Journal of Visual Languages and Computing (1991) 2, 163–188

VLP: a Visual Logic Programming Language

DIDIER LADRET AND MICHEL RUEHER

University of Nice—Sophia Antipolis, I3S-CNRS, Batiment 4, Avenue Einstein, Sophia Antipolis, 06560 Valbonne, France

Received 27 June 1990 and accepted 3 December 1990

In this paper, we argue that visual programming can be greatly enhanced by integrating some cognitive aspects when designing a visual language. We try to apply some interesting rules—established by Bertin on experimental evidence—to get graphical views which have a good picture quality and hence do not have to be read in a linear way. We introduce VLP, a visual logic programming language, which supports program composition and editing in both textual and graphical representations with correspondance between the two views of the program being automatically maintained by the system. The goal of this work is to facilitate the development and reuse of Prolog prototypes through a complementary use of graphical and textual views. In particular, we use graphics to put forward the relational nature of logic programming and, thus, help the user to get a deep understanding of the declarative semantics of his programs. We explain the motivation of design choices, provide an overview of system capabilities, and evaluate system advantages and drawbacks.

1. Introduction

VISUAL PROGRAMMING LANGUAGES [1] have often been presented as a means to build, understand and manage programs [2] in a more easy and efficient way. However, visual programming has not yet gained wide acceptance [3] and the proper role for graphics in programming remains controversial [4].

In this paper, we argue that the weak acceptance of visual programming and the (present) non-generality of visual programming can be greatly enhanced by integrating some cognitive aspects when designing a visual language. This is why we bring forward an approach based upon the following two points:

- First, we propose to comply strictly with some simple psychological laws concerning the perception of graphics.
- Second, we propound to use graphical and textual language in a complementary way.

Psychological laws governing the perception of visual aspects are not yet fully understood. However, Bertin used some interesting experimental evidence to establish a small set of rules [5, 6] that have to be observed each time we want to support efficiently communication by using graphics. We try to apply these rules in order to get graphical views that have the qualities of 'good pictures' and hence do not have to be read in a linear way.

Following Glinert [4], we propose an environment which simultaneously supports program composition and editing in both textual and graphical representations with correspondance between the two views of the program being automatically main-

1045-926X/91/020163 + 27$03.00/0

tained by the system. As a matter of fact, the mixed use of graphics and text is one of our main interests because a visual view of a program, as interesting as it might be, is often not sufficient to express all the semantics of a program, or is too exclusive and does not attract the experienced programmer used to a textual structure. Getting inspiration from multicode theory [7, 8, 9] in image mental structure, we believe that visual and textual representation have distinct optimal domains. Of course, as in the dual code theory [10], we think that the two representations used in parallel can sometimes give better results than only one.

Thus, we introduce in this paper VLP, a visual logic programming language based upon the above mentionned principles. VLP supports interactive graphical composition and execution of logic programs. The choice of investigating visual capabilities in logic programming has several motivations.

Logic programming has been used widely for rapid prototyping to clarify and to validate software specifications and to involve the user more-and-more in the software's definition. Now, areas where the application of graphics has already proven particularly successful include graphical specification and design aids. In fact, graphical representations are often helpful in understanding the relationships between objects (e.g. semantics nets, entity relations diagrams) and to capture the overall process (e.g. Nassi–Schneidermann charts, state transitions diagrams). Hence, one aim of this work is to use graphics to put forward the relational nature of logic programming, which is a key feature when prototyping.

Logic programming is difficult to use on non-trivial applications. There is clear evidence that the apparent 'naturalness' and 'simplicity' of Prolog are deceptive, and that users suddenly experience great difficulty writing and debugging declarative programs intented to solve anything other than toy problems [11]. Visualization of programs has proven its capability to enhance program readability and comprehensibility, and thereby to make them more maintainable and useful [12]. Thus, we expect that graphical views will bring out the program structure and help the user to get a deep understanding of the declarative semantics of logic programs.

The concise syntax of logic programming languages, e.g. Prolog, requires only a limited number of visual symbols. Moreover, Prolog is a 'dense' language so that a large part of a prototype of an information system can be held on the screen at the same time, thus making the graphical approach work for medium or large applications.

Kowalski [13] first outlined the close relationship that exists between logic programming and graphical languages used for rapid prototyping but he did not propose any formalism. Graphical environments for Prolog have already been developed by Dewar and Cleary [14] and by Eisenstadt and Brayshaw [11, 15]. However, their system only focussed on providing a graphical tracer for Prolog (i.e. visualization of execution tree of textual Prolog programs).

We have chosen to apply Bertin's laws of graphics. These laws are deduced from large-scale experiments and their aim is to allow more powerful communication by means of pictures [5]. By conforming to the laws of graphics, the resulting representation of Prolog programs is more expressive than the single classical textual program. Hence, we get rid of many constraints (like variable names) and give a graphic to be watched and not read. For the complementary use of the textual and the visual style, we will link their distinct optimal domains to the classification of Prolog

clauses when prototyping [16]. As a result, we obtain an understandable and attractive visual view of a Prolog program. Its most valuable points are:

- An improved visual comprehension, both static (the program text) and dynamic (its execution) that facilitates reusing and stepwise understanding.
- A higher level of abstraction for Prolog variables.
- A better support for annotation of programs (e.g. variable quantification, refinement with pre-conditions and post-conditions).
- A better support for the manipulation of Prolog programs in an iconic environment.

The resulting representation of Prolog programs will involve a strong encapsulation of components, and hence eliminates the *spaghetti ball* phenomenon inherently associated with flowcharts and other graphical representations based on directed graphs.

After recalling Bertin's rules of graphics, especially the eight visual variables, we present our approach on three levels:

- *Graphics*, using the laws of graphics to make an expressive and powerful visual view of a Prolog program.
- *Symbolics* to ease visual memorization and enforce a graphical normalization of software components, as in electronics or mechanics.
- *General*, using this visual view complementarily to a textual one in order to give the better suited view according to the kind of program manipulated.

Finally, we describe a short example in order to illustrate the main characteristics of the proposed visual logic programming approach.

This article assumes the reader is familiar with terminology of visual programming [1, 17] and with logic programming [18] and the Prolog language [19, 20].

2. The Laws of Graphics

The laws of graphics are the results of large-scale experiments and common-sense reasoning. They have been set up and formalized by Bertin [5, 6] in a geographical context. In this section we sum up his main results and present them in a visual programming perspective. Specific properties will be introduced when they are used for the design of VLP's features. Though, what follows in this chapter is in no way a presentation of *ex nihilo* beliefs from the authors of this paper.

Graphics use plane properties to express likeness, order and proportionality relationships. In one moment of perception, the ear perceives one sound when the eye sees a relationship between three sets (the three plane dimensions). So graphics are used to save time and memory, to have an instantaneous perception.

> Graphics is not an art, it is a system of strict and simple signs allowing a better understanding of set relationships existing in a graphic [5].

Graphics, like mathematics, deals only with similarity relationships or, on the other hand, difference relationships, order relationships and proportionality relationships between objects. The laws of graphics are derived from the structure and properties of visual perception. A picture is built upon three homogenous and ordered dimensions: X and Y orthogonal plane dimensions, the Z variation of the elementary spot (i.e. the

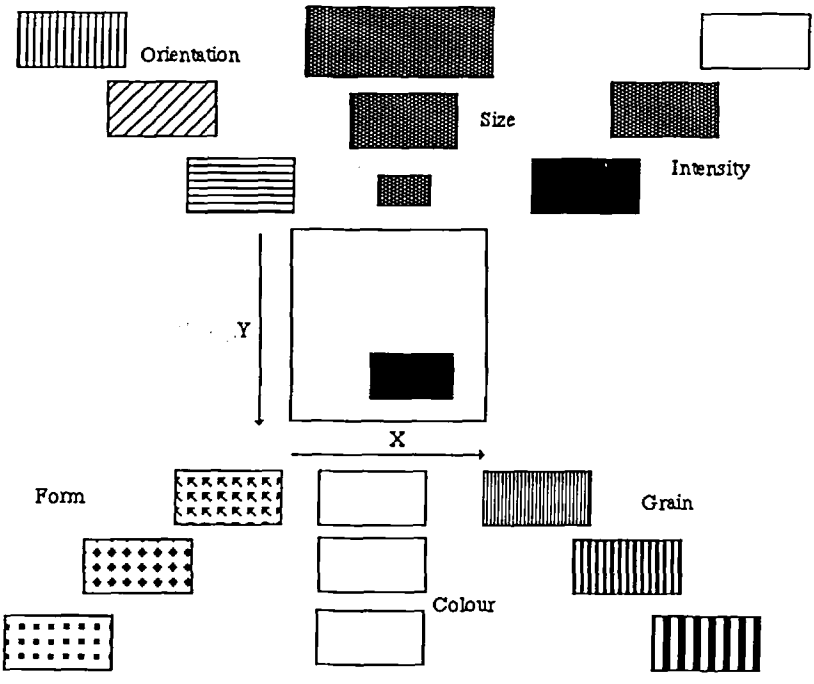

Figure 1. The six graphical variables [5]

inside of a form). Bertin uses six variables to characterize the Z value (cf. Figure 1 from Bertin [5]):

- The variations of size and intensity.
- The variations of colour, grain, orientation and form.

When designing a graphic, we have to take into account the different properties (established in an experimental way) of the eight visual variables. All the above-mentioned relationships can be expressed in a natural way by using the two dimensions of the plane, but the other six variables allow only some of them to be conveyed. As shown by Bertin:

> the size and intensity variables are best suited for order and difference relationships while colour, grain, orientation and form variables enable to express similarity relationships.

Thus, the former are called pictures variables (to represent main pictures), the latter separation variables (to select subpictures).

3. Presentation of Our Visual Programming Style

As it was explained before, we have chosen to investigate visual capabilities for Prolog because of both its concise syntax and its use for rapid prototyping where developers are familiar with graphical representations.

When building a visual representation of a Prolog program, two problems arise. The first is to represent a Prolog term (from now on called Prolog component, in analogy to electrical and mechanical components). It will be solved by using the four picture variables: X, Y, Z variations in size and intensity. The second problem is to represent several Prolog components on a graphic and to select them. It will be solved with the four separation variables: grain, colour, orientation and form. The properties of the eight visual variables will be explained when we use them.

Another point we will have to deal with is that the plane is continuous and homogeneous. According to Bertin:

> every visual variation is significant, and any variation without meaning is ambiguous.

Any visual run different from a straight line, any variation of distance between those lines or any angle different from the right angle are not neutral and create non-significant variations. As a counter-example, to let the user make free-hand drawing programming, as in PiP [21], may be hazardous.

First of all, we introduce the basic symbols of our Prolog components in order to enable a graphical normalization.

3.1. Level of the Symbolics

The choice of symbols or icons is mainly subjective and thus arguable. The choices presented have been made with care for simplicity and analogies to electronical components, for which a normalization exists. We now present the basic symbols. A Prolog program is made of Prolog clauses, built by logical combinations of Prolog components made of a *predicate* and its *arguments*. These two symbols are displayed in Figures 2 and 3.

The argument symbols will be displayed on each side of the predicate symbol. Of course, the predicate name can be replaced by a more expressive icon. The predicate symbol does not contain the predicate arity: that will be expressed by the number of arguments placed on the predicate symbol, as shown in Figure 4, which illustrates the analogy between a Prolog component and an electronic component.

We need visual variables that allow us to distinguish easily the arguments from the predicates. To this aim, we have used variation of size. The variations of *size* and *intensity* are used to transcribe information about the size and value of the represented

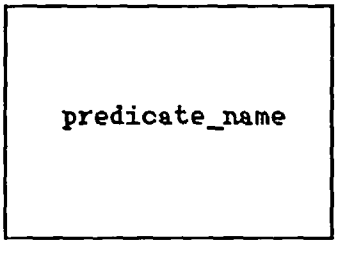

Figure 2. The predicate symbol

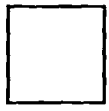

Figure 3. The argument symbol

object; on a geographical map, for instance, they can be used to represent the production or the size of a factory. But objects represented by variations of these variables are not all equally visible (for instance, one notices easily in Figure 1 that smaller objects are 'less visible' than taller ones). These four variables may be used in visual programming to describe hierarchies, for instance type hierarchies in Ada. But they have to be cleverly used. As a counter-example, in our first proposal [22, 23] the size of the different arguments was not the same for the head and body of a Prolog clause. It was an error because there is no hierarchy among the arguments of a Prolog clause.

Some objects, like the Prolog arguments, will have to be all equally visible and their equivalence will have to be easily noticed. There will be no hierarchy among them. So, visual variables like the picture variables will not be suited to this aim. Thus, we need the separation variables *colour*, *grain*, *orientation* and *form*, which all imply the same visibility. The interesting feature of those variables is their *selectivity*, i.e. their ability to distinguish, separate and select subpictures. Variations of size among the argument symbols and among the predicate symbols would be incorrect because there is no hierarchy among the Prolog predicates and among the Prolog arguments. All the corresponding symbols (presented in Figure 2) must have the same size in order not to imply an incorrect hierarchy or to prevent the visual recognition that will be achieved by the separation variables. But the variations of size and form between an argument symbol and a predicate symbol are used to distinguish them easily.

To express a *Prolog clause*, we will logically combine Prolog components inside the predicate symbol. We will consider looking at the inside of this symbol as zooming inside the Prolog component it defines (cf. Figure 5). Hence, we get a good *encapsulation*: a component is a black box with arguments for its connection, but that can be zoomed at to look at its definition. This can be compared to the activation of a file icon on a Macintosh opening the corresponding file. This encapsulation is enhanced by the components library we introduce in our environment. When building a clause, the programmer will have the opportunity to choose already defined components and to assemble them. This zoom will be a way to have an accurate description of *what* the component is.

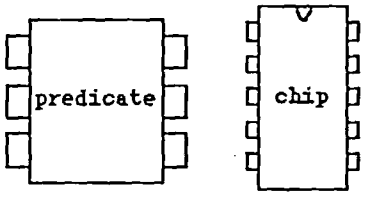

Figure 4. A Prolog component and an electronical component

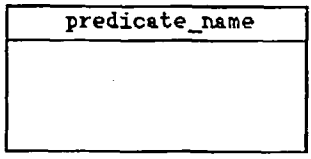

Figure 5. An expanded Prolog clause

We now make some remarks about the choices of the symbols:

- The rectangles of Figures 2 and 5 are similar to express the similarity between the specification of the component (Figure 5) and its definition (Figure 6). It is particularly interesting when defining recursive components; see for instance the descendants of Figure 12. In some existing visual languages (e.g. ThinkPad [24]) non-similar symbols (rectangles) are sometimes used to represent the same object: as an example, a binary tree and its two subtrees are not represented by similar rectangles. This will not help normalization and understanding.
- The fact that arguments are placed left or right does not mean that they have to be considered as inputs or outputs; in the same way that the left pins of a chip need not be connected to '−' or the right ones to '+'. Prolog is an inversible language, with no input/output notions, which is very useful in rapid prototyping. The place of the arguments is only to make the presentation easy. Of course, this place will have to be scrupulously respected to enable normalization. This will be eased by using a components library.

As Prolog is made of a few syntactical elements, the visual representations we have just presented are going to be used often and have a great potential of becoming an accepted symbolism. The user will have the opportunity to redefine it at will, but he will have to take into account practical aspects and the laws of graphics, in order not to lose the benefits of visual programming.

3.2. Level of the Graphics

We are now going to present the assembly rules of these symbols to get a Prolog program. To provide a significant advantage over a textual view, our visual aspect

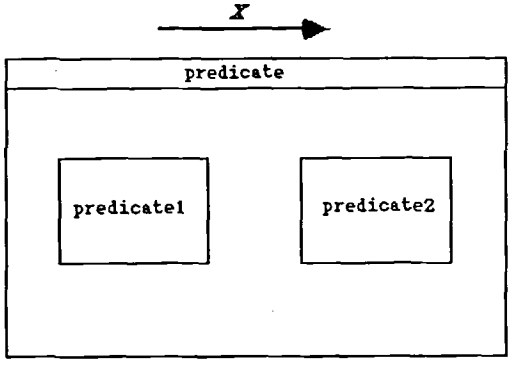

Figure 6. The conditional AND

must be useful and accepted. The assembly rules must be natural and powerful in order that a graphic can be watched rather than read. We are going to apply the laws of graphics to have a strict and rigorous algebra of construction. The separation variables allow us to select quickly identical variables without using their names. As a Prolog program is made of a logical combination of the above basic elements, we now introduce our way to express the conjunction and disjunction operators. In Prolog, the conjunction and disjunction operators are conditional [25]: the order of the clauses in a program defines the order of the evaluation (i.e. SLD resolution which processes the clauses of a procedure from top to bottom and evaluates the predicates from left to right inside a clause).

The X and Y plane dimension variables are ordered: the fact that a statement of a visual programming language is drawn higher than another or at the left of another implies that it will be executed first, because:

> the plane is ordered from top to bottom (gravity) and from left to right (time passing), the first order being stronger than the second [5].

3.2.1. Conjunction

To express the logical connection 'Conditional and', we employ the X plane dimension, as shown in Figure 6.

The logical part of this visual representation is the Prolog clause:

> predicate(...):—predicate1(...), predicate2(...).
> which states that predicate(...) is true if predicate1(...) and then predicate2(...) are true.

The picture variable X plane dimension helps to express the exact semantic of the conditional conjunction operator. Indeed, the fact that a **predicate1(...)** is drawn to the left of **predicate2(...)** implies that it will be executed first because the plane is ordered from left to right.

3.2.2. Disjunction

As expected, the Y dimension will be used to express the 'Conditional or' connection, as shown in Figure 7. But, as the order on the Y dimension is stronger than on the X

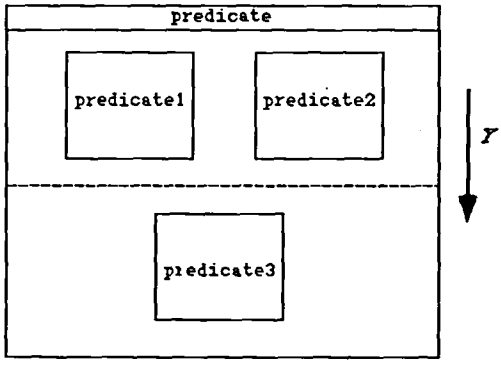

Figure 7. The conditional OR

dimension, we channel the *X* reading as shown in Figure 7. The dotted line indicates that changing of 'or' zones during execution is not definitive because of the Prolog backtracking mechanism. Of course, this proposition is subject to practical validation because it does not conform to the laws of graphics.

We have centered the predicate3 symbol on purpose because we think aligning it under the predicate1 symbol would have left an empty space that could lead to the conclusion that something is missing in the definition.

If the programmer builds a component with more than six alternatives[a] or with more than six components in an alternative, the visual representation may be a little confused. We decide not to visually represent these cases. As a matter of fact, such a number of components is a sign that the software being built is poorly modularized.

3.2.3. Using the Laws of Graphics for the Arguments

Because of the infinite number of possible constants, there is no other standard solution than writing the name of a constant inside the argument symbol (cf. Figure 8).

Of course, the user may build icons for a restricted number of frequently used constants, but the constants are usually very linked to a textual representation. We will see in Section 4 that Prolog clauses containing many constants are better suited to a textual manipulation.

On the other hand, for the Prolog clauses containing many variables, graphics gives us the opportunity to free ourselves from textual constraints (e.g. their names). As a matter of fact, for a type-free language like Prolog, variable names are not really important. The important thing is the concept of logical variable and the fact that the Prolog resolution unifies two variables sharing the same name. We will indicate unifiable variables by filling their associated symbols with identical patterns. We will see that the various patterns can also imply some information on the 'kind' of objects the arguments represent.

Despite of much interest in the use of colour, we will not use it since we are not sure how to best apply it for its selectivity rather than for its aesthetic qualities. The use of colour is critical: there is no rule to determine selective colours, and colour is not available on all graphical displays.

According to Bertin 'form is the least selective of all. Grain is selective' and can practically offer three levels of selection, shown in Figure 9. Grain is ordered, but unlike size and intensity, it does not weaken some subpictures because all the objects represented with variations of grain have the same visibility. As an example, an object with a taller grain is not more visible than an object with a smaller grain, while an

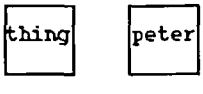

Figure 8. The name of arguments

[a] People usually restrain the number of parameters to 6 in graphical systems (e.g. SADT [26], C2 [4]). Of course, this number can be changed according to the user's graphical environment.

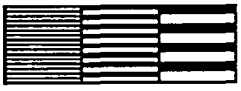

Figure 9. The grain's three level of selection

object with a larger size is more visible than an object with a smaller size. *Orientation is quite selective too* and offers four levels of distinction described in Figure 10.

The laws of graphics allow us, by combining only three levels of grain selectivity and four levels of orientation selectivity, to select and pinpoint easily a dozen distinct elements of a visual language program without using their names, freeing ourselves from a textual constraint. Furthermore, with only the three levels of grain and the four selective directions seen in the previous section, we obtain the palette of Figure 11.

The blank pattern is used to express the dummy argument ('_' in C_Prolog). We have 10 patterns to express 10 distinct variables. As an indication, only 2% of our implementations use more than 10 variables in one clause. The limited number of variables does not really restrict us in practice. Furthermore, building Prolog clauses with more than 10 variables is not really good programming, as the relationships becomes difficult to understand.

Of course, drawing two arguments with distinct patterns does not necessarily mean that the two corresponding variables are distinct, in the same way that writing two variables X and Y does not imply they are not unifiable. Figure 12 shows the application of these patterns on the classical **descendant/2** example.[b]

The descendant relation expresses that *a child is a descendant or a child of a descendant is a descendant*, whose Prolog clauses are, up to variable renaming,:

```
descendant (X, Y):- child (X, Y).
descendant (X, Y):- child (X, Z), descendant (Z, Y).
```

To express functional arguments, we will combine basic arguments symbols as shown in Figure 13, which represents the **tree(Root, Left son, Right son)** functional argument. To get the full argument, the user has to zoom on the functional symbol.

The visual aspect is also a good support for variable annotations. Usual annotations are the explicit universal quantification ($\forall$) for the correct use of negation [27], and the mode ($\downarrow$ known (ground) when executing the Prolog component/ $\uparrow$ ground after the execution of the Prolog component) for further optimizations and proofs [28] and transformations of the Prolog prototype into more deterministic languages [23]. We do not claim that $\forall$, $\downarrow$ and $\uparrow$ are especially meaningful symbols, but as they are

Figure 10. The orientation's four selective directions

[b] **Descendant/2** is the usual Prolog notation to specify that predicate **descendant** has an arity of two.

Figure 11. The palette of levels of selection

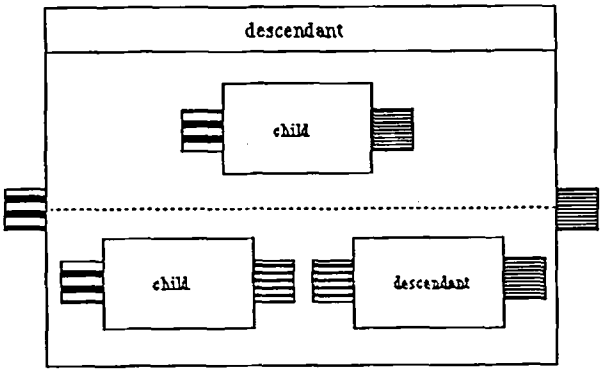

Figure 12. The descendant example

accepted by the Prolog community, we show how our visual representation supports them (cf. Figure 14).

3.2.4. *Using the Laws of Graphics for the Predicates*

The name of a predicate being a constant, we have to write it inside the symbol. A complete visual normalization is not possible because of the large number of possible predicates. However, it can be tried for a limited number of frequently used predefined predicates. For instance, with abstract and representational icons shown in Figure 15, we iconize four well known list manipulation predicates:member/2, not member/2, head/2, tail/2.

Another frequently used predefined predicate is the negation, implemented in Prolog by negation as failure [29]. As negation cannot be expressed by one image [30, 31], we cannot represent it with visual variables (as we represented and and or for instance). So we suggest using the symbol of Figure 16 to express the negation of a Prolog component.

Another frequently used predicate in Prolog programming is the cut. To represent

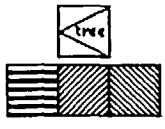

Figure 13. A functional symbol and its expanded graphical view

it, we propose to use the symbol of Figure 17:

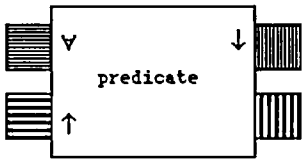

Figure 14. An example of variable annotation

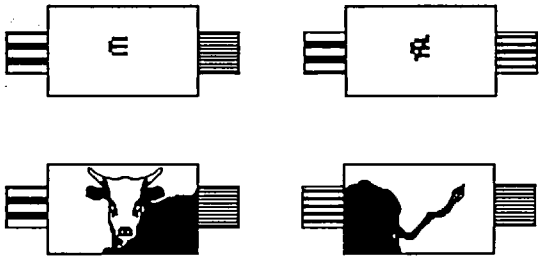

Figure 15. An iconic representation of member/2, not member/2, head/2 and tail/2

3.2.5. Structuration of Clauses

Of course, all the alternatives inside a Prolog clause can only be expressed in a single clause symbol if the arguments are identical. Otherwise, a duplication of clause symbols may be necessary (for clauses containing stop recursion rules, for example). The well known factorial example in Prolog have the direct visual representation of Figure 18. The factorial predicate states that Z is the factorial of $N$and has to be used as a function and also N must be known. The corresponding Prolog clauses are:

```
fact (0, 1).
fact (N, Z):- N > 0, N1 is N − 1, fact (N1, Z1), Z is N * Z1.
```

The variation of size between the two clauses symbols does not transgress the laws of graphics. As a matter of fact, the smaller size of the first fact/2 component symbol clearly indicates that it is less complicated than the second one. But, with a normalization, we can get the more pleasant visual representation of Figure 19.

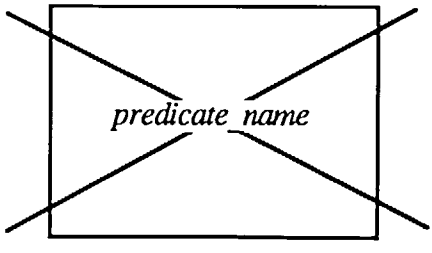

Figure 16. The negation symbol

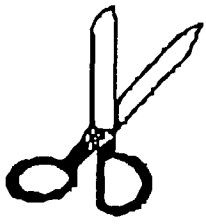

Figure 17. The cut symbol

Figure 19 corresponds to the following definition of the **factorial/2** component:

```
fact (N, Z):- N = 0, Z = 1
fact (N, Z):- N > 0, N1 is N - 1, fact (N1, Z1), Z is N * Z1.
```

Here, all the subcomponents have the same size because we have different alternatives of a single clause **fact/2**.

The graphical representation of the factorial is arguable. In particular, the symbols for '$-$' and '$*$' can be seen in different ways ($N1$ is $N-1$; N is $N1-1$; ...). This problem arises from the fact that **factorial/2** is not a relational predicate but a function. As our system has been designed to put forward the relational nature of Prolog clauses, the graphical representations of predicates, which actually correspond to functions, have a lot of chances to be ambiguous. Furthermore, it is difficult to represent graphically arithmetical predicates because arithmetic is already symbolical. So, the best thing to do is to manipulate arithmetic with the classical textual symbols.

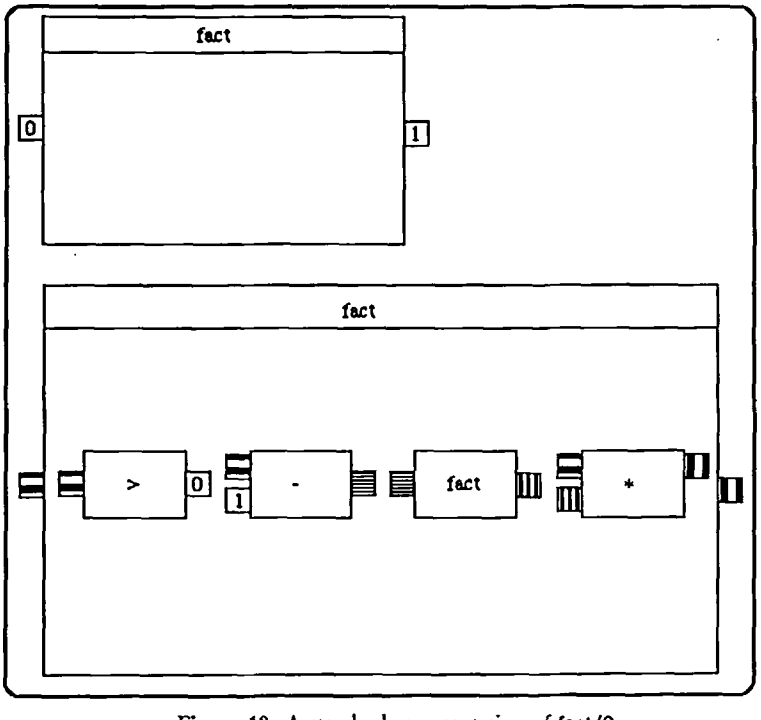

Figure 18. A standard representation of **fact/2**

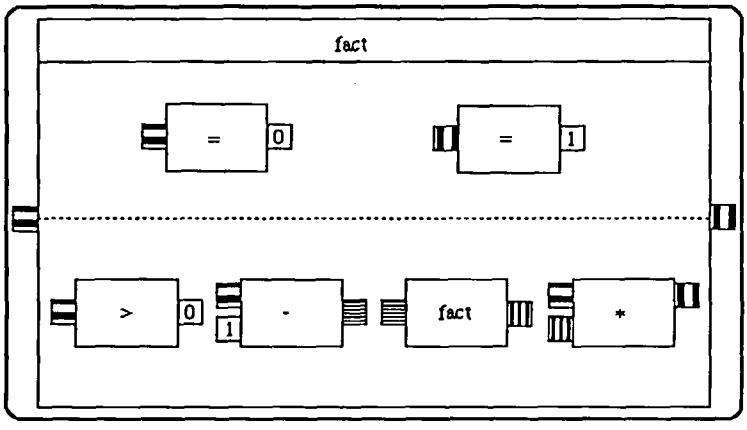

Figure 19. A normalized representation of fact/2

3.3. General Level

3.3.1. Using Visual and Textual Views in a Complementary Way

We believe that purely visual approaches, like Pict/D [32], FP-XL [33], are too exclusive to be completly general. Some programming tasks should not be done in a visual way, since the user will eventually become disappointed and will surely come back to the textual program. Taking inspiration from multicode theory [8] in image mental structure, we think that the visual and the textual representation have distinct optimal domains. However, as in the dual code theory [10], we think that the two representations used in parallel can sometimes give better results than only one.

Our experiments in Prolog lead us to think that some classes of clauses are better suited for a visual manipulation than others and vice versa. That is why we suggest to manipulate our program with multiple views, visual and textual. A Prolog clause better suited for a visual manipulation will not be textually displayed. Actually, it is useless to remind the user how the textual is less adapted than the visual. Conversally, if the user deals with a lot of Prolog facts with constants, best suited to a textual manipulation, it is hazardous to show him an awkward visual view! Of course, the user can always ask that some Prolog clauses be displayed both visually (for a better comprehension) and textually (for more efficiency).

To decide which kinds of Prolog clauses are to be manipulated according to which view, we use the classification of Legeard and Rueher [16], that distinguishes four classes of Prolog clauses:

- *Functional clauses* describing the relational semantics of the program. For instance, the clause descendant/2 is a functional clause of the descendants program. These higher level clauses mainly contain variables. The possible application of the separation variables to represent them implies an interesting visual manipulation.
- *Elementary clauses* insuring the independance of the functional clauses from the choice of data structures. Their semantics is linked to the one of the program. Examples will be given in the next section.

- *Utilitarian clauses* the usual utilitaries used by the basic or functional clauses, their semantics is not linked to the one of the program. For these two classes of clauses, the use of visual or textual programming is a matter of experiment. Some quite high level basic clauses using many structured variables are better manipulated textually, while some low-level utilitaries will have a good visual representation. In the general case, a textual view is more concise and efficient, while a visual view helps understanding.
- *Basic relations* describing the static semantics of the program. For instance, the facts child (lisa, nicole), child (lisa, charles),... are the basic relations of the descendants program. Since these relations contain many constants, they are better suited to a textual manipulation.

Finally, there may be a last class of Prolog clauses: *consistency control clauses* to ensure the correctness of the program. These clauses are easily visually represented too. Of course, the 'principles' given above are only recommendations to the user: he is always free to adapt them to his goals. In Section 4, these classes of Prolog clauses are detailed in an example.

3.3.2. Implementation

To ensure consistency and updates between the graphical and textual views, our software architecture is based upon an abstract syntax tree. Visual and textual views are decompilations of this abstract syntax tree and any modification to either of these two views are reported to the abstract syntax tree, which incrementally sends it back to the views. To assert Prolog clauses from the tuples of logic operators, it is necessary to normalize the logic formulas to obtain a single representation. We have chosen the disjunctive normal form because disjunction is usually represented by clauses having the same head, the predicates of the different rule bodies being connected by conjunctions. A first version on Macintosh with PrologII [19] has been made more powerful (this, however, is still an early prototype) on Sun workstations with Delphia-Prolog.[c]

4. Application

We illustrate our approach on a causal tree management Prolog program. A causal tree is a kind of decision tree which has a dreaded event as root, whose nodes represent events resulting from the conjunction or disjunction of the events associated with their children, and whose leaves are elementary events. These kinds of trees are used to model emergency system in such areas as aeronautics. We will only examine the *qualitative-analysis* function which is aimed at identifying all critical sets of causal trees, i.e. all minimal sets of events leading to the dreaded event.

A causal tree is described in Prolog by and-decompositions, or-decompositions and elementary events.

[c] Delphia-Prolog [34] is a modular Prolog interfaced with a graphical toolbox named Pixia.

4.1. Beginning the Session and Defining the Basic Relations

When we start up the VLP system, the standard menu bar with four pull-down menus appears on the screen (see Figure 20).

The FILE menu contains various items for:

- saving;
- storing into the differents components libraries (e.g. user library, project library, etc.);
- retrieving from the differents components libraries;
- changing between a textual and a visual representation;
- tracing the program;
- leaving the session.

The EDIT menu contains various items for:

- various items for combining components by means of and; and or,
- erasing,
- zooming,
- cut, copy and paste facilities.

The ARGUMENTS menu contains various items for:

- adding an argument;
- unifying two arguments;
- annotating arguments by means of:
  - quantification;
  - ↓ ↑ annotations.

The BUILT-IN menu contains all standard which are available in Cprolog.

First of all, let us define the basic relations by the following Prolog facts:

```
and-decomp (oi, a, b).   and-decomp (c, e, f).   and-decomp (d, f, g).
or-decomp (a, c, 7).     or-decomp (b, d, 8).    or-decomp (e, 4, 3)
or-decomp (g, 5, 6).     or-decomp (f, 1, 2)
basic-evt (1).           basic-evt (2).          basic-evt (3).
basic-evt (4).           basic-evt (5).          basic-evt (6).
basic-evt (7).           basic-evt (8).
```

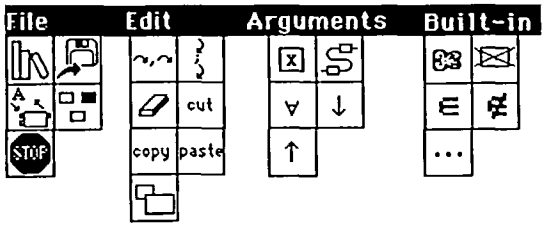

Figure 20. The standard menu bar and pull-down menus

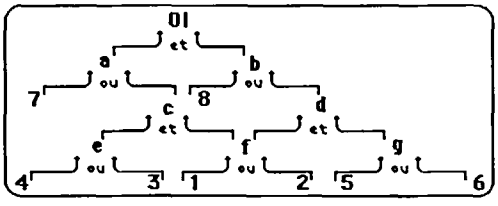

Figure 21. A graphical representation of the causal tree

The basic relations are often the first part of the Prolog program that is built, because these static relations are easily expressed due to the declarativeness of the Prolog programming language. These Prolog facts contain constants that are not easily visually represented but written in a graphic (even if an iconification of these constants is possible!). So, a visual representation of these basic relations is not well adapted and the basic relations will be better built on the textual view with a syntax directed editor. However, the data object which is modelized by those relations could be displayed by using specific visual formalisms (e.g. a graph like the one of Figure 21) but this is not yet possible in VLP. For future visual use, they can nevertheless be saved in the components library with a visual representation, as indicated in Figure 22.

4.2. Defining Functional Clauses

The definition of the qualitative-analysis/2 function is based upon the second order predicate setof/3 as shown by the following clause:

```
qualitative-analysis (Ev, All-Sets):-
    setof (S, set (Ev, S), Sets),
    critical (Sets, All-Sets).
```

Second order predicates are out of the scope of our relational symbolism and they have therefore to be represented either in a textual or in an iconic way; here, we have choosen the textual one but this choice is somewhat arbitrary.

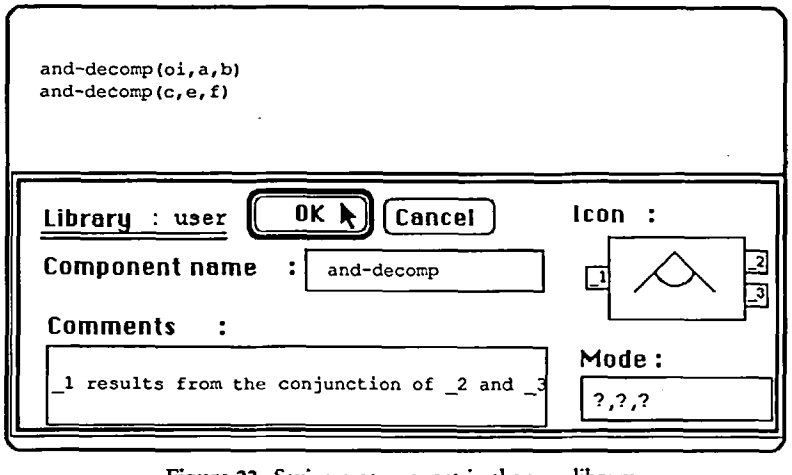

Figure 22. Saving a component in the user library

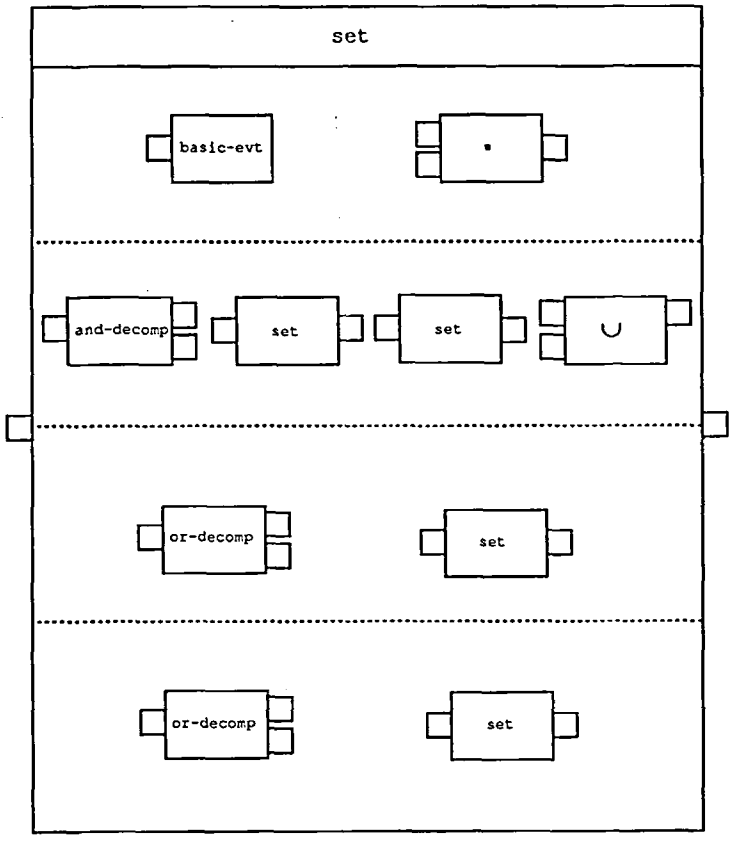

Figure 23(a). The structure of the set/2 relationship

The user can now ask to build visually the set/2 component. By reusing the and-decomp/2, or-decomp/2, basic-evt/1 components, the predefined icons (e.g. the list constructor '.', and the set union operator '∪') and doing logical connections, he gets the result of Figure 23(a).

Now, he just has to fill the arguments with the possible patterns or copy from a previous pattern in case of identical variables to obtain the set/2 component of Figure 23(b). When filling with a pattern, the palette is displayed. When unifying the two arguments, the current pattern is displayed at the left of the palette.

This figure expresses that the set of basic events leading to an event *Ev* is:

- the set containing only Ev if *Ev* is a basic event;
- the union of the sets of basic events associated with two sub-events, if the conjunction of these sub-events leads to *Ev*;
- the set of basic events associated with one of the sub-events connected to *Ev* by an or-decomposition relationship.

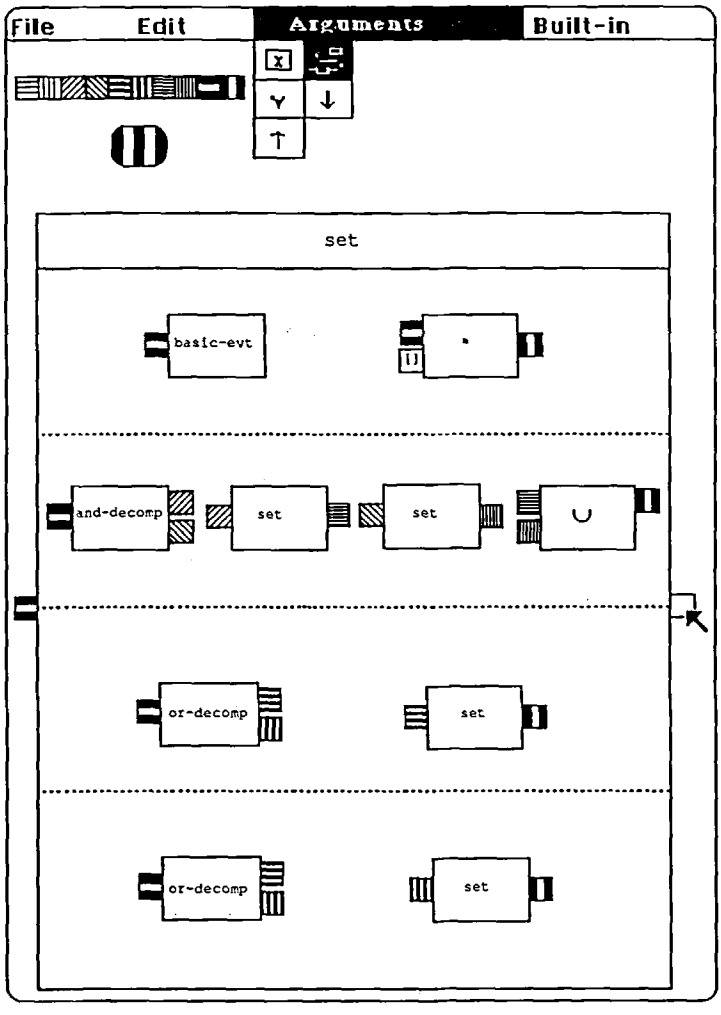

Figure 23(b). Using the standard pattern for defining arguments

Figure 23(b) corresponds to the following Prolog clauses.

```
set (Ev, [Ev]):-
   basic-evt (Ev).
set (Ev, S):-
   and-decomp (Ev, Sev1, Sev2),
   set (Sev1, S1),
   set (Sev2, S2),
   union (S1, S2, S).
set (Ev, S):-
   or-decomp (Ev, Sev1, Sev2),
   set (Sev1, S).
set (Ev, S):-
   or-decomp (Ev, Sev1, Sev2),
   set (Sev2, S).
```

We can notice that the variable-free graphical representation constrains the user to have a relational view of the invertible predicate.

The critical/2 relation which defines the critical sets (i.e. the minimal ones) is displayed in Figures 24(a–d).

Figure 24(a) describes the rewriting rule of critical/2 in critical'/3. It outlines better the encapsulation of these two relations than the corresponding Prolog clause given below.

```
critical (Sets, Criticals-Sets):-
  critical' (Sets, Sets, Criticals-Sets).
```

To define critical'/3 we will use both textual and graphical representations. As a matter of fact, the first clause required for defining critical'/3 contains two constants and a textual representation is thus well adapted:

```
critical'([], Sets, []).
```

The two other clauses needed for defining this relation contain many abstract variables and a graphical representation is better suited for them [cf. Figures 24(b)]. The corresponding Prolog code is:

```
critical' ([S|Others-sets], Sets, [S|Others-Criticals-Sets]):-
  retract (S, Sets, Sets-without-S),
  not (exist-subset(S, Sets-without-S)),
  critical' (Others-Sets, Sets, Others-Criticals-Sets).

critical' ([S|Others-Sets], Sets, Others-Criticals-Sets):-
  retract (S, Sets, Sets-without-S),
  subset (S, Sets-without-S),
  critical' (Others-Sets, Sets, Others-Criticals-Sets).
```

A normalized representation of the complete relation is displayed in Figure 24(c). We may remark that the graphical representation enables a better understanding of the overall structure than the textual representation.

The two elementary clauses used to define critical'/3 are shown in Figure 24(d); the corresponding Prolog clauses are:

```
subset (S, [Set|Sets]):-
  inclusion(Set, S).
subset (S, [_|Sets]):-
  subset (S, Sets).
retract (S, [S|Sets], [Sets]).
retract (S, [X|Sets], [X|Set']):-
  retract (S, [Sets], Sets').
```

4.3. Execution

To determine all the minimal sets of basic events that could lead to the dreadful event oi, we have to compute the question qualitative-analysis (oi, All-sets) which calls the non-deterministic predicate set/2. In order to illustrate the benefit of our approach when testing, we are now going to visualize the execution of set (oi, S). The synospsis of the execution is shown in Figure 25. The inverted display of or-decomp/3 indicates that this component is currently being executed. The shaded predicates have been backtracked. However, often the result of the Prolog execution is an

instanciation of the free variables into ground terms (i.e. constants), the results are displayed textually, as it can also be seen in Figure 25. The set (oi, [7,8] relation, displayed in the textual window, is the answers already given by execution of the set/2 component.

Here are some comments to explain the displayed state of the program. When computing set (oi, S), the first clause will not succeed because oi is not a basic event. But oi can be decomposed in a and b by using the second clause. The event a is

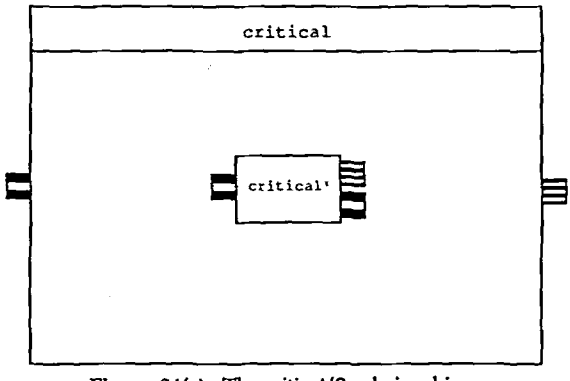

Figure 24(a). The critical/2 relationship

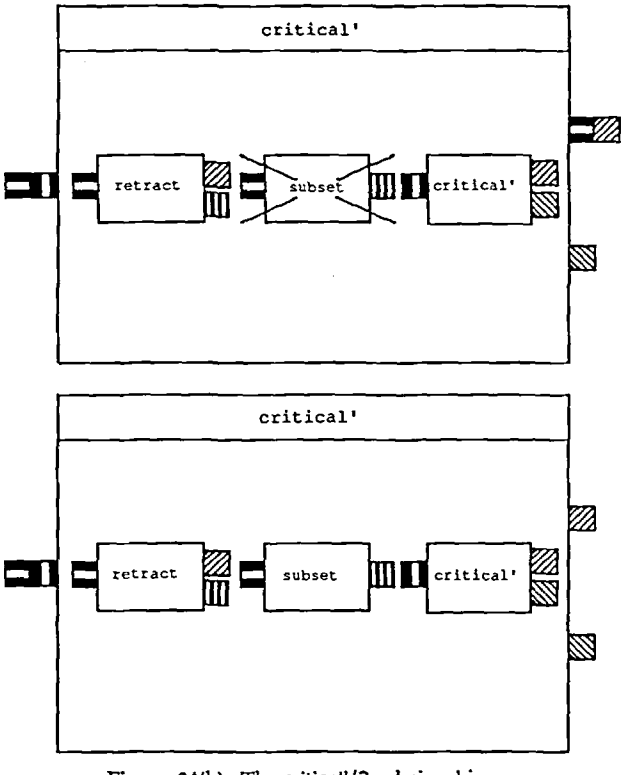

Figure 24(b). The critical'/3 relationship

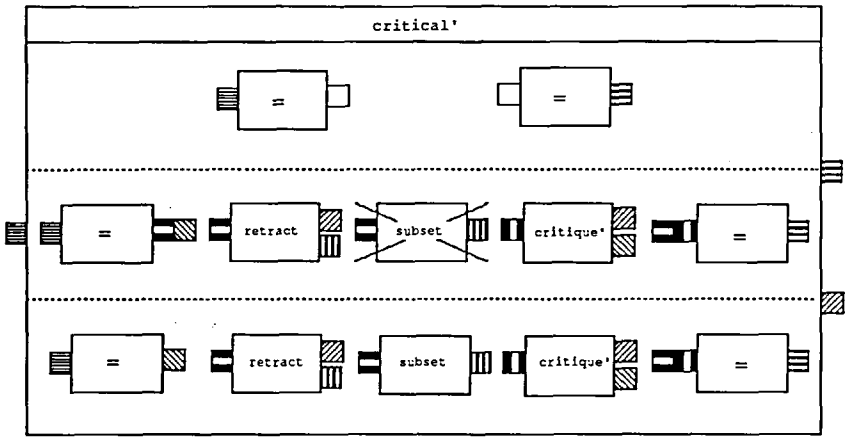

Figure 24(c). A normalize representation of critical'/3

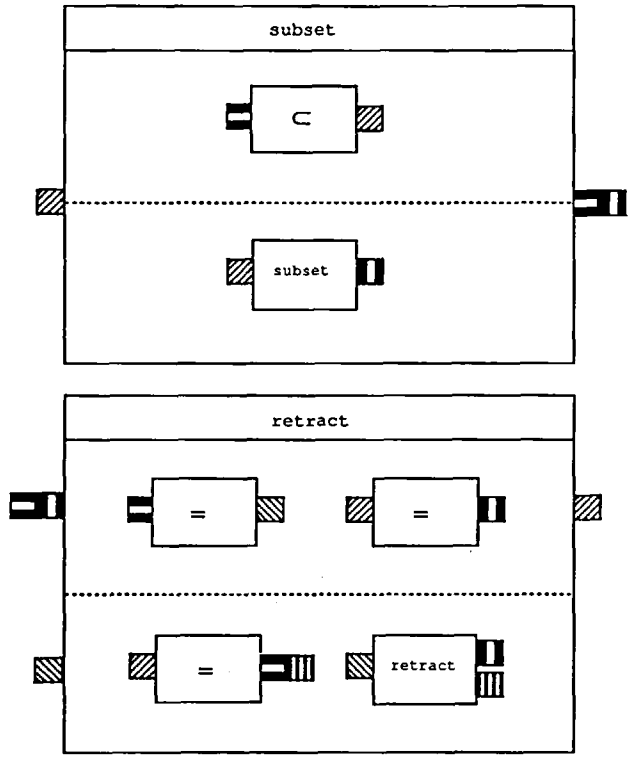

Figure 24(d). The subset/2 and retract/3 elementary clauses

neither a basic one nor can it be decomposed by the and-decomp/3 predicate. Hence the third clause will be used and succeed by computing set (a, [7]). The event b is not basic and cannot be decomposed by the and-decomp/3 predicate. It can be decomposed in 8 and d with the or-decomp/3 predicate. In this way set (b, [8]) can be computed and we get the first answer set (oi, [7, 8]).

Let us now examine the backtrack step: 'U' cannot be backtracked, but set (b,

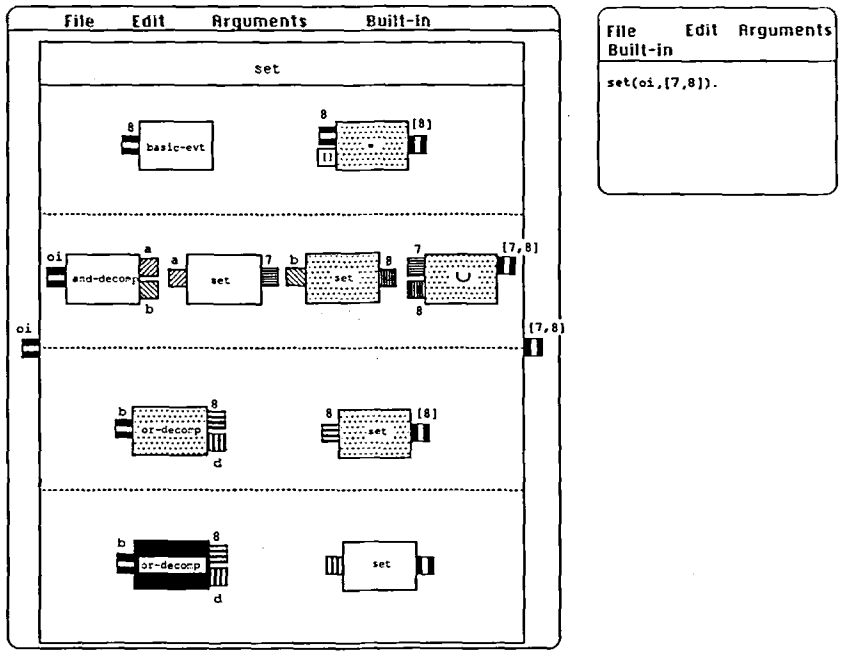

Figure 25. Execution of the set/2 relationship

[8]) can be undo. Thus, we will compute the forth clause where we are going to solve the set (d, S) goal.

5. Discussion

We have introduced the concept of VLP in this paper. VLP, a visual logic programming language, is based upon some simple psychological laws concerning the perception of graphics. This work was started with the aim of using graphics not as an art, but as a system of strict and simple signs allowing a better understanding of set relationships which exist in graphics.

Conforming to the laws of graphics, the resulting visual logic program is more expressive than just the textual Prolog program. The choices of symbols or icons for representing the basic objects (i.e. Prolog terms) have been made with care for simplicity and efficiency in order to enable a graphical normalization. The graphical assembly rules have been designed to yield an algebra of construction. The picture variables (i.e. X and Y plane dimensions) enabled us to express clearly the logical connections between components while the graphical separation variables (i.e. grain, orientation and form) allowed us to represent terms which can be unified without naming them. Hence, we are rid of many textual constraints and we produce a graphic to be watched. Our box representation of the clauses also involves a strong encapsulation which eliminates the *spaghetti ball* phenomenon of numerous graphical representations.

However, visual programming cannot efficiently cover all the requirements of software development. In particular, graphical interaction appears too cumbersome a means for entering elementary relations. Taking inspiration from multicode theory which argues that visual and textual representation have distinct optimal domains, we propose using graphical and textual languages in a complementary way. It is possible to link their distinct optimal domains to the classification of Prolog clauses when prototyping (e.g. textual representation of basic relations but also of second order clauses). But the best suited view cannot be defined for every kind of clause and hence we have to enable parallel use of both representations.

First experiments showed that the visual representation really improves both static and dynamic understanding of Prolog programs. Visual and iconic representations (e.g. critical'/3, not/1, ' ∪ ') allow a global understanding of the meaning of displayed relationships while encapsulating of the relations symbols enables the apprehension of their structure. The higher degree of abstraction for Prolog variables prevents both beginners and experienced users from viewing their programs in a procedural way. Hence, the use of identificator-free variables emphazise the relational nature of Prolog predicates (e.g. the invertibility of set/3). Of course, many examples do not become easier to program when using visual logic programming but the resulting program is easier to understand and errors such as loops will be detected faster. The construction of the program if not quicker, can at least be more convenient in a graphical way. For some kinds of clauses like basic relations, the textual representation remains a better support. Thus, using visual and textual representation of logic programs seems to be a promising approach for rapid prototyping and a valuable support for documenting, normalizing and hence reusing Prolog components. However, extensive experiments are still necessary to validate VLP and to refine it. Likewise, we still continue to explore the new works on the properties of graphics (e.g. [9]) in order to improve the visual representation in a systematic way.

VLP would of course have to be enhanced and improved in many ways for the system to become production quality software. First, the subset of built-in predicates which are defined in an iconic way should be extended to cover at least all standard predicates. The user interface can also be improved significantly. In addition, more work is required to support program debugging (e.g. visualization of the complete execution space, use of colours to highlight failure *vs.* success, etc.). These enhancements will be incorporated in future versions, but before implementing them we would like to get more experimental validation on VLP.

In this paper, we have only explored the capabilities of the laws of graphics for visual logic programming. Of course, the use of these laws would be of great interest for other programming paradigms. However, the application to other programming paradigms depends on the syntactical complexity of the underlying programming language. As a matter of fact, it would certainly be difficult to apply our approach to languages like Ada because we would need to define many basic symbols that could make visual normalization difficult. The use of the laws of graphics could also improve more conventional editors. For instance, a syntax driven editor should neither enable different indentations for the same syntactical structure, nor allow the use of the same font format for different objects (e.g. inherited features and local redefined features in an object-oriented programming language).

References

1. S. K. Chang (1987) Visual languages: a tutorial and survey. *IEEE Software* **4**, 29–39.
2. E. P. Glinert (1986) Towards "second generation" interactive graphical programming environments. In: *Proceedings of the IEEE Computer Society Workshop on Visual Languages*, pp. 61–70.
3. S. P. Reiss (1986) An object-oriented framework for graphical programming. *Sigplan Notices* **21**, (10), 49–57.
4. E. P. Glinert, E. P. Kopache, M. E. & D. W. McIntyre (1990) Exploring the general-purpose visual alternative. *Journal of Visual Languages and Computing* **1**, 3–39.
5. J. Bertin (1981) *Graphics and the Graphic Information Processing* Walter de Gruyter, Berlin.
6. J. Bertin (1983) *Semiology of Graphics* University of Wisconsin Press, Madison, U.S.A., 415 pp.
7. S. M. Kosslyn, S. Pinker, G. E. Smith & S. P. Schwartz (1979) On the demystification of mental imagery. *The Behavioral and Brain Sciences* **2**, 535–581.
8. S. M. Kosslyn (1980) *Image and Mind* Harvard University Press, Cambridge, Massachussets.
9. S. M. Kosslyn & C. F. Chabris (1990) Naming pictures. *Journal of Visual Languages and Computing* **1**, 77–95.
10. A. Paivio (1971) *Imagery and Verbal Processes* Holt, Rinehart and Winston, New York.
11. M. Brayshaw & M. Eisenstadt (1989) A practical tracer for prolog. HCRL technical report no 42, The Open University, Milton Keynes, U.K.
12. R. Baecker (1988) Enhancing program readability and comprehensibility with tools for program visualization In: *Proceedings of the ICSE-10.* London, pp. 356–366.
13. R. Kowalski (1984) Software engineering and artificial intelligence in new generation computing. *Future Generations Computer Systems* **1**, 39–49.
14. A. D. Dewar & J. G. Clearly (1986) Graphical display of complex information with a Prolog debugger. *International Journal of Man Machine Studies* **25**, 503–521.
15. M. Eisenstadt & M. Brayshaw (1988) The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. *Journal of Logic Programming* **5**, 277–342.
16. B. Legeard & M. Rueher (1989) An approach for prototyping in Prolog (in French). *TSI* **8**, 423–438.
17. B. A. Myers (1990) Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing* **1**, 97–123.
18. R. Kowalski (1983) *Logic for Problem Solving* Elsevier, New York.
19. A. Colmerauer (1985) Prolog in 10 Figures. *Communications of the AMC* **28**, 1296–1310.
20. W. F. Clocksin & C. S. Mellish (1987) *Programming in Prolog, 3rd edn* Springer Verlag.
21. G. Raeder (1985) A survey of current graphical programming techniques. *IEEE Computer* **18**, (8), 11–25.
22. M. Rueher, M. C. Thomas, A. Gubert, & D. Ladret (1986) A prolog based graphical approach for task-level specifications. In *Proceedings of the NATO Advanced Workshop on Languages for Sensor Based Control in Robotics.* Pise.
23. M. Rueher (1987) From specification to design: an approach based on rapid prototyping. In: *Proceedings of the 4th International Workshop on Software Specification and Design*, pp. 129–133.
24. R. V. Rubin, E. J. Golin, & S. P. Reiss (1985) ThinkPad: a graphical system for programming by demonstration. *IEEE Software* **2**, 73–79.
25. Gri (1981) *The Science of Programming* Springer Verlag, 364p.
26. D. T. Ross (1977) Structured analysis: a language for communication ideas. *IEEE-TSE*, SE-3.
27. Nai (1986) Negation and quantifiers in NU-PROLOG. In: *Proceedings of the 3rd International Conference on Logic Programming.* London, U.K., July, pp. 624–634.

28. P. Deransart (1989) Proofs of declarative properties of logic programs In: *Proceedings Tapsoft' 89.* Barcelone, 13–17 March, pp. 207–226.
29. K. L. Clark (1978) Negation as failure. In: *Logic and Data Bases* H. Gallaire & J. Minker (eds) New York, Plenum Press, pp. 292–322.
30. G. Cohen (1983) *The Psychology of Cognition, 2nd edn* London, Academic Press.
31. P. E. Morris &. P. J. Hampson (1983) *Imagery and Consciousness* London, Academic Press.
32. E. P. Glinert & S. L. Tanimoto (1984) Pict: An interactive graphical programming environment. *IEEE Computer* **17** (11), 7–25.
33. F. G. Pagan (1987) A graphical FP language. *Sigplan Notices* **22** (3), 21–39.
34. Delphia (1988) Delphia Prolog, User Manual, Release 1.5. Grenoble.