Graphical Representation of Logic Programs and Their Behavior

Hikmet Senay Santos G. Lazzeri

Department of Electrical Engineering and Computer Science
The George Washington University
Washington, D.C. 20052

Abstract

This paper describes a set of graphs that form a coherent framework for the design and implementation of a graphical environment for logic programming. The set includes three types of graphs which collectively represent the structure of logic programs and their behavior. The representational formalism underlying the development of these graphs is based on cyclic AND/OR graphs. One of the most important properties of the representation scheme presented in this paper is the ease with which the program behavior can be mapped onto the source code. This is accomplished by folding the graph that represents program behavior onto the graph that represents the source code. In our representation scheme, information regarding the generation of values for variables is represented using a separate set of binding dependency graphs which are particularly useful in debugging.

1. Introduction

It has long been recognized that graphical representation of programs can considerably increase programmers' ability to effectively develop, understand, and debug complex computer programs. This recognition has naturally led to the development of programming environments that provide graphical tools for editing, monitoring and debugging programs in a variety of programming languages. Although the development of graphical tools for logic programming is no exception, the tools that have been developed so far primarily concentrate on monitoring program behavior for debugging purposes. Any change on the program code still needs to be done textually. This dual mode of operation not only necessitates a major context switch for the programmer but also complicates the software development process by breaking the essential linkage between editing and debugging. In order to alleviate the problem, there is a need for a set of compatible representation schemes that enforce consistency in all phases of logic program development.

In this paper, we describe such a set of graphs that collectively represent the structure of logic programs and their behavior at different levels of detail. The structure of a logic program is represented by a static graph corresponding to its source code while its behavior for a given goal is presented in terms of a dynamic graph showing individual steps of the solution and a set of binding dependency graphs illustrating how values of variables are generated during execution. Since the behavior of a program is generally dictated by its structure, all these graphs are closely coupled to emphasize this causal relationship which is of primary importance in program development.

In order to make this relationship clearly visible, the representational formalism used is based on cyclic AND/OR graphs. This differs significantly from the traditional AND/OR tree-based representations which are commonly used by the previous graphical debuggers for logic programming. Although an AND/OR tree-based representation provides a clear account of the dynamic behavior of a logic program, it needs to be linked to the source code and the variable bindings generated during program execution. Establishing the linkage between an AND/OR tree and its source code is, however, not a trivial task since the source code in general cannot be represented by a similar tree structure. In our approach, this linkage is established by representing both the dynamic behavior of a logic program and its source code in terms of cyclic AND/OR graphs.

Besides making this linkage clearly visible to the programmer, the use of cyclic AND/OR graphs has other important benefits: (1) a considerable saving in screen space can be obtained due to the presentation of run-time information on a fixed-size graph even when the corresponding proof tree is infinite; (2) the declarative and procedural aspects of logic programming can be effectively conveyed using a single graph; and (3) the entire software development can take place in an integrated programming environment where both program editing and debugging are done graphically.

The only apparent drawback of using cyclic AND/OR graphs in presenting the dynamic behavior of a logic program is possibly hiding too much information, perhaps some crucial pieces, about program execution by confining the presentation to a fixed-size execution graph. Fortunately, this drawback can be alleviated by unfolding the graph which essentially maps to an AND/OR tree without any information loss as shown in Section 3.

2. Static Graph

A logic program is a finite set of clauses defining rela-
tionships between objects in a particular domain of discourse. In general, a clause is the most basic statement which is an implication of the form \( p \leftarrow q_1, \ldots, q_n \) containing precisely one atomic formula in its consequence (head) and a conjunction of \( n \) atomic formulas in its antecedent (body) where \( n \geq 0 \). Declaratively, a logic program clause states that, for each consistent instantiation of variables appearing in the clause, \( p \) is true if all \( q_1, \ldots, q_n \) are true. Procedurally, the same clause specifies a mechanism in which \( p \) is solved by first solving \( q_1, \ldots, q_n \). Since the method of solution is, in principle, determined by the simple structure of clauses, it is often sufficient to develop a logic program by concentrating on its declarative meaning. This is indeed of practical importance because the declarative interpretation of a logic program, which in essence reflects the structure of the underlying domain, is usually easier to understand than its procedural interpretation.

In a graphical environment, the declarative style of logic programming can conveniently be supported by representing logic programs in terms of AND/OR graphs which effectively present various relationships among program clauses, and thus, the implicit structure of the underlying domain.

In an AND/OR graph, a logic program (Horn) clause is represented by an AND node with a single incoming arc, corresponding to the clause head, and \( n \) outgoing arcs, each corresponding to an atomic formula in the clause body. In this representation, each arc is labeled with a predicate name and arity, and a set of arguments. For instance, a clause of the form \( p \leftarrow q_1, \ldots, q_n \) is represented by a subgraph shown in Figure 1. Declaratively, an AND node states that, for each consistent instantiation of arguments attached to the incoming and outgoing arcs of the node, the relation (predicate) labeling the incoming arc is true if all relations labeling the outgoing arcs are true. Since the declarative interpretation of a clause can be obtained from the corresponding AND node in a straightforward manner, the declarative style of logic programming is preserved on a clause-by-clause basis.

Although the declarative style of programming is often preferable in logic program development, the procedural side cannot be completely ignored for efficiency reasons. In order to facilitate the procedural interpretation of a logic program, it is customary to consider the entire set of clauses about the same relationship as a procedure definition where both the set of clauses belonging to the definition and the atomic formulas in their bodies are ordered in some consistent manner. Following the traditional procedure-oriented view of predicate definition, a set of clauses about the same relationship is represented by a procedure box in our AND/OR graph-based representation. In this representation, there is an outgoing arc from a procedure box to an AND node for each clause in the definition as shown in Figure 2. Since a procedure box gives a set of alternative clauses to solve a given goal whose name/arity matches with that of the corresponding procedure, it constitutes an OR node in an AND/OR graph. These alternatives are searched from left to right, in a way similar to the linear ordering of clauses, during unification (clause selection). Atomic formulas (subgoals) in a clause body are also ordered from left to right following the traditional ordering. Furthermore, it is straightforward to incorporate additional control constructs, such as cut, into a procedure definition in our representation scheme. Since the behavior of a program is also presented on the same graph, as described in Section 3, the effects of such control constructs can easily be conveyed along with the procedure definition.

The following classical example about the family relationships further illustrates the use of AND/OR graphs in representing logic programs. In Figure 3, the logic program whose textual form is given on the left is graphically presented by an AND/OR graph representing the static structure of the program on the right. When an AND/OR graph is used to merely display the static structure of a logic program, it is called a static graph. In Figure 3, the static graph has eight AND nodes (blank circles) which correspond to the eight clauses of the program. Since some of the clauses, such as all clauses of the male/1 relationship, are unit clauses, that is, clauses with no atomic formulas in their bodies, the AND nodes corresponding to them have no outgoing arcs. In addition to representing all individual clauses of a logic program, a static graph also displays the relationships among them which essentially reflects the structure of the underlying domain. The logic program shown in Figure 3 consists of five clause sets (procedures) each of which is represented by a procedure box. Of these five procedures, the one representing the father/2 relation is undefined as indicated by the nonexistence of any outgoing arc from the corresponding procedure box. However, it is straightforward to

![Figure 1](image1.png)

**Figure 1.** An AND/OR graph node representing a clause of the form \( p \leftarrow q_1, \ldots, q_n \).

![Figure 2](image2.png)

**Figure 2.** A procedure box with multiple clauses. (Only clause heads and the corresponding AND nodes are shown for simplicity.)
son(X,Y):-parent(Y,X),male(X).
parent(X,Y):-mother(X,Y).
parent(X,Y):-father(X,Y).
mother(sue,tom).
mother(jane,lisa).
male(tom).
male(bill).
male(joe).

Figure 3. Textual and graphical representations of a simple logic program about the family relationships.

define such procedures by drawing an outgoing arc from the procedure box for each clause, labeling the arc with arguments of the clause head, and then specifying each atomic formula in the clause body. An atomic formula (subgoal) in a clause body is simply specified by directing an outgoing arc from the AND node representing the clause to an appropriate procedure box, and then, specifying arguments of the atomic formula as a label for the arc.

Since an arc corresponding to an atomic formula in a clause body can be directed to any procedure box, including those whose definitions directly or indirectly include the clause being defined, it is possible to represent recursive logic programs using cyclic AND/OR graphs. This is important because most logic programs involve recursive definition of data structures that have no definite size, such as lists, trees and graphs. Because of their recursive nature, most logic programs cannot be represented in terms of tree structures such as those used in displaying the dynamic behavior of logic programs [1,2]. The cyclic AND/OR graphs, however, can effectively represent the static structure of recursive logic programs as well as their dynamic behavior.

In order to illustrate the representation of a recursive logic program using a cyclic AND/OR graph, consider the program shown in Figure 4(a). The cyclic AND/OR graph corresponding to this program is shown in Figure 4(b). In both figures, the atomic formulas in the program and their counterparts in the cyclic AND/OR graph are labeled with numbers to emphasize the one-to-one correspondence between the textual and graphical forms of the program. The recursive call (labeled 5) in the body of the second clause of location/2 is simply represented by directing the second outgoing arc from the corresponding AND node to the procedure box of location/2.

Although cyclic AND/OR graphs, as described in this section, are general enough to completely represent the source code of any logic program, they may become cluttered when a cycle resulting from a recursive call involves several procedure definitions. In such cases, directing an arc that represents a recursive call to an appropriate procedure box becomes difficult without crossing other parts of the graph. Since this is essentially a routing problem, it can either be solved by using routing techniques or be avoided by duplicating the procedure box without giving its definition. Our approach to handling this problem is based on the latter because of its simplicity.

Besides the difficulty involved in representing long cycles in a recursive logic program, the representation of large procedures also poses a problem. Even though AND/OR graphs are, in principle, sufficient to represent any number of clauses in a procedure definition using the procedure box symbol, introduced in this section, it is in practice impossible to completely represent a procedure with a large number of clauses. Another similar problem arises when an atomic formula has a large number of or
long arguments. Both of these problems are primarily attributable to the limited screen space in which an AND/OR graph, or any part of it, would be displayed. However, these problems can be alleviated by selectively displaying a small part of a procedure definition or of arguments.

3. Dynamic Graph

Cyclic AND/OR graphs used to represent static structure of logic programs, as described in the previous section, also form a natural framework to present the dynamic behavior of these programs. In general, the dynamic (run-time) behavior of a logic program is characterized in terms of successive goal reductions performed to solve a given goal (query) and variable bindings generated during the solution of that goal. While the declarative meaning of a logic program is sufficient to delineate its behavior, the procedural steps taken during the solution of a goal should also be presented to provide a clear account of the program behavior. Since a cyclic AND/OR graph has an unambiguous procedural interpretation, it can conveniently be used to display the dynamic behavior of a logic program by graphical animation of these procedural steps.

Even though there are very few systems that present behavior of logic programs graphically, the solution of a goal with respect to a logic program is customarily represented using graphical techniques in the literature. A proof tree representing successive goal reductions in the solution of a goal is an example of this practice. In a proof tree, the root node corresponds to the initial goal whose solution is represented by the entire tree. The internal nodes of the tree correspond to the goals which are obtained by goal reduction during the computation. The subtrees under these nodes again represent the solution of the corresponding goals. The solution of the goals at the leaf nodes are trivially obtained by matching these goals with unit program clauses. For instance, the solution of the goal given for determining the location of 'lincoln' according to the location program in Figure 4 is represented by the proof tree shown in Figure 5. Even though a proof tree gives a brief account of the solution steps performed during the computation, it only does so for the goals that have been solved in the process. However, it presents neither the goals that have been tried but unsolved nor the alternative clauses that could have been used to find an alternative solution. Nevertheless, both types of information are important in examining the program behavior.

AND/OR trees, that are also commonly used for presenting the behavior of logic programs graphically, eliminate these shortcomings as well as present the type of information conveyed by proof trees. An investigation of notational formalisms by Pain and Bundy [4] concluded that AND/OR trees offered the greatest potential in terms of clarity of explanation, but that they suffered from several deficiencies including:

(1) It is not immediately clear when a goal has successfully been solved;
(2) It is difficult to see which goals are outstanding at any moment - the current goal is not immediately obvious;
(3) The variable bindings are not clearly displayed;
(4) To keep the different environments of recursive calls clear, the variables have to be renamed - their origins are not always clear;
(5) There is no direct link to the clauses in the program.

In an attempt to alleviate these deficiencies, Eisenstadt and Bradshaw introduced AORTA diagrams [2] which are essentially augmented AND/OR trees. Although AORTA diagrams remove the first four deficiencies listed above, the link that they establish between the representation of program behavior and its source code is rather weak. According to Eisenstadt and Bradshaw, the user must obtain a code listing or display the code in a separate window on the screen, in order to clearly see the linkage between the clauses in the program and the AORTA diagram. For instance, the behavior of the 'location' program in solving the goal 'location(lincoln,X)' is given by the AORTA diagram shown in Figure 6. Even though the graphical conventions used in Figure 6 are slightly different than those used by Eisenstadt and Bradshaw, the essential characteristics of the representation are identical.

In Figure 6, the arcs with argument labels, that are incoming to the procedure boxes, indicate goals which are reduced to subgoals at the succeeding levels. The arcs represented by dashed lines show branches that have been traversed without success whereas the arcs represented by thick solid lines show branches that have been traversed with success. In fact, the subtree outlined by the thick solid lines exactly corresponds to the proof tree of the given goal. All other arcs which are represented by thin solid lines are the branches that have not been traversed during the program execution. Consistent with the graphical conventions used in Section 2, the rectangular boxes represent the OR nodes while the circles stand for the AND nodes of the AORTA diagram. In Figure 6, the circles are also coded to convey the information regarding the involvement of the corresponding program clauses in the

![Figure 5](image_url)

**Figure 5.** The proof tree representing the solution of `location(lincoln,X)` according to the location program given in Figure 4.
A circle with a minus sign indicates that the corresponding program clause has failed to unify with the goal. A circle with a plus sign indicates that the corresponding clause head has unified with the goal, but no proof has been found for its body. A circle filled with black indicates that the corresponding clause has been used in the solution of the goal while all other circles with empty interior represent clauses that have not been tried during the computation.

Even though augmented AND/OR trees, or AORTA diagrams, are useful to give a clear account of the dynamic behavior of logic programs, they are not very efficient in utilizing the limited screen space. When the computation involves several recursive calls or is nonterminating due to errors in the program, the screen space may exhaust very rapidly. Furthermore, establishing the linkage between an augmented AND/OR tree and the corresponding source code may require a considerable effort on the programmer's part when the program is complex. For instance, it is not clear what the code segment is that corresponds to the second clause of the location/2 procedure which appears at the right lower corner of Figure 6. Unless a complete definition of location/2 procedure is visible on the screen, the programmer has no way of extracting that information from the augmented AND/OR tree.

In general, using tree structures to present dynamic behavior of logic programs is bound to suffer from such difficulties because of the recursive nature of logic programs. However, it is possible to convey the same information without these difficulties by folding augmented AND/OR trees onto cyclic AND/OR graphs. The folding process is essentially equivalent to merging all procedure boxes having the same name into a single procedure box and redirecting all incoming arcs of these boxes to the combined procedure box. For instance, the augmented AND/OR tree shown in Figure 6 can be folded by breaking it at the points indicated by the shaded triangles, superimposing each subtree resulting from this process on top of the others in some orderly manner, and redirecting the broken arcs to the only procedure box that represents the location/2 relation. The result of this process, which is an augmented cyclic AND/OR graph, is shown in Figure 7. Essentially, this graph has the same structure as that of the cyclic AND/OR graph shown in Figure 4(b) which represents the static structure of the location program. The only difference between these graphs is the way they present clause arguments. In the static graph, all arguments are presented as they appear in the source code. In the other, some of the arguments are hidden for the sake of brevity and the ones that are visible are shown with their values. However, it is possible to make all arguments vis-

Figure 6. An augmented AND/OR tree displaying the execution of the 'location' program for the query ?- location(lincoln,X).
Figure 7. The final snapshot of the augmented cyclic AND/OR graph that represents the trace of the 'location' program following the completion of the solution for the goal 'location(lincoln,X)'.

Since both graphs that represent the static structure of a logic program and its behavior have the same structure, the graph that conveys the program behavior can be dynamically generated by superimposing the program trace onto the static graph. In practice, this is done by animating the program behavior on the static graph. Because of their dynamic nature, such graphs are called dynamic graphs. When coupled with facilities that provide the capability to move forward and backward in the trace, the dynamic graphs can effectively present the program behavior without any information loss. Alternatively, a dynamic graph can be completely inspected by unfolding the graph, which is the reverse of the folding process.

4. Binding Dependency Graphs

Another important piece of information that contributes to the delineation of logic program behavior is about how values of variables are generated during the computation. In general, the scope of a variable in a logic program is the instance of a clause in which the variable appears when that instance is created. This means a new set of variables must be created for each instance of a clause that involves in the computation. Values of these variables are determined by substituting terms for variables where a term may be another variable, a constant, or an application of a function symbol to arguments which are terms themselves. The substitution process essentially defines a binding for each variable which depends upon the other symbols. Often, the programmers need to inspect such dependencies to understand the program behavior. While this inspection is traditionally done by looking at the textual trace of a program, it is possible to facilitate the task by presenting the binding dependencies graphically.

In order to define a binding dependency graph, consider the insertion sort program given in Figure 8. In this program, the arithmetic comparison predicates, <=2 and >=2, are not defined because they are often provided as built-in system predicates.

1. sort([X[Xs], Ys];- sort(Xs,Zs), insert(Xs,Ys).
2. sort([], []). 
3. insert(X, [X]).
4. insert(Y, [Y], [Y], Ys);- X = Y, insert(Y, Xs, Ys).
5. insert(Y, [Y], [X], [Y], Ys);- X <= Y.

Figure 8. Insertion sort program.

When this program is run to solve the goal 'sort([2,1],[S])', the substitution sets which are shown in Figure 9 are generated. In Figure 9, there are seven substitution sets, S1-S7, each of which corresponds to the solution of one goal using an instance of one of the program clauses. The order in which these substitution sets are generated is indicated by the index number of each substitution set identifier. Each box representing a substitution set has three parts: (1) a top part which shows the goal being processed, (2) a bottom part which gives the head of the unifying program clause with unique variable names, and (3) a middle part which lists all variable substitutions obtained by unifying the goal at the top with the clause head at the bottom. In the middle part of each box, the arrows (=> or <=) indicate the direction of binding dependencies. The terms on the left of each box are the goal arguments whereas the ones on the right are the arguments of the unifying clause head. Variables that have been instantiated

Figure 9. Substitution sets generated by the 'insertion sort' program during the solution of 'sort([2,1],[S]).'
in the previous steps are surrounded by curly brackets which also include the values. During the unification, the values rather than the variable names are used, but the variable names are kept to uniquely identify how the values are obtained. Starting from a variable which is not surrounded by curly brackets, if the arrows are followed in the backward direction one can easily find all dependencies of the variable. Following the arrows in this fashion essentially defines a binding dependency graph for the variable at which the traversal started. For instance, starting from the input variable $S$ in the substitution set $S_1$, the entire binding dependency graph of $S$, which is shown in Figure 10, can be generated.

Binding dependency graphs are particularly useful in logic program debugging. Previously, they have been implicitly used in the context of rationale program debugging [5]. However, their use in a graphical programming environment has never been explored. When coupled with the graphical techniques described in the preceding sections, they not only facilitate the debugging process, but also provide an access mechanism with which the dynamic graph of a logic program can be browsed through. By pointing to an argument in the dynamic graph, it is possible to identify the procedures, or even the individual clauses, that involve in the generation of the selected argument.

![Figure 10. Binding dependency graph for $S$.](image)

5. Concluding Remarks

This paper described a set of graphs that form a coherent framework for the design and implementation of a graphical environment for logic programming. Using these graphs, it is possible to clearly establish the linkage between the static structure (source code) of logic programs and their dynamic behavior. This is accomplished by folding the augmented AND/OR tree, that represents the execution of a logic program, and superimposing the resulting graph on the static graph that represents the source code. Essentially, this process is information preserving. The graphs resulting form this process represent both the structural and behavioral aspects of logic programs. While static and dynamic graphs form the core of the representation scheme presented in this paper, they do not present information regarding the generation of variable bindings. This information is presented by a set of binding dependency graphs that illustrate how values of variables are generated during the computation.

The graphs described in this paper have been used to form a framework for the design of a graphical environment for logic programming [3]. By utilizing the static and dynamic graphs, this environment allows programmers to monitor the execution of logic programs during debugging. By clustering, hiding, and resizing the components of static and dynamic graphs, the programmers can focus their attention to particular program segments within this environment. Currently, we are in the process of extending this environment by incorporating binding dependency graphs which would further facilitate the debugging process. Our future plans also include the development of a graphical editor for logic programming with which the programmer can develop logic programs graphically.

References


