Implementing Lograph

Omid Banyasad
Philip T. Cox

Technical Report CS 2001-05

Nov 27, 2001

Faculty of Computer Science
6050 University Ave., Halifax, Nova Scotia, B3H 1W5, Canada

Implementing Lograph

Omid Banyasad *Philip T. Cox*

Faculty of Computer Science, Dalhousie University
Halifax, Nova Scotia, Canada

Abstract

Lograph is a non-deterministic visual logic programming language which provides the basis for the language LSD, a visual language for designing structured objects. Hence in order to implement LSD we must first implement Lograph. This raises many questions about visual logic programming languages, such as efficient execution, and clarity of the interface. Here we show how Lograph can be made deterministic and potentially efficient by ordering execution rules, cases of definitions and literals within cases and queries. We also discuss the conflict that arises between this ordering for efficiency and the clarity of programs, and present an interface mechanism to overcome it. Finally, we describe a prototype of Lograph which is currently under development.

1 Introduction

The work reported here is part of a continuing project aimed at creating new software tools for "design engineering", in which some of the concepts underlying software engineering tools are applied to the domain of structured object design. We briefly recap the main issues, as follows.

In [9] it was observed that in logic programming, terms (data) and literals (the components of algorithms) are uniformly represented, and that the execution mechanism transforms terms via unification. This led to the conjecture that visual logic programming might provide a basis for a design language in which design components and algorithm components are homogeneously represented in an integrated environment, and to a proposal for a Language for Structured Design (LSD).

The language underlying LSD is Lograph, a general, non-deterministic, visual logic programming language [7]. It should be noted that although various visual logic programming languages have been proposed, for example [11,13,15,16,19], Lograph has some properties that make it particularly suitable as the basis for a design environment. First, the semantics can be realised as graph transformations, and second, unification is replaced by two execution rules that reveal the details of unification rather than treat it as one large step [8]. Together, these properties allow an execution to be viewed as a movie depicting the morphing of a query into a result.

Lograph is extended to LSD by the addition of *components* and *operations,* logical manifestations of *solids* and *operations* on them in design spaces, formally defined in [10].

As mentioned above, Lograph is a non-deterministic language and so, therefore, is LSD as proposed in [9]. However, programs written in LSD should produce consistent results (solids) when executed, so the execution of LSD programs must be deterministic. We therefore need a deterministic execution model for Lograph, which is one of the issues addressed here.

The consideration of determinism raises a second issue. Since determinism in logic program execution implies ordering of the literals in the bodies of clauses, and ordering of clauses in the definition of a predicate, as in Prolog, how should such ordering be expressed in a visual logic programming language without resorting to a confusing network of lines? This issue is addressed in Section 4.

Finally, since these questions have arisen as a consequence of our implementation of deterministic Lograph on the way to LSD, we will discuss some of the features of the current Lograph editor environment.

Since Lograph syntax and semantics are essential to our discussion, in the next section we give a brief overview of Lograph. Our presentation is based on the more detailed descriptions in [5,7].

2 Lograph Syntax and Semantics

Lograph is a visual representation of *flat Horn clauses* which are a specific form of first-order predicate calculus formulae. The semantics of flat Horn clauses are defined by a set of deduction rules called *Surface Deduction*. The discussion of surface deduction, its soundness and completeness can be found in [8].

2.1 Lograph Syntax

A Lograph *program* is a collection of literal definitions with no terminals in common. A *literal definition* (or definition for short) is a set of cases with the same name and arity. A *case* consists of a name, a head and a body. The

head of a case is an ordered list of terminals of length n for some integer $n \geq 0$ called the *arity* of the case. The *body* of a case is a network of items called cells interconnected by wires. A *cell* is either a function cell or a literal cell.

A *function cell* consists of a name, a *root terminal* and a list of *terminals* of length $n \geq 0$ called the *arity* of the cell. A function cell is represented by an icon with the name of the function in the centre, a curved face containing the root terminal of the function on its peak, and a flat face along which the terminals of the function, represented by small circles, are arranged. Figure 1 shows a function cell named Function with arity 5. A function cell can have two possible orientations,  or  . Regardless of the orientation of a function cell, its terminals are ordered from left to right.

Figure 1: A function cell

A function cell with arity 0, also called a *constant*, has the simpler representation or  , where <name> is the name of the function.

A *literal cell* consists of a name and a list of *terminals* of length $n \geq 0$ called the *arity* of the cell. A literal cell is represented by a rounded rectangle with the name of the literal in the centre. The terminals of a literal cell are arranged along the perimeter starting from the *origin,* indicated by a clockwise-pointing arrowhead which may be placed anywhere on the perimeter of the cell. Figure 2 shows a literal cell named Concat with arity 3.

Figure 2: A literal cell

A terminal may occur in several cells and the head of a case. This is indicated by wires connecting the different occurrences. Hence saying "terminal A is connected to terminal B" is the same as saying "terminals A and B are the same."

2.2 Lograph Transformation Rules

Executing a Lograph program involves applying three Lograph execution rules to a *query,* which is a network of cells, none of the terminals of which occur in the program.

Since the underlying semantics of Lograph is provided by surface deduction, as mentioned earlier, the three Lograph transformation rules are, of course, pictorial manifestations of the three rules of surface deduction. The *Replacement* rule replaces a literal cell with a copy of the body of one of the cases of the definition with same name and arity as the literal cell, if such a definition exists. The terminals of the head of the case and the corresponding terminals of the literal are connected in the process. By *connecting* two terminals, we mean that every occurrence of one of the terminals is replaced by a new occurrence of the other.

The *Merge* rule can be applied to two function cells if they have the same name, arity and root terminal. First the corresponding terminals of the two function cells are connected, then one of the cells is deleted.

The *Deletion* rule applies to a function cell the root terminal of which has no other occurrences, removing the cell from the query.

Since a query provides the starting point for execution of a program, we use the phrases "execution of a program" and "execution of a query" interchangeably.

2.3 Lograph and Prolog

In Section 3, we present the restrictions we impose on Lograph to obtain a viable programming language. In this discussion we will make frequent comparisons to Prolog [12].

A Lograph definition is analogous to a set of Prolog clauses that define a predicate, and a case is analogous to a clause. A literal cell is analogous to a literal in the body of a clause or a query. A function cell corresponds to a term. A terminal occurring in a literal cell or in the head of a case corresponds to a variable. A variable with just one occurrence that is not the root of a function cell is analogous to an anonymous variable in Prolog.

Just as the execution of a query in Prolog aims to produce the empty clause, the goal of a Lograph execution is to reduce a query to an empty graph. It is important to remember, however, that our purpose in implementing Lograph is to provide a basis for LSD where the goal is to generate explicit components [4,9]. In that context, the graphical transformations accomplished by the merge and deletion rules (finer grained than unification in Prolog) are important; and the definition of "successful computation" depends to a large extent on the nature of the solids produced in the design space.

In Prolog, a failure occurs when two variables cannot be unified because they are bound to terms that begin with different functions. The analogy in Lograph occurs when the query is transformed into a graph containing two function cells with the same root, but different names or arities. Such cells cannot be merged, and therefore prevent the query from being transformed into the empty graph. Note that in Lograph, there are other untransformable configurations. The simplest example is a function cell the root of which also occurs as another terminal of the cell. This corresponds to cycles detected by full unification, which is not performed by Prolog.

3 Deterministic Lograph

In Lograph as described above, there are three sources of non-determinism: the choice of which execution rule to apply, which cell or collection of cells to apply it to, and for the replacement rule, which case of a definition to use. Because Lograph represents flat Horn Clauses graphically, it expresses this non-determinism in a natural way: however, to make Lograph viable as a programming language, we must impose restrictions similar to those imposed on general, first-order, Horn Clause resolution theorem to obtain Prolog.

In Prolog, the clauses that define a predicate are linearly ordered, indicating the order in which they will be applied to a query literal. Similarly, the literals in each program clause are linearly ordered, and are executed by resolution in that order. The order in which the search space is traversed is therefore well defined, and is exploited by the Prolog programmer.

Clearly, since Lograph is a first-order Horn Clause language like Prolog, we can aim for the same kind of implementation based on depth first search with backtracking instigated by failure, where failure in Lograph is defined by the occurrence of undeletable function cells, as discussed at the end of the last section. The restrictions we will make are analogous to the above restrictions inherent in Prolog: however, there are some important differences because Lograph is based on surface deduction rather than simple resolution.

As in Prolog, we need to impose two orderings on Lograph to obtain a well defined traversal of the search space: specifically, the order in which cases of a definition should be tried in applying the replacement rule, and the order in which literal cells in a query should be replaced. In addition, we need to decide on the order in which the three execution rules are to be applied, We address the latter issue first.

3.1 Order of Transformation Rules

In this section we show that if a particular ordering of the transformation rules leads to an execution that reduces a query to an empty graph, then any ordering will do the same.

As mentioned in Section 2.2, the deletion rule is applied to functions with dangling root terminals. Since a *deletable* function cell cannot participate in any other transformation rules, the order in which deletable functions are removed will have no effect on the rest of the execution.

Let us suppose that the replacement rule precedes the merge rule in the chosen rule ordering, and that the current query contains a pair {A, B} of function cells connected by their roots. Clearly A and B are compatible, since otherwise the query cannot be reduced to the empty graph. We have four cases to consider: either

(a) the query contains some literal cells, or

the query does not contain any literal cells, and either

(b) the roots of the two compatible cells identified above are not connected to any other terminals, or

(c) the roots of the two cells are connected to the roots of some other function cells, or

(d) the roots of the two cells are connected only to non-root terminals of some other function cells.

In case (b), the only transformations that can be applied to A and B is merge followed by a deletion. These transformations are independent of any others, and can therefore be applied immediately.

In case (c), since the query is eventually reduced to the empty graph, any function cell connected by its root to the roots of A and B must be compatible with them. Since every merge produces a function cell compatible with the merged cells, and therefore with any other compatible cells attached to them by root terminals, the order in which the cells in such a group are merged is irrelevant. Hence the merge of A and B can be performed before any other merges of cells in the group.

In case (d), suppose that the execution removes the "other" function cells before any other transformations occur. The "other" cells are removed either by (d1) deletion, or by (d2) merging followed by deletion.

In case (d1), we are left with an instance of case (b) or (c), so that merging A and B can be the next operation performed. Clearly the deletion of the "other" function cells does not depend on the presence of cells A or B, so the merging of A and B could be performed earlier.

In case (d2), we are left with an instance of case (b), (c) or (d). We deal with (b) and (c) as in the previous paragraph. As for (d), we need only note that cases (d) and (d2) cannot alternate forever since the transformations that occur in case (d2) strictly reduce the size of the graph, so we must eventually get case (b) or (c).

In case (a), no merging of A and B will occur until all replacements have been performed. Clearly, performing the replacements is not affected by the presence or absence of A or B, so we can merge A and B at any time.

Therefore, since the transformation rules can be applied in any order, we need to consider the best order in which to apply them. Obviously, if we are doing a depth first search of the solution space as in Prolog, then we should discover "nonunifiability" early. In Lograph, this means applying the merge rule as early as possible. The deletion rule does not really affect this since it just plays the role of "garbage collector"; however, if we are interested in useful animated visualisations of executions, then we might want to apply it early as well, in order to reduce clutter.

Another issue relating to the order of rule application is whether or not we want to optimise our search for a solution. Interestingly, the graphical structures built from function cells are very similar to structures for terms proposed in [5] to enable intelligent backtracking. If intelligent backtracking were to be implemented, the merge and deletion rules would not necessarily be applied as early as possible.

Finally, since Lograph replaces unification with explicit transformation rules, there may be ways to apply them which are better suited to the application. For example it might be possible in some circumstances to "batch" merges and deletions, applying them only occasionally between sequences of consecutive replacements.

3.2 Ordering Cases

Just as Prolog orders the clauses of a predicate definition, we need to order the cases of a definition in Lograph. The easiest way to do this is simply to label each case with a sequence number. The ordering can also be visually represented as a list of icons each bearing the name of one of the cases. The case at the head of the list has sequence number 1 and will be tried first in the replacement of a literal cell in a query. The list of cases can be rearranged by dragging and dropping entries in the list. This ordering of cases is illustrated in our discussion of the Lograph prototype in Section 4.1.

3.3 Ordering cells in cases and queries

As mentioned above, we need to specify the execution order of the cells in a case or query. Note that since merge and deletion can be applied at any time, we need concern ourselves only with the order of literal cells.

This leads us to an interesting problem. Like other visual languages, Lograph exposes the structure of algorithms without imposing needless sequentiality on them. However, we need to impose sequentiality for the sake of efficiency. This looks like the same problem that arises in implementing other visual languages, for example dataflow languages. In the case of dataflow languages, operations are partially ordered, and any linear order produced by topological sort will do [6]. In Lograph, however, the wires are not data flow links, so no suitable order can be automatically generated. It is therefore up to the programmer to specify an appropriate order. This is, of course, what Prolog programmers do, so deciding on literal cell ordering should not be a too great a burden for the Lograph programmer. Furthermore, the goal of this project is to implement LSD where the domain is the design of structured objects. As noted in [4] the design process has a natural order to it, so imposing an ordering on the cells in a case should be quite natural for the designer/programmer.

One obvious solution is to add special connections between literal cells to indicate execution order, like the synchros of Prograph [6]. However these would be far more intrusive than such synchronisation links in a data flow language where they are needed only occasionally.

Instead we use two representations of literal cell ordering analogous to the two representations of multi-layered images in Photoshop [1]. The first is a list of icons similar to the list of cases described above, and to the list of layers displayed in Photoshop's "layers" palette. The second representation, like the Photoshop image window, treats the window in which a case or query is displayed as a series of transparent layers each containing some of the items that make up the whole image. In Lograph, each layer contains one or more literal cells. Those in the topmost layer are to be executed first, followed by those in the next layer, and so forth. There is no ordering imposed on literals within a layer, allowing the programmer to group together literals which he or she knows could be executed in parallel.

As a clue to the ordering of cells in the layered window, the literal cell icons are painted in a range of shades of one colour, ranging from dark at the front to pale at the back.

Layers can be reordered by dragging their iconic representations in the list view. Layers are recoloured whenever they are reordered or a new layer is added. Literals can

also be moved from layer to layer. When new literals are added to the body of a case or a query, the colour of all layers are adjusted to their new values. When the number of layers increases the range of shades is subdivided resulting in less differentiation between layers. Clearly, as the number of layers grows, the programmer may have to rely more on the list view for ordering.

Note that during execution, when a literal cell is replaced by the body of a case, the layers of the case body are placed in front of the existing layers of the query.

4 Programming Environment

In this section, we discuss the editor environment of Lograph through a worked example. The example shows how to write a program in Lograph that computes the concatenation of two lists.

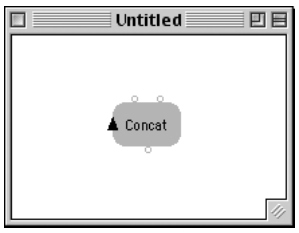

Figure 3: Program window

4.1 Editor

When Lograph is started, a menu bar containing, **File**, **Edit**, **Run**, and **Settings** appears together with an empty window named **Untitled** in which a program is created and maintained. Double-clicking inside this window creates a definition icon with no terminal, an origin on the left side, and the default name Un-named. We rename it Concat. Every definition icon has a sensitive boundary; that is, the cursor changes to ○ whenever it is near the perimeter of the cell, indicating that a a click will add a terminal. Figure 3 shows the program window after we have created a definition with arity 3 named Concat.

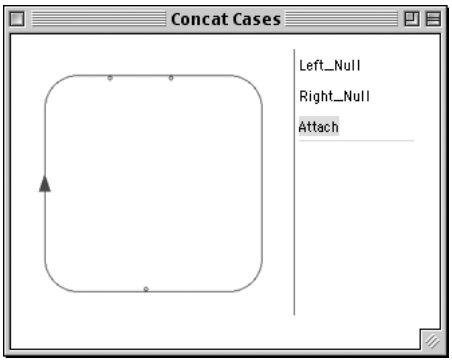

Figure 4: Cases window of Concat

We now double-click on the definition, opening its cases window, consisting of the list of cases on the right and a thumbnail of the selected case on the left. A new case is added to the definition of a literal by double clicking in the list. This creates a new case named Case N where N is the number of cases previously created. The name of a case can be edited at any time by selecting it in the list an typing. Note that our implementation allows cases to be named, which is not a feature of Lograph as described above. In our example, we have created three cases for the Concat definition and named them Left_Null, Right_Null, and Attach as shown in Figure 4. The order of cases can be changed by dragging them in the list. Notice that in Figure 4 the thumbnail of the newly-created selected case Attach shows only the head of the case with an empty body.

To define the body of a case, we double click on it in the case list or double click the selected case's thumbnail. This opens the case window consisting of a workspace to the left and a layer list to the right, both of them initially empty. The workspace contains the layered view of the case described above. New literal cells can be added to the workspace by dragging and dropping definitions from the program window or can be created by double clicking in the workspace. Definition icons in the program window literal cell icons in case windows are similar except for their colours, green and blue respectively. The colour settings for different icon classes can be customized.

In our current implementation, each layer contains only one literal cell. Consequently, when a literal cell is added to the workspace, a new layer is created to contain it, and its icon is used to identify the layer in the layer list. Editing the name or adding a terminal is accomplished in the same way for a literal cell as for a definition as described above. In addition, the origin and terminals of a literal cell can be dragged around its perimeter.

Double clicking in the workspace while holding down the "F" key creates a new function cell with root terminal and arity 0, named Un-named by default. The name can be edited and terminals added as described above. Double-clicking a function cell changes its orientation, from pointing-up to pointing-down or *vice versa*.

As the cursor passes over a terminal, the terminal turns red indicating that the terminal is "active". Clicking on an active terminal creates a "rubber band" between the terminal and the cursor. Clicking on another terminal in the same workspace creates a wire between the two terminals. Figure 6 illustrates the case windows for the three completed cases of Concat.

4.2 Animated Execution

Now that the definition of Concat is complete, we can use it to execute queries. A query window is opened by selecting **New Query** from the program's **File** menu. Editing in a query window is similar to editing in a case win-

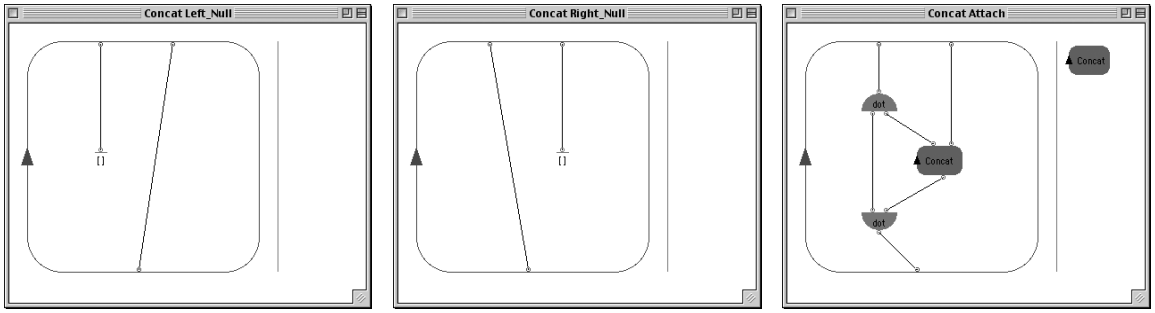

Figure 6: Case windows of Concat

dow. We create a new query as in Figure 5. Note that the constant X in this query is an example of a *neutral constant*, indicated by its solid root. A neutral constant is one which will not cause backtracking if it fails to merge with a connected function cell. The effect in this example is that execution will stop rather than backtrack in its search for the empty graph.

A query can be executed in three different modes: **Run**, **Single Step** and **Animate**. In the **Run** mode, the result is computed by Lograph interpreter and displayed in the query window. In **Animate** mode the interpreter displays an animation of each rule application. The deletion of a function is animated by fading out the function and releasing all the wires which shrink away from the disappearing function cell towards their other ends.

The merge rule is animated by morphing two function cells into one. Animation of the replacement is accomplished by expanding the replaced literal to the size of the case that replaces it, then fading in the body of the case together with the necessary connecting wires. Execution of queries in single step mode is in fact the pictorial manifestation of the Lograph interpreter engine process. Backtracking is also visualized by reversing these animations. The result of executing the query in Figure 5 is shown in Figure 5. In **Single Step** mode, the interpreter animates each step but stops between steps.

5 Current status and future work

5.1 Prototype

The heart of the prototype is the Lograph interpreter engine, a standard logic programming interpreter implemented in Java.

The features of Lograph presented here provide capabilities similar to Lisp and Prolog. Some of the conveniences of those languages need to be included as well, which we will now briefly describe.

In Lograph, a list can be represented by a nested structure of function cells as in Figure 5 where two structures built with cells named dot represent the lists [a,b,c] and [X,1,2]. Here the constant [] is interpreted as the empty list. Such structures can be abbreviated as they are in Prolog, by special constants such as [a,b,c]. In Prolog, however, a list can contain a

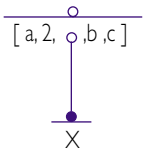

Figure 7: List constant cell with embedded terminal

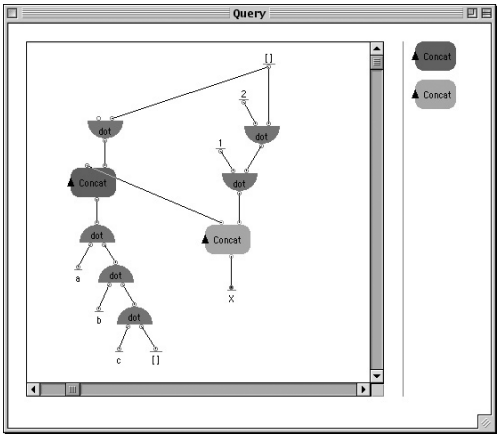

Figure 5: Query window

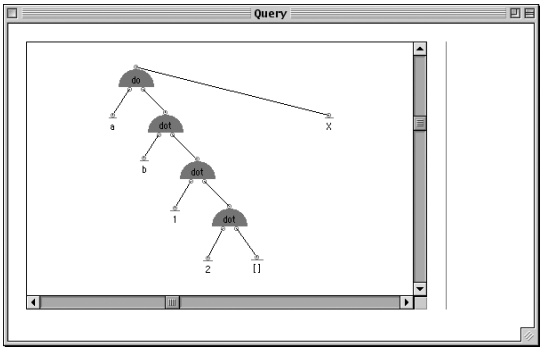

Figure 7: Query result

embedded as in Figure 7.

Although logic programming is not usually used for numerical problems, Prolog provides numbers as special constants and some basic arithmetic operations that compute functions of numbers in a data flow fashion. In the domain of structured object design, there is a clear need for numerical computation far more extensive than that normally expected in Prolog programs, so it will be necessary to extend Lograph to provide this capability beyond the mechanisms provided by Prolog, which are inconvenient at best.

5.2 Future work

As noted above, the Lograph is being implemented as the first step in implementing LSD. We now briefly describe some of the more significant steps that need to be taken to reach that goal.

The core of the current prototype is a Java implementation of a standard logic language interpreter. This has the usual advantages such as ease of development and debugging, cross-platform executability and so forth. It also has the usual disadvantages, such as low execution efficiency, and slow and limited graphics. Once we have passed the proof of concept stage, a more "industrial strength" implementation will be required, using more appropriate technologies and implementation techniques to obtain a fast and capable interpreter engine, able to support the heavy demands of the intended application.

Solids are 3D objects which manifest themselves in LSD programs as explicit components [9]. An LSD program is therefore necessarily three-dimensional, requiring an editor that operates in 3D. In the 2D editing environment described above, however, we have used layering in the third dimension to deal with the ordering of literals in cases. Clearly we need to extend this notion to a three-dimensional editor. One possibility might be to use transparency instead of the shading employed in the 2D scheme. More opaque objects would be analogous to darker literals in the upper layers of a Lograph case or query, and more transparent objects would be analogous to literals in the deeper layers. Under execution, the fully opaque objects in a query are the ones that undergo transformation first, and objects become more opaque the closer they get to being executed.

6 Concluding remarks

Implementing a prototype Lograph environment has raised various questions about visual logic programming languages in general and Lograph in particular.

For execution efficiency, it is necessary to restrict the Lograph language from a general first-order Horn-clause theorem-prover to an efficiently implementable language. Many of the restrictions are obvious counterparts of the restrictions inherent in Prolog, involving the ordering of clauses and the ordering of literals.

Since Lograph is based on surface deduction rather than simple resolution, it is also necessary to consider how its three execution rules should be ordered. We have shown that the merge and deletion rules can be applied at any time, and to simulate the search order of Prolog, should be applied as early as possible. However, it is possible that other execution orders could be chosen to suit certain applications.

Like other visual languages, Lograph exposes the structure of algorithms in a useful way. However, since it is not dataflow, there is no way to automatically linearise operations, so like the Prolog programmer, the Lograph programmer must take responsibility for this task. We have proposed and implemented a layering scheme similar to the layering of Photoshop images, for visualising the ordering of literal cells in a case.

7 Acknowledgments

This research was supported by Natural Sciences and Engineering Research Council of Canada Research Grant OGP0000124.

8 References

[1] Adobe Systems Inc., *Photoshop 6.0 User Guide*, (2000).

[2] Autodesk Inc., *AutoLISP Release 12 Programmers Reference Manual,* (1992)

[3] Bentley Systems Inc., *MicroStation 95 User's Guide*, (1995)

[4] O. Banyasad, P. T. Cox, *Solving design problems in a logic-based visual design environment*, Report CS-2001-04, Faculty of Computer Science, Dalhousie University, (2001).

[5] P.T. Cox, On determining the causes of nonunifiability, *Journal of Logic Programming* 4, American Elsevier (1987), 33-58.

[6] P.T. Cox, F.R. Giles, T. Pietrzykowski, Prograph: a step towards liberating programming from textual conditioning, *Proc. 1989 IEEE Workshop on Visual Programming*, Rome (Oct 1989), 150-156. Reprinted in *Visual Object-Oriented Programming: Concepts and Environments*, M. Burnett, A. Goldberg, & T.G. Lewis (Eds), Manning Publications (1995).

[7] P.T. Cox, T. Pietrzykowski, LOGRAPH: a graphical logic programming language, *Proceedings IEEE COMPINT 85*, Montreal (1985), pp 145-151.

[8] P.T. Cox, T. Pietrzykowski, Incorporating equality into logic programming via Surface Deduction, *Annals of Pure and Applied Logic 31*, North Holland (1986), pp 177-189.

[9] P.T. Cox, T. Smedley, LSD: A Logic Based Visual Language for Designing Structured Objects, *Journal of Visual Languages and Computing*, v9, Academic Press (1998), 509-534.

[10] P.T. Cox, T. Smedley, A Formal Model for Parametrised Solids in a Visual Design Language, *Journal of Visual Languages and Computing,* v11, Academic Press (2000), 687-710.

[11] K.M. Kahn, V.A. Saraswat, Complete Visualizations of Concurrent Programs and their Executions, Proc. 1990 IEEE Workshop on Visual Languages, (1990), 7-15.

[12] R.A. Kowalski, *Logic for problem solving*, North-Holland, (1979).

[13] M.A. Najork, S.M. Kaplan, The CUBE Language, *Proc. 1991 IEEE Workshop on Visual Languages*, (1991), 218-224

[14] A. Paoluzzi & C. Sansoni, Programming Language for solid variational geometry. *Computer Aided Design 24*, (1992), 349-366.

[15] L.F. Pau, H. Olason, Visual Logic Programming, *Journal of Visual Languages and Computing,* v2 (1991), 3-15.

[16] J. Puigsegur, W.M. Schorlemmer, J. Agustí, From Queries to Answers in Visual Logic Programming, *Proc. IEEE Symposium on Visual Languages,* (1997), 102-109.

[17] A. Rau-Chaplin, B. MacKay-Lyons, and P. Spierenburg (1996), The LaHave House Project: Towards and Automated Architectural Design Service, *Proceedings of the International Conference on Computer Aided Design (CADEX'96)*, IEEE Computer Society Press, pp 25-31.

[18] T.J. Smedley, P.T. Cox, Visual Languages for the Design and Development of Structured Objects, *Journal of Visual Languages and Computing,* v8, Academic Press (1997), 57-84.

[19] L.L. Spratt, A.L. Ambler, A Visual Logic Programming Language Based on Sets and Partitioning Constraints, Proc. 1993 IEEE Symposium on Visual Languages, (1993), 204-208.

[20] *Standard VHDL Language Reference Manual*-Std 1076-1987. IEEE (1988).