A Tree Representation for Parallel Problem Solving

L.V. Kalé[1]

Department of Computer Science, University of Illinois

1304 W. Springfield Ave., Urbana, IL–61801

Abstract

A tree-representation for problem-solving suited for parallel processing is proposed. We give a formal definition of REDUCE–OR trees and illustrate it with a detailed example. Each node of the proposed tree denotes a completely described subproblem. When literals share variables, it permits solutions from one literal to prune the search space for the other literals. Attempts to get such pruning with AND–OR trees lose a significant form of 'OR parallelism'. An alternative strategy for searching AND–OR trees leads to the SLD trees, which miss the 'AND-parallelism'. The REDUCE–OR trees are especially useful for problems with a generate-and-test flavor.

1 Introduction

Problem solving is one of the important areas in Artificial Intelligence research, with broad applications. In a problem-reduction system, typically, the problem is expressed as a conjunction of goal-literals. One is also given a set of *methods*. Each method can be used to solve a single goal-literal. It reduces a single goal literal to a sub-problem with zero or more goal-literals. Multiple methods can be used successfully to solve a given goal-literal. The complexity of problem solving systems has been growing consistently. To complete complex problem-solving tasks in a reasonable amount of time, one may have to take recourse to parallel processing. This is all the more likely because of the progress in device technology, which makes processors cheaper, but at the same time takes us closer to the limit of how fast single processors can perform. We are therefore interested in parallel problem solving systems.

The *AND-OR trees* have been the representation of choice for problem solving. This formalism gets complicated when multiple literals of a problem share a variable. Many solutions to this interdependence problem have evolved in past. However, none of these are suitable for parallel processing. Here, we present the *REDUCE-OR trees* as an alternative representation of the problem-solving process. Each node of this tree represents an independent piece of computation which can be solved without reference to any other nodes in the tree. Such pieces can be computed by different processes in a parallel system without undue communication. It captures important forms of parallelism, some of which are missed in the AND–OR as well as the SLD tree.

2 The REDUCE–OR trees

It is now well-established that theorem proving restricted to Horn clauses, problem-solving using problem-reduction, and pure logic programming are different views of the same activity [Kowalski, 1979; Loveland, 1978]. Different terminology has evolved in these domains. So, we now define the terminology used in this paper. The problem solving task is specified by a *top level query* (also called the *theorem*, *problem*, or *goal*), and a set of Horn clauses (*methods*, or *axioms*). A query is a set of positive literals. A Horn clause (simply *clause* in the following) is of the form $A \leftarrow B_1, ..B_n$, where A and B_i are positive literals. A positive literal consists of a predicate name followed by a parenthesized list of *terms*. A term is a constant, variable-name or a function symbol followed by a parenthesized list of terms. Predicate symbols, constants, and function symbols are identifiers that begin with upper-case letters, whereas variables begin with a lower case letter. In the above clause, A is said to be the head of the clause, and '$B_1, ..B_n$', is the body of the clause. A clause with an empty body is called a *fact* and is written simply as 'A.'. A clause that is not a fact is called a rule.

What properties should a tree representation[2] of parallel problem-solving have? It is important that each node of the tree represent an independent sub-problem, which can be solved without referring to the information at the nodes in the tree above it. This will facilitate solving different nodes or subtrees of the tree using different processors, without undue communication. To enforce this constraint, we attach a *partial solution set* (PSS) to each node of the tree. Each member of a PSS attached to a node N is a *substitution* that is a solution to the literal or query that labels N. The independence constraint then states that the PSS for each node must be computed using

[1]This work was supported by the National Science Foundation under grant NSF–CCR–8700988.

[2]For simplicity we talk about *tree* representations. The graph representation can be obtained by merging identical nodes.

information from the PSS's of its children only.

Another important criteria is the ability to handle sub-problems containing literals that share variables. Consider the query (admittedly contrived): 'Square(10,x), Cube(x,y)'. This requests finding values for x and y such that x is square of 10, and y is the cube of x. If the sub-problems are solved independently, the search space for Cube(x,y) is very large, even if we constrain all the numbers to be less than some upperbound. Instead, if we let the solution (or solutions, in the general case) of Square(10,x) to constrain the search space of Cube(x,y), we get a much smaller search space. One must then allow for and handle such ordering constraints. Notice that the order doesn't have to be total. In the clause for S in Figure 1, once a binding for x has been obtained by solving P(t,x), no further reduction in search space can be obtained by enforcing any order between Q and R. Any such ordering destroys opportunities for parallelism. So: Instead of just as a *set* of literals, the top level query as well as bodies of all clauses are assumed to be specified as *partial orders.* We represent this partial order in a graph notation called the Data Join Graph (DJG). Each literal of the query is an arc of a DJG. The nodes of the graph are joining points for data coming from different arcs. There is a single *start* node, with no incoming arcs, and a single *finish* node, with no outgoing arcs, in a DJG.

A few points should be noted. We do not *require* that when two or more literals share a variable, one of them must dominate the others in the partial order, i.e., that there be a single generator literal for every variable. In some cases, it may be more efficient to solve two literals sharing a variable independently, and then intersect the solutions. Conversely, sharing of variables is not necessary to impose an ordering. For example in 'GEN(x), TEST(x), EXTEND(x,y)', it may be prudent to wait for a certification from TEST, before wasting the efforts on extending the binding produced by GEN. Also, for some clauses, one order may be more efficient than another depending on (the values of variables in) the invocation. Here we assume for simplicity that a fixed DJG is used for every invocation. Preliminary ideas on some extensions to allow 'runtime' choice of graphs are in [Kale, 1985].

Consider the problem specified in Figure 1. (The figures appear at the end of the paper). The AND–OR tree for this problem is shown in Figure 2. Examining this tree in view of the criteria stipulated above, one notices some deficiencies in the representation. The independence criteria requires that P, Q and R be solved independently; then there is no way to constrain the search space for Q and R using the solutions obtained from P. Even if we ignore this constraint (accepting the resultant communication penalty, or limiting it by using a shared memory system), the AND–OR trees run into other serious problems. As shown in Section 3, all attempts to use the AND–OR tree in such a situation lead to either the loss of parallelism between Q and R (the AND parallelism) or the loss of OR parallelism in further exploring multiple solutions provided by P.

The REDUCE–OR tree for this problem is shown in Figure 3. The details are explained after the formal definition of these trees. Notice that there are multiple (two) nodes for different instances of literals Q and R. This is a major difference between the AND–OR trees and the REDUCE–OR trees. Another difference is the PSSs attached to all the nodes via dotted lines. Each REDUCE–node except the root corresponds to a clause of the program. (Many nodes may correspond to a single clause, in general). It is written as $R(H,P,PSS)$, where H is instantiated head of some clause C, and P is a partial order comprising the instantiated literals in the body of C. PSS is a set of substitutions that satisfy the literals in P, and hence H. In diagrams, a REDUCE node is denoted by a rectangle, with the head (H) in a box at its top left corner, and the DJG for P in the middle. The PSS is drawn as a box with double lines at the top. The OR node is written as $O(\sigma,G,PSS)$, where σ is a substitution, G is a literal, and PSS is a set of substitutions that satisfy G. It is diagrammed as an oval, with σ in the upper half, and G in the lower. σ stores the context of an OR–node as explained below.

Formal Definition: The REDUCE–OR tree for a query Q, w.r.t. a logic program P, is defined using mutually recursive rules. Rules (1) and (2) specify how to build the tree, and the rules (3) and (4) specify how to collect the PSSs. The root of the tree is $R(\Lambda,Q,PSS)$.

RULE 1: *The children of a REDUCE–node*: Let $R(H,P,PSS)$ be a REDUCE–node. Select a literal, say $G_k \in$ P. Let $P_1,..P_m$ be the predecessors of G_k in the partial order P. Select one OR–child for each i, $1 \leq i \leq m$, say $O(\sigma_i^j,P_i^j,PSS_i^j)$. ($P_i^j$ is an instance of P_i). From each PSS_i^j, select one substitution, say $\delta_i^j \in PSS_i^j$. Let $\pi_i = \sigma_i^j \cdot \delta_i^j$ and $\sigma = \pi_1 \cdot \pi_2 \cdot .. \cdot \pi_m$. For each combination of such selections, if the resultant σ is a consistent substitution, then there is a new OR–child : $O(\sigma,\sigma G_k,PSS')$. The first argument, σ, essentially remembers the 'context' of the OR–node. Without such a σ, the REDUCE–node R1 of Figure 2 would have no way of knowing that {x=A, y=D, z=F} is not a solution.
Informally: there is an OR–node for solving an instance of a literal G_k in P for every consistent composition of a solution to each predecessor of G_k in P.
Note that if $|P|=0$, (the node corresponds to a *fact*), there are no children.

RULE 2: *children of an OR–node:* Let $O(\sigma,G,PSS)$ be an OR node. For each clause of the form 'H_i:–Q_i' such that H_i unifies with G, the OR node has a child $R(\theta H_i,\theta Q_i,PSS_i)$. Here θ is the most general substitution to the variables of H_i so that it matches G. (i.e., $\theta H_i = \pi G$ for some π). Q_i is a partial order of literals.

RULE 3: Let $O(\sigma,G,PSS)$ be an OR–node with children $\{R(H_i,Q_i,PSS_i)\}$. Then,

$PSS = \{ \theta | \theta G = \pi_i H_i$, where $\pi_i \in PSS_i$, for some $i\}$.
Informally: any solution to a REDUCE–node translated via back–unification is a solution to its parent OR–node.

RULE 4: Let $R(H,P,PSS)$ be a REDUCE–node, where the partial order consists of literals $G_1, G_2, .. G_n$. Let the children of this node be denoted $\{O(\sigma_i^j, G_i^j, PSS_i^j)\}$, where G_i^j is an instance of G_i. Then,
$PSS = \{\theta | \theta = \pi_1 \cdot \pi_2 \cdot .. \pi_n$ where $\pi_i = \sigma_i^j \cdot \delta_i^j$, for some j, with $\delta_i^j \in PSS_i^j$, θ consistent $\}$.
Informally: a consistent composition of solutions to each literal of a clause is also a solution to its head.
Notice the special case: $R(H,\{\},\{\Phi\})$. i.e. the PSS for a REDUCE node corresponding to a *fact* is a singleton set with a null substitution.

We now illustrate the rules using the tree of Figure 3.
Application of Rule 2: the topmost OR node in the tree is O(ϕ,S(I,u,v),PSS). To generate its child, we use the only clause whose head unifies with S(I,u,v). Solving $\theta S(t,y,z) = \pi S(I,u,v)$ for a most general substitution θ we find $\theta = \{t=I\}$ (with $\pi = \{u=y, v=z\}$). So, we get the child REDUCE node: R(θ S(t,y,z), θ {P(t,x), Q(x,y), R(x,z)}, PSS), which together with the DJG for the clause leads to the node in the Figure.
Application of Rule 1: Consider the REDUCE node created in the application above. We select G_k = R(x,z) from the DJG of this REDUCE node. There is only one predecessor literal, $P(I,x)$, and there is only one OR–child labeled with an instance of this. So we select δ_1^1 = {x=B} from the appropriate PSS. As the 'context', σ_1^1, for that OR node is empty, $\pi_1 = \delta_1^1$. Also, because there is only one predecessor, $\sigma = \pi_1$ = {x=B}. So, we create an OR node: O({x=B},R(B,z),PSS), which appears at the bottom right in Figure 3. A more illuminating example would arise if the DJG contained another literal, T(x,y,z), which has all the other literals as predecessors. To understand the full generality of Rule 1, the reader should follow the creation of an OR node for an instance of T in that case.
Application of Rule 4: The topmost binding, say θ_1, in the PSS of the REDUCE node R1 with variables x,y and z is obtained using the PSSs of the OR nodes labeled P(I,x), Q(B,y), and R(B,z) as: $\theta_1 = \pi_1 \cdot \pi_2 \cdot \pi_3 = \sigma_1^1 \cdot \delta_1^1 \cdot \sigma_2^2 \cdot \delta_2^2 \sigma_3^2 \cdot \delta_3^2 = \{\} \cdot \{x=B\} \cdot \{x=B\} \cdot \{y=D\} \cdot \{x=B\} \cdot \{z=F\} = \{x=B,\ y=D, z=F\}$. (The j values have been arbitrarily assigned). Notice that picking the substitution $\{x=A\}$ from PSS_1^1 does not lead to a consistent solution only because it conflicts with σ_2^2 and σ_3^2, thus justifying the need to maintain the 'context' in σ at all OR nodes.
Application of Rule 3: The topmost substitution in the PSS of the topmost OR node is found by solving for a θ such that $\theta S(I,u,v) = \pi_1 S(I,x,y)$ where $\pi_1 = \{x=B, y=D, z=F\}$.

It can be shown that a substitution θ appears in the PSS of a REDUCE–node labeled with a query Q, *iff* θQ is a consequence of the clauses used to obtain the tree. We omit the proof here for lack of space.

3 Comparisons

AND–OR trees have been used to represent problem solving since the early days of AI [Gelernter, 1959; Slagle, 1963]. Early problem solving systems did not explicitly provide for sharing of variables among subproblems, and indeed many interesting problems can be solved using decomposition into independent subproblems only, without any sharing across literals [Slagle, 1963]. The problems arising out of such sharing were soon noticed, as demonstrated by the well–known *Fallible Greek* example (see [Levi and Sirovich, 1976; Loveland and Stickel, 1976]). Definitionally, the problem was handled with the notions of candidate solution graphs, and consistent solution graphs (CSG) [Nilsson, 1980]. A candidate solution graph for an AND–OR tree includes the root; Whenever it includes an AND node, it includes all its children, and whenever it includes an OR node, it includes one of its children. A solution graph is consistent if the unifying composition of substitutions on all its match arcs is consistent. The operational aspect of the problem was how to search the AND–OR tree for a CSG.

A simple strategy would be to pass upwards the *set* of solution bindings found at nodes in the tree, merging them at the OR nodes and *joining* them at the AND nodes to ensure consistency. However, as we saw in section 2, this does not let the solutions to one literal constrain the search space for the others.

Nilsson [Nilsson, 1980] discusses the importance of early pruning, and suggests a strategy to detect inconsistent partial solution graphs. This can result in pruning only if one of the subgoal has produced a binding, and no alternate binding is possible within the tree for that subgoal. So, this approach won't lead to any pruning in the problem of Figure 1, and may lead to full exploration of the subtree for Q(x,y), for example.

Another approach is to force each AND node to maintain a single consistent binding at a time [Conery and Kibler, 1985]. The OR nodes send up only one solution, and wait until the parent AND node detects some inconsistency, and asks for another solution (*backtracks* into the OR node). In Figure 2, once P sends a solution, {x=A}, Q are R can be started in parallel, with the constraint {x=A} helping to prune their search space. This also permits a form of OR parallelism since alternate methods beneath P can be working on different solutions to P at the same time. But, as only one solution can be passed up, this misses a form of OR parallelism involved in exploring multiple instances of Q (or R) in parallel.

Levi and Sirovich [Levi and Sirovich, 1976] define a search tree in which nodes represent proof trees (partial solution graphs), and branches are choice points. These trees turn out to be identical to the SLD trees [Kowalski and Kuehner, 1971]. (Logic programming systems search these trees depth–first, with a more efficient variable representation than proposed in [Levi and Sirovich, 1976] and eliminate their *AND* operator used to avoid backtracking the bindings. See [Bruynooghe, 1982] for a connection between them, although the ideas remained unconnected in that paper). This effects the pruning we desire, and also permits OR parallelism. However, it misses the AND parallelism now. As we can see in the SLD tree of Figure 4, Q and R have to be solved sequentially in all branches. Also, there are redundant subtrees in SLD trees whenever they contain independent AND subproblems: The two subtrees for R are identical in our example. The duplication can be avoided even on parallel processors using a method in [Ullman, 1984].

Thus, the attempts to adapt the AND–OR trees for parallelism run into three problems. Some lose the ability to prune the search–space in the case of shared variables. Others miss either the AND parallelism, or a form of OR parallelism.

A model proposed independently in [Singh and Genesereth, 1986] leads to trees similar to ours. However, it combines the OR nodes for a literal, and imposes an ordering on streams of bindings flowing through these nodes. As a result, it loses some parallelism, and is incomplete when the trees may contain infinite branches. It is thus similar to [Li and Martin, 1986] and [Wise, 1982] analysed in [Kale, 1987a].

4 Discussion

The parallelism exploited by the REDUCE–OR trees is best illustrated in generate and test computations. *Generate* must happen before test ('pruning'), may be AND parallel, and may return multiple candidates to be tested. We are able to exploit all these sources of pruning and parallelism, including that between *tests* for different candidates.

The REDUCE–OR trees were informally introduced in [Kale and Warren, 1984], which discusses the interconnection topologies suitable for parallel Prolog. We are developing a process model for parallel evaluation of Logic programs [Kale, 1987b]. based on REDUCE OR trees. It finds every solution even when the tree contains infinite branches. The process model provides efficient algorithms and data structures to use the tree–representation effectively. For example, it avoids the redundancies involved in computing consistent compositions of substitutions from the PSS' of children OR nodes. Extensions into heuristic problem solving are also planned.

References

[Bruynooghe, 1982] M. Bruynooghe, "The memory management of PROLOG implementations", in *Logic Programming*, K. L. Clark and S. Tarnlund, (eds.), Academic Press, 1982, 83–98.

[Conery and Kibler, 1985] J. S. Conery and D. F. Kibler, "AND–parallelism and Non–Determinism in Logic Programs", *New Generation Computing*, **3**, (1985), 43–70.

[Gelernter, 1959] H. Gelernter, "Realization of a Geometry Theorem–Proving Machine", *Proc. First International Conf. on Information Processing*, Paris: UNESCO, 1959.

[Kale and Warren, 1984] L. V. Kale and D. S. Warren, "A Class of Architectures for Prolog Machine", *Proc. of the Conference on Logic Programming, Uppsala, Sweden*, July 1984, 171–182.

[Kale, 1985] L. V. Kale, "Parallel Architectures for Problem Solving", Doctoral Thesis, Dept. of Computer Science, SUNY, Stony Brook, Dec. 1985.

[Kale, 1987a] L. V. Kale, "'Completeness' and 'Full Parallelism' of Parallel Logic Programming Schemes.", *Proc. of 1987 Symposium on Logic Programming*, San Fransisco, 1987, 125–133.

[Kale, 1987b] L. V. Kale, "Parallel execution of Logic Programs: the REDUCE–OR process model", *Proc. of Fourth International Conference on Logic Programming*, May 1987, 616–632.

[Kowalski and Kuehner, 1971] R. Kowalski and D. Kuehner, "Linear Resolution with selection function", *Artificial Intelligence*, **2**, (1971), 227–260.

[Kowalski, 1979] R. Kowalski, in *Logic for Problem Solving*, Elsevier North–Holland, New York, 1979.

[Levi and Sirovich, 1976] G. Levi and F. Sirovich, "Generalized And/Or Graphs", Artificial Intelligence 7(3), 1976.

[Li and Martin, 1986] P. P. Li and A. J. Martin, "The Sync Model: A Parallel execution method for logic programming", *Proc. of the third symposium on logic programming*, Salt Lake City, Sept. 1986, 223–234.

[Loveland and Stickel, 1976] D. W. Loveland and M. E. Stickel, "A Hole in Goal trees: some guidance from resolution theory", *IEEE Trans. on Comp. C 25(4)*, 1976, 335–341.

[Loveland, 1978] D. W. Loveland, *Automated Theorem Proving*, North Holland, 1978.

[Nilsson, 1980] N. J. Nilsson, in *Principles of Artificial Intelligence*, Tioga Publishing Co., 1980.

[Singh and Genesereth, 1986] V. Singh and M. R. Genesereth, "PM: A parallel execution model for backward–chaining deductions", KSL–85–18, Department of Computer Science, Stanford University, May 1985, revised June 1986.

[Slagle, 1963] J. R. Slagle, "A heuristic Program that solves symbolic integration problems in Freshman calculus", *J. ACM*, **10**, (1963), 507–520.

[Ullman, 1984] J. D. Ullman, "Flux, Sorting and Supercomputer Organization for AI Applications", *Journal of Parallel and Distributed Computing*, **1**, (1984), 133–151.

[Wise, 1982] M. J. Wise, "A parallel Prolog: the construction of a data driven model", *Proceedings of the 1982 Conference on Lisp and Functional Programming*, 1982, 56–66.

```
Top Level Query: S(I,u,v)?
Clauses:    S(t,y,z) ← P(t,x), Q(x,y), R(x,z).
                 P(I,A).    Q(A,C).    R(B,F).
                 P(I,B).    Q(B,D).    R(B,G).
                            Q(B,E).
```

Figure 1: A Problem Specification

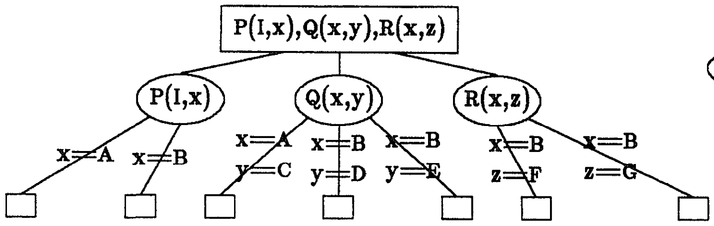

Figure 2: An AND-OR Tree

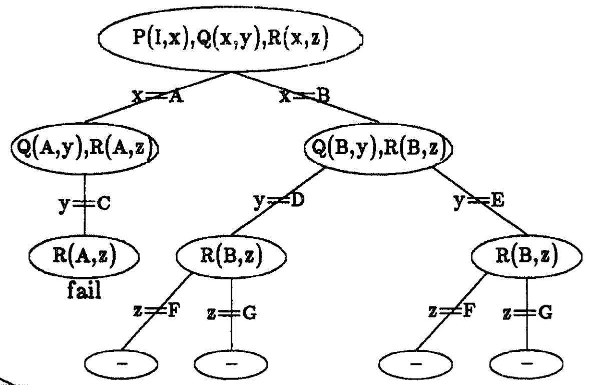

Figure 4: A SLD Tree

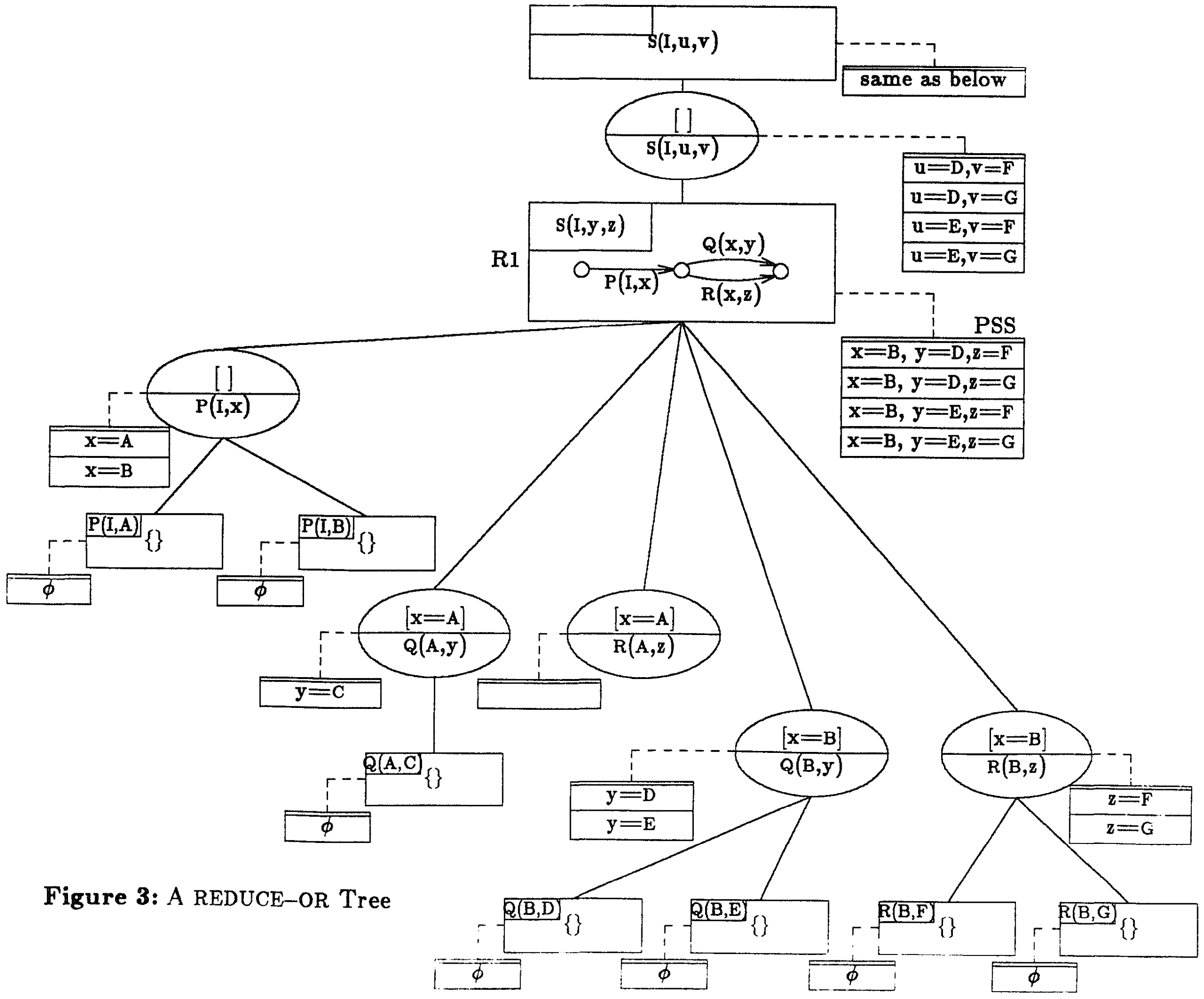

Figure 3: A REDUCE–OR Tree