SRI International

CAN DRAWING BE LIBERATED FROM THE VON NEUMANN STYLE?

Technical Note 282

June 1983

By: Fernando C. N. Pereira, Computer Scientist
Artificial Intelligence Center
Computer Science and Technology Division

This paper appears in *Database Week Proceedings of the Business and Office Applications*, San Jose, CA (May 23-26, 1983)

The work described herein was carried out at the Computer-Aided Architectural Design unit in the Department of Architecture of Edinburgh University, and supported by the grants GR/B 79493 and GR/A 88774 from the Science and Engineering Research Council. The preparation of paper was partly supported by SRI International.

SRI International®
333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

Can Drawing Be Liberated from the Von Neumann Style?

Fernando C. N. Pereira
Artificial Intelligence Center
SRI International
March 1983

Abstract

Current graphics database tools give the user a view of drawing that is too constrained by the low-level machine operations used to implement the tools. A new approach to graphics databases is proposed, based on the description of objects and their relationships in the restricted form of logic embodied in the programming language Prolog.

Acknowledgments

The work described herein was carried out at the Computer-Aided Architectural Design unit in the Department of Architecture of Edinburgh University, and supported by the grants GR/B 79493 and GR/A 88774 from the Science and Engineering Research Council. The preparation of the paper was partly supported by SRI International. I thank David Rosenthal, Peter Swinson, David Warren and Daniel Sagalowicz for their helpful comments.

1. What's in a Drawing?

The problems discussed in this paper and the solutions proposed arose from trying to simplify the writing of computer-aided design applications by defining design databases in logical rather than physical terms. From this new point of view, a design database is just a collection of logical statements about the objects being designed.

Apart from the basic relationships between parts of the design, a design database must contain statements that specify the *views* relevant to different aspects of the design activity. Drawings are an essential element of the design activity. Therefore, a design database must include statements that specify a visual representation for the objects described by the database. Furthermore, it is generally the case that the most convenient form of user access to a design database is through drawings of its contents.

A graphical interface to a database must provide means for **viewing** the contents of the database and for **building** objects through graphical operations. These two classes of operations correspond directly to the usual database operations of query and update. It seems to be the case that, with current implementation techniques for graphical interfaces, the computation of the graphical view of the database is related only in an "ad hoc" way to the translation of graphical input into changes to the database.

The main objective of this paper is to show how to give nonprocedural, logical descriptions of the **structural mapping** between objects and their graphical representations that can he used both for graphical output and for graphical input. In other words, the same description is used to view graphically the contents of the database and to **identify** what objects are meant in user commands given in graphical terms. We will see that the logical statements describing the structural mapping are runnable programs in the logic programming language Prolog [6, 18, 14].

For a single representation to be used both for viewing the contents of a database and for identifying which objects in the database correspond to features of the drawing, the representation must be **invertible**. With an invertible representation, we can both go from objects to their images and from images to those parts of of the database that correspond to the images. Invertibility imposes heavy constraints on the representation method. For example, a representation method based on a general-purpose programming language with graphics primitives (**procedural representation**) will not be invertible, because it manifestly impossible to derive from features of a drawing the parts of the program that produced those features. In contrast, a **structural representation** with a one-to-one correspondence between graphics primitives in the drawing and data structures in the database is clearly invertible.

CMU's Glide language for 3D geometric modeling [7] and the Sticks & Stones functional notation for VLSI design [4, 5] are extreme examples of procedural representation languages. Geometric modeling systems based on instantiation of prototypes [3] and graphics systems based on the same idea [17] stand at a halfway position. A graphics system with a simple display file is a straightforward example of a pure structural representation.

The nature of the two representation methods just described has critical practical consequences. As I noted before, some form of invertibility, the ability to go from a drawing to its representation, is essential for interactive editing. Structural representations are invertible, and therefore allow interactive editing but do not support general operations for putting pictures together from other pictures, deriving pictures from prototypes, or specifying classes of objects to be edited. In contrast, procedural representations support general languages for specifying complex pictures but do not support facilities for recovering the constituents of a picture from the finished picture.

The examples in the two sections that follw show that logic programming can indeed help us to combine advantages of structural and procedural representation methods. This is due to the fact that suitably constructed Prolog programs are **multi-purpose** [18], that is, the procedures in the program can be used both to compute some results from some inputs or to compute which inputs would generate given results. In other words, suitably constructed logic programs are invertible.

2. A Logic of Pictures

The discussion in this section builds on two empirical observations about physical assemblies (and pictures):

1. Complex objects are made of simpler objects

2. New objects (or pictures) are often made by replicating, with some transformation, some existing object (picture).

I will give now a logical description of the "arch" of Figure 2-1[1].

The structure of the arch can be represented by the following statements or **clauses**:

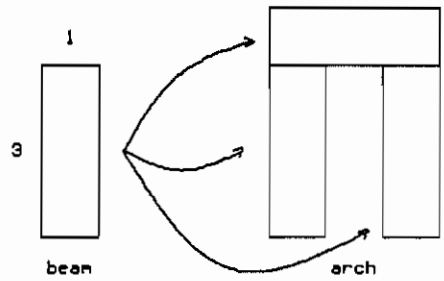

Figure 2-1: Arch

```
part(leftcolumn,arch).
part(rightcolumn,arch).
part(crossbeam,arch).

instance(leftcolumn,beam,id).                    (1)
instance(rightcolumn,beam,translate(2^0)).
instance(crossbeam,beam,rotate(-90,0^0):
                        translate(0^4)).

primitive(beam)

graphics(beam,line(0^0,1^0)).
graphics(beam,line(1^0,1^3)).
graphics(beam,line(1^3,0^3)).
graphics(beam,line(0^3,0^0)).
```

These statements are the application of a **predicate** (such as `part` to **arguments** (such as **arch** or `line(0^0,1^0)`). The arguments of a predicate are written after it in parenthesis, separated by commas. Each argument is a **term**, representing some object. Clause (1), for example, has the reading "`leftcolumn` is an instance of **beam** transformed by the identity transformation **id**."

A term may be:

- A **constant**, naming a specific object:

  `beam  line1  3`

- A **compound** term, which is a **functor** (the **principal** functor of the term) applied to a list of **arguments**, representing an entity built in the way named by the functor from the entities named by the arguments[2]:

  `3^5  translate(1^1)  line(1^1,2^2)`

  with principal functors `^`, **translate** and **line** respectively.

[1] All the examples are 2D for simplicity.

[2] For easier reading, a functor of two arguments (**binary**) may be written between its arguments instead of before the parenthesized list of arguments.

The notation I have just introduced is a fragment of the logical language of **definite clauses** [11]. Through the programming language Prolog, such definite clause **programs** can be run to check that objects satisfy given relationships or to compute which objects are in a given relationsbip with others.

The example program I have just given might seem no different from a program in some object definition language, such as Glide [7]. However, in such a language, names such as **line** or **translate** have a fixed interpretation. In contrast, in definite clauses a name, he it a predicate name, a functor name, or an atom name, has no predefined meaning. Thus, the clauses that were proposed to describe Figure 2-1 would have the same effect if **part** were replaced everywhere by **detail**. The names used in the example, however, are there for some reason: they were chosen to indicate the **intended** meaning of the clauses. That is, **part** is meant to represent tbe part-whole relationship, **instance** to represent the "transformed instance of" relationship, **translate** to represent a translation operator, : to represent composition of spatial transformations, and so on. But this intended meaning can only be made concrete if we add two ingredients:

- The viewing mapping between objects without parts (atomic objects) and graphic representations.
- The structural rules that determine the representation of complex objects from those of their components.

I will now discuss how to give structural rules, which are most important from the point of view of this paper. The question of the mechanisms for viewing the atomic objects will not be discussed further.

Structural rules are themselves definite clauses but are more complex than those shown above. The ones below are sufficient to complete the example.

```
in(Part,Obj) ⇐ basic_part(Part,Obj).
in(Part@Trans,Obj) ⇐
   instance(Obj,Proto,Trans),
   in(Part ,Proto).

basic_part(Obj,Obj) ⇐ primitive(Obj).
basic_part(SubPart,Obj) ⇐                    (2)
  part(Part,Obj),
  basic_part(SubPart,Part).
```

A clause of the form "P ⇐ Q, R, S" has the meaning that P is true if each of the **conditions** Q, R and S is true. Names starting with a capital letter are **variables**, standing for arbitrary entities. Thus, clause (2) above can be read as "for any SubPart, Part and Obj, SubPart is a basic part of Obj if Part is a part of Obj and SubPart is a basic part of Part." The relation **in(Part,Obj)** is intended to hold between an object Obj and any **virtual primitive** Part that is indivisible and contributes to the structure of Obj. A virtual primitive is either a **primitive**, satisfying the predicate 'primitive', or an object derived through the application of geometric transformations to a primitive.

To derive actual graphical representations we needboth structural rules and rules that relate the graphics of an object to the graphics of its parts. The relationship **visible**, defined hy the rules below, holds between an object and a graphic primitive if the primitive belongs to the graphics of one of the parts of the object.

```
visible(Obj,Prim) ⇐                          (3)
  in(Part,Obj),
  visible(Part,Prim).
visible(Obj@Trans,Prim) ⇐                    (4)
   visible(Obj,Prim0),
   apply(Trans,Prim0,Prim).
visible(Obj,Prim) ⇐ graphics(Obj,Prim).      (5)
```

The rules for the **visible** relationship can he understood as follows. Rule (3) states that Obj has graphics Prim if Part is in Obj and Part has graphics Prim. Rule (4) states that an object Obj transformed hy Trans (written as Obj@Trans) has graphics Prim if Obj has graphics Prim0 and Trans applied to Prim0 gives Prim. The predicate **apply** just applies geometric transformations, and therefore we do not need to examine it further. However, for what follows, it is important that **apply** be invertible in the limited sense of being able to compute one of its arguments, given the other two. This is not problematic, because a reasonable set of geometrical transformations will satisfy that property, being mathematically a group.

3. Pointing at Things

Pointing at a drawing with some pointing device[3] is the natural means of referring to the objects we want to operate on. In general, pointing operations are *ambiguous*, that is, several different objects could be meant by the operation. Whereas the identification of which objects are meant is, in general, a very difficult question, we will see later that, in the present framework, it is possible to

[3] In GKS terms, a **pick** or **locator** device [10], which is used to point at graphics or locations on the screen. I will leave aside the computer graphics issues of how the bottom-level graphics machinery derives from this input a hit set of atomic graphic objects (segments in GKS parlance).

derive reasonable guesses.

The usual situation in a complex drawing will he that a pointing operation will unambiguously identify a **hit set** of atomic graphical entities, and the identification task will be to go from those atomic objects to a set of objects in the database whose graphics include those graphical entities.

For power and generality, a pointing operation may be a combination of actual physical pointing with a specification of constraints (the **hit class**) to help identify the objects being referred to.

In conventional picture editors, identification is conceptually simple because of the limited repertoire of objects that may he built and referred to. Objects are either primitives (lines, text strings, etc.) or complex objects made of simpler objects, but in which the simpler objects have lost their individuality [10, 2]. Therefore, from the point of view of identification, we have a single level of atomic objects without any internal structure. In the same way, primitive text editors have commands to identify a particular character or a particular line in the text but cannot identify lines with a given internal structure. In picture or text editors of this kind, the pointing expressions of the user language are all of the form

<pointed positions, hit codes>

where the hit codes are used to select from all the (atomic) objects whose views contain some graphics in the hit set, those in some particular predefined subclass (for example, line segment, symbol).

Another kind of pointing, **specialization**, is available in some drawing systems [1]. The target set identified by a specialization is the set of all occurrences of a given subobject, where those occurrences have been created through instantiation of a common prototype. I will have more to say about prototypes and instances in Section 5. For now, it is enough to note that this kind of pointing operation may be seen as drawing an arbitrary distinction between objects that are the same from the user's point of view, but that do not come from the same prototype. For example, of two apparently identical arches X and Y, X could be an instance of an arch prototype, and Y an assembly of three instances of the beam prototype. X could be identified as a specialization of "arch," whereas Y could not. This shows particularly well how the internal organization of the drawing system imposes "ad hoc" constraints on the user language. At the machine level, instances of prototypes are stored in the database as descending from the prototype. It is therefore easy to go from the prototypes to the instances. The pointing operation does not identify "all X's," where X is a *description* of the prototype, but only "all descendants of X". The way in which the picture has been put together has here an incurable influence on the class of identifiable objects.

We have seen so far two kinds of seemingly arbitrary limitations in the user language for pointing:

- Loss of the substructure of objects
- Identification of pointed objects governed by features of the database that are invisible to the user.

By using the methods of the last section, we are going to see now how these problems can be solved.

4. Structural Rules and the Identification Problem

We can now give a quite general description of the identification problem in terms of structural rules. Essentially, we can use the fact that the predicate `visible` as defined in Section 2 can be inverted provided that the predicates `instance`, `part`, `graphics` and `apply` are also invertible. The first three are defined exclusively by atomic facts and therefore are trivially invertible. The predicate `apply` can also be made invertible as discussed in Section 2.

I assume the user input provides a hit set of atomic graphic objects as those generated by the `visible` predicate. The hit classes also derived from the user input are seen as names for conditions that an object must specify to be the target of the pointing operation. Then, the identification problem for a pair

<graphic object, hit class>

can be simply described by the rule:

```
target(Graphics,HitClass,Object) ⇐                    (6)
    visible(Object,Graphics),
    satisfies(Object,HitClass).
```

where `satisfies` defines whether an object satisfies the conditions named by a hit class.

The hit classes allowed by the user language could range from the names of primitives to complex conditions expressed in the same language as the structural rules. For example, the following clauses partially define an `arch` hit class:

```
satisfies(Object,arch) ⇐ arch(Object).
```

```
arch(Object) ⇐ instance(Object,arch,Trans).
arch(Object) ⇐                                    (7)
   setof(Part,basic_part(Part,Object),Set),
   arch_shape(Set).
```

Arches occur by this definition in two ways: as instances of an **arch** prototype, and as "ad hoc" arches made of independent parts partially defined by clause (7). This clause contains a new kind of condition, of the form

```
setof(thing, conditions, set)
```

which is satisfied when **set** is the set of **things** satisfying the **conditions**. Clause (7) can therefore be read as "an arch may be an object such that its (basic) parts form an arch." The definition of **arch_shape** is not very interesting at this point, and is therefore omitted.

In a more realistic setting, rule (6) would be replaced by a rule that takes into account the context-dependent **focus** of the interaction. We can formalize focus as a predicate that takes a context and an object and checks that the object is "in focus" in that context. The revised rule is as follows:

```
target(Graphics,HitClass,Object,Context) ⇐
  visible(Object,Graphics),
  satisfies(Object,HitClass),
  focus(Context,Object).
```

In our arch example, we could have the following definition for focus:

```
focus(arches,Arch) ⇐ arch(Arch).
focus(beams,Beam) ⇐ beam(Beam).
```

where the **arch** and **beam** predicates identify arches and beams, respectively. In a building application, pointing at a line segment might identify a wall if we are working at the floor plan level, but might identify a house if we are working at the site plan level.

This notion of focus will also be useful in viewing the database to avoid the presentation of irrelevant detail.

5. New From Old

In the examples of the last section, we have encountered a very simple mechanism for defining objects from others through transformations. The term "prototype" was used above to refer to objects that are transformed in this way. However, whereas the prototypes of the last section were fully defined, in general a prototype will only describe an aspect of an object, leaving undefined (**uninstantiated**) other aspects. Prolog programming gives us an effective means for dealing with partly defined or **parameterized** objects. For example, the following clauses define two parameterized prototypes **beam(Length)** and **arch(Width,Height)** leaving unspecified the length of beams of width 1 and the width and height of arches.

```
prototype(arch(Width,Height)).

part(leftcolumn(Height),
     arch(Width,Height)).
part(rightcolumn(Width,Height),
     arch(Width,Height)).
part(crossbeam(Width,Height),
     arch(Width,Height)).

instance(leftcolumn(Length),
         beam(Length),id).
instance(rightcolumn(Width,Height),
         beam(Height),
         translate((Width-1)^0)).
instance(crossbeam(Width,Height),
         beam(Width),
         rotate(-90,0^0):
         translate(0^(Height+1))).

prototype(beam(Length)).

graphics(beam(Length),
         line(0^0,0^Length)).
graphics(beam(Length),
         line(0^0,1^0)).
graphics(beam(Length),
         line(1^0,1^Length)).
graphics(beam(Length),
         line(0^Length,1^Length)).
```

These clauses correspond directly to their counterparts in Section 2, except that now object names are not atomic, but instead have arguments for the unspecified dimensions. Figure 5-1 shows some instances of the prototypes.

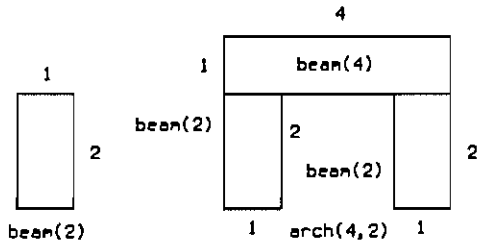

Figure 5-1: Parameterized Arches

Besides prototyping, other combinating operations between objects are easily defined. For instance, a **union** relationship to make a new object containing all the parts of two other objects can be interpreted by the following additional clauses for **in**:

```
in(Part,Obj) ⇐
   union(Obj1,Obj2,Obj),
   in(Part,Obj1).
in(Part,Obj) ⇐
   union(Obj1,Obj2,Obj),
   in(Part,Obj2).
```

. The specific union objects will be defined by clauses for **union**[4] . The two clauses for **in** state that a part is in a union object if it is in one of the two objects in the union.

Work is being done on even more powerful methods of object definition in definite-clause logic for graphics databases. Of particular interest are definition operators analogous to those provided in regular expressions and context-free grammars [9, 4] that can be used to define generic objects containing arbitrary numbers of parts where the location and dimensions of each part are defined relatively to those of the other parts and the overall assemblage is defined by further constraints. Such operators can be applied for example to fill a region with instances of a given prototype.

6. Updating the Database

The method of representation described in the preceding sections is very closely related to that provided by relational databases. All the basic object definition predicates (**part**, **instance**, **primitive**, **graphics** and **prototype**) can be identified with relations in a relational database. The predicate ***instance*** is the only one that has an implicit fuctional dependency, from its first argument to each of its other arguments. The techniques for updating Prolog databases described by Parsaye [12] may therefore be applied here.

Because, in our databases, objects are described in a piecemeal fashion, it is perfectly acceptable to modify only part of the description of an object, or to modify all objects that use a prototype by modifying the prototype. The easiest way to visualize what happens to the objects described by the database when the database is updated is to see an update as a mapping of an extensional version of the predicate **in** (the set of tuples in **in**) to another such set of tuples. The correspondence between updates of the base relations and changes in the extension of the **in** relation is defined by the structural rules.

Because objects containing instances of prototypes may change if the prototypes are changed, we need to be able to **copy** an object by creating a new object containing the parts of another directly without relying on the prototype instances contained in the old object. The old and new objects will be exactly the same from the point of view of the **in** predicate but their internal structure will be different, with the new object being a one-level assembly of atomic parts. The copy operation can be defined as a global database update operation along the lines

```
copy(Old,New) ⇐
  for_all(Part,
          in(Part,Old),add(part(Part,New))).
```

where **for_all** and **add** are extralogical operators[5] that iterate over the tuples of a relation and add new tuples to the database, respectively.

In a related paper [16], a method for updating and maintaining the integrity of Prolog design databases is discussed.

7. Implementation

SeeLog, a graphics front end for Prolog databases based on an early version of the methods described here has been implemented by combining a Prolog interpreter with the GiGo window graphics manager [13] on a VAX 11/750 under 4.1 Berkeley Unix[6].

SeeLog allows the user to give a declarative description not only of drawings but also of the screen layout in which the drawings are to be displayed. SeeLog manages the relationship between primitive objects and their graphical representations so that the display is automatically updated when the Prolog database changes and graphics input is efficiently related to primitive objects in the database. SeeLog is being reimplemented and extended to add a more flexible screen model and to improve the speed of redisplaying and input interpretation. SeeLog is being used to demonstrate Prolog programming techniques for computer-aided architectural design [15, 8].

Although the definitions of **in** and **visible** fully describe the graphical content of a database, in a large database it is not practical to rely solely on those

[4] Union is a predicate rather than a functor that combines two object names into a new object name to guarantee that spurious union objects are not generated when the definition of **in** is used for identification of objects rather than for viewing.

[5] The theory of giving nonprocedural meaning to such operators is discussed by Kowalski [11].

[6] Unix is a trademark of Bell Laboratories.

definitions to interpret user input. Some form of "backwards indexing" from parts to the objects that contain them is clearly desirable. It is possible to provide such information in the form of additional redundant clauses in the database. Of course, a specification of the functional dependency of the indexing clauses on the base clauses will be required to ensure correct updating.

8. Conclusion

A method for relating objects in a design database to their graphical representations has been presented. The foundation of the method is the use of definite-clause logic both as a description language and, in the form of Prolog, as a programming language for representing objects and the relations between objects and between objects and their graphical representations. For the point of view of graphical operations, the main advantage of the method is that the same logical statements can be used to compute the images of objects and to interpret graphical input in terms of the objects in the database. The method also makes the definition of parameterized objects as easy as that of constant objects. There is potential for extending this work to encompass more powerful object combination operations such as indefinite iteration and space filling.

References

[1] Applied Research of Cambridge, General Drafting System, System Overview, ref. no. GDS|S2. Cambridge, England, 1980.

[2] A. Bijl and J. Nash, "Progress on Drawing Systems," *Computer-Aided Design*, Vol. 13, No. 6, pp. 351-357 (November 1981).

[3] I.C. Braid, *Designing with Volumes*, Cantab Press, Cambridge, England, (1973).

[4] L. Cardelli, "Sticks & Stones: an Applicative VLSI Design Language,"Internal ReportCSR-85-81, Dept. of Computer Science, University of Edinburgh, (July 1981).

[5] L. Cardelli and G. Plotkin, "An Algebraic Approach to VLSI Design," *VLSI 81 Internotional Conference*(1981).

[6] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer Verlag, (1981).

[7] C.M. Eastman and M. Henrion, "GLIDE: A System for Implementing Design Databases,"Research Report76, Institute of Physical Planning, Carnegie-Mellon University, (December 1978).

[8] C. Giraud, The Presque-Half Plane: Towards a General Representation Technique. In preparation. Describes a new geometric modeling technique based on the combination of point set and edge representations.

[9] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Reading, Massachussets, (1979).

[10] ISO Working Group TC97/SC5/WG2, "Graphical Kernel System (GKS) - Functional Description,"ISO TC97/SC5/WG2 DocumentN117, International Standards Organisation, (January 1982).

[11] R.A. Kowalski, *Logic for Problem Solving*, North Holland, (1980).

[12] K. Parsaye, Database Management, Knowledge Base Management and Expert System Development in Prolog. In this proceedings.

[13] D.S.H. Rosenthal, "Managing Graphical Resources," *Computer Graphics*, Vol. 17, No. 1, pp. 38-45 (January 1983).

[14] P. Roussel, "Prolog : Manuel de Reference et Utilisation,"Technical Report, Groupe d'Intelligence Artificielle, U. E. R. de Luminy, Universite d'Aix-Marseille II, (1975).

[15] P.S.G. Swinson, "Logic Programming - a Computing Tool for the Architect of the Future," *Computer-Aided Design*, Vol. 14, No. 2 (March 1982).

[16] P.S.G. Swinson, F.C.N. Pereira and A. Bijl, A Fact Dependency System for the Logic Programmer. To appear in Computer-Aided Design.

[17] P.J.W. ten Hagen, T. Hagen, P. Klint, H. Noot, H.J. Sint and A.V. Veen, *Mathematical Centre Tracts*. Volume 130: *Intermediate Language for Pictures*, Matematisch Centrum, Amsterdam, (1980).

[18] D.H.D. Warren, L.M. Pereira and F.C.N. Pereira, "Prolog - The Language and its Implementation Compared with Lisp," *SIGPLAN/SIGART Newsletter*, ACM Symposium on A.I. and Programming Languages(August 1977).