Automatic Exploitation of Non-Determinate Independent And-Parallelism in the Basic Andorra Model

M.Olmedilla, **F. Bueno**, and **M. Hermenegildo**
{moa,bueno,herme}@fi.upm.es
Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - SPAIN

Andorra-I is the first implementation of a language based on the Andorra Principle, which states that determinate goals can (and should) be run before other goals, and even in a parallel fashion. This principle has materialized in a framework called the Basic Andorra model, which allows or-parallelism as well as (dependent) and-parallelism for determinate goals. In this report we show that it is possible to further extend this model in order to allow general independent and-parallelism for nondeterminate goals, without greatly modifying the underlying implementation machinery. A simple an easy way to realize such an extension is to make each (nondeterminate) independent goal determinate, by using a special "bagof" construct. We also show that this can be achieved automatically by compile-time translation from original Prolog programs. A transformation that fulfills this objective and which can be easily automated is presented in this report.

Keywords: language constructs, programming environments, formal program development methodologies, implementation issues, evaluation, simulation and benchmarking.

Corresponding author:

Manuel Hermenegildo
Facultad de Informatica
Universidad Politecnica de Madrid
28660-Boadilla del Monte, Madrid
SPAIN

Telephone:	+341-336-7435
Email:	`herme@fi.upm.es`
FAX:	+341-352-4819

Automatic Exploitation of Non-Determinate Independent And-Parallelism in the Basic Andorra Model[2]

M.Olmedilla, **F. Bueno**, and **M. Hermenegildo**
{moa,bueno,herme}@fi.upm.es
Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - SPAIN

Andorra-I is the first implementation of a language based on the Andorra Principle, which states that determinate goals can (and should) be run before other goals, and even in a parallel fashion. This principle has materialized in a framework called the Basic Andorra model, which allows or-parallelism as well as (dependent) and-parallelism for determinate goals. In this report we show that it is possible to further extend this model in order to allow general independent and-parallelism for nondeterminate goals, without greatly modifying the underlying implementation machinery. A simple an easy way to realize such an extension is to make each (nondeterminate) independent goal determinate, by using a special "bagof" construct. We also show that this can be achieved automatically by compile-time translation from original Prolog programs. A transformation that fulfills this objective and which can be easily automated is presented in this report.

Keywords: language constructs, programming environments, formal program development methodologies, implementation issues, evaluation, simulation and benchmarking.

1 Introduction

The Andorra proposal in [War88] (now being called the "Andorra Principle") pointed out that advantage could be taken of the execution of determinate goals ahead of their standard execution "turn." A goal is determinate if it can be determined that only one clause in the predicate definition will match with it [SCWY90]. The execution of deter-

minate goals has the very desirable properties of allowing to maintain a unique branch of computation, avoiding the complexity which is inherent to Prolog non-determinism (i.e. choice-points). Furthermore, the execution of determinate goals can further reduce the search space of other goals, and even make them become also determinate.

In addition, the Andorra proposal also defined a framework which allowed or-parallelism and also the and-parallel execution of *determinate* goals (determinate, or deterministic "stream and-parallelism"). This has been known as the Basic Andorra model, and is the basis for the Andorra-I language and system implementation [SCWY91b, SCWY91a]. It supports or-parallelism and also dependent and-parallelism for determinate goals, which gives rise to an implicit coroutining which resembles that of flat committed-choice languages.

Another and complementary way of achieving parallel execution which has also been identified [HR89, HR90] is to also run in parallel *nondeterminate* goals, but provided (or while) they are independent ("independent and-parallelism" –IAP). The execution of independent goals in parallel has the very desirable properties of preserving the program complexity perceived by the programmer [HR89]. One way to also include this type of parallelism is to further extend the above mentioned framework, giving rise to new models incorporating all of these types of parallelism, e.g. the Extended Andorra Model (EAM) [War90, HJ90], IDIOM [GSCYH91], etc. However, this requires the implementation of complex execution models. In this report we present an alternative which makes use of the existing implementation of the Basic Andorra model, throughout a technique for transformation of Prolog programs into programs that can be run in Andorra-I. This transformation allows the Andorra-I system to exploit either parallel execution of determinate goals, even if they are dependent, and or-parallelism (as per the Andorra Principle), as well as parallel execution of independent goals.

The technique is based on the use of determinate builtins to encapsulate independent goals, so that they would be run in parallel in Andorra-I. Additional considerations for a Prolog program to run in Andorra-I are also discussed. An algorithm for the transformation and our implementation of it are presented. This implementation makes use of compile-time tools that have been developed in the context of the &-Prolog system [HG90], a language implementation based on the ideas of Independent And-Parallelism.

In section 2 the background idea of the transformation is presented, then a first approach and some improvements to it can be found in section 3. The behaviour of the transformed programs is discussed in section 4. Our implementation of an automatic translator based on these ideas is presented in section 5, some examples on the behaviour of this translator in section 6, and our conclusions in final section 7.

2 Making Independent Goals Determinate

In the context of IAP, goals can be run in parallel provided they are independent, while in the Basic Andorra model goals can be run in parallel provided they are determinate. In order to introduce the possibilities for parallel execution of IAP in the model of execution of Andorra-I, a rewriting of independent goals could be done that made them become determinate. For these purposes the Prolog builtin bagof/3 can be used. This builtin predicate allows solutions for a predicate to be collected, and can be made determinate under some circumstances.

A goal `bagof(Term,Goal,Solutions)` is true if `Solutions` is the list of all solutions to `Term` for the execution of `Goal`, for the given instance of this goal. For example, the goals `bagof(X,p(X,Y),[a])` and `bagof(X,p(X,Y),[b])` are true for the given program:

```
p(a,1).
p(b,2).
p(c,3).
```

for which the goal `bagof(X,p(X,Y),L)` is nondeterminate, whereas other goals such as, for example, `bagof(X,Y^p(X,Y),L)` (i.e. goals which existentiate the variables which do not appear in the first argument) would be determinate, with `[a,b,c]` as the solution for `L`. Furthermore, the goal `bagof((X,Y),p(X,Y),L)` would also be determinate, with `[(a,1),(b,2),(c,3)]` as the solution for `L`. The latter will be preferred for the rewriting of independent goals in our case.

For those goals which are known to be independent, a transformation can be done at compile-time that encapsulates each of them in a bagof/3 goal (from now on, referred to as simply "bagof") in such a way that it becomes determinate. At execution-time, since Andorra-I supports (full) Prolog [SCWY91b], these goals will be found to be determinate, and therefore, executed in parallel. Then all solutions collected for each of these goals should be traversed, which can be done with a member/2 predicate, as explained in the next section.

There already exist algorithms for detecting independent goals at compile-time in a given program [MH90]. These take as input a Prolog program and annotate it with the correct conditions [HR90] for detecting independence among goals at run-time. The annotated program is thus runnable in independent and-parallel fashion in a system like &-Prolog. Therefore, these algorithms can be seen as performing a translation from Prolog to &-Prolog that makes the exploitation of IAP possible. In order to make this also possible in Andorra-I, a full transformation of a Prolog program can be done, using as a front-end the mentioned algorithms, and as a back-end a bagof encapsulation based on the ideas presented above.

3 The Transformation

Having as input an annotated &-Prolog program, which itself makes explicit the available (independent-and) parallelism via if-then-else constructions, the transformation that encapsulates goals in bagofs can be performed over the annotated goals. These goals would appear in the "then" part of the ite (if-then-else) connected by the '&' operator, which states that they can be run in parallel; otherwise they would appear in the "else" part of the ite connected by a comma ',' for sequential conjunction. The conditions for independence of the goals always consist of checks on the groundness and independence of variables. Thus

```
ground(X), indep(Y,Z) -> p(X,Y) & p(X,Z)
                       ; p(X,Y),  p(X,Z).
```

can be read as "if `X` is ground and `Y` and `Z` are independent, then run `p(X,Y)` and `p(X,Z)` in parallel, otherwise run them sequentially." This has the same meaning as the equivalent construction known as CGE (Conditional Graph Expression [MH90]), which in this case would be:

```
ground(X), indep(Y,Z) => p(X,Y) & p(X,Z).
```

The proposed transformation will then put these goals as the second argument of (separate) bagofs, and their respective variables in a tuple as the first argument of the respective bagof. Provided the bagof is constructed as determinate, these two goals will run in parallel in Andorra-I, achieving the desired effect. The only thing left is to obtain each of the solutions for each one of the goals and combine them in a cartesian product way (i.e. simulate a "join" operation, as discussed in section 4). To do this, advantage can be taken of the backtracking capability, present in Andorra-I as well as in Prolog: as all solutions are collected in a list structure, each of these can be inspected using a member/2 predicate, which will give in turn a solution for each goal; all solutions will then be traversed by backtracking, in a standard Andorra-I computation.

This transformation is reminiscent of the encapsulation of independent and-parallel goals in or-parallelism proposed in [CDO88], although it serves a quite different purpose. Here, our aim is to exploit the and-parallelism capability of the target system, resembling that of [Ued87], and as in this, it also can be seen as an optimization for exhaustive search programs, although the transformation performed can be viewed as opposite to that one.

Following the above guidelines, the previous example, after being transformed, would look like:

```
ground(X), indep(Y,Z)
   -> bagof((X,Y),p(X,Y),P1),
```

```
   bagof((X,Z),p(X,Z),P2),
   member((X,Y),P1),
   member((X,Z),P2)
 ; p(X,Y), p(X,Z).
```

Note that the original variables `X`, `Y` and `Z` have been used in the member/2 predicates; this poses no new problems, since execution of the bagofs does not further instantiate these variables. Also, all variables in the original goals have been included in the term that captures the solutions, the first argument of the bagofs. This ensures that bagofs are determinate and is general enough for our purposes; nonetheless, it can be further optimized, as we will see in the following.

First, there is no need to include in the tuple for the first argument of the bagofs variables whose values are known, i.e. variables which are ground at execution-time. One of these in the above example is the variable `X`: because the bagofs are only executed when the condition for independence succeeds, there it is already known that this variable is actually ground, therefore it can be omitted from the tuple, and this can reduce the size of the list structure for all solutions (i.e. `P1, P2`). This optimization has general applicability for every variable for which it can be known at compile-time that it would be ground at execution-time. This information can be obtained by a local analysis based on the inspection of the conditions of the ites, as in this example, or by a global analysis, commented below in section 5.

To ensure that the above optimization is correct in every case, the fact that all variables of a given goal can be ground has to be taken into account. In this case one may think that the bagof could be omitted, but there is no guarantee that the original goal, although ground, will be clause-determinate, and thus parallelism could be lost if it is not encapsulated. Therefore it is preferable to introduce the bagof with any one of the variables in the original goal, but then, as the list of solutions for this goal would have only one element, there is no reason to include the member/2 goal and it can be omitted.

Second, the original goals can have as arguments anonymous variables, i.e. variables whose values are not relevant for execution of the body of the clause in which they appear[3]. As before, there is no need to include these variables in the tuple for the bagofs, instead they can be existentiated in the second argument of the bagofs[4], so that their values are ignored, but the bagofs remain determinate. This optimization requires a renaming of these variables so that they can be referred to in the existentiation as well as in the goal.

In the example, let p/2 be p/3, adding to it an extra argument which is not relevant in the computation. Thus, the final complete transformation would lead to:

```
ground(X), indep(Y,Z)
```

[3]These variables appear usually as '_' in the program, but not necessarily.

[4]The findall/3 builtin can be used as syntactic sugar for this.

```
-> bagof(Y,A1^p(X,Y,A1),P1),
   bagof(Z,A2^p(X,Z,A2),P2),
   member(Y,P1),
   member(Z,P2)
 ; p(X,Y,_), p(X,Z,_).
```

A final point that should be noted here concerns the builtin predicates. In principle, most of them are themselves determinate, and thus it makes no sense to include them in a bagof. Moreover, we find that some considerations on the granularity of goals should apply to parallelization: only goals with a reasonable execution size should be parallelized, otherwise the extra work needed for running in parallel would overcome the advantage obtained from parallel execution. Thus, builtins in general should not be parallelized. These considerations have been taken into account in the above mentioned algorithms for annotation of parallelism, thus our transformation would deal only with goals already annotated.

4 Collecting Solutions in Andorra-I

The Andorra-I system implements the language Andorra-I Prolog [SCWY91b]. This subsumes standard Prolog, including side-effects and most standard builtins, with the cut pruning operator, and also the commit pruning operator, as in committed-choice languages.

A bagof builtin predicate is being implemented in Andorra-I with a semantics that resembles that of standard Prolog. Thus a general goal `bagof(Term,Goal,List)` should be true if `List` includes all solutions (abstraction made of their ordering) for the term `Term` for which the particular instance of Goal is true. If `Term` includes all the variables of `Goal` the bagof goal will be considered determinate. Otherwise, if `Goal` contains uninstantiated variables not included in `Term` the bagof goal will wait for these to get instantiated; this can be avoided if these variables are existentiated in `Goal`: their instantiation will not be considered relevant and the bagof goal will not wait for their values to execute.

In an Andorra-I computation two distinct phases are performed [SCWY91a]: in the determinate phase determinate goals are evaluated (in and-parallel); when this ends, the nondeterminate phase selects a goal for reduction and creates a choice-point, branches of this can be explored in or-parallel. If no particular control is specified, the goal reduced is the leftmost goal, as in Prolog. In the presence of failure, the branch failed is abandoned, and execution procceeds with the rest of the branches, backtracking is performed for this purpose, if needed; no backtracking is needed through determinate goals.

In this framework, the bagofs constructed as in section 3 will be executed in the determinate phase, provided that the conditions for independence succeed, giving all solutions for their variables, which will afterwards be traversed by the member/2 goals

in the nondeterminate phase, through backtracking. Two distinct matters should be discussed: how conditions will evaluate in this framework, and to which point collection for all solutions and traversal of them is worth doing.

Conditions for the annotated ites would evaluate in the same context as in Prolog if they are executed when leftmost. Otherwise, if their execution is performed before goals to its left, there is no guarantee that their evaluation will be as in Prolog. As we will see in next section, the algorithms for the transformation for IAP make several optimizations based on a global analysis of the program for the Prolog semantics; thus information on the state of instantiation of variables will be assumed as in a Prolog execution. If this is the case, the execution in Andorra-I should resemble that of Prolog (in fact, that of &-Prolog) and parallel expressions (whether conditional or not) executed only when leftmost. To achieve this, the parallel expressions should be sequentialized to their left[5].

Furthermore, as has been discussed elsewhere [GSCYH91, BH92], goals executing before the ites themselves could vary the state of instantiation of the variables involved in the conditions, therefore ites should also be sequentialized to their right. This will further restrict the parallelism available and can be avoided in Andorra-I if the determinate phase selects determinate goals in a left-to-right order. In this case, checks for ground/1 and indep/2 (in a conditional parallel expression) or the bagof/3 goals themselves (in unconditional expressions) would be selected before other determinate goals to their right. This is true for the body of the clause where the expression appears, but calls to the procedure of this clause do still need to be sequentialized.

On the other hand, the transformation using bagofs will collect all solutions and afterwards select the first of them by means of a member/2 goal to procceed with the execution. The rest of the solutions will be extracted if needed by backtracking. The efficiency gained with parallel execution of the bagofs can be reduced or even lost due to having to execute the goals for all solutions, if only the first is asked for. On the other hand, we regard that it will incur no penalty if the intended use of the program is to collect all solutions[6].

5 Algorithm for Automatic Translation

A general algorithm implementing the points mentioned in the preceding sections will do the following, where global analysis (possibly based on abstract interpretation) is optional:

1. Annotate goals to run in (independent-and) parallel, except builtins. Considerations on granularity apply. Optimizations on conditions also apply. Local and global analysis is used.

[5] For this purpose the sequentialization operator '::' of Andorra-I may be used.

[6] This supported by results obtained in [Ued87].

2. For each annotated goal the correspondent bagof is built. Optimizations for these apply. Local and global analysis is used again.

3. Consider the cut as a guard operator.

For an implementation of the translation algorithm, advantage can be taken of the tools already developed in the context of &-Prolog. Annotators for parallelism have been implemented that already supply the first item of the above. Also three global analyzers are implemented in the compile-time system of the &-Prolog, namely the "modes" analyzer [HWD92], the "sharing" analyzer [MH92] and the "sharing+freeness" analyzer [MH91]. Our objective has been, for simplicity, to implement the rest of the translation as part of this compile-time system, as can be seen in figure 1. This is specially relevant since the native global analyzer of the Andorra-I system [SCWY91b], itself based on some of the ideas of [HWD92, MH92, MH91], is also being incorporated as another analysis tool to the system presented in the figure.

The initial Prolog program is conveniently preprocessed for analysis based on abstract interpretation, then it is annotated for parallelism, and finally transformed into the output Andorra-I program. The back-end translator will deal with the bagof encapsulation, side-effects and cuts.

For the bagof encapsulation a local analysis for simplification of the bagofs, as mentioned in section 3, can be done. This analysis will look at the conditions of the ites to gather information on which variables are ground inside the ite. Nevertheless, the annotators themselves take advantage of information of the global analysis to simplify the conditions on the ites. Thus, information obtained by local analysis will not be complete. The solution to this is obviously to allow annotation plus translation with *only* local analysis, or both steps *plus* global analysis, both accessing the information of the latter. A raw algorithm for this step will:

1. For each clause,
   - locate an ite or CGE in its body,
   - perform 2 with the above construction,
   - sequentialize to its left and right (depending on the style of transformation desired - recall section 4),
   - iterate.

2. For each goal in the ite or CGE,
   - perform local analysis and access global analysis information: the set of ground variables for the goal at this point in the execution is made available,
   - perform 3 to build the bagof-member construction,

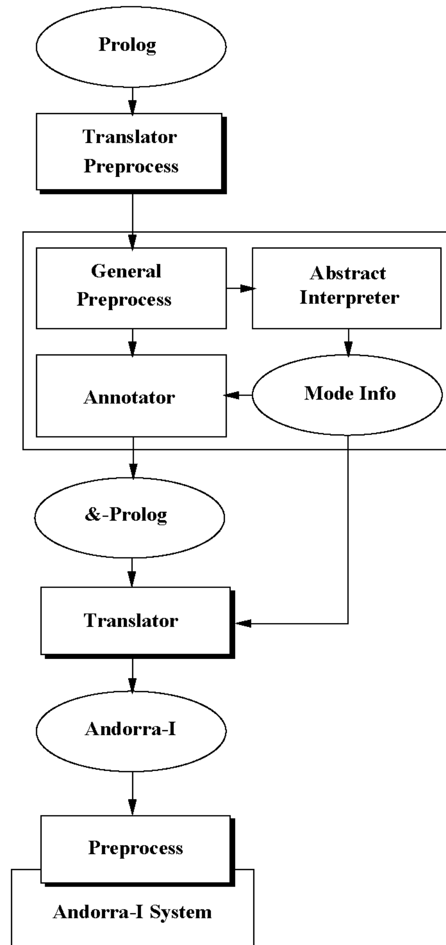

Figure 1: Overview of the full translator implementation.

- iterate.

3.
- obtain all variables for the goal,
- subtract to them those known to be ground,
- locate and subtract to the previous the anonymous variables, rename them apart in the goal,
- existentiate the renamed variables,
- create a tuple with the remaining variables,
- build the functors for bagof/3 and member/2

The algorithm just translates the &-operator construction of the ites or CGEs to the bagof-member construction that simulates the same behaviour. Additionally, the definitions for the new predicates ground/1, indep/2 and member/2, which appear in the translation process, must be included in the output program. In order for the output program to be runnable in Andorra-I, extra considerations must be done: influence of the cut in Andorra-I must be taken into account (much of this done by the Andorra-I preprocessor [SCWY91b]).

Andorra-I has inherited from the committed-choice languages the concept of *guard*. The body of a clause may be separated into the guard part of it and the body part, by a guard operator, which can be either one of the two pruning operators: cut (!) or commit (|). As commit is not included in standard Prolog programs, only the cut should be taken into account.

What has to be achieved is that only one guard operator is allowed in a clause, while in Prolog a clause body can have as many cuts as wanted. A simple way to overcome this is to fold the original clause, creating a new predicate defined by a single clause with a body composed by the goals from the original body which appeared after the first cut. Then this new clause can be recursively transformed in the same way.

6 Examples

This section is intended to demonstrate the behaviour of the implemented tool for the transformation algorithm presented in the preceding one. This will be done through the exposure of the translated output for a number of benchmark programs, showing the different issues of the translation.

We begin with a program presented by D.H.D. Warren [War77], which consists of two simple queries to a database made up of a number of (Prolog) facts, what makes the corresponding goals (pop/2 and area/2) highly nondeterminate, though independent. In its original formulation, the program was:

```
query([C1,D1,C2,D2]) :-
        density(C1,D1),
        density(C2,D2),
        D1 > D2,
        T1 is 20*D1,
        T2 is 21*D2,
        T1 < T2.

density(C,D):-
        pop(C,P),
        area(C,A),
        D is (P*100)/A.
```

From this version we reach the following one by a simple unfolding of the density/2

predicate definition, in order to avoid its original determinism. Thus we have:

```
query([C1,D1,C2,D2]) :-
        pop(C1,P1), area(C1,A1), D1 is (P1*100)/A1,
        pop(C2,P2), area(C2,A2), D2 is (P2*100)/A2,
        D1 > D2,
        T1 is 20*D1,
        T2 is 21*D2,
        T1 < T2.
```

Despite of the nondeterminism of the queries (now represented by the two pop/2 and area/2 conjunctions), these remain independent, and thus are eligible for and-parallel execution. This is captured by our compile-time tools, which give as output the following automatically translated version:

```
query([C1,D1,C2,D2]) :-
        bagof((C2,P2,A2),'query/1/1/$cge/1'(A2,P2,C2),L_bagof_1),
        bagof((C1,P1,A1),'query/1/1/$cge/2'(A1,P1,C1),L_bagof_2),
        member((C2,P2,A2),L_bagof_1),
        member((C1,P1,A1),L_bagof_2),
        D1 is P1*100/A1,
        D2 is P2*100/A2,
        D1>D2,
        T1 is 20*D1,
        T2 is 21*D2,
        T1<T2.

'query/1/1/$cge/1'(A2,P2,C2) :-
        pop(C2,P2),
        area(C2,A2).

'query/1/1/$cge/2'(A1,P1,C1) :-
        pop(C1,P1),
        area(C1,A1).

member(X,[X|_]).
member(X,[_|T]) :-
        member(X,T).
```

Note that the goals for the is/2 builtin have not been parallelized, though have been (safely) moved out to allow for and-parallelism of the other two. New predicates are created for the independent conjunctions (though not strictly necessary) and these encapsulated in bagofs.

For the next example we have the safe/1 predicate for solutions on the N-Queens problem, programmed in a manner such that negation is avoided:

```
safe([Q|Qs]):- safe(Qs), attacking(Q, Qs, A), A=no.
```

```
safe([]).

attacking(Q, Qs, yes):- attack(Q, Qs), !.
attacking(_, _,  no).
```

The recursive call is independent of the nondeterminate goal for attacking/3 in the body of the first clause for safe/1. In the standard execution of this program, the outer goal for this predicate is always ground, so now we have to encapsulate goals with ground variables, which gives us:

```
safe([Q|Qs]) :-
        bagof(Qs,safe(Qs),L_bagof_1),
        bagof(A,attacking(Q,Qs,A),L_bagof_2),
        member(A,L_bagof_2),
        A=no.
safe([]).
```

where the member/2 goal for the first bagof has been omitted, as it is unnecessary, as well as the ground variables in the second bagof. In fact the first bagof itself can be omitted, recalling that the safe/1 goal is determinate; this can not be done automatically unless a determinacy analysis is performed, similar to that of the Andorra-I preprocessor itself.

On the contrary, in the following example although both member/2 goals can be omitted, none of the corresponding bagofs can, as the encapsulated goals are in fact nondeterminate. This example consists of a common/3 predicate to check if a given list is a sublist of two other given lists, and thus its intended use is in queries with its three arguments ground:

```
common(L,L1,L2):-
        sublist(L,L1),
        sublist(L,L2).

sublist([],[]).
sublist([X|L],[X|L1]):- sublist(L,L1).
sublist(L,[_|L1]):- sublist(L,L1).
```

Being all variables in the subgoals ground, the translator selects any of them to put in the bagof (alternatively, a new variable will serve the same purpose). Thus the output program looks like:

```
common(L,L1,L2) :-
        bagof(L1,sublist(L,L1),L_bagof_1),
        bagof(L2,sublist(L,L2),L_bagof_2).
```

7 Conclusions

We have presented a transformation that fulfills our original objectives and which can be easily automated. We have shown that it is possible to introduce general independent and-parallelism in a system like Andorra-I, already supporting determinate dependent and-parallelism and or-parallelism, and that this can be done without modification of the implementation of the underlying execution model by means of program transformation. This makes the extension of the model much simpler an easier.

We also show that this can be achieved automatically at compile-time. An algorithm and its implementation have been presented, and some examples on its behaviour shown. Future work will include interfacing it with the Andorra-I system and bagof implementation, and possibly its integration with the system.

References

[BH92] F. Bueno and M. Hermenegildo. An Automatic Translation Scheme from Prolog to the Andorra Kernel Language. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 759–769. Institute for New Generation Computer Technology (ICOT), June 1992.

[CDO88] M. Carlsson, K. Danhof, and R. Overbeek. A Simplified Approach to the Implementation of And-Parallelism in an Or-Parallel Environment. In *Fifth International Conference and Symposium on Logic Programming*, pages 1565–1577. University of Washington, MIT Press, August 1988.

[GSCYH91] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: A Model Integrating Dependent-, Independent-, and Or-parallelism. In *1991 International Logic Programming Symposium*. MIT Press, October 1991.

[HG90] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[HJ90] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *Proceedings of the Seventh International Conference on Logic Programming*. MIT Press, June 1990.

[HR89] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[HR90] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[HWD92] M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 3(4):349–367, August 1992.

[MH90] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[SCWY90] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, April 1990.

[SCWY91a] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 825–839. MIT Press, June 1991.

[SCWY91b] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

[Ued87] K. Ueda. Making Exhaustive Search Programs Deterministic. *New Generation Computing*, 5(1):29–44, 1987.

[War77] D. H. D. Warren. *Applied Logic—Its Use and Implementation as Programming Tool*. PhD thesis, University of Edinburgh, 1977. Also available as SRI Technical Note 290.

[War88] D. H. D. Warren. The Andorra Model. Presented at Gigalips Project workshop. U. of Manchester, March 1988.

[War90] D. H. D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.