A HYBRID PARALLEL EXECUTION MODEL FOR LOGIC-BASED REQUIREMENT SPECIFICATIONS*

Jeffrey J.P. Tsai and Bing Li
Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
Chicago, IL 60607-7053
tsai@eecs.uic.edu

ABSTRACT

It is well known that undiscovered errors in a requirements specification is extremely expensive to be fixed when discovered in the software maintenance phase. Errors in the requirement phase can be reduced through the validation and verification of the requirements specification. Many logic-based requirements specification languages have been developed to achieve these goals. However, the execution and reasoning of a logic-based requirements specification can be very slow. An effective way to improve their performance is to execute and reason the logic-based requirements specification in parallel. In this paper, we present a hybrid model to facilitate the parallel execution of a logic-based requirements specification language. A logic-based specification is first applied by a data dependency analysis technique which can find all the mode combinations that exist within a specification clause. This mode information is used to support a novel hybrid parallel execution model, which combines both top-down and bottom-up evaluation strategies. This new execution model can find the failure in the deepest node of the search tree at the early stage of the evaluation, thus this new execution model can reduce the total number of nodes searched in the tree, the total processes needed to be generated, and the total communication channels needed in the search process. A simulator has been implemented to analyze the execution behavior of the new model. Experiments show significant improvement based on several criteria.

Keywords :

Logic-based requirements specification language, data flow analysis, parallel execution model, AND-OR parallel execution.

This research was supported partially by National Science Foundation and DARPA under Grant No. CCR9633536, by USAF Rome Laboratory under Grant No. F30602-95-1-0035, by Fujitsu Network Communication.

INTRODUCTION

Undiscovered errors in a requirements specification will be very expensive to be fixed in the software maintenance phase. Errors in the requirement phase can be reduced through the validation and verification of the requirements specification. Through the execution of the logic-based requirements specification, customers and developers can validate their disagreement or misunderstanding in the requirement phase. By reasoning over the logic-based requirements specification, developers can verify that desired properties such as, consistency, liveness, etc. have been preserved. However, the execution and reasoning of a logic-based requirements specification can be very slow. An effective way to improve their performance is to execute and reason the logic-based requirements specification in parallel. To reach this goal, in this paper we present a hybrid parallel execution model for logic-based requirements specifications such as FRORL (Tsai, J.J.P., 1987).

A logic-based specification consists of a set of clauses. For each clause, the clause head and body are formed by predicates. The execution of a logic-based specification is based on the resolution principle. A data value is transferred among clauses and within the clauses' bodies based on the unification rule. The execution of a logic-based specification is similar to a depth first search of an AND-OR tree (Cornery,J.S., et al, 1983), where AND nodes represent the predicates within a clause, and OR nodes represent the clauses unifiable within a calling predicate. The parallelism of a logic-based requirements specification can be extracted from this AND-OR execution model. It follows the top-down searching strategy of the AND-OR tree. The parallel execution of OR-branches is relatively easy to implement because the execution of clauses unifiable with a calling predicate are independent of each other. The parallel execution of AND-branches is much more complex since it may create binding conflicts during evaluation (DeGroot, D., 1988). An effective and common method to solve this problem is to rely on data flow analysis, which allows exactly one producer and multiple consumers for each parameter in a clause.

Currently, most approaches adopt the method of dynamic mode inference to do data flow analysis (Chen, A.C. at al, 1991). The basic idea of this approach is to rely on the user's query to generate data dependency information, then use this information to spawn processes which can be executed in parallel. The nature of all these methods is a demand-driven model which executes in a top-down fashion. Even though these approaches can extract AND-parallelism during the execution, their common drawback is that the dynamic determination of parameters' mode and parallel execution combination puts tremendous overhead on the model. This may jeopardize any advantage these parallel execution models might obtain. A few approaches for data dependency analysis by static mode inference (Chang, J.H., et al, 1985) is based on the analysis of the structure of a logic program to generate the parallel executable components. But again they suffer the problems of either being unable to generate all the solutions to the input logic program, or they sacrifice the degree of parallelism of the program. For example, in (Chang, J.H., et al, 1985), Chang et al. implement static data dependency analysis by considering only the worst case clause invoking pattern, which eliminates the possible advantage of generating multiple solutions of a logic program at the same time. As a result, the approach has to explicitly handle the backtracking problem during the parallel execution of the AND-OR tree.

In this paper, a novel parallel execution model for a logic-based requirements specification based on a new static data dependency analysis is presented. This model can preserve maximum parallelism while guaranteeing to generate all the solutions of a logic-based specification without backtracking. The power of this model comes from the fact that the static data dependency is capable of representing all execution sequences and all multiple solutions implied by a logic-based specification (Tsai, J.J.P. et al, 1998). Since the complete mode information is available through mode inference even without actual evaluation, the new parallel execution model can apply some extra analysis processes during compiling comparing with other approaches and obtain more parallel execution information of the underlying logic-based specification. The overall parallel execution behavior of the logic-based specification can thus be improved. Specifically, the new execution model extracts the necessary mode information from the static data dependency analysis to facilitate the combination of top-down and bottom-up searching of the AND-OR tree of a logic-based specification. In this process, all solutions implied by the logic-based specification can be explored and generated based on the same data dependency information, which may result in the nice property of preserving the correctness of the results while eliminating backtracking from consideration. Regarding the top-down evaluation direction of this hybrid model, it adopts the method similar to the current approaches, i.e., a predicate is executable only if all of its input parameters have received values from some other predicates within the same clause. This evaluation direction mainly reflects the interrelation among the predicates within a clause. On the other hand, the new evaluation along the bottom-up direction relies on the detection of ground variables from the static data dependency information to facilitate the evaluation of those predicates with all of its required value being available. This direction mainly deals with the possible adjustment of evaluation sequence among clauses of a logic-based specification. By adopting this strategy, the overhead of the parallel execution of a logic-based specification can be reduced. In this paper, other related issues of parallelizing a logic-based specification are also discussed. Among them are the elimination of backtracking in the process of searching and the multiple binding synchronization problem which exists in OR-parallelism.

The paper is organized as follows. Section 2 summarizes the current approaches. Section 3 provides an overview of the proposed new system and shows how it handles major problems in the parallelizing process. Section 4 discusses the static data dependency analysis method and the mode information it generates. Section 5 explains in detail the AND-OR parallelism based on the static data dependency information. Section 6 considers the efficiency of the new approach and compares the experimental results of the new model with other parallel approaches. Section 7 gives a brief summary.

RELATED WORKS

When realizing the parallelism of a logic program, various approaches exist for dealing with the three main problems: 1) realization of OR- parallelism; 2) realization of AND-parallelism; and 3) reduction or elimination of backtracking.

OR-Parallelism

If an OR-branch has more than one clause which can unified with a predicate (a goal), then these clauses can be executed concurrently. The successful evaluation of any one of these clauses indicates the successful evaluation of the invoking predicate. This kind of parallelism is called OR-parallelism. Because the execution of unifiable clauses in this case is independent of each other, the implementation of OR-parallelism is relatively simple. The main problem in handling the OR-parallelism is how to represent the different branches. From our observation, the following two classes of approaches exist in this area.

Solution synchronization Almost all systems and approaches realizing OR-parallelism fall into this category. The major problem which needs to be handled is the synchronization of the multiple solutions generated from different clauses (Butler, R. et al 1986). Warren provided a concise summary of the current approaches available within this category (Warren, D.H.D, 1987). These methods developed various synchronization and ordering algorithms to identify and control the solutions generated from the different branches of the clause. The final result of the OR predicate is basically a Cartesian product of all possible results generated from unifiable OR-clauses. In order to insure the proper cross production, various data structures and identifications attached to a result have been presented which determine the bindings generated by different clauses. The major data structures used under this context can be categorized as

- Binding Array (Butler, R. et al 1986)
- Synchronization timestamp (Halperin, M. et al, 1985)
- Version Vectors (Hausman, A, Ciepielewski, A., Haridi, S., 1987)

The data structures have the following common characteristics:

- Binding data structures defined at each OR-branch point exists when the OR-parallel point is first entered and exists throughout the parallel execution process until all the branches of the OR-parallel point have finished execution.

- Binding data structures are dynamically modified along with the execution at each branch of the OR-parallel point. This modification may come from the deeper-level predicates in the OR-parallel branches when that branch finishes execution and generates the corresponding binding.

The first point forces a system to bear a large overhead by maintaining many complex synchronization data structures. To make things worse, some of the data structures may be necessary for the current OR-parallel execution if some processes are not interested in the current variables at the current OR-branch point (Warren, D.H.D, 1987). The second point also makes the control of the synchronization data structures complex. Specifically, the cost for the creating and accessing variable bindings becomes prohibitively high. In order to cope with this problem, some of the current approaches consider a AND-OR search tree as a hierarchical structure and use the inheritance mechanism to reduce the complexity of maintaining the synchronization data structures (Hailperin, M. et al, 1985).
Since all possible OR-parallel branches are explored simultaneously, if all the solutions are guaranteed to be found while the OR-branches are explored, there is virtually no need for the backtracking in this case. The merging of OR-parallel processes has to co-operate with the spawning of AND-parallel processes. For example, in approach (Li, P. Peyyum & Martin j. Alain, 1986), the results are gathered at OR-branches by taking a cross production, and only selected information is passed to the above AND-branches.

OR-parallelism combined with backtracking This model (Westphal, H et al, 1987) adopts the lay approach in spawning the process. Only when a processor idle and asks for a process will another branch at an OR-parallel point be generated and executed. If a failure occurs, the remaining untouched branches of the OR-parallel point are searched. So it is a combined approach of the traditional sequential and the advanced parallel execution model. The incompleteness in the parallelism which originates from the inadequate number of process at evaluation time require the approaches can be considered as a special case of the first class with a special focus on the parallelism and a limitation on the number of process which can exist at the same time. So the implementation has to consider those controls usually occur in the sequential realization, like task switching and backtracking. This is a mixture of the OR-parallelism and the general sequential evaluation of a logic program. When the number of process exists at any time is limited to one, the execution is equal to a sequential approach.

AND-Parallelism

With AND-parallelism, all predicates within a clause body must be true for a clause to be successful. The exploration of the predicates within a clause body can be carried out concurrently. Handling the binding conflict (DeGroot, D., 1988) is the major task in any implementation, and a number of approaches exist.

Programmer's annotation This is the simplest method to handle AND-parallelism. It requires the users to explore and define explicitly the parallel component of a logic program (Clark, K, et al, 1984). Even though the control is relatively simple, but human's involvement is definitely not our intention.

Independent AND-parallelism In this kind of parallelism, there is no variable dependency between AND-parallel goals. Predicates which share variables must have these variables bound to ground upon entering the clause (Hermenegildo, M.V., Jacobs, D., et al, 1986). This approach restricts the AND-parallelism to a relatively simple case similar to OR-parallelism. A comparison of different independent AND-parallel models can be found in (Gupta, G., 1991).

Data flow analysis for dependent AND-parallelism One of the most popular methods is to assign only one producer for each parameter and allow multiple consumers of the same parameter in the program (Chang, A.C., et al, 1985). Data flow analysis is used to determine the producers. With a single producer, the dependency of variables between predicates can be monitored. The spawning of process is based on this data flow information. The data flow analysis method can be done either dynamically or statically, with the dynamic analysis being the dominant analysis method at the current time. Dynamic analysis generates binding information from the user's input at each step of unification. This variable instantiation changes during execution and is adjusted dynamically. Static analysis (Chang, A.C., et al , 1985) generates binding information which is independent of the user's input, so that the overhead of the parallelization can be reduced. But currently available static analysis approaches impose various limitations or restrictions on the results, because they either limit the number of solutions generated or the degree of parallelism. Other approaches impose the similar sequential restrictions to a logic program to achieve dependent AND-parallelism (Shen, K., 1992).

Backtracking

Handling backtracking is not so critical as AND-parallelism and OR-parallelism, because it is possible to totally eliminate backtracking from the parallel execution model (Chen, A.C., Wu, C.L. et al, 1991) if

- AND-OR parallelism is supported so that all branches at all choice-points can be searched simultaneously, and
- All solutions implied by the logic program can be expressed and generated from the parallel execution model.

Even though, some of the interesting development in various approaches which handle backtracking, not necessarily under the context of parallel logic program model, it is still worthy to be listed here. Also, there still exist a lot of approaches in the current logic program community which explicitly handle backtracking because of the insufficient power to guarantee the above two criteria. Basically, there are two different kinds of backtracking mechanisms, shallow backtracking and deep backtracking. Shallow backtracking backtracks to alternative candidate clauses when the head of the current candidate clause cannot be proved a success by the unification with the calling literal. Deep backtracking backtracks to any previously invoked procedure which has a choice-point. This is also called intelligent backtracking.

Intelligent backtracking In conventional backtracking scheme, where a failure occurs during the search, it backtracks to the nearest choice-point and tries other branches. In intelligent backtracking, instead of backtracking to the nearest choice-point, backtracking goes to the choice-point where the binding of the variable which causes the failure can be bounded to another variable (Kumar, V. & Lin, Y.J, 1987). Research in intelligent backtracking which handles other related aspects, like cut, tail recursion, etc., also exists. In (Codognet, C., 1988), the pruning of the parts which could be explored several times by the conventional backtracking scheme is studied.
Improving backtracking by using data dependency information was first proposed in (Conery, J.S., 1983). Various researches considered intelligent backtracking handling in the AND-parallelism of the AND-OR parallel execution model of a logic program were presented in (Borgwardt, P., Codognet, C. et al 1986).

1.1.1 **Semi-Intelligent backtracking** Semi-intelligent backtracking uses static data dependency analysis for the AND-parallelism. Chang and Despain (Chang, J.H. & Despain, A.M., 1985) use this technique to find the optimal choice-point for intelligent backtracking. Since the mode information is not sufficient to support the multiple solutions exploration, backtracking is necessary. The binding status considered is only the worst case binding combination, which may involve in inaccurate binding results, so it is called semi-intelligent backtracking.

Elimination of backtracking The AND-OR parallel execution model can completely eliminate backtracking from the execution of a logic program (Chen, A.C., Wu, A.M. et al, 1991). Few results have been produced because most current approaches are unable to find all the solutions implied by the underlying logic program. This has forced them to adopt some degree of the backtracking.

OVERVIEW OF A NEW APPROACH

In this section, we outline a new approach and discuss how it handles the problems mentioned in Section 2. Figure (1) shows the major components of the model and their interrelations. Our approach consists of the following four techniques.

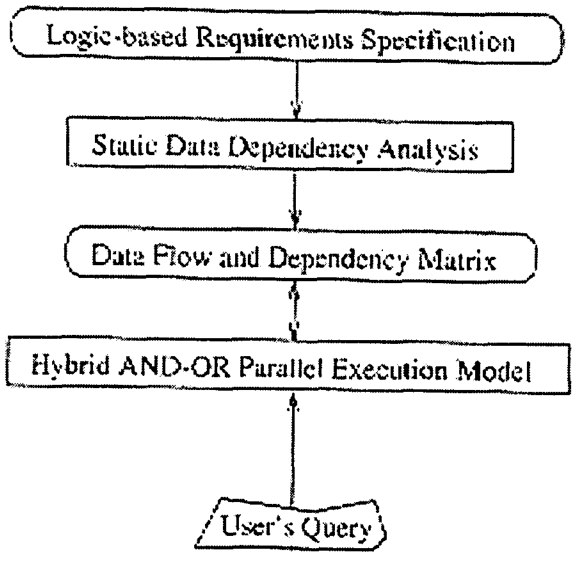
Logic-based Requirements Specification
Static Data Dependency Analysis
Data Flow and Dependency Matrix
Hybrid AND-OR Parallel Execution Model
User's Query

Figure 1: An Overview of the Parallel evaluation system

Static data flow analysis In this step the multiple execution sequences in a logic-based specification are found. These results are stored as two-dimensional data dependency matrixes and are consulted during execution to determine predicates that can be evaluated in parallel.

Hybrid AND-OR parallel execution model This AND-OR parallel model uses information gathered statically and the user's input to find the largest number of the predicates that can be executed in parallel. These predicates may come from different levels of the corresponding AND-OR tree. The parallelism of the predicates determined by the static data dependency information will be carried out in a bottom-up fashion. If no predicate can be executed in this way, the general top-down execution model will be used.

Simplified synchronization data structure The data structure used to support OR-parallelism can be realized with reduced complexity comparing with current approaches according to the hybrid execution model. Specifically, the duration of the synchronization data structures and the complexity of them can be reduced.

Backtracking elimination Since our approach can generate all the solutions from the static data dependency analysis, and all the branches at each choice point are searched simultaneously, we believe that our model can eliminate the backtracking by carefully implementing the AND- and OR-parallelism.

DATA FLOW ANALYSIS OF A LOGIC-BASED SPECIFICATION

A logic-based requirements specification is highly dynamic. A specification clause may be used to perform computations in several directions. The arguments of a clause may serve both as sources of input for the computation described by the clause, and as the location of output of that very computation. A predicate in a logic-based specification may have different meanings based on different environments. Most obviously, many predicates in a logic-based specification can be used in several directions, depending on the mode of the arguments. Considering the following example, *append* ([1,2], [3,4], *X*) appends two lists and returns the result in *X*, *append* ([1,2], *X*, [1,2,3,4]) obtains lists which could be appended to [1,2], in order to yield the list [1,2,3,4], and *append* ([1,2],[3,4],[1,2,3,4]) tests whether the two lists appended indeed yield the list [1,2,3,4]. As a consequence, terms in a logic-based specification may have multiple meanings. Most prominently, the list notation <<equation:datafl>> can either denote the construction of a list from an element *H* and a list *T*, or it can denote the destruction of an existing list in its head (*H*) and tail (*T*) components. Which of the two meanings is intended by the

specification depends on the sequence of clause evaluation. The order of the clause evaluation can be determined within the semantic domain of logic-based specification. The sequence of evaluation may change from input (query) to input (query). Such multi-directionality is the nature of a logic-based specification language and constitutes the main difficulty in parallelizing a logic-based specification. In order to preserve the completeness of a logic-based specification, we have to explicitly locate all possible data transformation paths from it in the parallelizing process. This phenomenon can be characterized as the mode information of parameters and data flow analysis in a logic-based specification (Chang, J.H. et al, 1985)
Traditionally, data flow and data dependency analysis have been investigated within the framework of logic programming. One approach to deal with the multi-directionality of a logic program has been to provide mode information for each clause (e.g., in Parlog, Concurrent Prolog). Bruynooghe (Bruynooghe, M., 1982) and Smolka (Smolka, G., 1984) have investigated the issue of verifying the consistency of mode declarations for arguments to predicates, as supplied by programmers. Automatic mode inference has been discussed by Mellish (Mellish, C.S., 1986) and Reddy (Reddy, U.S., 1984) recently by Bruynooghe (Bruynoogle, M. et al, 1987) and Mannila and Ukkonen (Mannila, H. & Ukkonen, E., 1987) most promising approach to mode inference so far has been presented by Debray in (Debray, S.K., & Warren, D.S., 1988). Debray uses an interpreter to reason upon the four-valued lattice of modes *don't know, empty, free*, and *ground*. Its purpose is to enable a compiler to optimize a logic program. Mannila and Ukkonen (Mannila, H. & Ukkonen, E., 1987) limit themselves to the simple modes *ground* and *unground*. Reddy relies on syntactic analysis to assign modes to predicates.

Mode Representation in the New Model

In this section, we present a new approach to mode inference which relies on a matrix representation of clauses to perform data flow and dependency analysis to support the new parallel evaluation model. In the new approach, we reason over the four-valued lattice of the following mode values:

- don't know ("?")
- input mode ("+")
- output mode ("--")
- empty (" ")

For each clause in the specification, we construct a two-dimensional matrix, with the literals in the clause along one dimension, and the variables of the clause along the other. This matrix is filled in with mode values by applying algorithms and constraining rules.

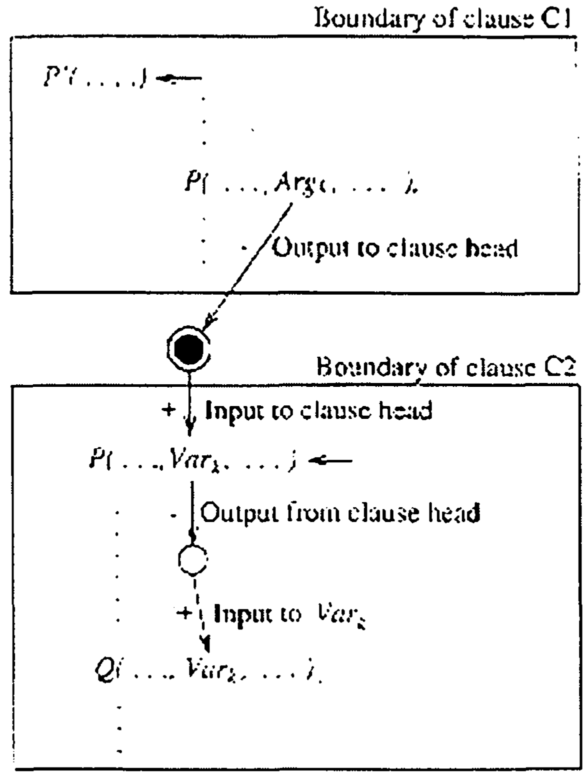

Figure 2: An illustration of data transformation by a clause head literal

Initially arguments of literals are marked as don't know ("?"), since no mode information about the parameters in a predicate is available. Arguments that do not occur in a corresponding literal are marked with " ". The next step is to collect and apply all possible known mode constraints of the literals within the rule clause such as those of user-defined predicates and the mode information for all built-in operators used in this rule clause.

The mode values of the arguments for all the literals in a clause body follow the normal definition of data flow: input is marked with "+", and output is marked with "--". In other words, an argument which generates data (exports a value for the argument) is said to have output mode and an argument which consumes data (needs a value to bind to) is said to have input mode (data is viewed to flow into the predicate across its "boundary").

But for a predicate in a clause head, the mode convention for its arguments should be the opposite of that in the arguments of predicates in the clause body. This is because a clause head serves dual roles: On the one hand it unifies with the calling (invoking) literal (and thus passes data between the calling literal and the clause). On the other hand it communicates data to and from the clause body through variables sharing the same name. We can interpret the communication of data to a clause in two steps: First we have a unification between the calling literal and the clause head of the invoked rule clause, then the data is passed between the head of the clause and the literals forming the clause body. The modes for the arguments of a clause head are normally viewed as based on the former role. However, when performing the analysis of a particular clause we treat the clause as a single unit, and therefore consider the modes for the arguments of the clause head from the latter point of view. Figure (2) gives an example of the two-staged process of data communication.

If a clause C_2 with the clause head P (..., *Vark*,...) is called within the body of a clause P' in C_1, then the literal P in C_1 may provide data to the clause C_2 via unification of the argument Var_k of P in C_2 against the parameter Arg_k of the call to P within the clause C_1. In that case, the mode of Arg_k of P in C_1 is output, whereas Var_k in C_2 has mode input, for the balance of flow to be preserved. However, in addition to receiving data from Arg_k in C_1, the variable Var_k of P in the clause C_2 also provides data to the predicates in the body of C_2 that share this variable. Since mode analysis is performed within the boundaries of a clause, the mode conventions for the clause head have to match that for the clause body. A "+" ("--") in the clause head therefore has opposite meaning in the clause body, and *vice versa*. The same holds true for data flow being communicated from the head of the clause C_2 to the calling literal P in C_1.

Data Flow and Dependency Analysis Algorithm

After we establish the matrix representation of mode information, we then apply the data dependency and flow analysis algorithm to the mode matrix of each clause. Applications of the algorithm are interleaved with applications of the constraining rules. Through this process, we will gradually build up the mode information for the clause under consideration. In order to handle multiple mode combinations more conveniently, we also introduce a new mode status "B", which stands for "both." It has a similar meaning as "?". The difference between "?" and "B" concerns the stage of analysis. "?" represents that we have no idea about the mode of an argument; "B" implies that we have already explored the argument and that both input and output modes are possible for this argument.

If we restrict our attention to an argument position in a literal, then we can refer to the literals as "producer" and "consumer." If a literal consumes the value for a variable generated by other literals, we call this literal as "consumer." If a literal produces (binds) the value for an ungrounded variable, we call this literal a "producer." The algorithms are based on two fundamental assumptions. These assumptions are also used to solve binding conflicts in AND-parallelism.

Unique-Producer Assumption Any argument, other than a constant, with the mode of "+" will demand exactly one producer for the value to be consumed by it.

Existence-of-Consumers Assumption A producer of a value (with the exception of dead-end values, which are generated but not used by any other literal in the clause) has at least one consumer.

From the initial matrix, the algorithm below calculates the mode matrix for a given clause. The matrix initially contains only " " and "?" information, with the exception of built-in predicates, and predicates for which information is already available from previous runs of the algorithm on other clauses. The algorithm is exhibited in two parts, the initialization of the matrix, and the dynamic analysis. In the following, let Lit be a sequence formed from the multi-set of the literals mentioned in a clause (note that a literal could occur more than once in the sequence), and let Arg be a sequence constructed from the set of arguments in the clause. Further, let *Arg* (*l*) be the subsequence of those elements in Arg, that occur in the literal with predicate symbol *l* (i.e., <<equation:dataf2>> " "} where $MM_{l,v}$ denotes the mode status of the variable *v* in the literal l in the matrix). In the algorithm, <<equation:dataf3>> ! represents " *there exists only one!*".

The data dependency and flow analysis algorithm deduces mode information by reasoning from the modes already known from the first part (1 --- 16) to generate the remaining unknown modes in the matrix (17 --- 29).

Algorithm 1 (Data Dependency and Flow Analysis)

```
1.   types MM: Lit x Arg $\rightarrow$ { "+", "-", "?", "B", " "}; Arg: set of argument(empty)
2.   for each literal l $\in$ Lit
3.      push l onto todo-stack
4.      Arg = Arg + Argument(l);
5.   if $\exists!$ v $\in$ Arg(l) . v is not a constant
6.      then $MM_{l,v} \leftarrow$ "-"
7.   for each argument v $\in$ Arg
8.      if v constant
9.      then for each l $\in$ Lit $\wedge$ $MM_{l,v}$ "?"
10.        $MM_{l,v} \leftarrow$ "+"
11.     if v is predicate
12.     then if all of v's arguments depend on constants
13.        then for each l $\in$ Lit $\wedge$ $MM_{l,v}$ "?"
14.           $MM_{l,v} \leftarrow$ "+"
15.       else for each l $\in$ Lit $\wedge$ $MM_{l,v}$ "?"
16.          $MM_{l,v} \leftarrow$ "B"

17.  While todo-stack is not empty
18.     l $\leftarrow$ pop todo – stack
19.     for each v $\in$ Arg(l)
20.        if $MM_{l,v}$ = "+" $\wedge$ v is not a constant
21.        then if $\forall$l' $\in$ Lit . $MM_{l',v}$ = "+"
22.           then fail
23.           if $\exists!$ l' $\in$ Lit . $MM_{l',v}$ = "?"
24.           then $MM_{l,v} \leftarrow$ "-"
25.        if $MM_{l',v}$ = "?"
26.        then if $\exists$l' $\in$ lit, l' $\neq$ l . $MM_{l',v}$ = "B" $\vee$ $MM_{l',v}$ = "?"
27.           then $MM_{l,v} \leftarrow$ "-"
28.           else . $MM_{l',v}$ = "B"
29.     call rule-based flow analysis
```

The algorithm examines the mode matrix in two directions. In the horizontal direction, the analysis is mainly based on the mode constraints existing within the arguments of a single literal, which may include built-in user defined predicates. In the vertical direction, the analysis is based on the above mentioned two assumptions.

Line 1 initializes the data dependency and control flow matrix "+" and "-" modes at this step are generated from user provided mode constraints and from *decomposition* in the rewriting process. This line also creates a set which holds all the arguments within the clause and initializes it to empty. Line 2 to 4 push each literal in the clause onto stack, todo-stack, which contains all remaining literals to be analyzed, at the same time these lines pick each argument from literal l and save it in *Arg*. Line 5 and 6 set a variable's mode to "-" if it occurs only once in its column. Since in this case it cannot have the "+" mode, it requires another appearance of the same variable to provide a value. Line 7 to 16 analyze the argument in *Arg* one by one to decide their mode if possible, based on the available modes. Among them, lines 8 to 10 set the modes of all occurrence of a constant within its column to "+", since a constant always provides values to others. Lines 11 to 16 handle the case of a predicate. If all the predicate's arguments depends on constant, then the mode of the predicate itself is set to "+" (Line 14), otherwise its mode should be "B" (Line 16).

Lines 17 to 29 apply mode inference operations and constraints to find new modes from the modes already in the matrix. It does this by checking each literal inside todo-stack. Note that l represents the current literal being checked (Line 18). Line 0 to 22 force failure if an argument other than a constant has all its appearances in different mode "+"; the intuitive idea behind it is that all the occurrences are consumers, and no producer exists. Line 23 to 24 change the only occurrence of the argument with unsearched mode to "-" (a producer) if all the remaining occurrences of the same argument are consumers. Line 25 to 28 deal directly with the undecided modes in the matrix. If there is no other cell within the same column which has undecided mode of "B" or unsearched mode of "?" other than the current cell, then the mode of the current cell should be "-" (Line 27), otherwise there is nothing we can say about its mode (Line 28). Line 29 calls the subroutines defined by Rule-1 through Rule-5. Those rules can also be written as condition statement inside the algorithm similar to the above cases. Because some of them are too long to be put in the algorithm, and some of them deserve some special attention for their importance, we write them in the format of a rule. Note again the effect of Line 29 is identical to a case check from Rule-1 to Rule-5.

Beside the operations described in the algorithm, the following rules constitute additional mode constraints and result in adjustment of modes within a single column or row of the mode matrix. Among them, some are used to eliminate mode status "B" from the matrix.

RULE-1 (Flow Analysis)

$MM_{l,v}$ = "-"

$\Rightarrow$ ($MM_{l,v}$ = "B" $\vee$ $MM_{l',v}$ = "?") $\wedge$ $l \neq l'$

Let $MM_{l',v}$ = "+"

RULE-1 (Flow Analysis)

$MM_{l,v}$ = "B"

$\Rightarrow$ $\forall l' \neq l$. $MM_{l,v}$ = "+"

Let $MM_{l',v}$ = "-"

These two rules reduce the number of arguments marked with mode "B". Further mode information to trim "B" could consist of mode declarations provided by the user, or mode information obtained by queries to the user. An example of the former is given in the following example.

Example 1 Consider the predicate

append ("+", "-", "-")

and

append ("-", "+", "-")

makes no sense when considering the procedural meanings implied by them. By eliminating them from the mode analysis, we can generate more accurate mode status. [1]

From here we can see that the user should describe explicitly the mode constraints among the arguments of a user-defined predicate in order to ease the analysis process. When the user builds a logic-based specification, he should also provide the mode constraints with each predicate he defines. Alternatively, when the operation of the transformed program does not meet the user's requirements, the user can force the selection among the multiple procedural interpretations generated due to several possible mode combinations.

Further rules constituting mode constraints are:

RULE-3 (Flow Analysis)

$MM_{l,v}$ = "-"

$\Rightarrow$ $MM_{l,v}$ = "-" $\wedge$ $l \neq l'$

fail

According to the unique-producer assumption, any mode information that includes more then one literal marked "-" for the same variable can be discarded. Thus this rule can be used to eliminate choices for modes marked "B", or terminate a particular search branch during flow analysis.

RULE-4 (Flow Analysis)

$MM_{l,v}$ =

$\Rightarrow$ l is a clause head $\wedge$

l' is a recursively literal in the clause body and matches with l $\wedge$ v and v' are the corresponding arguments

let $MM_{l,v}$ = opposite $MM_{l,v}$

This rule is built to handle the case that a recursive literal in the clause body calls the same clause head. Based on the different sign conventions between predicates in the clause head and body, we have to adjust the sign when this case occurs.

[1] This applies to append as a user-defined predicate, rather than append as obtained from the decomposition of a list construct since the latter has a determined mode assignment.

RULE-5 (Flow Analysis)

$MM_{l,v}$

$\Rightarrow$ l is a clause head $\wedge$

l' is a recursively literal in the clause body and matches with l $\wedge$

v and v' are the corresponding arguments $\wedge$

$MM_{l,v} \neq$ opposite $MM_{l,v}$

fail

If the corresponding modes are not opposite, as discussed earlier, then the mode assignments constructed so far are inconsistent, and we have to backtrack and attempt to find a different assignment.

In contrast to the data dependency and flow analysis technique presented here, Debray and Warren's approach lacks the power to handle multiple-mode combinations. As a result, the mode pattern of arguments of a predicate have to be generated by taking the least upper bound of all the possible success patterns generated by different calling patterns. Our approach explicitly considers a multiple-mode possibility of a logic-based specification and is flexible enough to determine all possible mode combinations. For Debray and Warren, a logic program may just mean a single mode transformation process, whereas in our approach it may result in different data dependencies and execution sequences. Therefore different target programs may be generated.

In addition, our approach performs significantly better than Debray and Warren's method. The general procedure of mode analysis in Debray and Warren's approach consists of a computation of the set of success patterns and calling patterns of a predicate, and a computation of the mode of a predicate by taking the least upper bound of all success patterns.

The first one contributes most to the complexity of the algorithm. In order to compute the success pattern, the control of the algorithm must go from left to right, according to the evaluation order assumed, through each predicate of a clause, and then calculate the possible mode changes of all parameters of the predicates in the clause body. This will contribute O (n^2) to the algorithm. But in order to calculate the mode modifications of each predicate in the clause body, the control must reclusively transfer to the analysis of success patterns of clause with the predicates as their head. So the overall complexity is O (n^2). The best case occurs when all the predicates in the clause body are defined by fact clauses. In this case the complexity will reduce to O (n^2). In the worst case (if there us a recursive call to the clause itself), the algorithm may not terminate.

Mode analysis in our approach proceeds sequentially through the matrix. If there exists a "B" in the matrix, then some modifications of the entries of matrix need to be applied. However, because of the close relationship between modes in the column direction, which is represented by the two assumptions, and the row direction, which is represented by the constraints of arguments within a predicate, a change of only a few of the "B"s to either "+" or "-"can completely specify all the modes in a matrix. So the average complexity of our algorithm is O (n^2). The best case occurs when al the parameters of the predicates in a clause body are constants or ground variables, in which case all the predicates become test conditions and the mode analysis can be accomplished in linear time. The worst cases arises when all the cells marked with mode 'B are totally independent such that the analysis of each predicate in the clause body must return to modify the matrix of the clause itself. Such backtracking may raise the complexity of the algorithm to O (n^3).

Example 2 In this example, we will use some clauses to illustrate the effect of the data dependency and flow analysis. The format of the clauses has been adjusted by rewriting rules to help facilitate the mode analysis. For details of these rewriting rules, please refer to (Tsai, J.J.P & Weigert, T., 1993). These clauses are:

1) $p(A_1) \leftarrow r(A_1,A_1)$

2) $r(A_1,A_2) \leftarrow s(A_1,N,W) \wedge t(W,A_2)$

3) $s(A_1,A_2,A_3) \leftarrow = (A_1,[]) \wedge = (A_2,0) \wedge (A_3,[])$

4) $s(A_1,A_2,A_3) \leftarrow car(A_1,X) \wedge cdr(A_1,L_1) \wedge$
"f" $(N,f(N)) \wedge = (A_2,f(N)) \wedge listcnstr(A_2,[A_2]) \wedge$
$append([A_2],L_2,A_3) \wedge s(L_1,N,L_2)$

5) $t(A_1,A_2) \leftarrow = (A_1,[]) \wedge t(A_2,[])$

6) $t(A_1,A_2) \leftarrow car(A_1,H) \wedge car(A_1,L_1) \wedge$
"g" $(H,g(H)) \wedge listcnstr(g(H),[g(H)]) \wedge$
$append([g(H)],L_2,A_2)$ $,L_2,A_2) \wedge t(L_1,N,L_2)$

For clause (2), we select one of two possible choices (see figure 3), in this case, the choice results in *r* has mode "+" in clause (1). From the mode information of *r*, we further partially infer the modes of the parameters of *s* and *t*. Because in column *W* there is no information about the mode status, it is marked with "B" in the two entries of the column. Mode status may change due to more specific mode information obtained at a later stage. Since the only entry in column *N* is not a constant, its mode has to be "-". (This entry is a "dead end", in that the value of this variable is not needed during the computation.) This information can be used to reduce the possible mode combinations for the clauses with the head *s*. In the following data dependency matrices shown, we mark entries that result form either the results of data flow analysis on other clause, or form Alg. 1 by circling them.
Figure 4 shows the data dependency matrix obtained for clause (4). The selected decomposition for the list constructs of the FOROL specification allows us to eliminate from consideration the situation where the argument A_3 of clause (4) has mode "+". This in turn determines completely the mode information for clause (2). The final matrix for clause (2) is given in Figure 5.
To analyze, clause (6) we obtain the mode of the clause head *t* from the matrix of clause (2). However, as Figure 6 shows, the resulting matrix is not consistent. Rule-4 of the second stage of the data flow analysis detects that the entries for the argument literal A_2 of clause

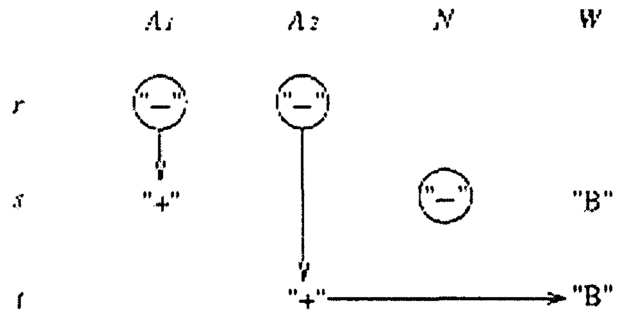

Figure 3: The partial result matrix for clause 2

(6) conflict in both having mode "-". However, the selection of this particular decomposition for the list construct in clause (6) does not jeopardize mode analysis. When we fail to find a consistent mode assignment, we will backtrack to the decomposition phase of rewriting and choose an alterative decomposition.[2] As a consequence, we also have to change this clause to

7) 7) $t(A_1,A_2) \leftarrow car(A_1,H) \wedge car(A_1,L_1) \wedge$
"g" $(H,g(H)) \wedge listcnstr(g(H),[g(H)]) \wedge$
$append([g(H)],L_2,A_2)\ ,L_2,A_2) \wedge t(L_1,N,L_2)$

[2] The decomposition of the list may also need to be changed in case the user requests other possible mode combinations, in which case we might have to explore all alternatives that can be generated that can be generated from using different ways of decomposing lists.

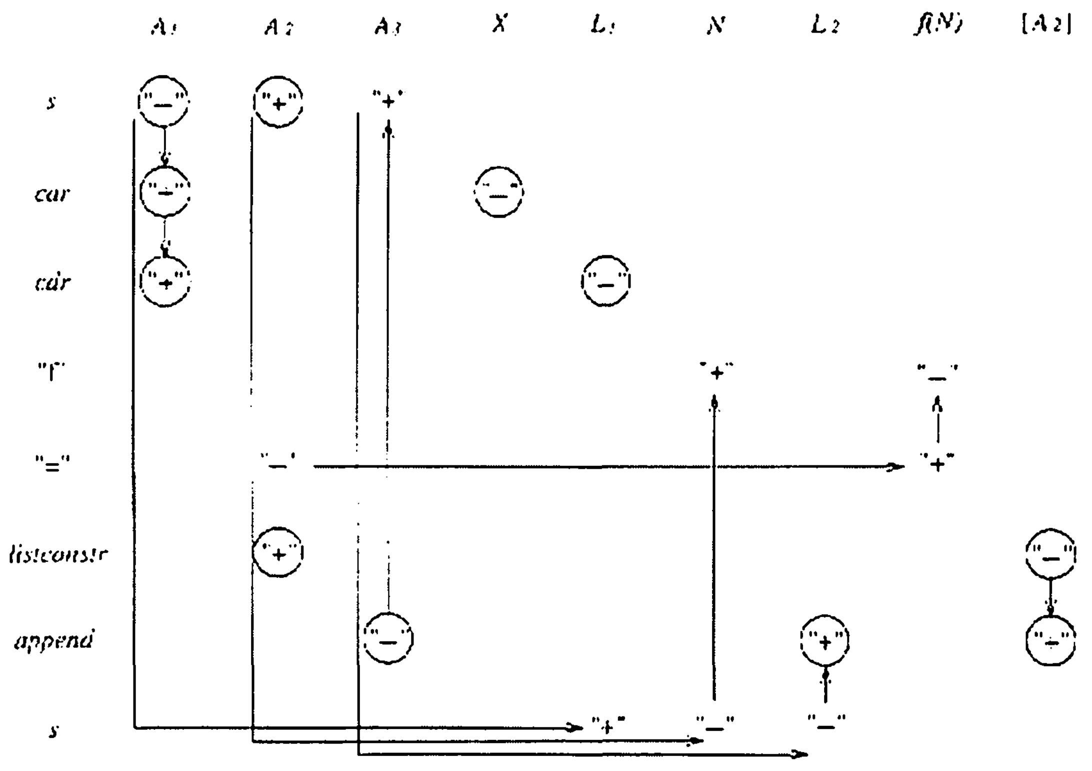

Figure 4: The final result matrix for clause 4

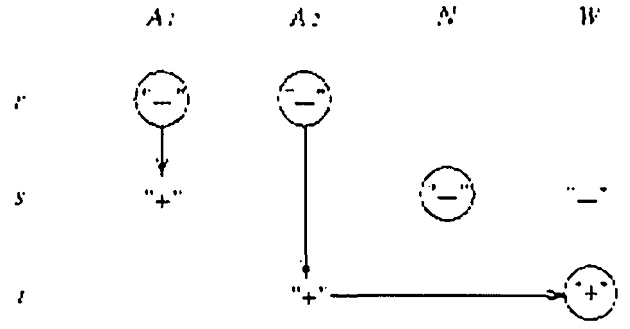

Figure 5: The final result matrix for clause 2

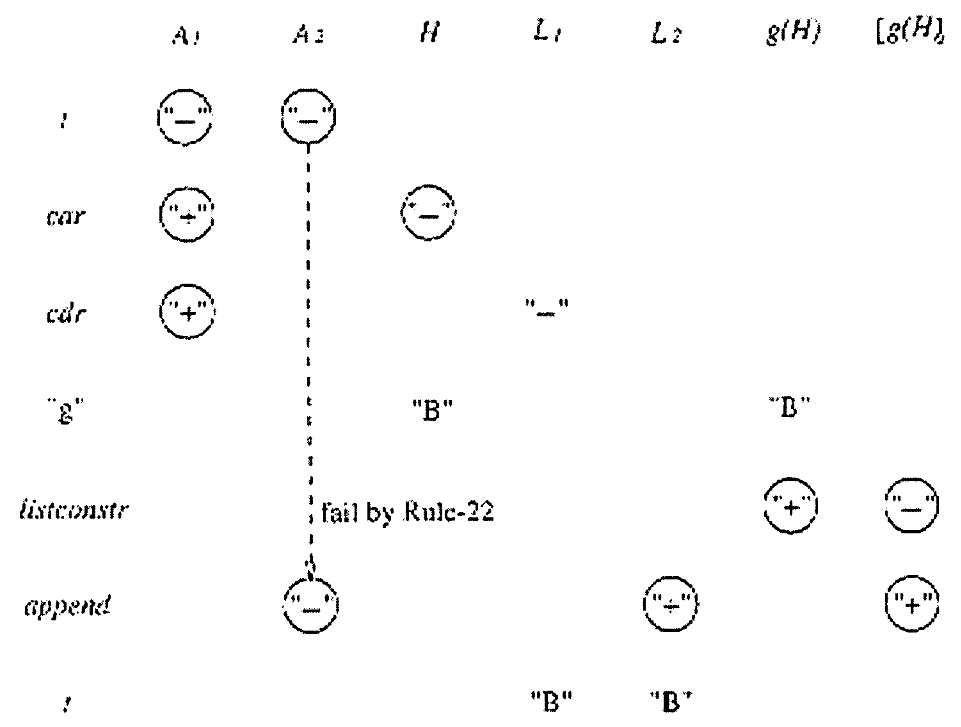

Figure 6: Initial attempt to construct a matrix for clause 6

Figure 7 shows the successful decomposition choice and the obtained mode assignments.
In Figure 8 and Figure 9, we show the final data dependency matrices for clauses (3) and (5), respectively. Note that for clause (3), the meaning of the = predicates vary based on different mode combinations of their arguments. The = predicate in the second row is that of a logical comparison since both of its arguments are grounded; the ones in the third and fourth rows are simply assignment expressions (e.g., A_2 will receive a value form the constant "O").

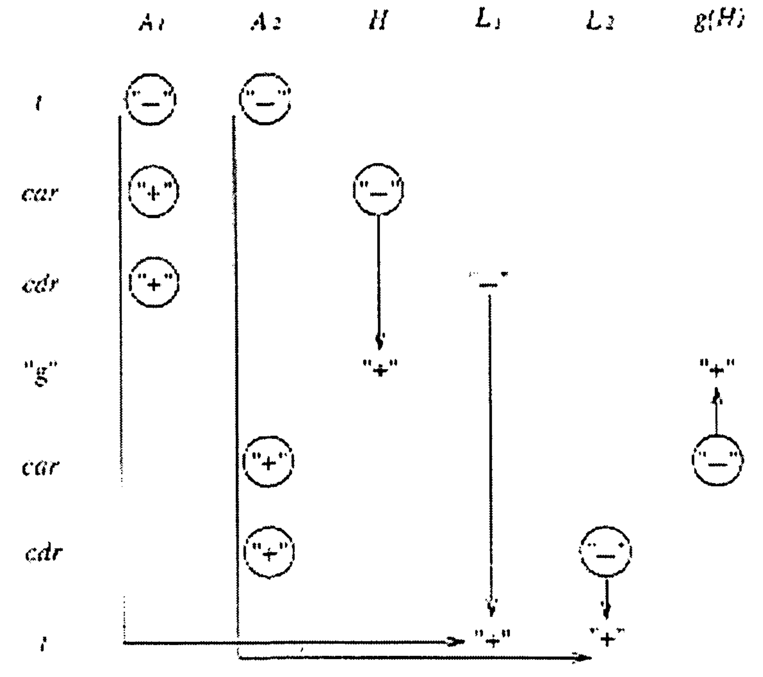

Figure 7: The final result matrix for clause 7

HYBRID AND-OR PARALLELISM IMPLEMENTATION

After the static mode analysis is finished, the system then uses the generated mode information to implement the parallelism of the original logic-based specifications.

The Usage of Mode Information in the Parallel Model

As mentioned above, most current approaches put tremendous overhead on the parallel execution system by doing dynamic analysis of the data flow, which is also called demand-driven. Few of them do static data flow analysis. But none of them solves the problem satisfactorily, mainly because their static approach cannot guarantee that it will generate all the solutions of a logic program implied. In our approach, by using the four-valued mode combination and generating all possible mode combinations using bi-directional mode ``B" in the matrix for each of the parameters of a predicate, all the solutions of a logic-based specification can be found from the matrix statically. The most significant advantage of this data dependency analysis is that the data flow information is available before any execution of the logic-based specification and any parallelization is attempted. Making use of this early available mode information is the key contribution of the new approach.

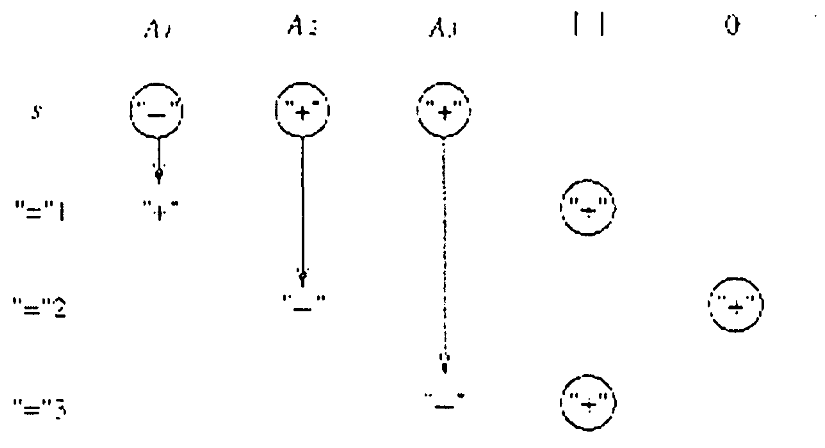

Figure 8: The final result matrix for clause 3

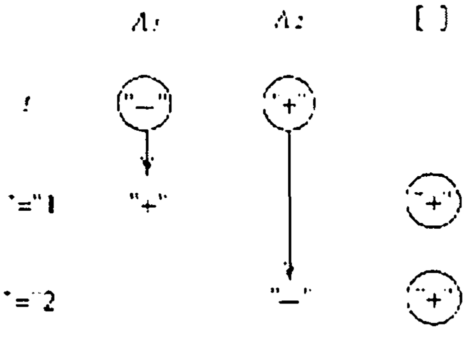

Figure 9: The final result matrix for clause 5

At each step of the dynamic evaluation of a logic-based specification, when the binding of a variable changes, the static data flow and dependency matrix is consulted to see if there exist some predicates in the logic-based specification clauses (not necessarily clauses from the same level) which can be evaluated in parallel, and the results generated from this simultaneous evaluation will enable further evaluation of other predicates in those clauses. These predicates can be executed in parallel and can come from different clauses which may reside in different levels in the searching tree. If there are no predicates which can be evaluated in the bottom-up fashion, then the general top-down evaluation process is invoked. The current approaches which employ mode information to solve binding conflicts generate and modify mode information at each step of the logic program. As a result, after each step of evaluation, mode analysis process is re-employed and mode status of arguments is changed. This changed mode information is used to facilitate further parallel evaluation of a logic program. So the parallel evaluation goes in a level by level fashion from the respective of a search tree, after each step, mode analysis has to be invoked to find a more specific mode for arguments of a predicate in the lower level. Instead in our model, since the complete mode information is available before any actual evaluation of the logic-based specification, the parallel evaluation can go more than one level in the search tree. So a predicate in a deeper-level of the search tree can be invoked if all its arguments which require value have been bounded by other predicates within the same clause, and this decision is made by checking the static mode information. In this case, the evaluation goes in a bottom-up fashion. That is, the parallel evaluation will explore predicates in a deeper-level in the search tree, collect the value there, and pass to other predicates within the same clause. The decision of when and where this bottom-up evaluation should be applied is solely decided by the static mode information. So the static mode information here serves to find predicates which can be executed in parallel from various levels of the search tree.

AND-OR Parallel Execution

In order to handle AND-OR parallelism, a bottom-up combined with a top-down evaluation scheme is used. Current approaches always work from the top of an AND-OR tree and search all the way down to the tree. In contrast, our approach can find all the mode combinations before the user's query is available, it is possible that while the execution continues, some information about the predicates which will be executed several steps later in the old top-down model is available in advance. If we carry out those operations first, then more information of mode for unsearched predicates can be gathered in advance and the efficiency of the whole process can be improved. The approach uses the results generated from the static data flow analysis, combined with the user's input, to find out the portion of the AND-OR tree of which the execution behavior is deterministic under the current variable binding. This portion of the AND-OR tree is executed in a bottom-up fashion. Compare this to other approaches, which rely completely on the user's input query to initiate the analysis, their execution is completely based on the dynamic determination of the data flow information. So the analysis is completely dynamic and executes in a top-down manner, which puts a tremendous overhead on the system.

The overhead for the new model is the cost of the static mode analysis. In the previous section the cost for static mode analysis has been discussed and shown that when a mode can be obtained, the cost for the new approach is better than current approaches in the best and average cases, and is the same in the worst case. So among the approaches applying mode to resolve binding conflicts, our approach is substantially better than the current approaches in obtaining the mode information. The new hybrid evaluation model also uses the mode matrices during execution to obtain the actual mode status for arguments of the predicates. This portion of the cost is the same among all the approaches which adopt mode information. The cost to finish a mode check for a predicate is O (n * m), where n is the average predicates within a clause, m is the number of arguments within a clause. If we assume there is a central process which contains mode matrices for all the specification clauses, then an extra overhead is two communication channels to sent the current binding to and receive the currently executable predicates from the centered process. In the other word, one of these two channels sends the mode status for input query, the other returns generated parallelly executable predicates. These predicates will then be assigned processes and begin evaluation. The new approach shows significant improvement over the current approaches when the corresponding search tree has a large height value.

The algorithm to check for predicates which can be executed in parallel begins by generating grounded variables. Dynamic checking of parallelly executable predicates focuses only on the detection of newly generated grounded variables. In our new model, instead of only dynamic evaluation, all possible variable bindings and execution information have been included in the statically generated data dependency matrix. A very important property of our data dependency matrix is that it has all the possible execution sequences based on the multiple data dependency information. The only role a user's query plays is to initiate the analysis. At each step of the execution, especially whenever a new binding of a variable is generated, this information is used to restrict the multiple data dependency choices existing in the data dependency matrix and the possible execution sequences they imply. This restriction may result in only one possibility for the execution of a clause. So a large part of the analysis of our approach is done statically instead of dynamically. The execution starts from the root of an AND-OR tree, which corresponds to the user's query, then at each step of the unification, a variable binding is generated which will restrict the meaning and execution behavior of the clause. The system locates the branch of the AND-OR tree, which may not be the predicates from a single clause based on the multiple binding, so that the execution behavior of this branch is restricted to a single choice, and then executes this branch in a bottom-up fashion. All the remaining branches which cannot be restricted to a single choice will be executed top-down. This is how multiple solutions of a query are derived. Even in the top-down case, the static data dependency matrix can still help. Since all the solution have been predicted and represented in the data dependency matrix, the place where this multiple choice occurs can be determined from the same matrix. So the generation of multiple choices is still under the control of the system.

Synchronization in OR-Parallel Execution Model

The problem of preserving the multiple choices generated from the OR-branches can be handled with the help of the same data dependency matrix. The major problem of OR-parallelism is to synchronize the multiple bindings of the same variable generated from different branches. A simple synchronization mechanism must be built according to the mode information included in data dependency matrix, considering that all the binding information is supported by these matrices. Based on the analysis of the AND-OR parallelism, there are two execution models, top-down and bottom-up. The realization of OR-parallelism is also closely related to these two models. During execution, if we find that the parallel execution of the logic program is in the top-down fashion similar to the current approaches, then the OR-parallel synchronization structure is identical to the current approaches using any of the currently available data structures and methods. If the execution proceeds in a bottom-up fashion, then the binding data structure will also be generated in a bottom-up fashion, which will simplify the data structures proposed for synchronization to only one level in the search tree. Compare with other approaches, which have to keep a binding data structure for all the processes existing at the same time, this is a major improvement in our approach. Also the creation of all these binding data structures can be delayed until all the variable bindings have been generated from all branches of an OR-parallel point. The current approaches have to generate a synchronization data structure upon the first entry of a OR-parallel branch point, and continue changing the synchronization data structure to reflect changes in the binding status of variables when the execution in any of its OR-parallel branches changes. Our approach reduces the life cycle of the synchronization data structures by waiting until they are actually needed to generate the complete binding combinations from OR-branches. Also the complexity of the OR-parallel binding data structure is sharply reduced because we use a one level control structure. In contrast, current approaches have to use a synchronization hierarchy when dealing with the OR-parallel synchronization problem.
The simplification of the synchronization data structure comes from the control structure of the AND-OR parallelism. The choice of a specific synchronization data structure does not affect the new model's performance.

Calculation of the EXEC Set

In order to find predicates which can be evaluated in parallel, a search of the data dependency matrices is necessary to find all predicates which can be evaluated concurrently. The predicates are calculated dynamically according to the binding status of the clause and the static data flow and dependency matrix. All these currently valuable predicates are gathered into a data structure which has enough space to hold the syntax structure of any of the clauses in the logic-based specification. This data structure is called **EXEC**, which represents all the predicates at the current stage whose parameters' requirements for value in order for them to be valuable have been satisfied by the other predicates within the respective clauses. This set will include the parallelly executable predicates at any time in the process. The formal definition of **EXEC** is followed.

DEFINITION

EXEC is a set which contains the predicates from the logic-based specification clauses which can be executed in parallel. A predicates in **EXEC** if all of the input variables of the predicate have been bounded by other predicates within the same clause.
The predicates within this set can come at any time during the process from clauses of different levels in the search tree, based on the binding status of the parameters within the predicate. The binding generated from the execution of this set will enable both the calculation of the new **EXEC** set and the general top-down evaluation of the logic-based specification, according to the binding status of the other predicates in the clause. If the predicates form more than one level are found to be executable, then they will be included in the new **EXEC** set, otherwise, the general top-down evaluation is invoked. So the proper calculation and application of this set become the center point of the whole approach.
In the following, an example is used to illustrate how this mechanism works. The example is based on the data flow and dependency matrices generated in Example 2, where generating the mode information is discussed.

Example 3. Consider the case where the specification is called with a mode combination of r (bound, unbound), i.e., r is called with the first parameter bounded to a list and the second parameter unbounded. Current approaches will first check the predicates in the body of the clause (1), which corresponds to the AND-parallelism at the first level. In this case, there is no possible parallel execution, so a process

corresponding to predicate s will be created and the OR-parallelism of the predicate s is explored, i.e. checking continues in a top-down fashion. In our approach, after the user's query is available, the input mode combination of r (-, +) is checked with the data dependency matrices to find the possible execution set **EXEC**. From the matrices for clause (1), we can see that the list provided from the clause head predicate r is consumed by the first parameter of the predicate s. Since the predicate is a user defined predicate, the search for EXEC will go into the OR-branches. Based on the algorithm described below, all the predicates in the clause (2) can be executed simultaneously since for all of these predicates, all their parameters with the input mode have been satisfied. Thus after examining the data dependency matrix for clause (2), **EXEC** = {=1, =2, =3}. Meanwhile the search for executable predicates is also done at clause (3). In this case, the two predicates car, and cdr are found to be executable. Finally **EXEC** becomes {=1, =2, =3,car, cdr}. In this example, instead of only the predicate s in the clause (1) being executed, the five predicates in **EXEC** can be executed in parallel.
After the execution of the predicate s in clause (1) is completed, the value bounded in X_1 by the execution of predicate s is passed to the predicate t in the same clause, and the same process continues for t. The **EXEC** set corresponding to the first step of the execution of predicate t will be {=1, =2,car, cdr}, which comes from clause (4) and clause (5) respectively. Please notice that within a clause, the behavior of our approach is similar to Chen and Wu's model [Chen, A.C. & Wu, C.L., 1991]. A predicate within a clause has to have all its parameters with input mode be matched with a producer in order for the predicate to be executable. So after the parallel execution of the above **EXEC** set, the next **EXEC** set of s will be executed. The same process continues until the parameter B_2 and B_3 in clause (3) receive a value from the execution of the clause.
As this example shows, the execution of the new model relies on the statically built data dependency matrix to spawn as many parallelly executable processes as early as possible during the dynamic evaluation of the logic-based specification.
In the following, an algorithm is introduced, which calculates the data structure **EXEC** set at each step whenever a new binding is introduced in the execution of the logic-based specification. The algorithm uses the **EXEC** to hold the predicates which are found to be parallelly executable by the algorithm. In the algorithm, **EXEC** is an external variable, similar to that in language C. The reason is that in any invocation of the algorithm, predicates which may be included in **EXEC** may come from various clauses in the logic-based specification, and the algorithm has to recursively call itself for each of the clauses involved in the calculation. In order to preserve the predicates in the **EXEC** set in the process of a multiple clause call, and form the complete image after the calling is finished, the data structure must be external. The same thing happens with the data structure **CLAUSE**, which holds the clauses which contributes to the current **EXEC** set. Other predicates in the clause in this set will be searched for further evaluation when the current parallel evaluation of the **EXEC** is over.

Expanded is a two-dimensional data structure. It is built up for each of the clauses in the logic-based specification, which forms the first dimension. The second dimension is similar to the **EXEC**, which holds the predicates in each clause which has been expanded or searched. It is similar to the dead set in the conventional DFS or BFS [Luger, G.F. & Stubblefield, W.A., 1989] algorithms. The only difference is that we have to consider the case when the clause is recursively called, the predicate may be moved from the *expanded* set, which means that the predicate has to be reevaluated under the new binding status in the new round of evaluation. Expanded_1 is a duplicate of *expanded* except that it is defined locally. It is used to hold the partial results during the execution. In the algorithm, we assume there are N clauses in the logic-based specification, and the specification is called with the predicate Pred_i, with the parameter of a, in the clause body of clause_j. Further assume that clause Clause_j has p predicates within its body, marked from 1 to P.

Algorithm 2 Calculate_Current_EXEC (Pred$_i$ (para$_1$,......para$_a$),Clause$_j$)

```
External: EXEC;
    External: Clause;
    Static: expanded[1..N]
    expanded$_1$[1..N];

    for each l, ($1 \le l \le p \wedge l \ne i$) \$Pred$_l \in$ expanded[j] $\wedge$ Arity(Pred$_l$)==b
if for each v $\in$ Arg(Pred$_l$)
            if MM$_{\text{Predl,v}}$ = "+" $\wedge$ v is not a constant
        then $\exists$Pred$_k \in$ Clause$_j$, s.t.
                (Pred$_k \in$ expanded[j] $\vee$ k == i) $\wedge$ (MM$_{\text{Predk,v}}$== "—");
    then
            if Pred$_l$ is not a user defined predicate
```

```
                **EXEC = EXEC +** $Pred_l$;
            $expanded_1[j] = expanded_1[j] + Pred_l$;
                **CLAUSE = CLAUSE + Clause**$_j$;
        else if $Pred_l$ is a user defined predicate
            for each $Clause_m$,s.t. $Head(Clause_m) == Pred_l$
                **Calculate_Current_EXEC**($Pred_l(para_1,\ldots\ldots para_b)$,$Clause_m$);
    if none of these predicates exist, then report "evaluation finished";
    $expanded[j] = expanded_1[j]$
```

The algorithm is called with a clause and its clause-head predicate as parameters. First the algorithm checks each of the predicates within the same clause which are not expanded predicates to see if any of them has all of its consumer parameters satisfied by some producer within the expanded set of the same clause. If no such predicate exists, then the evaluation of the clause has been finished. Otherwise if a predicate found is not user-defined one, then the predicate and the corresponding clause are added to the data structure. If the predicate is user-defined one, then the corresponding clause is called with the appropriate parameters. With user-defined predicates, the parallelly executable predicates found in the subsequent evaluation will be added to the **EXEC** set, and the corresponding clauses will be added to the **CLAUSE** set. But the expanded set will be filled by the procedures called later.

Hybrid Execution Algorithm

In this subsection, another algorithm is introduced, which explores the hybrid AND-OR parallelism of a logic-based specification. This algorithm is initially called when the user's query is available. The user's query predicate and the clauses unifiable with the predicate consist of the initial calls to the previous algorithm to calculate the first **EXEC** set. After the set is constructed, all predicates in this set are evaluated in parallel by creating a process for each of them. Then the same exploration for the **EXEC** continues for the clauses which have contributed to the previous set and the parallel evaluation process proceeds for another **EXEC** set generated. In this recursive process, if the look-ahead process goes only one level deep, then we know that the new scheme does not contribute to the current evaluation of the clause, and the conventional top-down evaluation process is adopted. If the evaluation of a clause terminated, then the results generated from the evaluation of the clause is gathered to form a complete answer set and passed to the upper level in the search tree. This part is discussed further in the next section when the data structure for the OR-parallel synchronization is discussed. When there is no clause in the current **CLAUSE** set, which contributes to the calculation of the previous **EXEC** set, then the process terminated.
Suppose the query is a predicate *Pred* with a parameter of m. Both sets **CLAUSE** and **CLAUSE**$_1$ hold the clauses which contribute to the calculation of the previous **EXEC** set. Since **CLAUSE** will change in the process of the calculation for the next **EXEC** set, a duplicate **CLAUSE** data structure is created to keep the set consistent in the calculation process of the next **EXEC** set.

```
**Algorithm 3 Parallel_Evaluation**(($Pred(para_1,\ldots\ldots,para_m=\})$)
**EXEC** = 0;

      for each clause $Clause_i$, s.t. $Head(Clause_i) == Pred$
     **Calculate_Current_EXEC**($Pred(para_1,\ldots\ldots,para_m)$,$Clause_i$);

    for each of the predicates in EXEC, create a parallel process to evaluate it;
      do
       **EXEC** = 0;
       **CLAUSE**$_1$ **= CLAUSE;**

      for each clause $Clause_j$ in $CLAUSE_1$,s.t. the head of $Clause_j$ is
      $Pred_j(par_1,\ldots\ldots,para_j)$

         **Calculate_Current_EXEC**($Pred_j(para_1,\ldots\ldots,para_j)$, $Clause_j$);
if the evaluation of the current clause goes only one level deep
         the general top-down parallel evaluation scheme is adopted for the clause;

     if the evaluation of $Clause_j$ has been finished
        form the cross product for the result of the clause from the results
                generated from each predicates in the clause;
        Add the result generated at this step to the storage area of the OR-parallel processor one
        level higher;

    for each of the predicates in EXEC, create a parallel process to evaluate it;
    until all of the clauses in CLAUSE_l have finished evaluation.
```

EFFICIENCY CONSIDERATIONS AND EXPERIMENTAL RESULTS

In this section, the simulator is used to demonstrate the executional behavior of the conventional and the new parallel searching methods. For each set of simulations, the percentage of improvement of the new model over the conventional model is given, i.e., (1 -- New_Val / Old_Val)%.
The parameters for describing the search tree are adjusted in their full range to explore all possible combinations of the searching. The three constraints used are:

1. Error_Rate of the leaf nodes in the search tree. The higher the value, the more the leaf nodes will have fail truth value.

2. Deep_Jump_Distance of the minimum and maximum deep jump distances. This serves as the basis for the new evaluation model. The higher the range of the value, the deeper the jump can be.

3. Deep_Jumping_Rate of deep jumping. The higher the value, the more likely deep jumping will occur within the searching tree.

In the table, *mh* means the minimum height of the search tree, *xh* means the maximal possible height of the tree, *mw* means the minimum width of the AND-parallel branches, which is also the minimum number of predicates within a clause, and *xw* is the maximum width of AND-parallel branches.

Simulation for Various Values of Error_Rate

The following table shows the effect of the error rate on the execution behavior.

Results for the testing of various values of Error_Rate							
	mh=3 xh=5 mw=6 xw=10	mh=6 xh=9 mw=5 xw=7	mh=8 xh=13 mw=3 xw=5	mh=15 xh=20 mw=1 xw=3	mh=20 xh=30 mw=1 xw=2	mh=30 xh=50 mw=1 xw=2	Avg.
Error_Rate = 0.00							
Total Evalua. Steps	5.88%	2.07%	2.50%	15.79%	29.17%	30.30%	**9.76%**
Total Nodes Searched	---	---	---	---	---	---	**---**
Total CPU Rounds	28.57%	6.04%	13.67%	53.51%	64.99%	71.28%	**30.79%**
Total Channel Costs	-27.27%	-2.96%	-0.28%	12.90%	30.43%	31.25%	**3.28%**
Ttl Wtd Channel Cst	-15.29%	-0.63%	0.05%	13.61%	30.43%	31.25%	**14.68%**
Error_Rate = 0.50							
Total Evalua. Steps	14.29%	20.00%	45.45%	57.89%	51.85%	57.14%	**48.63%**
Total Nodes Searched	---	21.43%	41.67%	75.76%	54.17%	60.61%	**59.48%**
Total CPU Rounds	26.67%	44.90%	70.00%	89.03%	79.38%	83.48%	**81.59%**
Total Channel Costs	14.29%	---	45.00%	67.90%	55.32%	60.00%	**53.28%**
Ttl Wtd Channel Cst	22.81%	12.90%	51.32%	65.77%	58.45%	61.29%	**59.62%**
Error_Rate = 0.99							
Total Evalua. Steps	14.29%	20.00%	45.45%	57.89%	51.85%	62.86%	**48.63%**
Total Nodes Searched	---	21.43%	41.67%	75.76%	54.17%	60.61%	**59.84%**
Total CPU Rounds	26.67%	44.90%	70.00%	89.03%	79.38%	83.48%	**81.59%**
Total Channel Costs	14.29%	4.17%	45.00%	67.90%	55.32%	60.00%	**53.69%**
Ttl Wtd Channel Cst	22.81%	27.02%	51.32%	65.89%	58.45%	61.29%	**59.99%**

From the table, we can see that with increasing error rate, all of the average improvements of the new execution model have values substantially better than the previous case in the corresponding column. For each row, where the depth of the evaluation changes, it can be seen that the improvement for deeper evaluation is generally better than for the shallow evaluation.
This is true because the new approach can go into deeper levels more quickly than the general approach. So while the error rate increases, those leaf nodes with error as their truth value can be discovered earlier than the general approaches can discover them, and short cuts can also be discovered much earlier. The new approach focuses on deep jumps, which is directly related to the depth of a tree. For the shallow search tree, the search does not go deep into the tree, and the advantage of the new approach cannot overcome the overhead of the new approach. So for deep search processes, the advantage for the new approach is obvious.

Simulation for Various Values of Deep_Jumping_Distance

In the following, the effect of the various depths and widths of search trees is analyzed using the deep jump distance.

Results for the testing of various values of Deep_Jumping_Distance							
	mh=3 xh=5 mw=6 xw=10	mh=6 xh=9 mw=5 xw=7	mh=8 xh=13 mw3 xw=5	mh=15 xh=20 mw=1 xw=3	mh=20 xh=30 mw=1 xw=2	mh=30 xh=50 mw=1 xw=2	Avg.
Min_Deep_Jump_Distance = 1, Max_Deep_Jump_Distance = 2							
Total Evalua. Steps	14.29%	10.00%	18.18%	31.58%	33.33%	31.43%	**27.52%**
Total Nodes Searched	---	42.86%	41.67%	46.79%	33.33%	33.33%	**39.87%**
Total CPU Rounds	26.67%	32.65%	40.00%	62.92%	57.85%	54.71%	**56.10%**
Total Channel Costs	14.29%	16.67%	35.00%	44.44%	36.17%	33.85%	**35.66%**
Ttl Wtd Channel Cst	22.81%	16.13%	30.79%	40.08%	37.19%	33.65%	**34.94%**
Min_Deep_Jump_Distance = 2, Max_Deep_Jump_Distance = 4							
Total Evalua. Steps	14.29%	20.00%	18.18%	47.37%	40.74%	51.43%	**39.45%**
Total Nodes Searched	---	21.43%	41.67%	65.15%	45.83%	54.55%	**52.29%**
Total CPU Rounds	26.67%	44.90%	40.00%	80.94%	67.69%	78.33%	**73.86%**
Total Channel Costs	14.29%	---	10.00%	56.79%	44.68%	53.85%	**43.03%**
Ttl Wtd Channel Cst	22.81%	12.90%	15.79%	56.17%	47.29%	55.16%	**50.84%**
Min_Deep_Jump_Distance = 4, Max_Deep_Jump_Distance = 7							
Total Evalua. Steps	14.29%	20.00%	54.55%	57.89%	55.56%	54.29%	**49.54%**
Total Nodes Searched	---	21.43%	50.00%	75.76%	58.33%	57.58%	**60.13%**
Total CPU Rounds	26.67%	44.90%	78.00%	89.92%	82.77%	80.99%	**81.59%**
Total Channel Costs	14.29%	---	55.00%	69.14%	59.57%	58.46%	**54.92%**
Ttl Wtd Channel Cst	22.81%	12.90%	61.84%	67.88%	62.68%	59.71%	**60.49%**

Obviously the behavior of the new execution model improves substantially when the deep jumping distance increases. This best demonstrates the improvement of this new model over traditional evaluation models. The execution also improves with increased depths. This point was analyzed when the effect of Error_Rate was discussed.

Simulation for Various Values of Deep_Jumping_Rate

The next table examines the effect of the rate of deep jumps during the search process.

Results for the testing of various values of Deep_Jumping_Rate							
	mh=3 xh=5 mw=6 xw=10	mh=6 xh=9 mw=5 xw=7	mh=8 xh=13 mw=3 xw=5	mh=15 xh=20 mw=1 xw=3	mh=20 xh=30 mw=1 xw=2	Mh=30 Xh=50 Mw=1 Xw=2	Avg.
Deep_Jumping_Rate = 0.1							
Total Evalua. Steps	---	---	18.18%	15.79%	11.11%	5.71%	**9.18%**
Total Nodes Searched	---	---	16.67%	28.79%	12.50%	6.06%	**16.99%**
Total CPU Rounds	---	---	34.00%	41.51%	22.15%	11.55%	**22.60%**
Total Channel Costs	---	---	15.00%	22.22%	12.77%	6.15%	**12.71%**
Ttl Wtd Channel Cst	7.02%	---	16.58%	17.62%	13.59%	7.03%	**10.49%**

As excepted, increase in the deep jumping rate improves the performance of the new execution model more and more substantially. Since the advantage of this new approach comes solely from the deep jumping in the execution and hybrid execution models, the results is quiet noticeable.
In other simulations, we found that with increases in the variable dependency rate, the improvement of the model decreases. The improvement, in the worst case, still achieves an average rate of about 20%. The result also show that as the values of non-determinism rate increases, the new parallel evaluation model demonstrates better evaluation behaviour.

CONCLUSION

To improve the performance of the validation and verification process of logic-based requirements specifications, we presented a hybrid parallel execution model. It handles AND-parallelism, OR-parallelism, and eliminates of backtracking related to a logic-based specification. The significance of this model is

- It is the first model to discuss the parallel execution of a logic-based requirements specification. Current approaches all focus on the parallelism of various logic programs.

- The new model uses a static data dependency approach to generate mode information for each of the variables in the predicate of a clause. This static mode information is the basis for the whole parallel evaluation process. The new model is guaranteed to find the maximum degree of AND- and OR-parallelism by adopting a new hybrid execution model. This model finds a parallel executable set of predicates at each step when a new variable binding is created based on the static mode information.

- Because the static data dependency analysis is guaranteed to find all possible execution paths and all solutions of a logic-based specification, this model can eliminate the backtracking structure from the execution control structure.

- At each step of the execution, the correct processes are divided into several clusters by the clauses. These clauses have predicates which are currently executable or have been active because of the static analysis. As a result, the total idle time of communication channels needed for the parallel processes is substantially lower than the idle time of the current approaches, where it is possible for a communication channel to be idle for a long time before a message is collected from deep predicate of one branch and been sent to the deep predicate of another branch.

- During the AND-parallel execution, if intermediate nodes do not contribute to the binding of the related variables, we do not need to generate these intermediate processes, which reduces the overhead of the parallelizing scheme. During the OR-parallel execution, the creation of the binding structure can also be delayed until the variable bindings actually are generated form the branches of the OR-parallel points.

- Since at each step the search goes deep into the AND-OR tree of a logic-based specification and the evaluation is applied to the clauses which have predicates currently executable, it is possible to apply the classical short-cut algorithm here earlier than other approaches can and thereby reduce the space of the AND-OR tree searched to generate the complete multiple solutions. As a consequence, the time and space saved using the short-cut algorithm will be maximized.

- The hybrid execution of OR-parallelism reduces the complexity of the synchronization data structures used for management of different bindings generated from different branches. This is reflected in the possibility of delaying the creation of binding data structures and the one-layered binding data structure synchronization. The bottom-up execution will first go deep into the AND-OR tree hierarchy, a data structure at an OR-parallel merging point will not be created until all the variable bindings are actually generated from the evaluation of the OR-branches below the OR-parallel point. According to the bottom-up execution model, all the bindings generated from different branches of an OR-parallel predicate are already available before the merging algorithm at the same OR-branch point is called. So there is no need to maintain synchronization data structure for more than one level below the OR-parallel branch point. This will sharply reduce the complexity of creation and maintenance of all the synchronization data structures used by most of the current approaches without any limitation on the level of information a binding list needs to maintain. The only overhead we have here is a tag which distinguishes solutions generated one level deep from one OR-parallel sub-branch from others.

REFERENCES:

P. Borgwardt, "Distributed Semi-intelligent Backtracking for a Stack-based AND-parallel Prolog," **IEEE 1986 Symposium on Logic Programming**, pp.211-222, Salt Lake City, Utah, Sept. 1986.

M. Bruynooghe, "Adding Redundancy to Obtain More Reliable andMore Readable Prolog Programs," **Proceeding of the first International Logic Programming Conference**, Marseille, France, 1982.

M. Bruynooghe, B. Demoen, A. Callebaut and G. Janssens, "Abstract Interpretation: Towards the Global Optimization of Prolog Programs," **Proceedings of the 4th IEEE Symposium on Logic Programming**, San Francisco, California, September 1987.

R. Butler, E.L. Lusk, R. Olson and R.A. Overbeek, "ANLWAM: A parallel Implementation of the Warren Abstract Machine,"Internal Report, Argonne National Laboratory, U.S.A., 1986.

R. Butler et. al., "ANLWAM: A Parallel Implementation of the Warren Abstract Machine," {\em Internal Report, Argonne National Laboratory, U.S.A., 1986.

J.-H. Chang and A.M. Despain, "Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis," **IEEE 1985 Symposium on Logic Programming**, pp.10-21, Boston, Massachusetts, July 1985.

J.-H. Chang, A.M. Despain and D. DeGroot, "AND-parallelism of Logic Programs Based on A Static Data Dependency Analysis," Digest of Papers, **COMPCON** Spring '85, pp.218-225, Feb. 1985.

C. Chen and C.-L. Wu, "A Parallel Execution Model of Logic Programs," **IEEE Trans. on Parallel Distributed Sys.**, pp. 79-92, VOL. 2, No. 1, January 1991.

K. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," Research Report DOC , Department of Computing, Imperial College of Science and Technology, 1984.

Codognet and P. Codognet, "Non-deterministic Stream AND-Parallelism Based on Intelligent Backtrackings," **Logic Programming, Proceedings of the Sixth International Conference**}, pp.63-80, Lisborn, Portugal, June 1989.

Codognet, "Yet Another Intelligent Backtracking Method," Logic Programming , **Proceedings of the Fifth International Conference and Symposium**}, pp.~447-465, Seattle, Washington, August 1988.

J. S. Conery, "The AND/OR Model for Parallel Interpretation of Logic Program," PhD Thesis, Dept. of Information and Computer Science, Univ. of California, Irvine, 1983.

T. Conlon, "**Programming in PARLOG,**" publisher: Addison-Wesley Publishing Company, 1989.

S. K. Debray, "Static Analysis of Parallel Logic Programs," Logic Programming, **Proceedings of the Fifth International Conference and Symposium**, pp. 711-732, Seattle, Washington, August 1988.

S. K. Debray, "Flow Analysis of Dynamic Logic Programs," **The Journal of Logic Programming**, Vol. 7: pp. 149-176, 1989.

S. K. Debray and D. S. Warren, "Automatic Mode Inference for Logic Programs," **The Journal of Logic Programming,** Vol. 5: pp. 207-229, 1988.

DeGroot, "Alternate Graph Expressions for Restricted AND-parallelism," **COMPCON** Spring '85, pp. 206-210, Feb. 1985.

DeGroot, "Restricted AND-parallelism and Side Effects," 1987 **IEEE Symposium on Logic Programming** , pp. 80-89, San Francisco, California, August 1987.

D. DeGroot, "Restricted AND-parallel Execution of Logic Programs," **Parallel Computation and Computers for Artificial Intelligence**, pp. 91-107, editor: Janusz Kowalik, publisher: Kluwer Academic Publishers, 1988.

P. Dembinski and J. Maluszynski, "AND-parallelism with Intelligent Backtracking for Annotated Logic Programs," 1985 **IEEE Symposium on Logic Programming**}, pp.~29-38, Boston, Massachusetts, July 1985.

Gupta, et. al., "IDIOM Integrating Dependent and-, Independent and-, and Or-parallelism," **Logic Programming, Proceedings of the 1991 International Symposium**}, pp.152-166, San Diego, California, October, 1991.

M. Hailperin and H. Westphal, "A Computational Model for PEPSy," **Technical Report CA-16, ECRC**, 1985.

B. Hausman, A. Ciepielewski and S. Haridi, "OR-parallel Prolog Made Efficient on Shared Memory Multiprocessors," 1987 **IEEE Symposium on Logic Programming**}, pp.~69-79, San Francisco, California, August 1987.

M. V. Hermenegildo, "An Abstract Machine for Restricted AND-parallel Execution of Logic Programs," **Third International Conference on Logic Programming**}, "Lecture Notes in Computer Science," Springer-Verlag, Vol. 225, 1986.

M V. Hermenegildo, R. Warren and S. K. Debray, "Global Flow Analysis as a Practical Compilation Tool", **Journal of Logic Programming** pp. 349-366, 1992:13.

D. Jacobs and A. Langen, "Compilation of Logic Programs for Restricted AND-parallelism," **European Symposium on Programming**}, Nancy, France, 1988.

V. Kumar and Y.-J. Lin, "An Intelligent Backtracking Scheme for Prolog," 1987 **IEEE Symposium on Logic Programming**, pp.406-414, San Francisco, California, August 1987.

Peyyun P. Li and Alain J. Martin, "The SYNC Model: A Parallel Execution Method for Logic Programming," **IEEE 1986 Symposium on Logic Programming**, pp.223-234, Salt Lake City, Utah, Sept. 1986.

F. Luger and W. A. Stubblefield, **Artificial Intelligence**, Benjamin/Cummings Co. 1989.

Mannila and E. Ukkonen, "Flow Analysis of Prolog Programs," **Proceedings of the 4th IEEE Symposium on Logic Programming**, San Francisco, California, September 1987.

C. S. Mellish, "Some Global Optimizations for a Prolog Comiler", {\it **Journal of Logic Programming**, Vol. 2, No.1: pp. 43-66, April 1986.

P. Raman and W. Stark, "Fully Distributed AND/OR-parallel execution of logic Programs," Logic Programming, **Proceedings of the Fifth International Conference and Symposium**}, pp.~1189-1203, Seattle, Washington, August 1988.

U. S. Reddy, "Transformation of Logic Programs into Functional Programs", **Proceedings of the 1984 International Symposium on Logic Programming**, Atlantic City, New Jersey, IEEE Computer Society: pp. 187-196,February 1984.

E. Y. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," {\em TR-003}, **ICOT-Institute for New Generation Computer Technology**, Japan, Jan. 1983.

K. Shen, "Exploiting Dependent And-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS)," Logic Programming, **Proceedings of the Joint International Conference and Symposium on Logic Programming**}, pp.717-731, 1992.

G. Smolka, "Making Control and Data Flow in Logic Programs Explicit," **Proceedings of the 1984 Symposium on LISP and Functional Programming**, Austin, Texas, August 1984.

K. Steer, "Testing Data Flow Diagrams with PARLOG," Logic Programming, **Proceedings of the Fifth International Conference and Symposium**, pp. 96-110, Seattle, Washington, August 1988.

P. Tinker and G. Lindstrom, "A Performance-Oriented Design for OR-parallel Logic Programming," Logic Programming, **Proceedings of the Fourth International Conference**, pp. 601-615, Melbourne, Australia, May 1987.

J.-P. Tsai, T. Weigert and H. C. Jang, "A Hybrid Knowledge Representation as a Basis of Requirement Specification and Specification Analysis," **IEEE Transactions on Software Engineering**, Vol. SE-18, No. 12, pp. 1076-1100, December 1992.

J.J.-P. Tsai and T. Weigert, **Knowledge-Based Software Development of Real-Time Distributed Systems**}, World Scientific, 1993.

J. P. Tsai, B. Li, and T. Weigert, "A Logic-Based Transformation System," **IEEE Transactions on Knowledge and Data Engineering**}, Vol. 10, No.1, pp. 91-107, Jan. 1998.

J. P. Tsai, A. Liu, E. Juan, and A. Sahay, "Knowledge-Based Software Architecture: Acquisition, Specification, and Verification," **IEEE Transactions on Knowledge and Data Engineering**, Vol. 11, No. 1, pp. 187-201, Jan./Feb. 1999.

D. H.D. Warren, "The SRI Model for Or-Parallel Execution of Prolog-Abstract Design and Implementation Issues," 1987 **IEEE Symposium on Logic Programming**, pp.92-102, San Francisco, California, August 1987.

H. Westphal, et al., "The PEPSys Model: Combining Backtracking, AND- and OR-parallelism," 1987 **IEEE Symposium on Logic Programming,** pp. 436-448, San Francisco, California, August 1987.

N. S. Woo and K.-M. Choe, "Selecting the Backtracking Literal in the AND/OR Process Model," **IEEE 1986 Symposium on Logic Programming**, pp.200-210, Salt Lake City, Utah, Sept. 1986.