EFFICIENT OR-PARALLEL INTERPRETATION OF LOGIC PROGRAMS HANDLING MULTI-BINDINGS.

Gonzalo Escalada-Imaz
Artificial Intelligence Research Institut (IIIA)
Spanish Scientfic Research Council (CSIC)
Campus UAB, s/n. 08193 Belaterra
Barcelona, Spain
gonzalo@iiia.csic.es

Ana M. Matínez-Enríquez
Advanved Studies Center (CINVESTAV)
National Technique Institut (IPN)
Av. IPN N2508, 07000 Mexico D.F.
Mexico
AMMARTIN@mvax1.red.cinvestav.mx

SUMMARY
We propose a new interpreter model for exploiting Or-parallelism in Logic Programming. Or-parallelism is applied to solve a goal-literal and it consists in executing in parallel one process for each clause whose head literal unifies with the goal-literal. Most of the existing interpreters exploiting Or-parallelism fall down in the combinatorial explosion of the huge search space which models the interpretation problem of Logic Programs with a strong no-determinist nature. This is one of the cases when a Logic Program as Datalog is employed to enquire Deductive Data Bases. In fact, the subsequent Or-branching of the mentioned search space swiftly increase the time and the number of parallel processes needed to solve a given problem. Thus, our aim is to propose a new approach for carrying out the Or-parallelism in an efficient way avoiding, as much as possible, the impractical process of considering one by one the different paths appearing in the enormous search space. For this purpose, the state interpret does not rely in the classical representation of known interpreters (Set-of-goals, Simple-binding-variables). A state of the interpreter represents a set of similar states (to be defined) and has the form (Set-of-literals, Multiple-binding-variables). The multi-bindings represent efficiently common simple bindings existing in different paths of the search space problem. To improve the interpretation time two operations on multibindings are defined: the merging and simplification of multibindings. The proposed method can achieve in a best case, an exponential gain in the number of processors, required space memory and execution time. In deterministic problems, the proposed interpreter has a minimun overhead with respect to chronological backtracking. Other good characteristics of the method are discussed throughout the article.

1.-INTRODUCTION

The declarative characteristic of Logic Programming that separates representation and control makes it suitable to design parallel interpreters. One important goal in specifying such a parallel interpreter is reducing the solving time of a given problem with a reasonable number of processors. Numerous methods have been published and important projects are being developed to deal with this objective.

Two kinds of parallelism arise when interpreting a Logic program with a parallel architecture: Or-parallelism and And-Parallelism.

- The former searches in parallel for the solution of one goal-literal by executing one process for each clause whose head unifies with the goal-literal. In performed classical interpreters, a process is associated with one different path of the search space of the problem. But, that technique obviously does not avoid the exponential increasing of the open processes because is clear that the number of search paths of a exponential search space is equally exponential.

- The And-parallelism is applied to solve the clause body activating parallel processes for the literals in the body of a clause. To take profit of this kind of parallelism with low on line execution overhead is a more difficult problem with respect to the Or-parallelism.

Different approaches have been proposed in the literature to decrease the execution time by taking advantage of a parallel architecture.

1) Or-parallelism [Ha-90, Sz-89, Wa-87a, Wa-87b] computes in parallel the clauses that unify with the chosen literal-goal.

2) Restricted And-parallelism [Cd-85,Cl-88,Co-83,De-84,Fa-87,Lm-86,St-85,He-86, He90,He-92]. Roughly, the body literals in a clause are decomposed in independent sets (non-sharing variables). The execution of literals in a same set are serialised. The solutions of one literal for each set are carried out in parallel.

3) Full And-parallelism [Kal-87, Wi-86a, Wi-86b]. Some methods solve in parallel all the body literals and perform additional unification operations for checking compatibility between instantations of the shared variables.

4) Parallel first order unification (fine granularity): Some parallel first order unification algorithms have been proposed [Ja-87,Ya-84]. This problem have been proved no to be in NC [Dw-84], only matcher algorithms [Dw-88] can be solved in parallel poly-logarithmic time. This is an important results because matcher, instead of first order unification algorithms are enough for enquering, e.g. with a Datalog language, Deductive Data Bases.

The published methods propose to exploit one of the three kinds of parallelism Or, Restricted-And and Full-And, or making a combination of Or-parallelism with one of the possible And-parallelisms.

Our aim is to propose a new approach for carrying out the Or-parallelism in an efficient way avoiding, as much as possible, the impractical process of considering one by one the different paths appearing in the enormous search space. For this purpose, the state interpreter does not rely in the classical representation of known interpreters (Set-of-goals, Simple-binding-variables). Rather than, a state of the interpreter represents a set of similar states (to be defined) and has the form (Set-of-literals, Multiple-binding-variables). The multi-bindings represent efficiently common simple bindings existing in different paths of the search space problem. To improve the interpretation time two operation on Multi-bindings are defined: the merging and simplification of multi-bindings. The proposed method can achieve in a favourable non-determinist search space, an exponential gain in the number of processors, required space memory and execution time. In deterministic problems, the proposed interpreter has a very low overhead with respect to chronological backtracking.

The basics of the proposed approach for a sequential interpreter have been first presented and formalised, in order to prove any kind of pertinent properties, in [Esc-89, Es-90a, Es-90b, Es-91]. Some previous studies and weaker results of the Or-parallel interpreter presented here are given in [Es-93b, Es-95]. Although the interpreter presented here focuses only on processing Or-parallelism, it could be also combined with independent And-parallelism [Es-92a, Es-92b, Es-92c] and restricted And-parallelism [Es-93a, Es-94a, Es-94b]. The previous mentioned works contain many analytical results, conversely to the most known sequential and parallel approaches in which generally are missing or very few properties are proved. However, no empirical results of the proposed interpreter here have been obtained yet comparing the proposed approach with other existing ones which indeed it seems a standard way of pondering and classifying the efficiency of a particular interpreter.

But lastly, some recent interpreters, called Multi-path in [Tu-95, Tu-94] and Multi-log in [Sm-94] are emerged relying on the same basic idea of combining depth-first and breadth-first search, given rise in our case to the handling of Multi-binding. These three approaches have close underlying principles. Although, neither Multi-path nor Multi-log approaches are not rigurously analysed, they are justified by well founded practical conjectures and also, they show the gain in efficiency of their approaches with respect to the more classical depth first search with any variant of Intelligent Backtracking. Then, a rational "symbiosis" of the analytical results determined for the Multi-binding proposal and the advantageous empirical results found for Multi-path and Multi-log methods allows us to expect significant improvements in the design of sequential and parallel interpreters following this research line which is based on convenient combining of both strategies Depth-first and Bread-first. As it has been mentioned, the conclusion obtained by such a "symbiosis" must be rationally bound by the fact that the three models share the idea of using Depth-first and Bread-first search techniques, but they are not identical. For example, recent proposals are also based on Simple-binding state representation of the interpreter.

A big drawback of the previous methods exploiting Or-parallelism is that they fall down easily in the combinatorial explosion inherent to the solution of a Logic Programming problem. Indeed, the successive decomposition of one process in several processes for each Or-point choice could induce an exponential number of processes. For instance, let us assume that we have to solve a set of goals {P1, P2,..., Pn,..}. If we have 2 independent solving paths for each goal the number of processes created will successively grow up, i.e. 2, 4, 8, ...,2^n,... In most proposed parallel interpreters, a process is in charge of solving the inferences forming a path in the search space beginning from the root or an interior node and ending in a leaf. Hence, in these classical interpreters the exponential increasing of processors is also exponential. In this context, a particular processor in a parallel architecture is constrained to work with many processes which causes huge overheads by change contexts. As a direct consequence of having a limited number of available processors (fixed for a certain machine) and a fast increasing in the number of required processes, these ones <u>have to be serialised</u>. All these aspects can even almost eliminate the gain in efficiency got by parallelism.

The proposed method presents some good characteristics: A) It enables us to master the above mentioned combinatorial explosion in fairly many problems. The number of processes could be exponentially smaller than that needed by existing interpreters (using Or-parallelism). B) Regarding the total running time, some good features appear in the our proposal. Indeed, the limitation in the required number of processes reduces the communication costs and also, it <u>avoids to serialise</u> the processes. Furthermore, the handling of multi-bindings benefits better of the techniques of sharing contexts than with simple binding classically used. C) The reduction in the required memory space in the proposed interpreter (even exponential in a best case) follows from the fact that a multi-binding is equivalent to a set of multiple bindings but represented in a compact and hierarchical way.

The proposed method has also other complementary aspects which we are going to treat throughout the paper. For example, the problem of computing multiple bindings is efficiently solved with the multi-binding representation where the operations on these objects (that will be defined) avoid an exhaustive Cartesian product explicitly executed in known methods to determine the whole intentional solutions of a goal-literal. Also, the operations on multi-binding may employ fine grain granularity parallelism. Another good point is that all the solutions of a Logic Program problem are determined simultaneously.

The next section describes the method in an informal way. The third one defines formally the multi-bindings and the required operations on these objects. After this, the general schema of the proposed parallel model is more precisely described and it is illustrated with examples. The fifth section, discusses some implementation aspects. In section six, some analysis showing the applicability and efficiency of the approach are given. Next section briefly proposes a scheduler (The main proofs concerning the assertions made throughout the paper can be found in [Es-89,Es-90]).

2.- INFORMAL PRESENTATION OF PARALLEL INTERPRETERS

In this section, we give a rough description of the proposed interpreter. In an attempt to be objective, we have taken an example from the bibliography [Lk-88] and by studying this example, we explain in an informal way the previous methods and the proposed one. This example also serves to point out the good features of the proposed method.

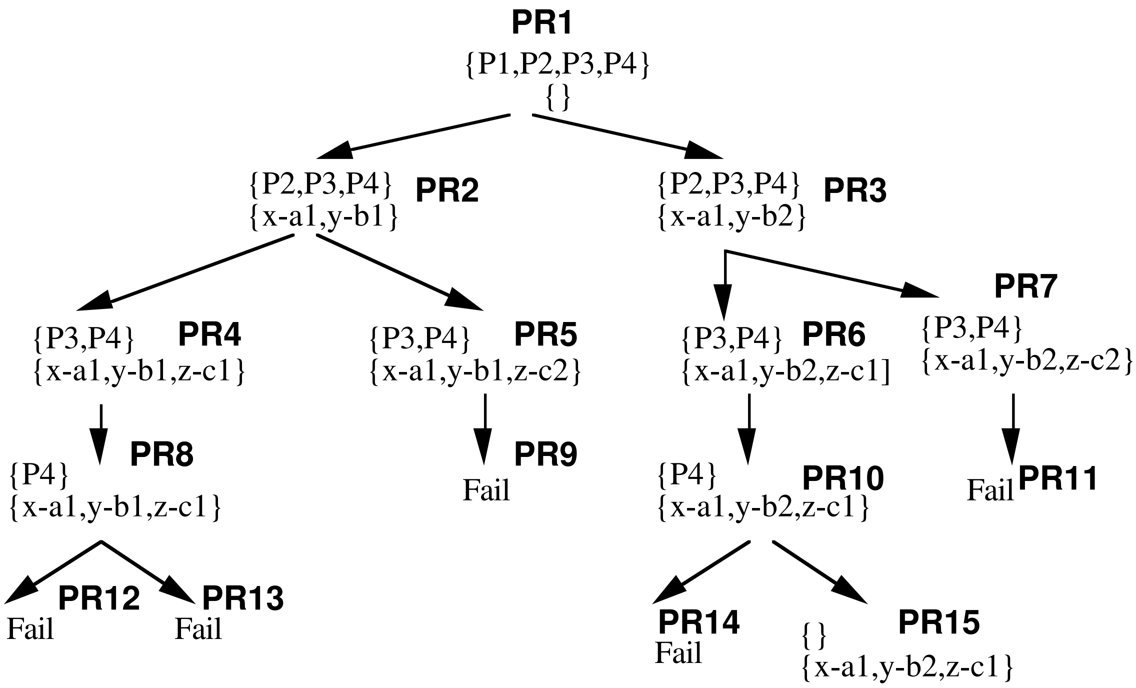
Let us look at the following problem given in [Lk-88].

1) ← P1(X,Y),P2(X,Z),P3(Z),P5(Y,Z)

2) P1(a1,b1)←	4) P2(a1,c1)←	6) P3(c1)←	7) P4(b1,c2)←
3) P1(a1,b2)←	5) P2(a1,c2)←		8) P4(b2,c1)←

Remark: As the number of active processes is proportional to the number of paths in the search space problem, the total number of open processes is exponential in no-determinist Logic Programming problems, in both cases opening a process for each node and assigning a process for each path. Hence, we simplify the description of the classical and proposed approach considering that each node is solved by a process.

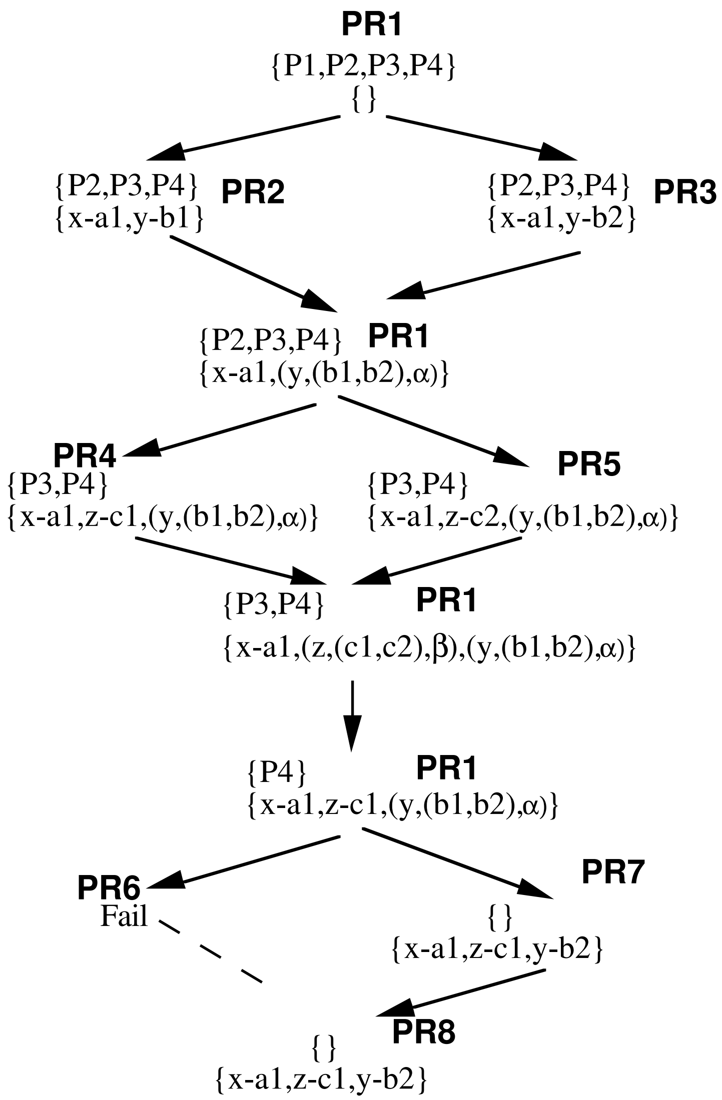
Standard Or-parallel interpreters (see figure below): The first goal is the conjunction {P1,P2,P3,P4}. A standard Or-parallel interpreter begins with trying to solve P1. As there exist two possible unifying head clauses (2 and 3), the process associated with the solution of P1 is converted into two new processes. Now, each one of these processes has in turn two new possible continuations because P2 could be unified with clauses 4 and 5, taking into account their actual binding variables environments, and so on. The drawing shows the different processes activated versus the time. As can be seen, in this case the number of processes open at the same time is four {PR4,PR5,PR6,PR7} (same number by associating a process to each path in the search tree).



To pass from one state to the next, there exist several ways. The most efficient and most commonly accepted is as follows (let us consider the solving of P3 in process PR6):

- The literal-goal is confronted with a head clause having the same predicate symbol. To solve P3, only clause 6 could be applied and the unification bound z to c1: {z-c1}.
- Afterwards, {z-a1} (in general is a set of binding variables) is unified with the binding environment, here, {x-a1, y-b1, z-c1}. The successful unification leads to a new process PR10 with the same bindings {x-a1, y-b1, z-c1} (in this particular case).

Multibindings interpreter MSI: As the standard methods, the MSI begins by trying to find the solutions of P1. As there exist two candidates clauses 2 and 3, two processes PR1 and PR3 are carried out in parallel. The process PR1 does not die after creation of the son processes. When both have ended, they return the sets of bindings and then the process PR1 makes an operation of merging them to get a multibinding represented by
{x-a1,(y,(b1,b2),α)}.
Now, the interpreter goes on by solving the next literal P2. Then, the process performs an operation, that we call simplification, between the obtained multibinding and each one of the bindings coming from each of the two possible unifications {x-a1,z-c1} and {x-a1,z-c2} with clauses 3 and 4. For each of these successful simplification operations (both in this case), one process is open PR4 and PR5. Thus, when these processes have finished and given back their obtained solutions in the form of multibindings, the top process merges these multibinding in an equivalent one and the steps are continued so on with literal P3.



Brief comparative discussion: although the definition of multibinding and the operations defined on them (merging and simplification) have not been defined yet, some observations can be made.

- The number of processes open in each moment (at most 2) is smaller with MSI and can be exponentially smaller than with previous interpreters.
- Concerning the space memory is clear that the multibinding represents an important gain (as we will see, it could be exponential in a better case) because several bindings could be merged in only one.
- The point open for discussion concerns total running time and it will be discussed in the section of analysis and comparisons. To study this point we will envisage several factors as the number of processes running concurrently, the probability of having to serialise processes, the longest resolution path to solve a goal literal, the necessity of obtaining one or all solutions, etc.

<u>**Notation**</u>. In the following of the article, aiming to be slightly more formal, we note the binding x-t by x≈t and we call it equation; a set of equations will be called a system; and finally a multibinding will be denominated multisystem.

3.- MULTISYSTEMS (Definitions)

The following definitions give a formal description of multisystems and the two operations used on these structures:
1) the merging of multisystems and,
2) the simplification of a multisystem together with a system.

System of equations: A couple of first order terms t and t´ is an equation and is noted t≈t´. A system of equations is a set of equations {t1≈t2, t2≈t2´, .., tn≈tn´}.

A solution of a system S: is a substitution {x1/t1,..,xn/tn} such that substituting simultaneously each variable xi in S for its corresponding term ti makes both terms equal in the equations.

Example: {x/a, y/f(a,z)} is a solution of {f(a,b)≈f(x,b), y≈f(x,z)}. There are systems without solutions, e.g.{a≈b}.

A Simplified System: is of type S={x1≈t1,..,xn≈tn}, the xi´s being distinct variables. **Its free variables**: are the variables in {t1,.,tn} not included in {x1,.,xn}

Remark. In the literature, a simplified system is often called a set of bindings or also, a binding environment of variables.

Equivalent Systems: Two systems are equivalent iff they have the same solutions.

Example: S1={x≈g(u),x≈g(v)} and S2={x≈g(u),u≈v} are equivalent systems.

Proposition. A non-siimplified system {t1≈t1´, t2≈t2´, ..,tn≈tn´} with at least one solution can be simplified in another equivalent simplified system {x1≈t1,...,xn≈tn} (in general, this simplification is not unique).

Definition. We note SIMPL(S) the operation (function) that simplifies a system S obtaining another simplified system S´ equivalent to S. SIMPL(S)=Fail iff S has no solutions.

Complexity of Simplifying Systems. The simplification operation of S corresponds to the obtaining of non-idempotent unifiers for S, hence it has a lineal [Pa-78].

Example. If S={f(a,b)≈f(x,b), y≈f(x,z)} then SIMPL(S)=S´={x≈a, y≈f(x,z)} is a simplified system of S.

The proposed parallel interpretation model is based on processing and propagating multisystems. These structures consist in compact representations of several systems (where common equations to several systems are represented only once).

Multiequation. It is a triplet $(x,(t1,..,tn),\alpha)$ and represents, or is equivalent to a set of equations $x1 \approx t1, x \approx t2.., x \approx tn$; α is a selector such that when it is instantiated to a value $k \in \{1,..,n\}$ the multiequation is equivalent to the equation $x \approx tk$.

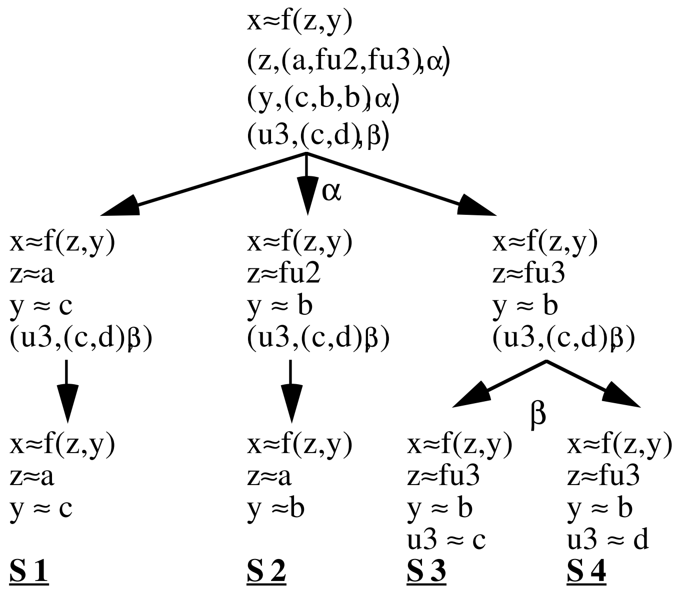
Multisystems: A multisystem $MS=Q \cup MQ$ is formed by a simplified system Q and a set of multiequations MQ, some of them linked by the same selector.

Simplified Multisystems. There are multisystems such that each first variable in $Q \cup MQ$ and in MQ do not appear more than once as first variables in $Q \cup MQ$.

Example: $MS=\{x \approx f(z,y),(z,(a,fu2,fu3),\alpha),(y,(c,b,b),\alpha),(u3,(c,d),\beta)\}$ is a multisystem with $Q=\{x \approx f(z,y)\}$ and $MQ=\{(z,(a,fu2,fu3),\alpha),(y,(c,b,b),\alpha),(u3,(c,d),\beta)\}$. Also, it is simplified multisystem because $\{x, z, y, u3\}$ appears only once as first variables in MS.

In the following, we will deal with only simplified systems and multisystems.

Graphical version of equivalent Set of Systems to a Multisystem: A multisystem MS is equivalent to a set of simplified systems $\Sigma(MS)$. Obtaining such systems could be done by mean of a tree whose nodes are multisystems and are defined progressively by the instantiations of the selectors. The leaves correspond to the equivalent systems to the multisystem. For instance, the tree and the systems of the previous multisystem in the last example are:



The selectors α that could be expanded in a node $Q \cup MQ$, must verify the conditions that:
1) A multiequation $(\mathbf{x},(..),\alpha)$ with α and variable **x** is in MQ and 2) **x** is free in Q
For example α is the only selector that could be expanded in the root node of the figure above because variables z and y fulfil the two cited conditions.

If α is the extended selector in a node, the i-th successor is defined by removing the multiequations with selector α and adding the system of equations obtained by instantiating α to i. If no selector can be expanded in a node, then its unique successor is defined by the system of equations in the node.

For the previous multisystem MS, the equivalent set of systems is {S1,S2, S3, S4}, as show in the figure above.

Recursive Definition of Equivalent Set of Systems to a Multisystem. A multisystem MS is equivalent to a set of systems $\Sigma(MS)$, defined by:

1) $\Sigma(\{\})=\{\{\}\}$

2) $\Sigma(S)=\{S\}$

3) $\Sigma(MS)=\Sigma(Q\cup MQ)=\cup_{\alpha=1,..,n}\Sigma(Q\cup Q_{\alpha i}\cup MQ^{\alpha})$, where

- α is a selector of at least one multiequation having arity n whose first variable x is also a free variable in Q;

- $Q_{\alpha i}$ is the simplified system obtained from multiequations in MQ linked by α when the selector is instantiated to i; and

-MQ^{α} is the set of multiequations in MQ not linked by selector α.

Remark: Intuitevely the multisystems are designed to capture as much as possible the shared equations of the systems represented by the multisystem.

Example: It is easy to see that multisystems are a compact and hierarchical way of representing a set of simple systems. That means that a set of systems can be represented with multisystems with significantly less space memory, and in a best case with exponentially less space memory. Let us look at the following multisystem:

(x1, (a11,....,a1n), α1)

(x2, (a21,....,a2n), α2)

....

....

(xn, (ak1,....,akn), αk)

Developing its corresponding tree, it is easy to check that this multisystem is equivalent to n^k systems, each one having k equations. However, the number of terms of the multisystem is only n.k terms.

In the parallel interpreter MSI model:
The And-processes need to simplify multisystems and the Or-parallel processes to merge multisystems in only one preserving the total set of systems represented by the merged ones. These two operations are defined below. Σ

MERGING OF MULTISYSTEMS

Assuming that we have a set of multisystems MS1,...,MSn with their respective equivalent set of systems, i.e.

$$\Sigma(MS1)=\{S11,..,S1k(1)\},.....,\Sigma(MSn)=\{Sn1,..,Snk(n)\},$$

the merging of these multisystems, noted

MG(MS1 U MS2 U ..UMSn)

consists in the obtaining of a new multisystem $\Sigma(MS)$ verifying

$$\Sigma(MS)=\{S11,..,S1k,S21,..,Sn1,..,Snk(n)\}.$$

As this operation is quite complex to present here, but not to implement, we will illustrate only a simple case considering that MS1,...,MSn are simple systems S1,...,Sn. The whole description of the operation with multisystems can be found in [Es-89,Es-90].

Merging of systems: The merging of a set of simplified systems {S1,...,Si,...,Sn} in a simplified multisystem verifying MS=Q U MQ , $\Sigma(MS)=\{S1,..,Sn\}$ is done as follows:
1) Q are the common equations to all the systems Si.
2) A multiequation (x,(t1,.,ti,.,tj,.,tn),α) is in MQ for each variable x such that x is linked to a term ti in Si with tj=x if there is not an equation x≈tj in Sj. These multiequations shared the same selector.

Example: Let us consider the three simplified systems given below. The figure illustrates the merging of the three systems:
1) The unique common equation to the three systems is $x \approx f(z,y)$, hence $Q=\{x \approx f(z,y)\}$.
2) The equations with variables z, y and u3 verify the point 2 of the Merging operation and therefore are represented in three multiequations $(z,(z,fu2,fu3),\alpha)$, $(y,(c,b,b),\alpha)$ and $(u3,(u3,u3,c), \alpha)$ linked by a same selector α.

$x \approx f(z,y)$ $y \approx c$	$x \approx f(z,y)$ $z \approx fu2$ $y \approx b$	$x \approx f(z,y)$ $z \approx fu3$ $y \approx b$ $u3 \approx c$

$x \approx f(z,y)$
$(z,(z,fu2,fu3),\alpha)$
$(y,(c,b,b),\alpha)$
$(u3, (u3, u3, c), \alpha)$

Remark. In fact this representation can be compacted avoiding the occurrences verifying the situation of u3 [Es-89, Es-90].

Theorem: If $MS=MG(S1,..,Sn)$ then $\Sigma(MS) =\{S1,..,Sn\}$.

Proof: The equivalence between a set of systems and the multisystem obtained by Merging is defined more precisely and its correctness proved in [Es-89,Es-90].

SIMPLIFICATION OF MULTISYSTEM MS

The objective is to simplify a non-simplified multisystem: $SIMPL(S \cup MS')$. The proposed Or-interpreter needs to handle non-simplified multi-systems when the following condition is not fulfilled by MS: There could be a set of common first variables to Q and MQ. This is equivalent to calculate a simplified multisystem MS verifying:

$$\Sigma(MS)=\{SIMPL(S \cup S') \neq Fail: S' \in \Sigma(MS')\}.$$

(As has been mentioned, $SIMPL(S \cup S')=Fail$ only if $S \cup S'$ has no solutions). In other words, the simplification of S and MS´ consists in the obtaining of a simplified multisystem MS equivalent to the set of systems obtained by simplification of S with each system S´ included in the equivalent set of systems to MS´ (and the removing of systems $S \cup S'$ with no solution). This relationship is noted by:

$$\Sigma(MS)=\Sigma(SIMPL(S \cup MS'))=\{SIMPL(S \cup S') \neq Fail: S' \in \Sigma(MS')\}.$$

A <u>simple algorithm</u> to get $SIMPL(S \cup MS')$ consists in developing the whole tree for MS´, determining $\Sigma(MS')$, computing exhaustively $\{SIMPL(S \cup S') \neq Fail: S' \in \Sigma(MS')\}$ and finally merging these systems $\Sigma(MS')$. But, as it can be seen, this operation is proportional to the number of systems represented by the multisystem. Instead, the proposed algorithm significantly improves the previous "brute force" algorithm, by taking advantage of the compact and hierarchical representation of multisystems:

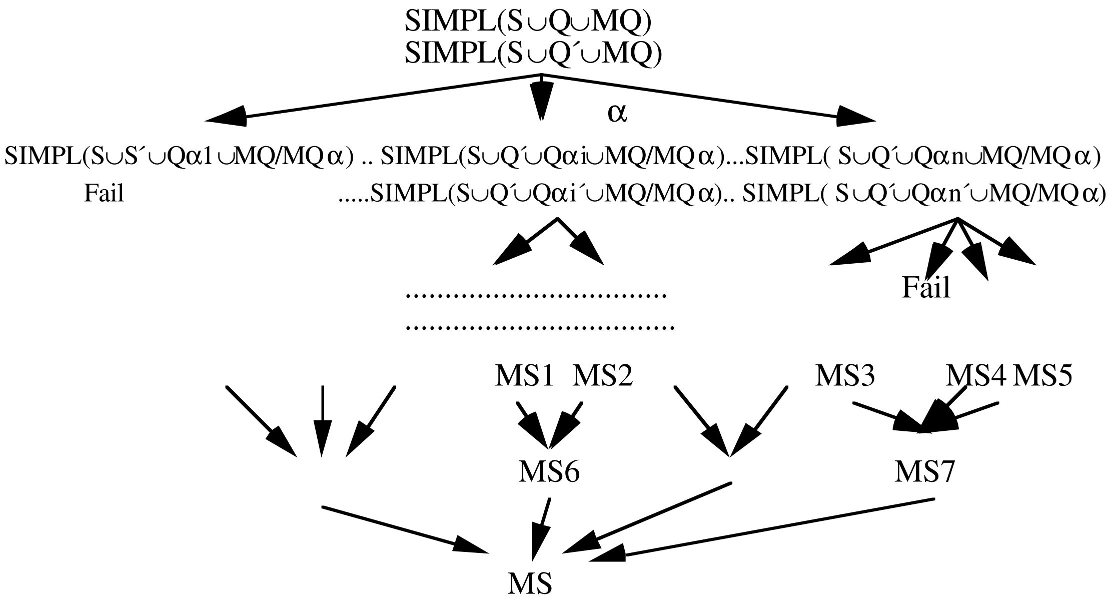
Proposition: If S1 is a simplified system then there exists a simplification of S1 U S2 such that: $SIMPL(S1 \cup S2)=S1 \cup S2'$ (if $S1 \cup S2$ has any solution).

That is, the simplification of a simplified system S1 with another one S2 can be got conserving the simplified one S1.

Proposition. Let $MS=Q \cup MQ$. Then $SIMPL(S \cup MS)= SIMPL(S \cup Q \cup MQ)=$

SIMPL(SIMPL(S ∪ Q) ∪ MQ)= SIMPL(S' ∪ MQ).

Graphical representation of the simplification of a multisystem MS: is defined by the following steps diagram (**MQ**α denote the multiequations linked by the same selector α and **Q**α**i** the set equations formed from MQα by the instantiation of selector α to i):



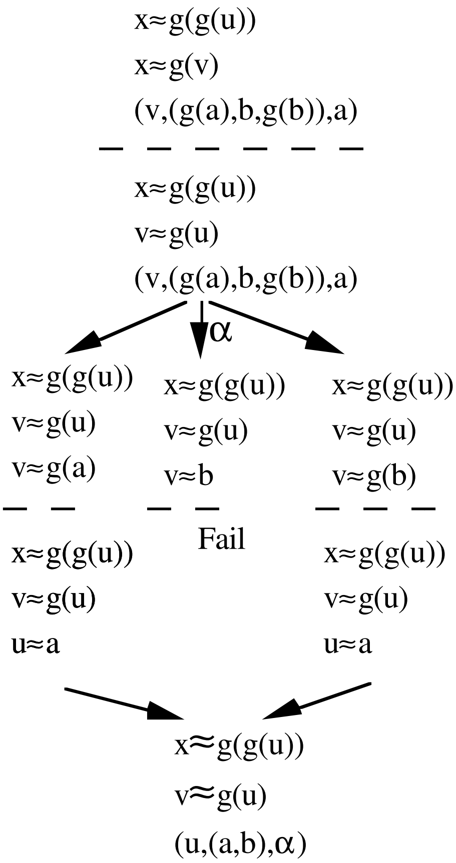
First, the system S∪Q is simplified in a system S∪Q´. This operation leads to a multisystem (S∪Q´∪MQ) which is unlikely in a simplified form: there could exist equations x≈t in S∪Q´ and multiequations (x,(..),α) with the same variable x. The involved selectors α in such situation are expanded one by one. The i-th node successor is defined by: 1) adding the system Qαi and 2) removing the multiequations MQα.
This process of expansion selector and simplification of systems goes on for each developed node until one of the two cases applies:
- The simplification of the system fails,
- After simplification of the system, we obtain a simplified multisystem.
As the process of simplification ends by finding nodes in one of these cases, the obtained multisystems are merged according to the order of previous expansion of selectors. Note that some multisystems can be "Fail" and hence, in the merging phase they will not be considered.

Recursive Definition of the simplification of Multisystems: The algorithmic simplification of a non-simplified multisystem MS=Q∪MQ is defined as follows:
1)SIMPL(S ∪ { })={S}
2)SIMPL(Fail ∪ MS)=Fail
3)SIMPL(S ∪ MS)= S ∪ MS, if S∪MS is simplified,
4)SIMPL(S ∪ MS) = $\cup_{\alpha=1,..,n}$SIMPL(S∪$Q_{\alpha i}$∪MQ^{α}), if S∪MS is a non-simplified multisytem and where α, $Q_{\alpha i}$ and MQ^{α} are as described in definition of Σ(MS).

Theorem: If MS=SIMPL(S U MS´) then Σ(MS) ={SIMPL(S∪S´)≠Fail: S´∈Σ(MS´)}.
Proof: In [Es-89,Es-90].

In a general case, SIMPL(S U MS)={MS1,...,MSk}, where MSi are simpler and smaller than MS. The proposed schema applies immediately MG(MS1,...,MSk)=MS' in order to take advantage of further SIMPL operations because, as it is showed in section 5, MG operations are executed with low computational cost and however SIMPL(S' U MS') operations can significantly decrease the computational cost with respect to a more exhaustive algorithm obtaining $\Sigma(MS) = \{SIMPL(S \cup S') \neq Fail: S' \in \Sigma(MS')\}$.

Example: Let us assume that we have to simplify SIMPL(S U MS), with S={x≈g(u)} and MS={x≈g(g(v)),(v,(g(a),b,g(b)),α)}. The steps are as indicated below:



First, the global set of equations is simplified:
SIMPL(S U Q)=SIMPL({x≈g(g(u)),x≈g(v)}) ={x≈g(g(u)),v≈g(u)}=S U Q´. Now, the simplified multisystem is not expressed in a simplified form because the equation v≈g(u) and the multiequation (v,(g(a),b,g(b)),α) share the same variable v. Therefore, in order to simplify this multisystem, we have to extend the selector α and then continue the simplification process with systems. The two multisystems got (with α=2 the simplification fails) are in a simplified form and so no further simplification is necessary. Finally, the two multisystems are merged into one.

The operations in detail are the following:

Example: Assume we require to calculate SIMPL(S U MS), with S={x≈g(g(u))} and MS={x≈g(v)} ∪ {(v,(g(a),b,g(b)),α)}. The successive steps are as indicated below:

SIMPL(S∪ Q∪ MQ)=

SIMPL(SIMPL(S ∪ Q)∪ MQ)= SIMPL(SIMPL({x≈g(g(u)),x≈g(v)}) ∪ MQ}=

SIMPL({x≈g(g(u)),v≈g(u)}∪ MQ)=

SIMPL(S ∪ Q´∪ MQ)=

SIMPL(S´ ∪ MQ)=

SIMPL({x≈g(g(u)),v≈g(u)} ∪ {(v,(g(a),b,g(b)),α)})=

$\cup_{\alpha=1,2,3}$SIMPL($S' \cup Q_{\alpha i} \cup MQ^{\alpha}$)= ={

SIMPL({x≈g(g(u)),v≈g(u), v≈g(a)}),

SIMPL({x≈g(g(u)),v≈g(u), v≈b}),

SIMPL({x≈g(g(u)),v≈g(u), v≈g(b)}) }=

={{x≈g(g(u)),v≈g(u), u≈a}, "Fail", {x≈g(g(u)),v≈g(u), u≈b}}=

{MS1, MS3}={S1, S3}

Finally, the proposed schema applies

MG(S1, S3)={x≈g(g(u)),v≈g(u), (u,(a,b),α)=MS'

Therefore, SIMPL(S U MS)=MS'.

4.-THE PARALLEL INTERPRETER

We now describe the essential aspects of the parallel interpreter based on a Von Neuman architecture.

As previously mentioned, the literals in a selected clause body are resolved sequentially, whereas successor processes of an Or-node are performed in parallel. There are three kinds of processes: And, Or and SIMPL. Their different activities are succinctly described.

AND-Process: It is associated with the global solution of a selected clause HC<-L1,..,Ln. The And-process receives a multisystem MS and a goal literal GL from its Or-process father such that HC and GL have the same predicate symbol. When an And-process is open, it executes the following operations:

It checks if clause HC<-L1,..,Ln applies to reduce the goal literal GL doing two operations. First, it simplifies SIMPL({GL≈HC})=S and second, SIMPL(S U MS), where MS is the received system. If SIMPL({GL≈HC}) or SIMPL(S U MS) fail, then the And-process stops after sending back the Fail signal to its father, otherwise it goes on. At this stage, SIMPL(S∪MS)=MS1≠Fail, and hence the clause can be applied for sequential solving beginning by its first body literal L1.

OR-Process: An Or-process is activated to solve L1 which activates concurrently one And-process for each clause having the same predicate symbol that L1 and sends them MS1. After terminating, each i-th successor And-process returns a Multisystem MS1i, or the ´Fail´ signal if there is no solution, using the corresponding clause.

Once all the successor multisystems MS1i are available, the Or-process merges them in a unique multisystem MS2 that is sent back to the father And-process. If all the received signals were Fail then the Or-process would return the "Fail" signal too.

And-process: The previous steps are successively repeated for the other literals L2,...,Ln in the body clause. Thus, the multisystem MS2 is sent to other Or-processes open to solve the second body literal L2. The steps go on until either one literal Lj, j≤n, has no solution, or a multisystem MSn is obtained for Ln. In the former, the And-process returns a Fail signal, in the latter, it sends the multisystem MSn to its Or-father as the outcome of the computation of its associated clause HC<-L1,..,Ln.

SIMPL-processors. The calculus of SIMPL(S∪MS) could be improved significantly by using parallelism with a reasonably small number of processors. The principle is: each time a selector is extended, a SIMPL-processor is activated for each possible value of the selector. These processors do sequentially:

1) To calculate the corresponding SIMPL operation ; 2) To activate the successor SIMPL-processors when the expansion of a new selector is required; 3) To wait for the replies of

these successors and to merge the returned multisystems, and finally, 4) to send back this obtained multisystem.

Example: Let us suppose that the simplification of the last example in the previous section has to be done in a step SIMPL(S∪MS) of the interpreter. In this case four SIMPL-processes could be open. The first one calculates the simplification of the root node in the illustrate tree and opens three new SIMPL processes for the computation of nodes corresponding to the three instantations of selector α. When these three parallel SIMPL processes have ended, the root SIMPL-process merges the two resulting multisystems in the final one, leaf of the drawn graph.

Example: To illustrate the proposed interpreter, let us see how it runs on the logic program:.

	Co: <-Px,Qy,Rx	
C1: Pa <-	C5: Qu <- Mu,Nv	C9: Rd <-
C2: Pz <-Sz	C6: Ma <-	C10: Ra <-
C3: Sb <-	C7: Na <-	
C4: Sc<-	C8: Qc <-	

Initially, a root And-process is created to solve the conjunction {Px,Qy,Rx}.

1) To solve Px, an Or-process is open which in turn activates two And-processes in parallel with clauses C1 and C2. Both And-processes receive MS={}. The first C1 And-process executes:

SIMPL({Px≈Pa}) ={x≈a}=S and then SIMPL(S∪MS)= SIMPL({x≈a}∪{})={x≈a}.
As it is a terminal process (the body is empty), the C1 And-process returns {x≈a}.
The second C2 And-process performs SIMPL({Px≈Pz})={x≈z}=S and afterwards, SIMPL(S∪MS)= SIMPL({x≈z}∪{})={x≈z}. In this stage, a new Or-process is open for Sz sending it the multisystem {x≈z}. The two terminal clauses C3 and C4 return the multisystems {x≈z,z≈b} and {x≈z,z≈c} respectively and after being merged by the Sz Or-process, {x≈z, (z, (b, c), α1)} is produced. This one is sent back by the Sz Or-process to the And-process C2. Hence, this one receives the multisystems {x≈a} and {x≈z,(z,(b,c),α1)} and merges them into the multisystem:
MS1={(x,(a,z),α),(z,(b,c), α1)}.

2) Next, the Co And-process goes on by activating an Or-literal for Qy which in turn opens two And-processes associated with clauses C5 and C8, sending them the multisystem MS1. The C5 And-process performs first, the simplification:
SIMPL({Qy≈Qu})={y≈u}=S and then SIMPL({y≈u}∪ MS1)={y≈u} ∪ MS1.
This path continues by opening two Or-process for Mu and Nv which obtains {y≈u,u≈a,v≈a}∪MS1 with terminal C6 and C7 And-processes. The other C8 And-process sends back, as another solution, {y≈c ∪ MS1}. At this stage, the Qy Or-process merges the two multisystems and obtains MS2={(y,(u,c),β),(v,(a,v),β),(u,(a,u),β) ∪ MS1}.

3) Following this process yields the final multisystem:
MS3={x≈a,(y,(u,c),β),(v,(a,v),β),(u,(a,u),β)}, which could be easily translated in the two possible solutions {x≈a,y≈a} and {x≈a,y≈c} for the initial problem.

<u>Remark</u>: The implicit parallel SIMPL steps have not been described for the required limit space of this article.

5.- IMPLEMENTATION ASPECTS

Usually, a binding environment (a set of variable-term pairs) is implemented by pointers from the variables to its bound terms. Terms are represented by d.a.g.s. as classically [Pw-

78,Hu-78]. An equation x≈t is represented by a pointer from x towards the d.a.g. node representing t. With multiequations (x, (t1,..,tn),α) the implementation is almost the same. The selector α is a simple integer. The variable x is associated with one pointer to the integer α and another to the list of pointers corresponding to the nodes representing the terms t1,.. and tn. Thus, if the integer takes the i value then i-th equations are selected from the multiequations having this selector α.

In an Or-parallel interpreter a problem emerges by the instantation of shared variables in brother processes. As different instantations can take place from distinct successors of an Or-node, a kind of distribution of certain instantations has to be provided.

Several techniques have been developed in the literature. The goal is to avoid the coping of the whole binding environments from the father to its son nodes. Indeed, as the tree depth increases the copies lead to big overheads that could annul the gain of parallelism.

The best known implementation techniques are the Directory Trees [Ch-83, Ch-86], the Hash Windows [Bo-84,Cr-85,Bcs-88,Fa-87] and the Binding Arrays [Wa-84,Wa-87,Ba-88,Bu-88,Cl-88]. In order to design a good implementation mechanism for shared variables, two main aspects must be considered:
- The structure updating time to change a processor from a process to a new one.
- The access time of a process to a global variable.
It is obvious than the lowest overhead is obtained when less structure data are used and the reading of a variable´s value is done swiftly. Next, we discuss briefly the published techniques.

Directory trees
Data structure. The following data structure are employed:
1) A tree T whose form is the same that the Or search tree.
2) Each node in T is associated with:
2a) A pointer to the list of new created variables by the present open clause.
2b) A pointer to the list of ancestor variables of the open clause, some of them with their corresponding term pointer if they are tied.

A tree T in the memory is created by the open processors: each processor receives a pointer to a node in T and when it activates its n successors it adds n son nodes to the received node. Also, it sends the node addresses to the corresponding successor processors (each one is associated with a new node).

Initialisation and updating. The change of a process requires leaving the data structure of a process and charging the data corresponding to the new next process.

The access to a global variable. The access to a variable begins by searching for it in the local list to the process. If it is not found, then it is looked for in the list of ancestor nodes, in the path going down from the root to the concrete node, following the branches in tree T. Several variants of this method exist.

Hash Windows. The difference between the Directory Tree and the Hash Windows relies on the fact that the reading of a variable´s value is not done by scanning a list but by means of a Hash table which enables a quicker access time, but with more required memory.

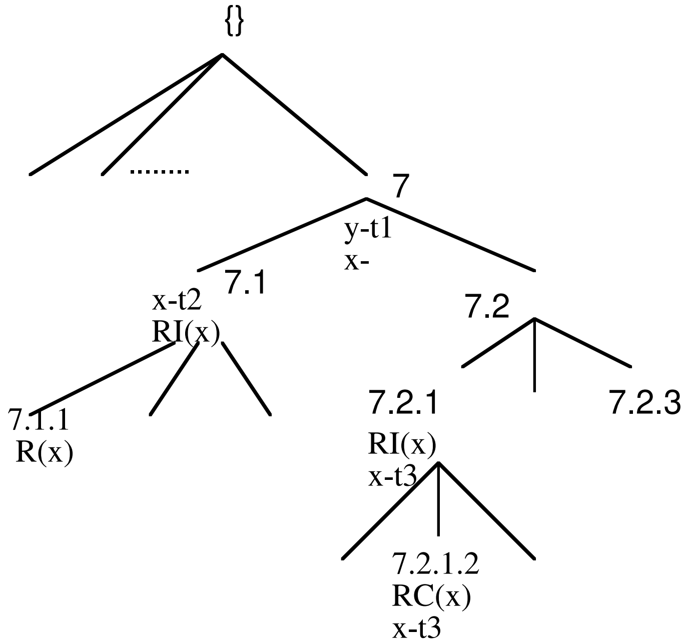
Binding Array. In this approach all, ancestor and created variables of a process, which could solve several subsequent Or-literals, are ordered in a vector called the Instantations Vector. The gain is the constant time of accessing to a variable. The drawback is the cost of updating the context when one processor changes of process because a new Instantation Vector must be charged. In this case, the strategy of the scheduler must be to assign a new process to a processor which is "near" the previous process of the processor, "near" meaning with few differences in the instantations variables.

Proposed implementation approach. The *published implementation approaches can also be applied to the interpreter with multisystem* taking into account the small differences in the implementation of an equation and a multiequation. But here, we describe briefly another implementation structure that presents some small advantages with respect to the existing ones: the updating is null and the reading time is linear in the tree size for each variable (proved below).

Data structure. Each process has reserved a hash table in the memory in such way that the reading of a variables value is done in constant time. Two following structures are needed:
- The tree T as previously;
- Each variable has two structures (in addition to those needed to implement the equation or multiequations). i) A flag that indicates if the variable was instantiated by the same process that created it. ii) A pointer to the creator node of the variable.

The access to a global variable. This time is proportional to the number of nodes into the path going down from the creator to the reading node of the variable. The updating overhead is nil because the structures are created dynamically and the variables are referenced by the two nodes in the tree (the one corresponding to the node creator and the present one trying to read the variable).

Example (see figure below): The notation i.j.k.... follows the breadthwise first right-left order of nodes in the tree. Each node corresponds to an Or-process and so it is associated with an integer succession i.j.k.... Node 7 creates x and y (they appear for the first time in the clause considered in this node). y is instantiated to a term t1 in this process but x remains free without instantation. After, nodes 7.1 and 7.2.1 Read and Instantiate x (RI(x)). Let us focus on node 7.2.1. As the flag of x is nil (x has not been instantiated in its creation) then x is instantiated to t3 after having checked that its value is also empty in its previous ancestors (only 7.2). Later, in 7.2.1.2 the value of x is read from node 7.2.1. As it has the value t3 there, it is copied in node 7.2.1.2 RC(x).
The reading of y´s value at any point is straightforward because the flag of y indicates that y has been instantiated in its creation.



Proposition:
- Variables instantiated by its creator process permit a constant reading time.
- The total reading time of a variable is in O(K + N), where N is the total number of reading operations and K is the size of the searching tree (strictly the subtree going down from the creator node of the variable).

Proof. The first statement follows from the constant time of accessing to a value in a Hash Table. To check the second statement, we can consider a strategy of reading and coping a value of a global value such that: A variables value is copied in all the nodes between the reading and creator node (for instance, in the previous example, when x is read in node 7.2.1. from node 7, the reading operation has crossed throughout node 7.2 and thus, the nil value could be affected to x in node 7.2 also). With this strategy a path in the tree T is crossed throughout only once for each variable.

Merging. The merging of multisystems is made by Or-processors following the given definition. The conversion of n equations in one multiequation and the translation of an equation in a multiequation (see definition of Merging) are straightforward. The simplification operation uses direct access to local variables (in present clauses) or indirect access (by means of the tree T) to global variables and instantiates the variables employing the described data structures.

6.- ANALYSIS AND COMPARISONS

To show the efficiency of the proposed method we give first an illustrative example:

Example: Let us consider the following simple Logic Programming:

$\leftarrow$ P(x1) , P(x2) , ..., P(xk) , P(x1,x2,...,xk)

P(a11)$\leftarrow$ P(a21)$\leftarrow$ .... P(ak1)$\leftarrow$ P(a1i_1,a2i_2,..,aki_k)$\leftarrow$

P(a12)$\leftarrow$ P(a22)$\leftarrow$ .... P(ak2)$\leftarrow$

…………………………………………

…………………………………………

P(a1n)$\leftarrow$ P(a2n)$\leftarrow$ .... P(akn)$\leftarrow$

This Logic Programming problem has only one solution {x1≈a1i_1,x2≈a2i_2,..,xk≈aki_k}. Some comparative results are the following, considering there is no serialisation of processes.

	Total running time	Number of processes	Required space memory
proposed method	O(k.n)	n+1	O(k.n)
standard methods	O(k.n)	O(n^k)	O(n^k)

Although in this case, the total running time is similar in both cases, due to the growing up of processes in the standard methods, some serialisation would be imperative. Thus, the real running time of a standard method should be a function depending also on the number of created processes, namely O(n^k).

A comparative discussion about the efficiency of several parallel interpreters must envisage at least these three parameters to solve a Logic Programming problem:
- The number of used processors,
- The total running time, and
- The required space memory

We are going to analyse these matters and others derived from them, comparing the proposed Or-interpreter with other standard Or-models described in the second section.

We use the following notation characterising an interpreter to solve a certain problem:
- Tpi (resp. Tsi), the running time of the proposed interpreter (resp. a standard one) ;
- Npi (resp. Nsi), the number of used processors by the proposed interpreter (resp. standard one);

- Mpi (resp. Msi), the space memory required by the proposed interpreter (resp. a standard one).

(the proofs of the next claims are only pointed out in this paper).

Lemma:

- We have always that Nsi ≥ Npi.
- The ratio Nsi / Npi could be exponential.

Proof: It could be proved [Es-89] that for each state in the proposed method where a process is decomposed in n processes there exists a process in a standard interpreter that is decomposed in the same number n of processes. However, the reciprocal is not true because of the existing mergings of several processes into only one in the proposed method. The second part of the lemma could be shown by analysing the example given in this section.

Lemma: Let Spi (resp. Ssi) be the number of serialised processes in a concrete parallel machine using the proposed interpreter (resp. a standard interpreter) to solve a problem.
We have Ssi ≥ Spi and the ratio Nsi / Npi could be exponential.

Proof: It follows from the previous lemma and from the fact that the number of processes is fixed for a parallel machine.

Lemma: When all the solutions are looked for:
Tsi ≅ Tpi iff no serialisation is needed and,
Tsi ≥ Tpi , otherwise.

Proof: The complete proof turns out to have many details. A sketch of it is as follows.
In a standard interpreter Tsi is related to the longest path in the search space modelling the problem because no branches could be pruned as they could lead to an eventual solution. A similar assertion happens with the proposed method (although strictly there is some additional cost for merging states and multisystems). However, it seems clear that in a real case some processes have to be serialised. Then, using the previous lemma, Ssi ≥ Spi implies Tsi ≥ Tpi.

It is almost always the case that all the solution are searched but let us suppose a different situation where only one solution is enough.

Lemma: Let us note Lmax (resp. Lmin) the longest (shortest) path in the search space of the set of paths leading to a solution. We have:

Tsi=O(Lmin) ; Tpi=O(Lmax) with Nsi ≥ Npi
iff
no processes have to be serialised and only one solution is searched .

Proof: It follows from the previous lemmas.

With this lemma we can see that running time to solve a problem with an Or-standard interpreter could be less than with our proposed interpreter, although using a bigger number of processes and in the particular context of searching only one solution and having a machine with much more processors than processes. However, this case is almost hypothetical because:
1) Usually, you search for all the solutions and 2) The increasing number of processors in the standard interpreters make compulsory the serialisation of processes and this factor could be exponentially bigger than in the proposed one.

Concerning the space memory, we can state:

Lemma: Mpi ≥ Msi and the ratio Mpi/Msi could be exponential.

Proof: The first statement Mpi ≥ Msi follows from the fact that an implicit equation in a multisystem can be common to a sub-set of simple systems Σ(MS) handled by standard interpreters. To show the second claim it is enough to consider the example in section 3.

Now, let us suppose that we have to resolve problems of a certain size with a small parallel machine having a small number of processors. In this case, the efficiency of both methods, the proposed one and the existing one, is strongly related to its efficiency in the computing on a monoprocessor machine, namely serialising all the processes. In this particular situation, we can use the following result

Lemma:
- The merging of multisystems MS1,..,MSn is in O(|MS1U....MSn|), where |MS1U....MSn| notes the total number of terms in the multisystems;
- If the time to calculate SIMPL(MS U S) is noted T, and the time to calculate {SIMPL(S∪S´)≠Fail: S´∈Σ(MS)} is noted T´ the we have: T´ ≥T and the ratio T´/T could be exponential.

Proof: Its proof is given in [Es-89].

This lemma leads to the following result:

Lemma:
If T is the time to resolve a problem for the proposed interpreter with only one processor and T´ is the time required by a standard parallel interpreter then T´≥T and the ratio T´/T could be exponential.

Proof: The complete proof is given in [Es-89]. It follows from the fact that mergings are additional operations done in the proposed method with respect to the classical ones and that for each merging MG(MS1 U .. U MSn) there exists at least one simplification SIMPL(MS U S) operation. Thus, the present method loses time when a merging is done but it gains when a simplification SIMPL(MSUS) is performed instead of {SIMPL(S∪S´)≠Fail: S´∈Σ(MS)} done in the standard cases. However, by the previous lemma, the time lost is proportional to the terms in (MS1 U ..U MSn) and the simplification SIMPL(MS U S) may require a time exponentially less than {SIMPL(S∪S´)≠Fail: S´∈Σ(MS)}.

7.-THE SCHEDULER

The scheduling problem of a set of processes problem appears when the number of active processes is bigger than the number of processors in the machine. In this case, the scheduler is in charge of the assigning of some processes to the available processors. Thus, we discuss two strategies that could be suitable to gain performance using the proposed interpreter. Indeed, it could be the case that to assign a particular process instead of another one to a processor allows us to diminish the overhead due to change contexts.

The two possibilities derive from the double possibility of using 1) a pure interpreter as the proposed one, or 2) an hybrid interpreter resulting of the proposed and a standard one.

1) In this case no particular strategy has to be applied. The processes can be applied to the processors in the same order as they are stopped as a consequence of the busy situation of processors. This results in a mixture of parallel and sequential interpreters relying on multisystems (the sequential one has been described in detail in [Es-90]).

2) The proposed interpreter is based on a certain scanning of the search space and the merging and simplification of multisystems. These two operations can be applied in a hybrid multisystem-standard interpreter. When all the processors become busy (a process has been

assigned to each of them) then, the scheduler could organise the pending assignments of processes to processors by giving rise to the merging and simplification of multisystems.

Thus, let us suppose that if we exclusively applied the standard interpreter we would consume too many processes. Thus, taking advantages of the significantly less number of required processors of the proposed method to solve a problem, we may implement the following strategy. The "suspicious" long paths (determined by a pre-processing step for example) are resolved by processes formed by the standard approach. The "suspicious" shorter and homogeneous paths are resolved by a multisystem interpreter. Following this strategy, we will merge multisystems coming from different branches developed to solve the same literal goal and verifying: 1) its length is similar, or 2) they have been serialised by saturation of the processors.

The advantages of such a hybrid system are that, when the number of processors is big enough to enable that part of the search space could be solved by a standard method then, we could cope with the advantages of both interpreters: The longest paths on the one hand and the major remaining of the search space on the other hand, are resolved relatively fast and with a small number of processors.

8.-CONCLUSION

It is easy to verify that the multisystems represent a set of binding environments (systems of equations) in a compact form and that even an exponential reduction in space memory could be reached. This is a straight consequence of the fact that common equations to several systems are represented only once. Also, as we have seen in the previous section 5, the implementation of multisystems does not require complex or particular data structure which facilitates the implementation and reduces overheads.

Concerning the processing time, some similar assertions to that made above with respect to space memory could be made in favour of the proposed approach. In fact, the described operation of simplification of a system and a multisystem are faster than the operations classically effected with simple systems (binding environments).

Let us consider for instance, the simplification between a system S and a multisystem $Q \cup MQ$. With our approach the calculus SIMPL($S \cup Q$) is done only once whereas with conventional binding environments (or systems), it could be repeated for the number $|\Sigma(MS)|$ of systems represented by the multisystem. A similar statement applies for further simplifications made in the next steps. The simplification performed in a node of the tree (built by the selector expansion mechanism) of the SIMPL($S \cup MQ$) operation is done only once whereas with the conventional methods it could be done as many time as leaves are found going down from this particular node to the leaves.

Thus, the total number of operations are fewer (can be exponentially fewer) with multisystems (and with the defined operations over them) than with classical systems. Furthermore, the proposed method can take advantage of parallelism in doing simplification operations.

Concerning implementation aspects, the existing approaches (directory trees, binding arrays, hash windows, etc.) could be employed in the proposed parallel interpreter. But, a more adequate data structure has been described which presents some variations with respect to the published ones.

The claims presented by lemmas in the last section make evident the following positive aspects of the proposed method compared to the known ones:

- It uses fewer number of processes (exponentially fewer in a best case) and so in many cases, this could be a relevant characteristic because the number of processors is limited in each particular parallel machine.
- It serialises less processes (exponentially less in a best case). This point implies that the maximal Or-parallelism is exploited and hence, a faster resolution of the problem can be found.
- It is quicker in a monoprocessor or in a machine with few processors. This is an important good feature when you have to resolve some problem in fairly small machines. Also, the claim could be extrapolated to the case where there are too many open processes for the available number of processors.
- When all the solutions are searched, the proposed method runs faster than the previous ones. This is a consequence of the facts: 1) without process serialisation the running time is similar for both interpreters (the proposed and the standard one), but 2) the serialisation reaches a much higher degree with standard methods.
- Only when one solution is searched and when the number of processes is smaller than the number of available processors, the proposed method could be slightly slower than standard ones. But, that appears as a hypothetical and no real situation.
- It requires less space memory, exponentially less in a best case.

It can be noticed that the method possesses other good characteristics: all the solutions are determined simultaneously which could be of particular benefit when Logic Programming is used in a Logic Data Base framework; there is no loss in the declarative expression richness; no backtrack is needed neither must any particular information be got before or during the execution which reduces significantly the overheads.

ACKNOWLEDGEMENTS. We thanks the anonymous referee for his comments and specially for providing us a reference relating our work with other ones which share certain mechanism analogy. This work has been supported by the ARREL Project TIC-92-0579-C02-01 and the collaboration between IIIA-CSIC (Spain) and CINVESTAV-IPN (Mexic) supported by the CSIC (Spain) and the CONACyT (Mexic) organisms. We thank also the COMPULOG-NET organisation for all the facilities conceded us, including a grant to cover the expenses needed to participate in the workshop.

9.-REFERENCES

[Bcs-88] Baron, U.C. Chassin de Kergommeaux J. and Syre J.C. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. Proc. of the Int. Conf. on 5th Generation Comp. Systems, 1988.

[Bo-84] Borgwart P. Parallel Prolog Using Stack Segments on Shared Memory Multiprocessors. Proc. of the Int. Symp. on Logic Programming, 1984.

[Ba-88] Brand P. and Almgren J. Wave-front Model for Scheduling in Or-parallel Prolog. Technical Report, SICS, July, 1988.

[Bu-88] Butler R. et al. Scheduling Or-parallelism: An Argonne Perspective. Proc. of the Fith Int. Conf. and Symposium on Logic Programming. Seatle, August, 1988.

[Ca-88] Calderwood A. Scheduling Or-parallelism in Aurora - the Manchester Scheduler. Technical Report, Manchester University, May, 1988.

[Cd-85] Chang, J. and DeGroot, D. And-Parallelism of Logic Programs Based on a Static Dependency Analysis. Digest of Papers of Compcon, Spring 1985.

[Ch-83] Ciepilewski A. and Haridi A. Storage Models for Or-parallel Execution of Logic Programs. Technical Report TRITA-CS-8301. Royal Institute of Technology, Stockholm, Sweden, 1983.

[Ch-86] Ciepilewski A. and Husman B. Performance Evaluation of a Storage Model for Or-Parallel Execution of Logic Programs. Proc. of the Symp. on Logic Programming, Salt Lake City, Sept. 1986.

[Cl-88] Clark K.l. Parlog and its Applications IEEE Trans. on Software Engineering, Vol 14, N 12, Dec. 1988.

[Co-81] Conery, J.S. Parallel Interpretation of Logic Programs. Proc. of the Conf. on Functional Progr. Languages and Comp. Arch., 1981.

[Co-83] Conery, J.S. And-Parallelism in Logic Programs. Proc. of the Int. Conf. on Artificial Intelligence, 1983.

[Cr-85] Crammond J. A Comparative study of Unification Algorithms for Or-parallel execution of Logic Languages. Proc. of the Int. Conf. on Parallel Processing, 1985.

[De-84] DeGroot, D. Restricted And Parallelism. Proc. of the Int. Conf. on 5th Generation Comp. Systems, 1984, ICOT, Tokio, Japan .

[Dw-84] C. Dwork, P. Kanellakis., P.C. Stockmeyer. On the sequential natura of unification. Journal of Logic Programing, Vol 1, N 1, pp 35-50, 1984.

[Dw-88] C. Dwork, P. Kanellakis., P.C. Stockmeyer. Parallel Algorithmes for term matching. SIAM Journal of Computing. Vol 17, N 4, 1988.

[Es-89] Escalada-Imaz G. Un schéma de contrôle en Logique du Premier Ordre orienté vers la recherche de toutes les solutions. Research Report LAAS-CNRS 89230. 74 pp. Julio 1989.

[Es-90a] Escalada-Imaz G. Optimisation d'Algorithmes d'Inférence Monotone en Logique des Propositions et du Premier Ordre. PhD. thesis. University of Toulouse (France). Also Research Report LAAS-CNRS 89.204. 181 pp. Julio 1989. *Fourth Chapter*.

[Es-90b] Escalada-Imaz G. Un schema de contrôle de l'inférence en Logique du Premier Ordre orienté vers la recherche simultanée de toutes les solutions. Revue d'Intelligence Artificielle. Hermès (ISSN 0992-499x). Vol 4, N4, pp 49-116, Diciembre 1990.

[Es-91] Escalada-Imaz G. Control de la Inferencia en Lógica del Primer Orden basado en la Búsqueda Simultánea de Todas las Soluciones. IV Congreso Asociación Española de Inteligencia Artificial, IV CAEPIA-91. Madrid. pp 59-73. Octubre 1991.

[Es-92a] Escalada-Imaz G. A parallel Interpretion Model of Logic Programs based on Multisystems. Research Report IIIA-CSIC 92-14. 10 pp. July 1992.

[Es-92b] Escalada-Imaz G. A Parallel Interpretation Model of Logic Programms Based on Multisystems of Equations. En: Parallel Computing and Transputers Application. (Valero, Oñate, Jane, Larriba, Suárez, eds.). IOS Press, Amsterdam. Parte I, pp 745-754. September 1992.

[Es-92c] Escalada-Imaz G. An Efficient Parallel Interpreter of Logic Programs. I Congreso Nacional de Programación Declarativa, PRODE-92. (Eds: Hermenegildo M, Moreno J.J.). Madrid. pp 69-80. Octubre 1992.

[Es-93a] Escalada-Imaz G. Exploiting Restricted And-parallelism and Or-parallelism in Logic Programs with Multisystems. Workshop on Parallelism and Implementation Technologies. Madrid. pp 13-20. Mayo 1993.

[Es-93b] Escalada-Imaz G. A new Approach for Exploting Or-parallelism in Logic Programming. Research Report IIIA-CSIC 93-04. 19 pp. February 1993.

[Es-94a] Escalada-Imaz G. A Multisystem-Based Parallelism-Interpreter of Logic Programs with Restricted And-Parallelism and Or-parallelism. Research Report IIIA-CSIC 94-03. 18 pp. Enero 94.

[Es-94b] Escalada-Imaz G. A Multisystem-Based Parallelism-Interpreter of Logic Programs with Restricted And-Parallelism and Or-parallelism. IEEE International Conference on Systems, Man and Cybernetics, IEEE-SMC-94. Texas, USA. pp 402-408. Octubre 1994.

[Es-95] Escalada-Imaz G., Manyà F. Reducing Time by avoiding Process Serialisations in Or-Parallel Interpretation of Logic Programs. IEEE International Conference on Systems, Man and Cybernetics, IEEE-SMC-95**.** Vancouver (Canada), Octubre 1995**.**

[Fa-87] Fagin, B.S. A Parallel Execution Model for Prolog. Report N UCB/CSD 1987. University of California, Berkeley. Nov. 1987.

[Ha-87] J. Harland and J. Jaffar. On parallel Unification for Prolog. New Generation Computing, 5, pp 259-279, 1987.

[Ha-90] B. Hausman. Handling speculative work in or-parallel prolog: Evaluation results. North American Conferencia on Logic Programing, pp &21-736. Austin, Texas, 1990.

[He-86] M. V. Hermenegildo, R.I. Nasr. Efficient Management of Backtracking in AND-parallelism. Third International Conference on Logic Programming. LNCS 255, pp 40-55.

[He-90] M. Hermenegildo , F Rossi. On the Correctness and Efficiency of Independent And-parallelism in Logic Programming. pp 369-390. MIT Press, 1989.

[He-92] M. Hermenegildo, R. Warren, S. Debray. Global Flow Analysis as a Practical Compilation. Journal of Logic Programming, 13 (4): 349-367, 1992.

[Hu-78] Huet G. Résolution d´équtions dans des lengages d´ordre 1,...,ω. Thèse d´Etat. Paris VII. 1976.

[Ka-87] L. Kale. Comlpetness and Full Parallelism of Parallel Logic Programing. Fourth IEEE Simposium on Logic Programing. 125-133, IEEE, 1987.

[Lm-86] Li, P.P. and Martin A.J. The Sync Model: A Parallel Execution Method for Logic Programming. Proc. of the Symp. on Logic Programming, Salt Lake City, Sept. 1986.

[Lk-88] Lin Y. Kumar V. An execution Model for Exploiting AND-parallelism in Logic Programs. New Generation Computing, 5, Pags. 393-425, 1988.

[Mu-89] K. Muthukumar, M. Hermenegildo. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. 1989 International Conference on Logic Programing. pp 80-101. MIT Press, 1989.

[Pw-78] Paterson M.S. and Wegman M.N. Linear Unification, J. of Computer Science 16, 2. 1978.

[St-85]Shapiro E.T. and Takeuchi A. Object Oriented Programming in Concurrent Prolog. New Generation Computing, Vol. 1, pp-25-48, 1985.

[Sm-94] D.A. Smith. Why Multi-SLD Beats SLD (Even on a Uniprocessor). PLILP-94, pp 40-56. LNCS 844. Springuer-Verlag.

[Sz-89] Szeregi. Performance analysis of the Aurora Or-prolog System. 1989 North American Conference on Logic programing. MIT Press. 1989.

[Tu-95] J. Tubella, A. González. Exploiting Path Parallelism in Logic Programming. Proc. of the Third Euromicro Workshop on Parallel and Distributed Processing PDP'95, pp. 164 - 173. San Remo, Italy. Juanuary 1995. IEEE Computer Society Press, 1995.

[Tu-94] J. Tubella, A. González. A Partial Breadth-First Execution Model for Prolog. In Proceedings of the 6th Int. Conf. on Tools with Artificial TAI´94.

[Ya-84] H. Yasuura. On parallel computational Complexity of unification. Proceding of the International Conference on Fith Generation Computer Systems 1984. ICOT. pp 235-243.

[Wa-84]Warren D.S. Efficient Prolog Memory Management for Flexible Control Strategy, Proc. of the Int. Symp. on Logic Programming. 1984.

[Wa-87]Warren D.H.D. The SRI Model for Or-Parallel Execution of Prolog- Abstract Design and Implementation Issues. Proc. of the Symp. on Logic Programming, 1987.

[Wa-89] D.H.D. Warren. The SRI Model for OR-Parallel Execution of Prolog: Abstract Design and Implementation. International Symposium on Logic Programming, pp 99-102. San Francisco, IEEE Computer Society, 1987.

[Wa-90] D.H.D. Warren. Or-parallel Execution Models of Prolog. Proceedings of TAPSOFT-87. Lecture Notes in Computer Science. Springer-Verlag, 1987.

[Wi-86a]Wise M.J. Experimenting with Epilog: Some Results and Preliminary Conclusions. Proc. of the 13th. Int. Symp. on Computer Architecture, 1986.

[Wi-86b]Wise, M.J. Prolog Multiprocessors. Prentice-Hall, 1986.