected by the user, 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 which are not considered as visible (i.e. which are neither in the dynamic context, nor in the parameter context).

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 any 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.

Therefore, in the LIKE system, the program context is twofold. The *dynamic context* contains a numbered list of strongly typed objects (all the objects known by the CAD system). This dynamic context also (possibly) contains real variables which were explicitly created by the user, and variables which values are references to embedded contexts. The *parameter context*, displayed in menu, contains the (input) parameters of the program, together with their values for the running example. This structure is represented in the next figure.

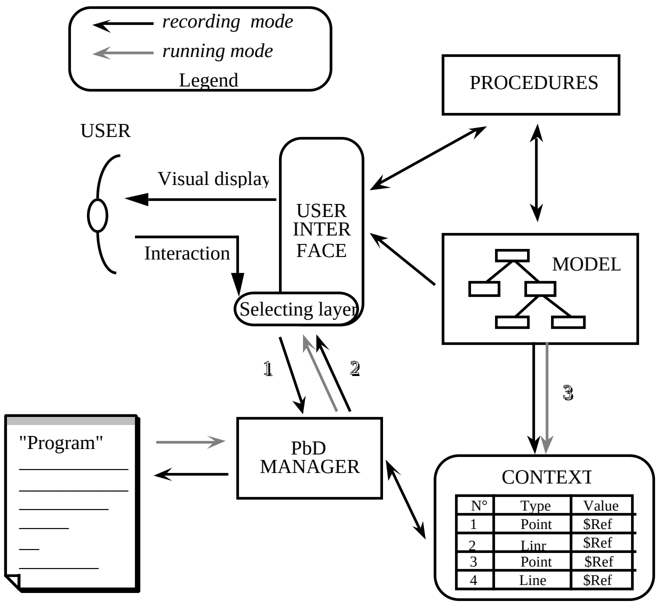

Figure 3: LIKE structure

The system provides debugging facilities. This means that the context management is really dynamic. Implicit variables are renumbered when actions are added or dropped in debugging mode. The system also identifies the functions that shall be invalidated by the deletion of previous constructive functions.

This management constitutes the dynamic context management of the program. 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, during the program 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 which results in a different number of *object creations shifts* the "naming" of further objects. Consequently, each reference on these shifted variables has to be modified. Moreover, the debugging suppression of actions which generate objects requires the checking and the possible invalidation of further actions which use these particular objects. All this management shall 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, has no longer 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 numbers of variables may vary. We will see in the following section how this problem may be solved, first for subroutines, and then for control structures.

In recording mode, each input (command or value) from the user goes through the PbD manager (arrow 1 and 2). 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 replaced by entity identifications. Commands are recorded in the program, together with operators of the display calculator which are considered as commands. Each entity identifier is replaced by the corresponding variable, by means of looking up the dynamic context. The dynamic context manager is notified for every entity creation (arrow 3), and new internal variables are created and are assigned to entity identifiers.

4.1. Context structuring for subroutines

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 as soon as they are selected by the user through a menu during program construction. Then the parameter context of this subprogram is shown to the user in a menu to enable assigning values to these parameters (see section 5.1).

In our present implementation, subroutines have unique output parameters which consist in the set of every graphical entities 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 towards 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.

4.2. Management of control structure contexts

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, 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. Users only specify the two first loops. During the second turn, the context of the first loop may be accessed to specify recurrence relationships (see section 5.3).When these two loops are specified, the PbD manager automatically switches to the running mode to perform the number of loops specified by the control predicate.

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 these of all the including branches. Figure 4 shows a structured program and its current context at the underlined point.

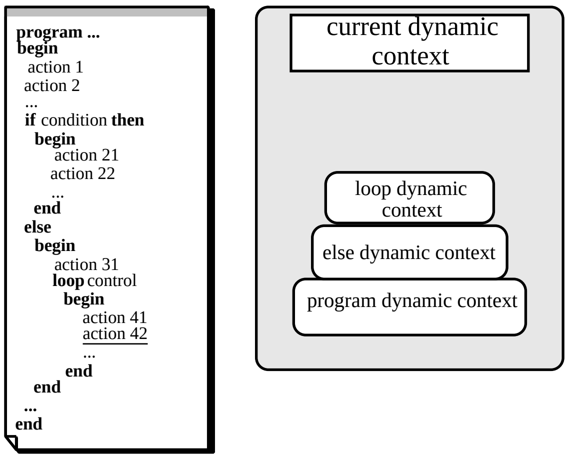

Figure 4: Visibility of the dynamic context

5. Dialogue protocol in the LIKE system

5.1. Controlling expressions and modes

While sequential visual PbD only requires "spying" user interactions, structured PbD requires PbD manager specific commands that enable opening (IF, THEN, ELSE, FOR, WHILE) and closing (END) structures and branches. Control predicates are specified by means of Boolean expressions. In alternative structures, they allow switching between two possibilities, the two branches of the alternative. In iterative structures, they control the end of the iterative process. They are computed before or after each iteration, and their value has the meaning of exit or continue.

In all cases, this requires PBD environments to be able to build, record and evaluate user defined expressions. This is the role of the display calculator that exists at the user dialogue level. Moreover, execution of iterative structures requires the PbD manager to shift from the recording mode (the mode where the system spies the user) to the running mode (the mode where the system replays the program).

5.2. Dialogue protocol for subroutine

When, during the design of the example, users want to call subroutines, they select the *call_routine* command and select in the displayed menu the name of the subroutine. The parameter menu is displayed to allow the user to assign current values to the parameters of the subroutine, and the subroutine runs.

Assigning values to parameters involves only graphical interactions. Users successively select each parameter and enter a value. If the parameter is a graphical object, this object shall belong to the context of the embedding program. If the parameter is a numerical or a Boolean object, it shall be introduced using the display calculator, and, if the corresponding expression involves variables (either internal variables or parameters of the calling program), the PbD manager checks if these variables belong to the context of the embedding program. These expressions are recorded in the program.

The geometric positionning of the output of the subroutine in coordinate space of the embedding program is done by a specific function available on the CAD system. The user defines a transformation (which may include shifting, rotating and/or scaling) and then "activates" this transformation. The CAD system computes the corresponding transformation matrix and applies it to each entity created by the subprogram.

5.3. Dialogue protocol for alternative constructs

In alternative structures, program statements are different in each branch. Therefore, two examples have to be provided. In some cases, the two branches may be defined from the same root example (e.g., to design two different screw heads, according to the screw diameter). In some other cases, two completely different examples are needed (e.g. to draw a circle centred on the intersection of two lines when they intersect, and to draw a parallel to both lines when they are parallel).

Adapting to both cases involves two user interfaces. When selecting an IF structure, the user is required to define, with the display calculator, the Boolean expression. This expression (e.g. *distance_from* Point_1 *to* Point_2 > 10) is evaluated, and the system opens the right branch. Nevertheless, the user may enforce the jump to the other case, in order to define the corresponding example. If the cases are successively defined, both sets of created objects appears on the display. However, the PbD manager forbids accessing the wrong context. When the definition of the second branch requires a completely different example, users define one branch and close (ENF_IF) the IF structure. Then, they change the program parameters, and require the running to be performed. According to the parameter values, the other branch is opened to permit the other example definition.

5.4. Dialogue protocol for loops

Halbert [9] splits iterations into two categories: *set iterations*, and *recurrent iterations.* Set iterations consist of applying the same sequence of actions to sets of objects: the first loop applies to the first object, the second one applies to the second one, and so on. Introducing this capability is straightforward on CAD systems, because it already exists in the interactive use. The group designation (e.g., rubber rectangle) permits the definition of the set. The action to be performed (e.g., to change its colour) is directly defined on group entities. The system generates a set iteration in the program.

Recurrent iterations are iterations where each loop is defined according to the previous one. There is a specific case of recurrent iteration which is also straightforward, namely the geometric transformation. It consists in rotating or shifting "n" times one (single or group-structured) object. Users define the transformation, build the numerical expression (that may refer to other objects), and select the object to be duplicated. Thanks to the structured management of the control structure context, this is also stored as an iteration in the program.

The specificity of the LIKE system is the capability to define general purpose recurrence-based loops, in two forms. (1) The *for* structure permits to define iterations whose number of loops is known before running the loop. (2) The *repeat* structure permits to define iterations which control predicate is based on the loop computed values.

In both cases, the iteration definition consists of four steps: (1) selecting the structure (*for* or *repeat*); (2) defining, on the example, the first loop; (3) defining, still on the example, the second loop; (4) defining the control predicate as a Boolean expression specified with the display calculator.

During the second loop, the system generates commands in the same order. It only requires the user to provide its operand(s) (designation of graphical objects). The context management of this second loop, called the *recurrence loop*, is very specific: *only the objects created or referenced during the first loop*, or *created by earlier steps of the second loops*, are visible (i.e. selectable). If objects are the same in both loops, these references are considered to be constant. If objects selected during the second loop belong to objects which were created during the loop, recurrence relationships are defined.

In the *for* loop, the control predicate is only allowed to involve objects that belong to the enclosing context of the iteration structure. In a *repeat* loop, only these objects, and the objects created during either the first loop or the second loop are allowed. Designation of an object created during the first loop is stored without modification in the program. Designation of objects created during the second loop stands for a recurrence reference to objects of the current loop and are propagated by the PbD manager from loop to loop.

When the loop structure is closed (*end_loop*), the system performs the automatic running of remaining loops.

Let us give an example. Assuming we want to draw a stack, each element height is half the previous one. This is a recurrence definition, and it cannot be built by geometric transformation. We want to start from a given square (already existing in the drawing), and to stop the iteration process when the height of the last element raise a given minimum (*min*). The result would be:

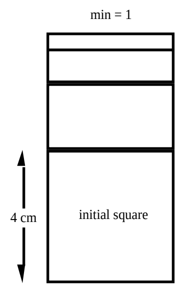

Figure 6: The required result

Assuming we have a 2D system (e.g., LIKE) that builds unbounded lines. We use two constructive functions, and one numerical access function available on a display calculator:

- *Create_parallel,* requires unbounded line and real value (distance). It creates a parallel new line from the given one (Fig 7).
- *Trim_line* requires three unbounded lines, and creates a bounded line (linear segment) between the intersection on lines 1 and 2 and lines 2 and 3 (Fig 7).
- *Length* is a numeric function which requires a bounded line and returns its length.

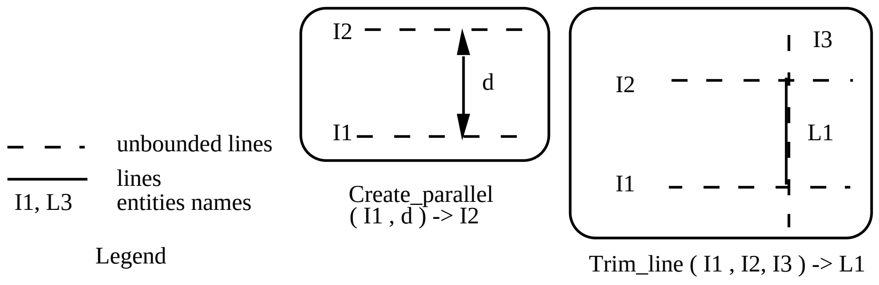

Figure 7: The two available constructive functions

Starting from the described initial state shown in figure 6 (initial square), we select the *repeat* command and define the first loop by the following sequence of actions (Fig. 8), that builds the objects pointed (Fig. 9):

create_parallel	**(I2, Length (L2) / 2)**	**->**	**I5**
trim_line	**(I2, I4, I5)**	**->**	**L5**
trim_line	**(I4, I5, I3)**	**->**	**L6**
trim_line	**(I5, I3, I2)**	**->**	**L7**

Figure 8: Example-based definition of the first loop

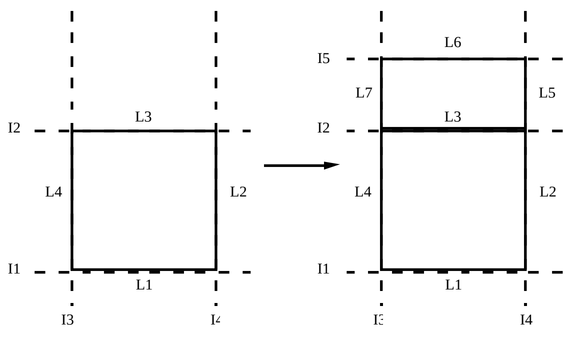

Figure 9: Initial and first loop graphical states

Then we select the *second_loop* command and design the next loop. The command are generated by the system that requires only their parameters:

```
create_parallel   ( I5, length ( L5 ) / 2 )     ->    I6
trim_line         ( I5, I4, I6 )                ->    L8
trim_line         ( I4, I6, I3 )                ->    L9
trim_line         ( I6, I3, I5 )                ->    L10
```

The PbD manager checks the references in the two loops. (1) Some references do not change (I3, I4). Therefore they are stored as constant references in the program. (2) Some references do change (I2 -> I5, I5 -> I6, L2 -> L5). Therefore these references must be interpreted as recurrence relationships. The program stored for the general loop is as follows, where *c•i* stands for a reference to the i-th object of the current loop, and *p•i* stands for a reference to the i-th object of the previous loop (recurrence relationships):

```
create_parallel(p•1, length (p•2)/2)   ->    c•1
trim_line          (p•1, I4, c•1)      ->    c•2
trim_line          (I4, c•1, I3)       ->    c•3
trim_line          (c•1, I3, p•1)      ->    c•4
```

Then the user selects the *until* command and the Boolean predicate controlling the iteration structure is expressed using the display calculator. This expression is defined by referencing an entity that results from the second loop:

length(L8) < 1.0.

Therefore it is considered as referring to the current loop:

length(c•2) < 1.0.

The final program where:

- I2, I3, I4, L2 are references to entities of the enclosing branch,
- each reference within the loop contains a couple [a, b], where a is the reference for the first loop (loop initialisation) and the second one is the reference for the iteration loop, is as follows:

```
repeat
          create_parallel   ( [I2, p•1] , Length ( [L2, p•2] ) / 2 )     ->    c•1
          trim_line         ( [I2, p•1] , [I4, I4], [c•1, c•1])          ->    c•2
          trim_line         ( [I4, I4], [c•1, c•1], [I3, I3] )           ->    c•3
          trim_line         ( [c•1, c•1], [I3, I3], [I2, p•1] )          ->    c•4
          until   Length ([c•2,c•2]) < 1.0
end_repeat
```

Figure 10 : The implicit final program

When the user selects the *end_repeat* command, the PbD manager iterates the loop structure until the control predicate turns to *true*. Therefore this new dialogue capability results useful even if the user is only interested in the example.

6. System evaluation and further research

The system presented in this paper has been developed under contract with an Aerospace company, in order to enable geometry generation for different components that belong to the same micro-waves guide parts families. Due to their machining process, the variant programs should contain both IF THEN ELSE constructs (when some dimensioning are less than a given minimum, some fillets should no longer exist and the topology changes) and loop structures (each part should contain a lot of small holes which number depends on the size of the component). None of the parametric or variational systems available on the market was able to create such parts families. The parts family used as test case for the system was developed both by conventional variant programming technique (in C language) and by PbD on the LIKE system. Both processes were done by engineers and involved one week for analysis. The program development itself required one month in C, and four days using the LIKE system. The system has been in use during about 6 months with the same kind of time ratio. Afterwards (and for other reasons) the Company changed its manufacturing process and the use of the system stopped.

The choice done in the design of the LIKE system was to capture the user interactions at the syntactical level. By reference to the well-known Seeheim model [27], user inputs were captured after the presentation component. The rational for this choice was to investigate the capability to add a PbD manager to pre-existing interactive systems with minimal modification of the kernel of these systems. Moreover, to enable new release of the support system without changing the PbD manager, LIKE only interprets its own specific commands (see section 3.2); semantics of native system commands are assumed to be unknown by LIKE. As a counterpart of this design choice, (1) it is not possible or not easy [33] to generate usual textual programs, able to be run on a different interactive system, (2) it is not possible to remove from the program the trace of the communication tasks such as zoom, and, (3) it is not possible to stop the program *before* creating any entity, without accessing its textual representation. Because the PbD manager is only notified for an entity creation *after* this creation, it is only possible to set breakpoints after such creations.

With the support of the ESPRIT PLUS Project[4] we are now developing a new PbD manager. This system is based on the same context management and dialogue protocol as the one presented in this paper, but it captures user inputs at a different capturing level. In this system, called EBP [33][30], user interactions are captured at the semantic level, i.e., by reference to the Seeheim model, just before the application interface component. This system (1) enables the generation of neutral programs based on a standard Application Programming Interface (API), [33] (2) removes all the traces of communication tasks and of non productive dialogue sequences, from generated programs, and, (3) like parametric or variational systems, enables debugging programs through interactions with the example. This new system is now running in 2D, and we are extending it in 3D, and implementing recursive function calls. This system has been tested, within the PLUS project, by the parts supplier intended to use it to generate portable parts libraries conforming to ISO CD 13584 [29]. The first time comparison has shown that the program development required time was divided by about 10 to conventional (textual) programming.

CONCLUSION

Computer graphics development does not only need efficient approaches and programming paradigms for software designers. It also requires from end-users to be able to program their own system. In the business application area, spreadsheets have already shown how user-oriented programming may be useful.

During the past ten years, lots of improvements have been achieved to elaborate a programming paradigm called Example Based Programming (EBP) or Programming by Demonstration (PbD), which enables end-users to develop implicitly graphical programs through the interaction with an example of these programs. Unfortunately the programs elaborated in that way were computationally incomplete.

Developing complete PbD systems needs either (1) to be able to manage program objects despite the fact that their number may change from any running, either (2) to develop dialogue protocols that enable end-users to graphically specify control structures or to develop inference mechanisms that identify implicit control structures.

In this paper we have presented the LIKE system, a computationally complete PbD system in the CAD area which is based on the dialogue protocol approach. The management of the program (implicit) objects is based on a structurization of the program *dynamic context*. The inputs of the control structure predicates are done on a *display calculator*. The dialogue protocol enables specification of recurrence relationships through purely graphical interactions. This dialogue protocol is efficient enough to enable dividing the time involved in program development by a factor of about ten.

[4] ESPRIT PLUS Project N°8984, Parts Library Usage and Supply

This paper proves that the PbD approach is now a mature enough technology, enabling graphic systems end-users to really program their system.

References

1. BOEHM C., JACOPINI G. : Flow diagrams, Turing machines and languages with only two formation rules, Comm. ACM, 9, 5, 1966.
2. CHANG S.-K., ICHIKAWA T., LIGOMENIDES P. : Visual languages, Plenum Press, New-York, 1986, 460p.
3. CUGINI U., FOLINI F., VICINI I. : A Procedural System for the Definition and Storage of Technical Drawings in Parametric Form, Proc. of EUROGRAPHIC'88, 1988, pp. 183-196.
4. CYPHER A. : Eager: Programming Repetitive Tasks by Example, Proc. of CHI'91, ACM, New Orleans, 1991,pp. 33-39.
5. CYPHER A. : Watch what I do, Programming by Demonstration, MIT Press, 1993, 635p.
6. DUFOUR J.F.: Programmation et résolution de problèmes de construction géométrique, BIGRE ,67, Jan. 1990, pp..136-147.
7. 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.
8. GLINERT E. : Visual Programming Environments, IEEE Comp. Soc. Press Tut., 1990, 600p.
9. HALBERT D. : Programming by example, PhD. Thesis, Berkeley, California, 1984, 121 p.
10. HIRAKAWA M., TANAKA M., ICHIKAWA T.: An iconic programming system: HI-VISUAL, IEEE Trans. on Soft. Eng., 16, 10, 1990, 1178-1184.
11. JACKIW R. : The geometer's sketchpad, Key curriculum Press, Berkeley, CA, 1992.
12. KURLANDER D. : Graphical Editing by example, PhD Thesis, Columbia University, 1993.
13. LIEBERMAN H. : Constructing Graphical User Interfaces by Example, Proc. of Graphics Interface '82, Toronto, 17-21 May 1982, pp. 347-354.
14. MAULSBY D. : The Turvy Experience, in CYPHER 93, pp. 139-270.
15. MAULSBY D., WITTEN I., KITTLITZ K., FRANCESCHUN V. : Inferring Grahical Procedures: the Compleat Metamouse, Human-Computer Interaction, 7, 1, 1992, pp. 47-89.
16. MILLS H. The New Math of Computer Programming, Comm. ACM, 18, 1, 1975.
17. MO D. : Learning Text Editing Procedures from Examples, M.Sc. Thesis, Calgary, 1989.
18. MYASHITA K., MATSUOKA S., TAKAHASHI S., YONEZAWA A. : Interazctive Generation of Graphical User Interfaces by Multiple Visual Examples, Proc. UIST'94, Marina del rey, 2-4 Nov., 1994, pp. 85-94.
19. MYERS B. : Creating User Interface by demonstration, Perspectives in Computing, Academic Press, 22, 1988, 276 p.
20. MYERS B. : Report on the end-user programming working group, in Languages for developing user interfaces,Myers Ed., Jones and Bartlett, Boston, 1992, pp. 343-366.
21. MYERS B. : Taxonomies of Visual Programming and Program Visualization, J. of Visual Lang. and Comp., 1, 1990, pp. 97-123.
22. MYERS B., GIUSE D., DANNENBERG R., VANDER ZANDEN B., KOSBIE D., PERVIN E., MICKISH A., MARCHAL P. : GARNET: Comprehensive Support For Graphical, Highly Interactive User Interfaces, IEEE Computer, 23, 11, 1990, pp. 71-85.
23. NARDI B. A. : A small matter of programming, perspectives on end user computing, MIT Press, Cambridge, Massachusetts, 1993, 157p.
24. NEWELL R., PARDEN G., PARDEN P. : Parametric Design in MEDUSA system, CAPE'83, Proc. of Computer App. in Production and Engineering, Amsterdam, 25-28 April 1983.
25. OLSEN D. R., DANCE J. R. : Macros by Example in a Graphical UIMS, IEEE Comput. Graphics and Appl., Jan. 1988, pp. 68-78.
26. OWEN, J.: Algebraic solution for geometry from dimensional constraints, ACM Symp. Found. Solid Modeling, Austin, Texas, 1991, pp. 397-407.
27. PFAFF G. : User Interface Management Systems, Proc. Workshop on User Interface Management Systems, Seeheim, Germany, Springer-Verlag, 1985, 224p.
28. PIERNOT P., YVON M., : The AIDE Project: an application-independant demonstrational environment, in Cypher 93, pp. 384-401.
29. PIERRA, G., AIT AMEUR, Y. : Logical Model for Parts Libraries, ISO-CD 13584-20, 1994.)
30. PIERRA, G., POTIER, J.C., GIRARD, P. : Design and Exchange of Parametric Models for Parts Library, 27th International Symposium on Advanced Transportation Applications, ISATA'94, Aachen, Germany, 31st October - 4 November 1994 , pp. 397-404.

31. PIERRA, G., POTIER, J.C., GIRARD, P. : The EBP system : Example Based Programming for Parametric Design, Workshop on Graphics and Modelling in Science and Technology, Coimbra, Portugal, 27-28 June 1994.

32. POSWIG J, VRANKAR G., MORAGA C.: Interactive anmiation of visual program execution, IEEE Symp. on Vis. Lang., 1993, pp 180-187.

33. POTIER, J.C., GIRARD, P., PIERRA, G., BESNARD F., "Génération graphique interactive de programmes de géométrie paramétrée", Premier Colloque International Productique Robotique du Sud Europe Atlantique, Bourges, 1-2 Juin 1995, (to be printed in RAPA, Hermès).

34. ROLLER D., SCHONEK F., VERROUST A. : Dimension-driven geometry in CAD : a survey, in Theory on practice of geometric Modeling, Springer Verlag, 1990, pp. 509-523.

35. SASSIN M. Creating User-Intended Programs with Programming by Demonstration, Proc. IEEE Symp. on Visual Languages, Saint-Louis, Missouri, 4-7 Oct. 1994, pp. 153-160.

36. SHU N. : Visual programming languages a perspective and a dimensional analysis, in Visual Languages, Plenum Press, New-York, 1986, pp. 10-34.

37. SMITH D. : Pygmalion : A Computer Program to Model and Simulate Creating Thought, Basel, Stuttgart, Birkhauser, 1977, 187p.

38. SOLANO L., BRUNET P., Constructive constraint-based model for parametric CAD systems, CAD, 26, 8, 1994, pp. 614-621.

39. STASTKO J.: Using direct manipulation to build algorithms animation by demonstration, Proc. of CHI'91, 1991, pp. 307-314.

40. 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.

41. VAN EMMERIK M. : Interactive design of parametrized 3D models by direct manipulation, PhD Thesis, Delft University, NETHERLAND, 1990, 141p.2.

42. VERROUST A.: Construction d'objets géométriques définis par des contraintes. BIGRE, 67, Jan. 1990, pp. 62-74

43. WILDE N., WYSIWYC (What You See Is What You Compute) Spreadsheet, IEEE Symp. on Vis. Lang., 1993, pp 72-76.