MULTIPROCESSING OF LOGIC PROGRAMS

Guo-Jie Li and Benjamin W. Wah
Department of Electrical and Computer Engineering
and the Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 W. Springfield Avenue
Urbana, IL 61801

ABSTRACT

In this paper we will discuss issues related to the multiprocessing of logic programs. Due to space limitation, we will only discuss issues related to the unambiguous specification of meta-knowledge, the synchronization of parallel executions, and the scheduling of parallel tasks. Many of the issues discussed here are still open at this time.

1. INTRODUCTION

Programming in logic has been recognized as a suitable programming methodology to manage and manipulate knowledge in artificial intelligence (AI) applications. As complexity of these applications grows, the computational efficiency of logic programs is an increasingly important concern. There are two basic approaches to improve the computational efficiency: the use of domain knowledge as well as meta-knowledge and parallel processing. Starting with a given amount of knowledge, parallel processing can always be used to improve the computational efficiency. On the other hand, meta-knowledge, which are control information to guide the search, can be considered as a complement to searching and parallel processing. More meta-knowledge means less searching. An example is illustrated in the design of good heuristics based on previous experience of solving a given problem. With sufficient knowledge, the problem can be solved efficiently, even on a serial processor.

In general, the total computing power of a parallel computer system is used to characterize the rate at which a given problem is solved. However, due to nondeterminism in AI computations, a high computing power does not always imply a high throughput. Since most AI applications involve heuristic searches, resources may be devoted to fruitless searches, which increase the total computing power but does not decrease the time to find a solution. What is important is the time to solve a given problem. Unfortunately, early experience with multiprocessor architectures for Hearsay-II [5], Eurisko [8], OPS-5 type forward chaining rule-base expert systems [6], and others have led to a belief that parallel AI programs will not have a speedup of more than one order of magnitude. This belief was based on the adaptation of serial AI algorithms to parallel processing and the numerous difficulties involved in designing parallel AI systems, but not in showing that parallelism in AI problems was inherently poor. The use of multiprocessing systems may involve the design of entirely new parallel algorithms for AI processing.

Another misconception about parallel processing is that it can be used to extend the solvable problem size of AI problems. Due to the high complexity of these problems, parallel processing is useful *in improving the computational efficiency*, but *not in extending the solvable problem size* [18]. That is, parallel processing alone cannot circumvent the difficult of combinatorial explosion. The power of parallel processing cannot be overemphasized and must be combined with heuristics to solve complex problems. Currently, the method to combine heuristics and massively parallel processing is still an open problem. In the Japanese Fifth Generation Computing System (FGCS) project, parallel processing is applied to knowledge and language based systems, but no direct research on artificial intelligence is involved. The publication in 1985 of the Sixth Generation Computing System development proposal shows a serious intention in Japan to go beyond the current FGCS activities and tackle the AI aspects of computations on broad front [1].

AI computations are characterized by their nondeterministic and dynamic behavior. That is, to evaluate a logic program, it is difficult to plan in advance the procedures to execute and schedule with the available information. This nondeterminism arises because almost any intelligent activity is likely to be poorly understood, and efficient deterministic algorithms to solve many AI problems are unavailable. When a problem becomes well understood and can be solved by an efficient deterministic algorithm, we usually cease to consider it 'intelligent.' Combinatorial searches are, therefore, inevitable for many AI problems [18, 9]. Further, the required operations are normally data-dependent and are defined at execution time. This, coupled with the nondeterministic nature of executions, forbids effective static analysis of the program.

The essential issues in designing a multiprocessing system to support AI applications can be classified into the representation level, the control level, and the processor level (Figure 1). The starting point in conventional computations is an algorithm, which is a consequence of the accumulation of the complete knowledge and meta-knowledge to solve the problem, and a complete understanding of the computations. On the other hand, AI computations may start with incomplete knowledge about the problem. AI languages, such as Prolog, supports the easy accumulation and access of knowledge but are rather poor in

Representation level
- Hierarchy of meta-knowledge
- Domain-knowledge representation
- AI languages and programming

Control level
- Partitioning and restructuring
- Synchronization
- Scheduling

Processor level
- Microlevel architectures
- Macrolevel architectures
- System level architectures

Figure 1. Design issues of AI architectures.

Research supported by National Science Foundation Grant DMC 85-19649.

IEEE International Conference on Systems, Man, and Cybernetics, 1986.

supporting the manipulation of its semantic information. As a result, multiprocessing controls of these programs, which are developed for sequential processing, cannot be inferred completely from the programs themselves and must rely on either expensive compile-time analysis or specifications by programmers. For example, functional dependencies of subgoals in a clause of a Prolog program are meta-knowledge that cannot be specified directly by a programmer. To decide whether two subgoals can be processed in parallel, it is necessary to know whether they are functionally dependent or have any shared variables. Prolog designers resorts to compile-time analysis or user-specification of the modes of variables and run-time consistency checks of incompletely specified modes. The specification of modes is an unnatural way of specifying concurrency and dependencies and can be avoided if a better method is provided to specify the known meta-knowledge and semantics.

With an appropriate method to specify the meta-knowledge, it is necessary to partition a problem (or program) into a set of subproblems (or processes), to provide facilities for these processes to communicate, to synchronize the concurrent processes to guarantee a correct solution, and to schedule processes to processors to solve the given problem efficiently.

The lowest level in the design process consists of the development of microlevel architectures to support primitive operations, the design of macrolevel architectures that contain assemblies of primitive architectures, and the design of the complete system. The design process is iterative, namely, solutions for one level are developed based on assumed solutions of other levels and are refined when all the levels are evaluated together. In this paper, due to space limitation, we will only discuss issues related to the use of meta-knowledge to specify uncertainities in logic programs, and the synchronization and scheduling of parallel execution of logic programs. Some of these control issues are analogous to those in conventional multiprocessing systems, while others are related to knowledge representations and AI languages.

2. META-KNOWLEDGE IN AI APPLICATIONS

Knowledge has a hierarchical structure. Typically, domain-knowledge refers to objects, events, and actions per se. Meta-knowledge is the knowledge about domain knowledge. Likewise, meta-meta-knowledge is the knowledge about meta-knowledge, and higher-level meta-knowledge can be defined similarly. As an example, Figure 2 shows the domain knowledge and several levels of the meta-knowledge for Quicksort. The lowest level in this hierarchy is implemented as a computer program, while other levels contain knowledge known to users.

Meta-knowledge can be divided into two classes: deterministic and statistical. Issues related to these two classes are discussed here.

(a) *Deterministic Meta-Knowledge*

A programming methodology that allows the natural and unambiguous specification, management, and consistency checks of deterministic meta-knowledge in a multiprocessing environment is necessary.

Deterministic meta-knowledge are conditions and rules that are related to the correct execution of the algorithm. A minimal set of this knowledge must be maintained to result in correct execution. Currently, Prolog allows programmers to specify the sequence of executions in a serial computer, but the meta-knowledge to specify the correct execution in a multiprocessing environment is incomplete. In pure logic programs, knowledge is represented in a declarative representation, which stresses the static aspects of knowledge and leaves the control to the compiler/interpreter. A clause in a logic program represents a logic relation. For example, a :- a_1, a_2, a_3. means that 'a' is implied by a_1, a_2, and a_3, but nothing about their functional dependencies is represented. The sequence of executions in a sequential computer is correct because a definite search order is imposed. However, in a multiprocessing environment, the precedence relationships among subgoals must be known to the scheduler before subgoals can be scheduled.

Level 3 meta-knowledge:
- Common sense knowledge about using Levels 2 and 1 meta-knowledge.

Level 2 meta-knowledge:
- Meta-knowledge about using Level 1 meta-knowledge.
- Tradeoffs in sorting a list of elements

Level 1 meta-knowledge:
- Domain-specific meta-knowledge:
  - The operation SPLIT splits a list into two smaller sub-lists.
  - The operation APPEND combines two sorted sub-lists together.
  - Domain-specific meta-knowledge about other sorting algorithms.
- Control meta-knowledge:
  - The operation SPLIT must be completed before the sub-lists can be sorted.
  - The operation APPEND cannot start until both sub-lists have been sorted.
  - Other control knowledge.

Computer implemented program:
- Procedural form.
- Declarative form.

Figure 2. Hierarchy of meta-knowledge.

The major difference between Prolog and a conventional programming language, such as Pascal, is that the type and meaning of variables and functions in Prolog are dynamic and query dependent, and this knowledge cannot be completely specified as in conventional languages. Due to the dynamic nature of Prolog, the semantic meaning of subgoals and operations must be specified, which results in the precedence relationships to be used by the scheduler in a multiprocessing system. In logic programming, the method to represent the semantic information in a general and efficient way is open at this time.

One approach to utilize the declarative power of Prolog but not its inability to specify semantics is to define the problem such that there are no functionally dependent subgoals. This approach generally results in inefficient execution because functional dependencies and precedence relationships are resulted from better understanding of the problem, which can help to reduce the problem complexity. For instance, a list of n elements can be sorted by finding one of the n! permutations that contains the sorted elements. The n! tasks are independent and can be searched in parallel. In contrast, an algorithm, such as Quicksort, contains functionally dependent subgoals and can sort the list in n·log n time. In general, the deeper that we understand the problem to be solved, the larger is the set of necessary precedence restrictions, and the more efficient is the solution to the problem.

In summary, essential issues in deterministic meta-knowledge and semantic information in Prolog are (i) the unambiguous specification of meta-knowledge, (ii) the consistency verification in the knowledge base, and (iii) the method to discover and learn new meta-knowledge.

(b) *Statistical Meta-Knowledge*

Statistical meta-knowledge can be used to enhance the computational efficiency of evaluating a logic program. The probability of success of a subgoal and the associated search cost have been found to be useful in guiding the search [10]. In general, clauses in Prolog with the same head should be ordered such that those that are likely to succeed with a smaller expected search cost are searched first. In contrast, subgoals within a clause should be ordered such that those that are likely to fail with a smaller expected search cost are searched first. Probabilities have

been used in expert systems, such as MYCIN, to guide the search [12]. Search costs have been used in reordering subgoals in a clause to reduce the expected total search cost [19].

In summary, essential issues in statistical meta-knowledge are (i) the search of appropriate statistical metrics, and (ii) the dynamic learning and refinement of these metrics.

3. SYNCHRONIZATION

A multiprocessing system must provide a synchronization mechanism to ensure the proper order of execution. In a logic program, dependent subgoals communicate with each other through shared variables, and the reads and writes of these shared variables must be synchronized.

Due to the special programming style of logic programs and the dynamic nature of unifications, synchronizations in parallel logic programs is more difficult than that of conventional programs. Several parallel languages, such as PARLOG [3] and Concurrent Prolog [16], have been developed, but only simple synchronization problems have been addressed. Active research are still ongoing for some of the unresolved synchronization problems. In what follows, we will briefly discuss these issues.

(a) *Meta-Level Control of Logic Programs*

If all dependencies and precedence relationships are represented by special meta-rules, then the interpretor/compiler can generate the necessary synchronization primitives. The difficulty of this approach is to find appropriate meta-rules, that is, one needs to find the conditions under which dependencies will occur. These meta-rules must be sufficient such that all precedence relationships can be derived unambiguously, and must be precise such that the execution order can be determinated easily. An important consideration is the scope within which the meta-rules can be applied. Common-sense meta-rules should be included to operate on more specific meta-rules specified by the programmers.

Several researchers have addressed this problem. Gallaire and Lasserre have proposed the use of meta-knowledge expressed as a general or special control strategy in a Prolog-like interpreter [7]. This separates the control into two levels but is difficult to extend into higher levels of the meta-knowledge. In LP, a Prolog equation-solver learning system [17], the control information is expressed in a declarative representation, and inference is performed at the meta-level. Search at the object-level is replaced by search at the meta-level. Research in this area is necessary to provide a practical method to specify the needed synchronization through meta-knowledge.

(b) *Embedding Synchronization Information in Programs*

Synchronization among subgoals can be achieved by adding synchronization primitives, such as modes of variables and concurrent statements, in the logic program. These synchronization primitives should be powerful enough to specify all the dependencies and yet simple enough to reduce the control overhead. Current solutions are either inadequate to represent all the precedence relationships or too restrictive in the possible concurrency.

Precedence relationships can be divided into two classes. One is the functional dependencies or semantic dependencies, which specifies that an event cannot begin until another event has finished. Note that functionally dependent subgoals must share variables, but that subgoals which shared variables may not be functionally dependent. Another kind of dependencies is shared-variable dependencies. For subgoals with only shared-variable dependencies, any permutation of the execution order is allowed except that one of these subgoals must be evaluated first to avoid binding-conflicts. Each permutation of the execution order forms a precedence graph, and the choice of the precedence graph is a scheduling problem. Distinguishing between these two types of dependencies will reduce the overhead of parallelism detection and simplify the selection of the proper execution order.

The synchronization of parallel logic programs is more difficult than conventional programs because the modes of variables in logic programs may not be fixed and may depend on the queries at run time. As a result, requiring the users to specify the modes or performing complicated input-output analysis would result in over-specifying or under-specifying the degree of concurrency. The synchronization problem is further complicated by the dynamic data dependencies that can only be detected after unifications. For instance, variables X and Y are independent in subgoal k(X,Y) but will become dependent when X and Y are unified with Z at run time.

In PARLOG, a parallel logic programming language [3], every argument has a mode declaration that states whether the argument is input (?) or output (^). In Concurrent Prolog [15], a "read-only" annotation (?) is used. Both mode declarations in PARLOG and read-only annotations in Concurrent Prolog do not distinguish between functional dependencies and shared-variable dependencies. In both languages, a fixed execution order is imposed on subgoals with only shared-variable dependencies.

Another way to specify the concurrency is to use different symbols to distinguish between "parallel AND" and "sequential AND," such as "," and "&" in PARLOG. It should be noted that precedence relationships are a partial order, so the above distinction between "sequential AND" and "parallel AND," which is a linear order, is not sufficient to specify all the precedence relationships.

In PARLOG and Concurrent Prolog, guarded clauses are used as another synchronization mechanism. A guarded clause has a format: h :- g | b., where g is the *guard* of the clause and b is its *body*. Subgoals in the body can only be evaluated when all subgoals in the guard have succeeded and values bound have been committed to the body. Distinguishing the guard from the body cannot completely specify the precedence relationships because there may exist precedence relationships in both the guard and the body.

Recently, some researchers are trying to simplify logic languages by introducing directionality of information in logic programs, that is, fixing the modes of variables [14]. Obviously, this approach will degrade the expressive power of logic programs to that of first-order functional programs. Up to now, there does not exist a language that has good expressive power and yet flexible enough for parallel executions.

In summary, the essential issues in embedding synchronization information in logic programs are (i) the design of adequate primitives to represent dependencies, (ii) the development of methods to distinguish between functional dependencies and shared-variable dependencies, and (iii) the design of strategies to cope with run-time data dependencies.

(c) *Modularization of Logic Programs*

Currently, synchronization of logic programs is specified at the clause level. An issue related to the synchronization and, in general, correct execution, is that this granularity may be inappropriate. Due to the lack of a structured method of organizing knowledge, it is difficult to design a correct and testable logic program at this time. Modularization of logic programs will simplify the synchronization and allow programs to be designed more reliably. In the evolution of conventional high-level languages, we can see a trend in the increase of modularity and structure of programs. Research is urgently needed in this area to develop structured methods to organize knowledge and design more reliable logic programs.

(d) *Synchronization and Object-Oriented Style*

Object-oriented programming replaces conventional operator/operand concept with new ones—messages/objects. "Sending an object a message" actually refers to a conventional function call. This approach grants systems greater independence from the objects they contain. As a result, synchronization through shared variables is eliminated, and many problems existing today with shared variables in logic languages can be bypassed.

Some researchers have adapted logic programs to object-oriented programming [16]. In Concurrent Prolog, objects communicate with each other by instantiating shared variables, hence the access and update of common variables become the communication of messages among subgoals. This approach does not simplify the synchronization problem, but merely shifts the responsibility from one part of the system to another. Problems related to the detection of parallelism and specification of execution order remain to be solved.

Although a direct translation of logic programs into an object-oriented programming style does not simplify the synchronization problem, object-oriented programming remains to be an attractive approach because it decouples direct communications through shared variables. Research is needed to develop a programming methodology to combine the power of logic and the loose coupling of object-oriented languages.

4. SCHEDULING

Scheduling is an important consideration in the computational efficiency. Scheduling for logic programs is more difficult than conventional programs because one cannot accurately predict in advance the desirable number of concurrent activities and select the best subgoals to solve the problem. On the other hand, scheduling is essential to avoid the exponential complexity of exhaustive enumeration.

In general, the dynamic computational model of logic programs can be represented as a token-flow graph, which reflects process call-return relationships from an AND/OR-graph representation and the data dependencies. The general token-flow graph consists of four kinds of nodes, and-decomposition, or-decomposition, and-join, and or-join nodes. The tokens passed along the edges can be demand tokens, data tokens, or control tokens.

The aim of scheduling is (i) to determine the active nodes to be evaluated when free processors are available, (ii) to properly designate the modes of variables, and (iii) to determine the number of processors to be used to solve a given class of problems cost-effectively. Due to the nondeterministic and dynamic nature of logic programs, a number of issues must be resolved.

(a) *Selection Strategies*

Suppose in the course of evaluating a logic program that n active nodes and k, $k<n$, processors are available. The ideal scheduling algorithm should select k nodes from the n active nodes such that this decision will minimize the expected computational time. It is difficult to design an optimal selection algorithm because (i) the success probabilities and expected search costs of nodes are estimated heuristically, (ii) the success probabilities and average search costs are dynamically changing during the search, and (iii) there may exist problem-dependent precedence restrictions. As a result, a global best-first search or dynamic scheduling is impractical at the present time.

In general, active nodes distributed in multiple potential solution trees may be selected for evaluation. Active nodes can be AND-nodes, which represent alternative clauses with the same head in Prolog, and OR-nodes, which represent subgoals in a clause. *OR-parallelism* refers to the strategy in which multiple potential solution trees are evaluated in parallel, whereas *AND-parallelism* refers to searching a potential solution tree in parallel. Immediate descendents of an OR-node can generally be searched in parallel because different clauses are involved. On the other hand, immediate descendants of an AND-node represent subgoals of a clause that may be dependent and hence cannot be evaluated in parallel. It is difficult to determine in advance whether AND-parallelism or OR-parallelism is better because there is no unique criterion to order the AND-nodes and OR-nodes [10].

In the published literature, one can find different points of view with respect to OR-parallelism and AND-parallelism. Many researchers have suggested that OR-parallelism is easier to implement and will significantly improve the computational efficiency of logic programs [13, 2]. OR-parallelism is beneficial when all solutions to a given problem have to be found. However, when not all solutions are sought and most of the descendant of an OR-node will lead to solutions, not all of these descendant will have to be evaluated before the computation is stopped, and OR-parallelism is not necessary. In contrast, some advocates of AND-parallelism believe that AND-parallelism is related to deterministic computations and hence may offer more speedup than OR-parallelism [4]. This may run into the risk of dedicating all resources to a single failing solution tree since the number of potential solution trees is extremely large and not all of them will succeed.

The desirable choice of parallelism should be determined according to the success probabilities and the average search costs of subgoals and clauses. AND-parallelism should be used on solution trees that will most likely lead to solutions. In contrast, if a few solutions are sought and most of the solution trees will not lead to solutions, then OR-parallelism should be used to find potential solution trees that will fail as soon as possible, and hence these hopeless solution trees can be pruned early in the search process.

In summary, the essential issues in selecting nodes to expand in parallel include: (i) acquiring the meta-information on success probabilities and average costs of each node, (ii) finding a good strategy to uniquely order the AND-nodes and OR-nodes, and (iii) finding an efficient method to update the success probabilities and average costs of nodes according to run-time information.

(b) *Data Dependencies*

The parallel scheduling of logic programs is complicated by the dynamic functional and shared-variable dependencies. Moreover, the success probabilities and search costs are dependent on the modes of variables. When a subgoal or clause is evaluated before another subgoal or clause, the modes of its variables may be different than if it were scheduled later. As a result, the associated search cost and success probability will change.

If functional dependencies exist, then the scheduler must find out these dependencies in run time and observe these dependencies in generating the schedule. If there are no functional dependencies but only shared-variable dependencies, then the scheduler may need to compare the search costs of all possible permutations of scheduling. In both cases, the overhead of dynamic scheduling algorithms under dependencies is so high that render them impractical.

A reasonable approach is to statically order all sibling subgoals and clauses at compile time and schedule them according to a parallel depth-first strategy. In our previous research, we have proposed an efficient method to order sibling nodes in the AND/OR-tree representation of logic programs [11]. The best search order with the minimum expected total cost is chosen from all possible permutations of a set of sibling nodes without violating their dependencies.

(c) *Granularity of Parallelism*

The proper number of processors to evaluate a given class of logic programs in parallel must be determined at the design stage. Due to the underlying tree structure of logic programs, some processors have to be idle when nodes close to the root of an underlying AND/OR-tree are evaluated. The proper number of processors should be chosen to match the inherent parallelism in the class of problems to be solved.

In multiprocessing of logic programs, equal-sized grains or tasks should be distributed evenly to all processing units. However, it is seldom possible to do so because the problems to be solved are usually irregularly-structured, and the workloads may be data-dependent. An important related issue is, therefore, the study of dynamic distribution and load balancing of workload.

5. FINAL REMARKS

In this paper we have identified some essential issues in multiprocessing of logic-program evaluations. Synchronization and scheduling are major issues in the multiprocessing control level. To guarantee the correctness of execution, information on dependencies must be provided by either meta-rules or specification of modes of variables. Due to the very high complexity of optimal scheduling algorithms, dynamic scheduling of logic programs is impractical. The future lies in better understanding of meta-knowledge and in better programming methodologies to manage the knowledge in a multiprocessing environment and design reliable logic programs.

REFERENCES

[1] Science and Technology Agency, *Promotion of Research and Development on Electronics and Information Systems that may Complement or Substitute for Human Intelligence*, Tokyo, Japan, 1985.

[2] A. Ciepielewski and S. Haridi, "Execution of Bagof on the OR-Parallel Token Machine," *Proc. Int'l Conf. Fifth Generation Computer Systems*, pp. 551-560, ICOT and North-Holland, 1984.

[3] K. Clark and S. Gregory, "Note on System Programming in PARLOG," *Proc. Int'l Conf. Fifth Generation Computer System*, pp. 299-306, ICOT and North-Holland, 1984.

[4] A. M. Despain and Y. N. Patt, "Aquarius--A High Performance Computing System for Symbolic/Numeric Applications," *Proc. COMPCON Spring*, pp. 376-382, IEEE, Feb. 1985.

[5] R. D. Fennell and V. R. Lesser, "Parallelism in Artificial Intelligence Problem Solving: A Case Study of Hearsay-II," *IEEE Trans. on Computers*, vol. C-26, no. 2, pp. 98-111, Feb 1977.

[6] C. Forgy, A. Gupta, A. Newell, and R. Wedig, "Initial Assessment of Architectures for Production Systems," *Proc. National Conf. on Artificial Intelligence*, pp. 116-120, AAAI, Aug. 1984.

[7] H. Gallaire and C. Lasserre, "Metalevel Control For Logic Programs," in *Logic Programming*, ed. K. L. Clark and S. A. Tarnlund, pp. 173-185, Academic Press, New York, NY, 1982.

[8] D. B. Lenat, "Computer Software for Intelligent Systems," *Scientific American*, vol. 251, no. 3, pp. 204-213, Scientific American Inc., Sept. 1984.

[9] G.-J. Li, *Parallel Processing of Combinatorial Search Problems*, Ph.D. Dissertation, Purdue University, W. Lafayette, IN, Dec. 1985.

[10] G.-J. Li and B. W. Wah, "MANIP-2: A Multicomputer Architecture for Evaluating Logic Programs," *Proc. Int'l Conf. on Parallel Processing*, pp. 123-130, IEEE, June 1985.

[11] G.-J. Li and B. W. Wah, "How Good Are Parallel and Ordered Depth-First Searches?," *Proc. of Int'l Conf. on Parallel Processing*, 1986.

[12] Van Melle, E. H. Shortliffe, and B. G. Buchanan, "EMYCIN: A Domain-Independent System that Aids in Constricting Knowledge-Based Consultation Programs," *Machine Intelligence: Infotech State of the Art Report 9*, Infotech International, London, England, 1981.

[13] T. Moto-oka, H. Tanaka, H. Aida, K. Hirata, and T. Maruyama, "The Architecture of a Parallel Inference Engine (PIE)," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 479-488, ICOT and North-Holland, 1984.

[14] U. S. Reddy, "On the Relationship Between Logic and Functional Languages," in *Functional and Logic Languages*, ed. D. DeGroot and E. G. Lindstrom, Prentice Hall, Englewood Cliffs, NJ, 1985.

[15] E. Shapiro and A. Takeuchi, "Object Oriented Programming in Concurrent Prolog," *New Generation Computing*, vol. 1, no. 1, pp. 25-48, OHMSHA Ltd. and Springer-Verlag, 1983.

[16] E. Y. Shapiro, A Subset of Concurrent Prolog and its Interpreter, Technical Report TR-003, ICOT, Tokyo, Japan, 1984.

[17] B. Silver, *Meta-Level Inference: Representing and Learning Control Information in Artificial Intelligence*, North-Holland, Amsterdam, Netherlands, 1986.

[18] B. W. Wah, G.-J. Li, and C. F. Yu, "Multiprocessing of Combinatorial Search Problems," *Computer*, vol. 18, no. 6, pp. 93-108, IEEE, June 1985.

[19] David H. D. Warren, "Efficient Processing of Interactive Relational Database Queries Expressed in Logic," *Proc. 7th Int'l Conf. on Very Large Data Bases*, pp. 272-281, Cannes, FFrance, 1981.