A Parallel Logic Programming Language for PEPSys

Michael Ratcliffe and Jean-Claude Syre

ECRC, Arabellastr. 17, 8000 Muenchen 81, West Germany

Abstract

This paper describes a new parallel Logic Programming language designed to exploit the OR- and Independent AND-parallelisms. The language is based on conventional Prolog but with natural extensions to support handling of multiple solutions and expression of parallelism.

1. Introduction

PEPSys (Parallel ECRC Prolog System) is a research project started in mid 1984 in the Computer Architecture Group of the European Computer-industry Research Centre (ECRC). Its general goals are to study and evaluate new and practicable solutions to the problems of parallel logic programming. Although the project aims at investigating parallel computer architectures for logic programming, it began with an attempt to define the application programmer's needs [Ratcliffe and Robert, 1985], as well as a study of existing parallel logic models [Syre and Westphal, 1985]. These considerations led us to define a high level language for parallel logic programming, which offers facilities for sequential programming, as in conventional Prolog, as well as others allowing the expression of AND-parallelism, OR-parallelism, controlled by the programmer. Beside the language definition, we have also defined a parallel computational model, and we are currently writing a compiler for the language generating a parallel intermediate-level language. This will be used for an implementation of PEPSys on a commercial multiprocessor system. Simultaneously, we are studying parallel computer architectures adapted to our approach. These will be evaluated by simulation.

This paper focuses on the PEPSys high level language [Ratcliffe and Robert, 1986] . Section 2 presents the objectives of the language and compares them with other approaches. Section 3 describes the main characteristics of the language, and an example showing its main features. Section 4 gives results obtained by a high level interpretation of the language, combined with an evaluation of the execution of some application programs. Section 5 discusses the useful extensions that are felt important but kept aside for further study. Section 6 presents the current status of the activities in the PEPSys project.

In addition to the writers of this paper, the co-authors of the work presented here are Max Hailperin, Philippe Robert, and Harald Westphal. The PEPSys project team includes also Uri Baron, Jacques Chassin de Kergommeaux, Bounthara Ing, and Donald Peterson, all full time researchers at ECRC, most of whom have contributed to the definition of the language by their comments or the use of it.

2. Objectives of the Language

In order to present the objectives of PEPSys, we would first like to situate our approach among the numerous proposals under study for parallel logic languages. Due to lack of space, we will restrict our short review to the most characteristic works. More on the subject can be found in [Syre and Westphal, 1985] [Gregory, 1986] [Crammond, 1985]

2.1. Committed Choice Languages

Historically, the Committed Choice Language approach has received considerable attention [Clark and Gregory, 1986] [Shapiro, 1986] [Ueda, 1985] . Although we do not intend to fight against this class of languages (we find it quite complementary to our own), this approach lacks at least two important features:

- It is not really user-oriented: Almost all these languages (Parlog, Concurrent Prolog, KL1) suffer from a complicated semantic definition, which is reflected in the numerous existing implementations and discussions around both the languages and their implementations. Their most recent derivatives, e.g. Kernel Parlog, Flat CP and the Logix System [Silverman and Houri, 1985], or Flat GHC have reduced capabilities in order to simplify and clarify the semantics of the guards in a clause. The OR-parallelism, a very important source of useful parallelism, is somewhat difficult to handle because it is basically excluded by the principle of guards. The annotations in CP or the modes in Parlog, even if they make the writing of programs not too difficult, often lead to a painful re-reading of the program.

- It is not really implementation-oriented: this approach often leads to an explosion of processes which often bring very little parallelism for very much control and synchronization overhead. It is our strong feeling that the user must be able to control the parallelism to avoid such situations.

This class of languages seems to be well adapted to some applications such as systems programming, simulation or expression of control for numerical programming, where it is useful to have some kind of synchronization mechanisms. Although different but probably aimed at the same applications is Delta Prolog [Pereira et al., 1986] in which explicit events are managed by the programmer to express the synchronization.

2.2. "All solutions" Languages.

By contrast, another class of languages avoids implicit or explicit synchronization constructs, and concentrates on pure parallelism (AND-parallelism, OR-parallelism, induced parallelism). This approach has also been addressed by many people, and at different levels: automatic parallelism detection [Chang and Despain, 1984], pure OR-parallel models [Ciepielewski and Hausmann, 1986] [Kumon et al., 1986], more complete systems including a language like PRISM [Kasif et al., 1983], an execution model such as PIM-D [Ito and Masuda, 1984] or Argonne Labs ANLWAM [Butler et al., 1986]. Among those works, only PRISM really addresses the problems from the language level to the implementation on parallel processors.

2.3. Objectives of the PEPSys Language.

The main objectives of the PEPSys language are the result of the careful study of the existing proposals, complemented by an analysis of the programmer's requirements. As part of a much larger project involving a computational model, and execution models, with which it is totally consistent, it provides an integrated solution to the problem of parallel logic programming. This is often not the case with other approaches which only deal with one facet of the problem. Let us briefly define the objectives of the PEPSys language:

- **An "all-solutions" language:** PEPSys is targeted for non-deterministic logic computations which lead to potentially several solutions, all of them being considered useful to the user. Decision making systems, and open problems often need this non determinism, simply because the "right" solution cannot be defined by the program, but only by the user inspecting the several solutions obtained from his query.

- **A language with a declarative semantics:** while it is true in some cases that one knows the behaviour of a predicate and its clauses (as is necessary in Parlog, for example), our approach is to retain the declarative property. Our language is compatible with Prolog, which may be useful when the user writes sequential portions of his program (and there are always sequential parts in a program).

- **A flexible language**, both at the expression level and at the implementation level. The flexibility at the expression level is conveniently achieved only if the programmer is given enough explicit control capabilities.

- **A language to produce easy-to-write, easy-to-read programs**, with a simple syntax, and a clear semantics, even at the expense of some additional writing. Punctuations and special keyboard characters are not felt the best way to express such important features as asynchronism or parallelism, especially when the program is to be updated.

The next section will present the language. It attempts to fulfil these objectives with three basic ideas: modularity, explicit independent AND-parallelism, and user-controllable OR-parallelism.

3. The PEPSys Language

3.1. Modules and Interfaces

When developing any large piece of software, imposing a modular structure on a language greatly aids compile-time error checking and analysis, particularly when compiling a small part of a very large program. PEPSys modules are completely self-contained. This means that all predicates accessible from within the module must be either explicitly declared within it or else explicitly imported into it. Similarly, any definitions required by other modules must be explicitly exported from the module containing the definition.

In order to achieve such a *closed* structure, two built-in predicates are used to declare inter-module interfaces:

```
?-export( [exp/3]).
?-import_from( 'other_module.par', [imp/5] ).
```

The definition of the predicate *exp/3* is exported from the current module and is available for importing into any other module. The definition of the predicate *imp/5*, defined in the module *other_module.par*, is imported into the current module and therefore available for use within it. It is not possible to implicitly import predicates, other than the standard built-in predicates, since no *global* declaration is provided. This restricts the scope of any possible name clashes to a local module.

A PEPSys program has two types of modules: serial and parallel.

Serial modules contain conventional Prolog code and use the the normal Prolog depth first execution strategy with backtracking to select alternative clauses. The only unusual feature is that all the clauses for a predicate must be grouped together. Access to any other predicates defined within other serial modules is provided directly by the declarations illustrated above.

Access to predicates defined in parallel modules is also provided through the same interface mechanism but the actual usage of such predicates must only occur from within the built-in predicates *oneof/1*, *bagof/3* and *setof/3*. The *oneof* predicate is used to obtain a single result, the first in time, generated by a predicate call whilst the other two collect alternative solutions from the predicate call. These three predicates will all fail on backtracking.

Parallel modules contain only the parallel PEPSys language. This includes most of the usual Prolog predicates, plus *oneof/1*, *bagof/3* and *setof/3*, but excludes all side-effect predicates (e.g. *assert/1*, *retract/1*, *read/1*, *write/3*, *!/0* etc.). Parallel modules may use predicate definitions declared within other parallel modules using the same interface declaration convention as used in serial modules. It is **not** allowed to import a predicate from a serial module into a parallel one and it is not necessary to use the built-in predicates *oneof*, *bagof* and *setof* when using predicates defined within other parallel modules.

These two module types are distinguished by the file name extension used to contain their code. Thus serial modules

have the extension *.ser* and parallel modules the extension *.par*. This follows the approach used in ECRCProlog [Estenfeld and Meier, 1986].

This structure separates the parallel and sequential parts of a program in a clear way. The programmer is relieved of the burden of having to imagine and manage complex interactions between asynchronous concurrent processes whilst still having access to powerful side-effect facilities. Large application programs are easy to manage and comprehensive compile-time error checking is facilitated.

3.2. AND-Parallelism

The parallel PEPSys language supports the parallel execution of Independent goals. Two goals are considered Independent only if they have either no uninstantiated shared variables or else cannot instantiate any shared uninstantiated variables to different values*. In this way, the overheads of general AND-parallelism is avoided.

Progress has been made in automatically detecting such independent goals. However, with state-of-the-art technology, we still believe the programmer to be the best guide, particularly to decide what is worthwhile parallelism (i.e. worth the overheads).

Goals which are independent in this sense are separated with the *independent* operator (#) instead of the usual comma (,). This indicates to the compiler and/or runtime system that these goals may be safely executed in AND-parallel mode.

The language places no restriction on the number of solutions each goal of an AND-parallel execution produces. There are no constraints as to how parallel constructs may be nested.

3.3. OR-Parallelism and Predicate Properties

In a parallel module, as well as being grouped together, the clauses of a predicate must be preceded by a *properties declaration*. This declaration contains additional information about the predicate to that expressed within the clauses themselves. This information is used to express whether it is worthwhile executing all clauses concurrently (i.e. OR-parallelism), whether the clause ordering is significant, and the number of valid solutions the predicate is allowed to generate. Some of this information is semantic in that it reflects on the meaning of the clauses written as the predicate's definition, whilst some is pragmatic in that it is merely advice to a compiler or runtime support system.

Property declarations have the form:

```
-properties( <list_of_properties> ).
```

Three properties are supported by the language; they are completely orthogonal:

- the *solutions* property - specifies whether all or only one of the solutions the predicate is able to generate are to be considered as valid.
- the *clauses* property - specifies whether the ordering of clauses is significant or not.
- the *execution* property - specifies whether executing all clauses concurrently is likely to be useful or not

It is important to note the fundamental difference in nature between the *solutions* and *clauses* properties and the *execution* property. The latter has no effect on the semantics of the predicate definition. It merely acts as advice to a run-time scheduler, effectively saying "if there are enough free resources, then allocate the execution of these clauses to different processes". Thus, the parallelism exploited may be constrained to the resources available.

Conventional sequential execution is also embedded in this scheme with the use of the following property declaration:

```
-properties([solutions(all),
            clauses(ordered),
            execution(lazy)]).
```

In this example, the *solutions* property allows all the solutions the predicate can generate to be considered as valid, the *clauses* property forces the clause ordering to be significant. That is, solutions from the first clause will be returned before any from later clauses. Finally, the *execution* property recommends *lazy* execution. This invokes the usual execution mechanism of generating choice-points and backtracking to these when failure occurs. However, note again that the *execution* property is purely advisory; there is no observable difference to the user if the execution property had been processed as if it were *eager*, that is, if all clauses had been executed concurrently. The resulting solutions from different clauses would still be ordered in the sequence order of the clauses generating them, and all the solutions generated would still be valid. The only difference would be that the backtracking mechanism could have been replaced by parallel execution. In general, it is assumed that any implementation would only exploit parallelism where it is recommended to do so.

Although there is no *cut* operator in the parallel language, its effect can be simulated using the *solutions* property. If the *solutions* property is defined as *one*, meaning that only the first solution generated in time is considered valid, then this is equivalent in conventional Prolog to having a *cut* as the last goal in every clause. If the predicate also has the property *clauses(ordered)*, then the solution generated must come from the first clause able to generate a solution. Using this mechanism it is possible to devise a general program transformation for any Prolog predicate with *cuts* into the PEPSys language.

3.4. An Example Program

This now seems an appropriate point at which to look at the PEPSys code for a short program. For this purpose, we will present a PEPSys coding for the *n-queens* program (see fig 3.4).

*This is also referred to as *restricted AND-parallelism* in the literature.

```
/*-----------------------------------------------------------------------*/
/*     PEPSys n-queens Programme              (c) copyright ECRC GmbH    */
/*     ==========================                 Muenchen 1986          */
/*   Authors: M. J. Ratcliffe and P. Robert.                             */
/*   Description:  serial module of the 'n-queens' program               */
/*   Entry point: go/1 ... argument is the integer board size            */
/*-----------------------------------------------------------------------*/

?- export( [go/1] ).                                  /* User entry point */
?- import_from( 'queens.par', [get_solutions/2] ).

go( Size ) :-
        bagof( Soln, get_solutions( Size, Soln ), Solutions ),
        member(S, Solutions), writeln( S ), fail.
go( Size ).

/*-----------------------------------------------------------------------*/
/*     PEPSys n-queens Programme              (c) copyright ECRC GmbH    */
/*     ==========================                 Muenchen 1986          */
/*   Authors: M. J. Ratcliffe and P. Robert.                             */
/*   Description:  parallel module of the 'n-queens' program             */
/*   Entry point: get_solution/2 ... called from serial module           */
/*-----------------------------------------------------------------------*/

?-export( [get_solutions/2] ).      % Export entry point

-properties( [] ).
get_solutions(Board_size, Soln) :- solve(Board_size, [], Soln).

% Accumulate the positions of occupied squares
-properties( [solutions(all),clauses(ordered),execution(lazy)] ).
solve(Bs, [square(Bs, Y) | L], [square(Bs, Y) | L]).
solve(Board_size, Initial, Final) :-
        newsquare(Initial, Next, Board_size), solve(Board_size, [Next | Initial], Final).

% Generate legal positions for next queens
-properties( [solutions(all),clauses(ordered),execution(lazy)] ).
newsquare([square(I, J) | Rest], square(X, Y), BoardSize) :-
        I < BoardSize, X is I + 1, snint(Y, BoardSize),
        not(threatened(I, J, X, Y)) # safe(X, Y, Rest).
newsquare([], square(1, X), BoardSize) :- snint(X, BoardSize).

% Generate all possible positions for the next queen
-properties( [solutions(all),clauses(unordered),execution(eager)] ).
snint( X, X ).
snint( N, NPlusOneOrMore ) :-  M is NPlusOneOrMore - 1, M > 0, snint( N, M ).

% Check whether queens on squares (I, J) and (X, Y) threaten each other
-properties( [solutions(one),clauses(unordered),execution(lazy)] ).
threatened(I, J, X, Y) :- I = X.
threatened(I, J, X, Y) :- J = Y.
threatened(I, J, X, Y) :- U is I - J, V is X - Y, U = V.
threatened(I, J, X, Y) :- U is I + J, V is X + Y, U = V.

% Checks whether square(X, Y) is threatened by any existing queens
-properties( [solutions(one),clauses(ordered),execution(lazy)] ).
safe(X, Y, []).
safe(X, Y, [square(I, J) | L]) :-
        not(threatened(I, J, X, Y)) # safe(X, Y, L).
```

Figure 1: A PEPSys Coding of the n-queens Program

This program is coded in two modules. The serial module, *queens.ser*, contains the user interface whilst the parallel module, *queens.par*, contains the parallel code. The interface between the two is provided by the *get_solutions/2* predicate. This is called from the serial module using the *bagof/3* predicate to collect all the solutions.

This program exploits both OR- and AND-parallelism. OR-parallelism is used to generate, and continue processing with, all the possible positions for the next queen. These are generated by the *snint/2* predicate. AND-parallelism is then used in the *newsquare/3* and *safe/3* predicates to execute the validity tests on the newly generated position of the next queen in parallel.

It is informative to consider the property declarations of the parallel predicates a little more closely:

- *get_solutions/2*: the properties are defaulted since only the *solutions* property is relevant and this is defaulted to *all*.
- *solve/3*: all solutions are required but it is not worth executing the two clauses in parallel.
- *newsquare/3*: all solutions are required but it is not worth executing both clauses concurrently since unification, which will mostly choose the first clause, will decide between them.
- *snint/2*: this predicate generates the OR-parallelism by generating all possible row positions in a column for a new queen.
- *threatened/4*: it is only necessary to prove one condition of a false position for a queen but the tests are not complex enough to be worth parallel execution.
- *safe/3*: this is a vector operation to try and find any previously placed queen threatening the newly placed one.

4. Language Evaluation

Any computer language is valueless without an implementation so we have written an interpreter for our language in CProlog. This interpreter can also generate an execution trace file. This file can than be interpreted by an analysis program in terms of parallel concurrently executing processes.

The analysis of the trace file makes several assumptions to simplify its work. It must be remembered that the purpose of this analysis is to evaluate a PEPSys coding for a program rather than predict the performance on a particular system; it measures the amount of parallelism expressed by the code. The main assumptions are presented below:

- each goal executes in one time unit
- no overhead for process splitting
- unlimited resources are available
- all AND-parallel goals are executed to completion
- OR-parallel processes split after unification

The assumption of unlimited resources violates a fundamental principle of the PEPSys project, namely that the amount of parallelism exploited should be restricted by the resources available. However, when investigating how much parallelism is expressed within a program and estimating what resources it could usefully utilise, this assumption is reasonable.

It is in fact an option that all OR-parallel processes are split after unification. The alternative is to perform unification in the child process(es). This case is a little naive and results in the creation of processes which may fail after performing unification, thus wasting the overhead of process creation. The assumption presented here can be thought of as representing a perfect indexing scheme in the selection of candidate clauses. The real situation lies somewhere between these extremes; we hope nearer the latter than the former!

Using these tools, the execution of five PEPSys programs was analysed. The table below summarises the results:

Program Name	Total No of Goals Executed	Maximum No of Active Processes	Speedup Factor
Arg1	31202	1200	267
Four Queens	1261	40	17
Map1	5700	130	56
Path_search	3223	28	8
Warplan	71796	78	10

The speedup factors quoted above are calculated by dividing the total number of goals executed by the execution time (which is also measured in numbers of goals). This is valid when all solutions are generated but is otherwise questionable.

The first program, *arg1*, is the salt and mustard problem re-coded from the original written at Argonne labs. The second is a specific example of the *n-queens* program discussed above; in this case two solutions are generated. The *map1* program is an implementation of the map colouring problem; this coding follows that used in [Ciepielewski et al., 1985] and exhibits much OR-parallelism. The *path* program is a simple heuristic search application implemented at ECRC. Using a representation of a public transport network it generates all reasonable ways of travelling between two nodes. The network used here has only 17 nodes. The final program, *warplan*, is a simple re-coding of Warren's original. Here it is solving a block's world problem presented in the same paper.

The graph below shows the number of concurrent processes (y-axis) as a function of time (x-axis) for two of the programs analysed. The solid line corresponds to the *four queens* problem and the dashed line to the *map1* problem:

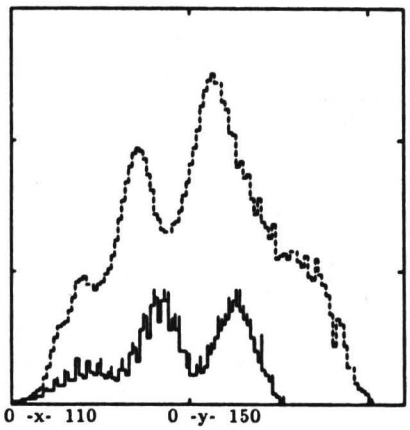

Figure 2: The Number of Concurrent Processes in Two PEPSys Programs

Using the language, the programmer can tune the parallelism in his program. The effects of adjusting the granularity of the parallelism in this way is illustrated in the table below. This shows the speedup factor for a single program (a soccer team selection problem) as a function of the amount of parallelism added to the basic sequential source program:

amount of OR-//ism	NONE	FULL	FULL	FULL
amount of AND-//ism	NONE	NONE	PART	FULL
execution time	816	38	32	10
speedup factor	-	21.5	25.5	81.6

Another result of this preliminary evaluation of parallel programs concerns the potential dangers of not controlling the parallelism. A program can generate a large amount of parallel activities, but many of them may be just duplicates: some OR branches in the program may produce the same intermediate solutions, and every parallel path generated afterwards will perform identical computations. The activation of AND-parallel branches can lead to a cross product of intermediate results which are just permutations of the same subresults. In the same vein, an OR-parallel branch can lead to so many parallel computations that it would saturate any multiprocessor system. The PEPSys language offers the programmer an adequate set of explicit constructs to adjust and refine his source programs in a clear and simple way.

We find these results encouraging in that our language does indeed lend itself to the expression of significant parallelism. It has even proved possible to get significant gains from simple translations of conventional Prolog programs.

5. Proposals for Language Extensions

When coding programs, we have come to recognise some limitations in the language. These proposals are designed to eliminate these.

The parallel PEPSys modules are completely static in nature. Whilst this presents a very clean language, it also has problems, particularly when it comes to comparing the efficiency of some Prolog programs with their PEPSys equivalents. This proposal provides a global dynamic database within the parallel environment with the unusual property, when compared with conventional Prolog, of not guaranteeing coherence. There are three areas in which this could be useful: lemmas (avoiding repeatedly computing the same intermediate solutions), constraining the search space (heuristic rules may be dynamically adjusted to generate only reasonable solutions) and process synchronisation (a process may actively wait for another to transmit a message).

In some cases it is desirable to modify a particular predicate's properties for a particular call of that predicate. Only modifications which do not override the basic nature of the predicate definition are allowed:

```
clauses:      "unordered" to "ordered"
execution:    "eager"     to "lazy"
```

No modification of the *solutions* property is necessary since the built-in predicate *oneof/1* accomplishes the same function. Such modifications may also be used with built-in predicates.

A vector relation is a relation between corresponding elements of lists (vectors) which can be executed concurrently for each set of corresponding elements. In the basic language, this is expressed by using recursion to select the list elements and executing the recursive goal call in AND-parallel mode. This has been identified as an important source of parallelism in Prolog [Ratcliffe and Robert, 1985]. The exploitation of this parallelism is inefficient because a recursive goal call must be executed before the next vector element process can be generated. A special syntax for vector operations would save the execution time of n goals, where n is the length of the vector, and initiate the concurrent processes faster.

The unification between a goal and a clause head is often augmented by the execution of a few simple tests. Such goals are really unification constraints and should be expressed as such. By compiling such constraints into the unification process the overall efficiency of execution will be improved. The so-called *flat guarded languages* essentially use guards in this manner.

The use of guards could be introduced in the style of P-Prolog at Keio University. In this case guards could be used to express a predicate able to produce multiple solutions but commit execution to a single clause. Currently this can only be done using multiple layers of predicates.

6. Current and Future Work on the PEPSys Project

The parallel programming language described in this paper is one activity of the PEPSys project. Associated with it, we have defined a parallel logic computational model. This model handles Sequentiality, AND-parallelism and OR-parallelism, in a resource-limited environment, with as little overhead as necessary. When the resources become saturated, the potentially parallel processes are executed sequentially, and conversely, a parallel process which was forced to run sequentially may be retroactively made parallel with a minimal overhead if resources become available. This facility outperforms the existing models. It is more general than the models defined by Hermenegildo (MCC, AND-parallel only) or Ciepielewski (SICS, OR-parallel only), or even Overbeek

(ANL, only deterministic AND-parallel branches). It uses an improved Hash Windows scheme for representation of data structures, and thus is more efficient than the Kabu Wake method (recopy of the whole state to start retroactive parallelism), and the other models mentioned above (however the ANL model, also in use at SICS and Manchester University with the concept of "favored" branches, seems to be very efficient, too).

From the language and the model definitions, we are currently working in three areas of interest:

- Language and applications: we are writing application programs, to test the language, as well as to evaluate our global approach. The programs are adaptations of sequential ones (PRESS, Logic Circuit Fault Finder), or they are developed from scratch with an initial parallel analysis of the problem (Public Transportation System Adviser, Tourist Information Adviser).

- Direct implementation on a commercial multiprocessor system. The PEPSys language and the computational model are currently being implemented on a Siemens MX-500 multiprocessor (similar to a Sequent Balance 8000 system). A compiler, using the technology developed at ECRC for the ECRC compiler and for the Sequential Inference Processor, is under test.

- Evaluation of execution models of multiprocessor systems adapted to PEPSys. We are defining new architectural models adapted to the computational model. We will implement a set of simulation tools that will evaluate those models on benchmark programs written in the PEPSys language.

Further results will include the performance evaluation of the direct implementation, the simulation of new parallel computer architectures for logic programming (in our class of languages, i.e. not the Committed Choice class), and a revised version of the programming language.

7. Conclusion

In this paper we have defined the main features of a parallel logic language which covers most of the user's requirements for programming lage scale applications containing potential AND and OR-parallelisms, as well as sequential portions. The modularity and some built-in predicates allow for a clean, structured programming methodology, with well-defined interfaces between sequential and parallel modules. Within a sequential module, conventional Prolog is used, with all the facilities to interface with the user or the system. Within a parallel module, a predicate is written in a familiar syntax close to prolog, and augmented with a Property Declaration. This Property Declaration defines the parallel behavior of the predicate, as seen by the programmer with regard to its use in the program. It also provides some flexibility in that the programmer can tune the real parallelism he wishes to have. The explicit expression of parallelism is considered better than any other solution, since it reflects more the specification of the initial problem. Some examples (most of them taken from existing sequential implementations), and their pseudo-dynamic evaluation by the interpreter, have shown a promising parallel behavior, which should be even better in the programs whose specifications use directly the capabilities offered by the language

The basic language is associated with a computational model and is currently being implemented on a commercial multiprocessor system. Real world (but still small) applications are being written to evaluate its capabilities.

We are aware that the language presented here is not the "ultimate" parallel logic language, but may be the starting point for an extended one, incorporating more tools to express other kinds of concurrency, such as those existing now in Parlog, GHC, or Delta Prolog. In the same vein, extensions to an asynchronous data base allowing loose, chaotic synchronisation, or vector parallelism constructs, may appear useful in real applications. This will be studied in conjunction with the other work in the PEPSys system at ECRC.

References

[Butler et al., 1986]
R. Butler, E. Lusk, W. McCune, and R. Overbeek.
Parallel logic programming for numeric applications.
In Ehud Shapiro (editor), *Third International Conference on Logic Programming*, pages 375-388. London, July, 1986.

[Chang and Despain, 1984]
J.H. Chang and A.M. Despain.
Semi-intelligent backtracking of Prolog based on a static data dependency analysis.
Technical Report internal report, University of California Berkeley, 1984.

[Ciepielewski and Hausmann, 1986]
A. Ciepielewski, B. Hausmann.
Performance Evaluation of a Storage Model for OR-parallel Execution of Logic Programs.
In *Proc. 1986 Symposium on Logic Programming*, pages 246-257. Salt Lake City, Utah, September, 1986.

[Ciepielewski et al., 1985]
A. Ciepielewski, S. Haridi and B. Hausman.
Initial Evaluation of a Virtual Machine for Or-parallel Execution of Logic Programs.
In UMIST (editor), *IFIP TC-10 Working Conf. on Fifth Generation Computer Architecture.* IFIP, Manchester, July 15-18, 1985.

[Clark and Gregory, 1986]
K. Clark and S. Gregory.
PARLOG: Parallel Programming in Logic.
acm Transactions on Programming Languages and Systems 8(1):1-49, January, 1986.

[Crammond, 1985]
J. Crammond.
A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic Languages.
In D. DeGroot (editor), *Int. Conf. on Parallel Processing*, pages 131-138. IEEE, St. Charles, Ill., August, 1985.

[Estenfeld and Meier, 1986]
K. Estenfeld and M. Meier.
ECRC Prolog User's Manual Version 1.2.
Technical Report LP - 13, ECRC, September, 1986.

[Gregory, 1986]
Steve Gregory.
Parallel Logic Programming: The State of the Art.
Internal Report, ECRC, May, 1986.

[Ito and Masuda, 1984]
Noriyoshi Ito and Kanae Masuda.
Parallel inference machine based on the dataflow model.
In *Intl Workshop on High level Computer Architecture 84*, pages 10. LA, May, 1984.

[Kasif et al., 1983]
Simon Kasif, Madhur Kohli, and Jack Minker.
PRISM A Parallel Inference System for Problem Solving.
In Nucleo de Inteligencia Artificial, Universidade Nova de Lisboa (editor), *Proc. 1983 Logic Programming Workshop*, pages 123 - 152. Maryland USA, June, 1983.

[Kumon et al., 1986]
K. Kumon, H. Masuzawa, A. Itashiki.
Kabu-Wake: A new parallel inference method and its evaluation.
In *Proc. IEEE COMPCON 86*, pages 168-172. San Francisco, March, 1986.

[Pereira et al., 1986]
L.M. Pereira, L. Monteiro, J. Cunha, and J.N. Aparicio.
Delta Prolog: a distributed backtracking extension with events.
In Ehud Shapiro (editor), *Third International Conference on Logic Programming*, pages 69-83. London, July, 1986.

[Ratcliffe and Robert, 1985]
M. J. Ratcliffe and P. Robert.
The Static Analysis of Prolog Programs.
Technical Report CA-11, ECRC, October, 1985.

[Ratcliffe and Robert, 1986]
M. J. Ratcliffe and P. Robert.
PEPSy: A Prolog for Parallel Processing.
Technical Report CA-17, ECRC, March, 1986.

[Shapiro, 1986]
Ehud Shapiro.
Concurrent Prolog: A Progress Report.
IEEE Computer 19(8):44-58, August, 1986.

[Silverman and Houri, 1985]
W. Silverman, A. Houri.
Logix, Installation Manual for Release 1.1
Weizmann Institute of Science, 1985.

[Syre and Westphal, 1985]
Jean Claude Syre and Harald Westphal.
A Review of Parallel Models for Prolog.
Technical Report CA-07, ECRC, June, 1985.

[Ueda, 1985]
Kazunori Ueda.
Guarded Horn Clauses.
Technical Report TR-103, ICOT, June, 1985.