Deductive Databases and P Systems

Miguel Angel GUTIÉRREZ-NARANJO

Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: `magutier@us.es`

Vladimir ROGOZHIN

The State University of Moldova
60 Mateevich str., MD-2009
Chişinău, Moldova E-mail: `rv@math.md`

Abstract. In computational processes based on backwards chaining, a rule of the type $A \leftarrow B_1, \ldots, B_n$ is seen as a procedure which points that the problem A can be split into the problems $B_1, \ldots, B_n$. In classical devices, the subproblems $B_1, \ldots, B_n$ are solved sequentially. In this paper we present some questions that circulated during the Second Brainstorming Week related to the application of the parallelism of P systems to computation based on backwards chaining, and we illustrate them with the example of *inferential deductive process*.

1 Introduction

In computational processes based on backwards chaining, a rule of the type $A \leftarrow B_1, \ldots, B_n$ is usually seen as a procedure which points that the problem A can be split into the problems $B_1, \ldots, B_n$ with the hope that $B_1, \ldots, B_n$ are simpler than A. In the case of getting $B_1, \ldots, B_n$ solved, we also have a solution for A via this rule.

This is the case of pure Prolog [2, 10] where $A \leftarrow B_1, \ldots, B_n$ is a definite clause and $A, B_1, \ldots, B_n$ are positive literals. Prolog uses SLD resolution to find an *answer* to the *goal* A, with SLD coming from Linear resolution for Definite clauses with Selection function. This selection function considers sequentially the list of current subgoals $B_1, \ldots, B_n$ and chooses one of them (in standard Prolog the selection function always takes the leftmost literal). The process of finding an answer for the chosen subgoals generates new subgoals, hopefully simpler than the previous one. The computation ends when trivial subgoals are reached.

The selection mapping is necessary because classic computational devices work sequentially, so we need to fix an order between the tasks.

In this paper we present some questions that circulated during the Second Brainstorming Week related to the application of the parallelism of P systems to the computation based on backwards chaining.

2 Logic Programming

Although the computation based on backwards chaining is a general procedure in computer science, we focus our attention on Deductive Databases and Logic Programming.

The way of representing information in Logic Programming (see, e.g., [1, 3, 6, 8, 4]) is via a set of clauses. These sets of clauses are *logic programs*. Roughly speaking, a clause is a first-order rule, where both sides of the rule consists of atoms, i.e., a predicate applied to some arguments. Formally, a clause is a formula

$$\forall x_1 \ldots \forall x_s A_1 \vee \ldots A_k \vee \neg B_1 \vee \cdots \vee \neg B_n,$$

where $x_1, \ldots, x_s$ are all the variables that occur in the atoms $A_1, \ldots, A_k, B_1, \ldots, B_n$. A clause [1] is a *Horn clause* if it contains at most one positive literal *(atom)* and it is a *definite clause* if it contains exactly one positive literal. For example

$$\forall X \forall Y\ daughter(X, Y) \vee \neg female(X) \vee \neg mother(Y, X)$$

is a definite clause. This universally quantified formula is usually written as

$$daughter(X, Y) \leftarrow female(X), mother(Y, X).$$

The positive literal, i.e. the *conclusion* of the implication is usually called the *head* of the clause. The rest of the literals, the *premises*, is known as the *body* or the *tail* of the clause. Definite clauses can consist on a single positive literal. They can be considered as rules with no tail or no conditional sentences, such as

$$female(anne) \leftarrow$$
$$mother(mary, anne) \leftarrow$$

These clauses are *facts*. A *substitution* $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$ is an assignment of terms t_i to variables V_i. If a substitution is applied to a clause, then we get an instantiated clause, where all occurrences of the variable V_i is replaced by the term t_i. For example, if the substitution $\theta = \{X/anne\}$ is applied to the clause C:

$$daughter(X, Y) \leftarrow female(X), mother(Y, X),$$

then we get the clause $C\theta$:

$$daughter(anne, Y) \leftarrow female(anne), mother(Y, anne).$$

A substitution θ is a *unifier* of the atoms A and B if $A\theta = B\theta$.

Logic programs compute through a combination of two mechanisms: unification and resolution. From any two clauses with complementary literals A and $\neg A$ the inference rule of *resolution* derives a new clause as consequence. For example, from

$$daughter(anne, Y) \leftarrow female(anne), mother(Y, anne)$$
$$female(anne) \leftarrow$$

we obtain the clause $daughter(anne, Y) \leftarrow mother(Y, anne)$. The deduction process is goal driven in the following way. If we have the *program*

$$daughter(X, Y) \leftarrow female(X), mother(Y, X)$$
$$female(anne) \leftarrow$$
$$mother(mary, anne) \leftarrow$$

[1] The basic difference between program clauses and database clauses is the use of types.

and we want to know if $daughter(anne, mary)$ is true, first we build the *goal*

$$\leftarrow daughter(anne, mary),$$

i.e., the one-literal clause $\neg daughter(anne, mary)$. The atoms $daughter(anne, mary)$ and $daughter(X, Y)$ unifies with the substitution $\theta = \{X/anne, Y/mary\}$. By using resolution with the first clause of the program and the unifier θ we get the new goal

$$\leftarrow female(anne), mother(mary, anne).$$

As we saw before, this step can be seen in a procedural mode. The problem of deciding if $daughter(anne, mary)$ is true has been split into two subproblems: Decide if $female(anne)$ and $mother(mary, anne)$ are true or not. But they are true because they are claimed by our program, so $daughter(anne, mary)$ is true.

When the reasoning system solves the *goal* Q it gives us an answer. There are two types of outputs given by the system with respect to the type of the goal Q:

1. If the goal does not contain variables, then we have a decision problem, and the possible answers are `Yes` or `No`. In this case the system decides if the goal can be or not derived form the program.
2. If the goal contains variables, then the system outputs the *unifier* θ such that the instantiated goal $Q\theta$ can be derived from the problem. This unifier represents the *answer* to the question, and obviously several unifications θ that make the goal $Q\theta$ true can exist.

In our example, we deal with a decision problem. After the first step the subgoals $female(anne)$ and $mother(mary, anne)$ have to be solved. This is done sequentially in classical devices with only one processor. We wonder whether it is possible to use P systems for these problems. We think that it would be very interesting to use the parallelism of P systems to solve all subgoals in a parallel manner.

3 P Systems

Now we are going to give some hints about a general representation of a set of *typed definite clauses* (*Deductive Database*) and of the *inferential deductive process* in the frame of hierarchical P systems [9] with active membranes [5].

Let us consider a Deductive Database (DDB)

$$\begin{array}{lllll} Q_1 \leftarrow & P_{11}, & P_{12}, & \ldots, & P_{1m} \\ Q_2 \leftarrow & P_{21}, & P_{22}, & \ldots, & P_{2m} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ Q_n \leftarrow & P_{n1}, & P_{n2}, & \ldots, & P_{nm} \end{array}$$

We assume for simplicity that we have the same set of parameters $\{x_1, \ldots, x_s\} \in D^s$ from the same domain D for all literals $Q_i,\ i \in \{1, \ldots, n\},\ P_{ij},\ i \in \{1, \ldots, n\},\ j \in \{1, \ldots, m\}$, and the same order of parameters $(x_1, \ldots, x_s)$ for the heads of all rules. The literals from tails are allowed to have any order of parameters.

We can *ask goals* presented as literals with constant terms and/or variables in the set of parameters to the *inferential deductive machine*. The constants will be denoted by c_i and variables will be denoted by v_i. We assume that goal have the same order of parameters $(x_1, \ldots, x_s)$ as the heads of DDB clauses.

Now we will give the general model of *logic inferential deductive machine* and some ideas how it can be represented in the framework of P systems. Consider the DDB described above and the goal Q. The logic inferential deduction process will be performed recursively according to the following steps:

ALGORITHM: *SOLVE*
INPUT: Q

PART 1: From the goal to the axioms:

Step 1 *Head unification*: Unification of Q with all heads Q_i from DDB in parallel. As a result, every head Q_i for which the unification process succeeded will get the set of unifiers θ_i.

Step 2 *Body unification*: For every head Q_i for which step 1 succeeded and the tail is not empty, *body unification* process will be performed with all subgoals P_{ij} in parallel, that is, the algorithm $SOLVE$ will be launched in parallel for every subgoal P_{ij} with input $P_{ij}\theta_i$. In the case of *facts* (rules with empty tail), the system returns the unifier θ_i of the head Q_i.

PART 2: From the axioms to the goal

Step 3 *Atom unification:* For every rule for which step 2 succeeded, the unification of results of all subgoals is performed.

Step 4 *Union of the results:* Since every particular rule from the DDB gives us some set of unifiers (solutions), one should consider the union of all these sets as a solution of Q.

OUTPUT: There are possible two cases:

1. The result to be the set of all unifiers $\Theta = \{\theta | DDB \vdash Q\theta\}$. In other words, the result will be the set of all unifiers θ for which $Q\theta$ could be derived from DDB
2. The result to be `Yes` in the case $\Theta = \{\theta\}, Q = Q\theta$ and `No` in the case $\Theta = \emptyset$

In this way we have got two types of parallelism here:

1. For each head Q_i a process which unifies it with Q and which unifies results of subgoals P_{ij} is created;
2. For each subgoal P_{ij} of Q_i the solving process is created.

In Figure 1 the general scheme of the deduction process and of the parallel processes interactions is presented. For each process from the scheme a membrane is created. In this way one can treat the tree of the processes interaction as a hierarchy of membranes of the P system solving the problem.

One can define two general types of membranes:

1. Membranes which stay for goals and subgoals representation. Membranes of this type perform *steps 1* and *4*. In *Step 1* they create submembranes which stay for the heads of the clauses. In *Step 4* it collects results of the *inferential rules* execution.

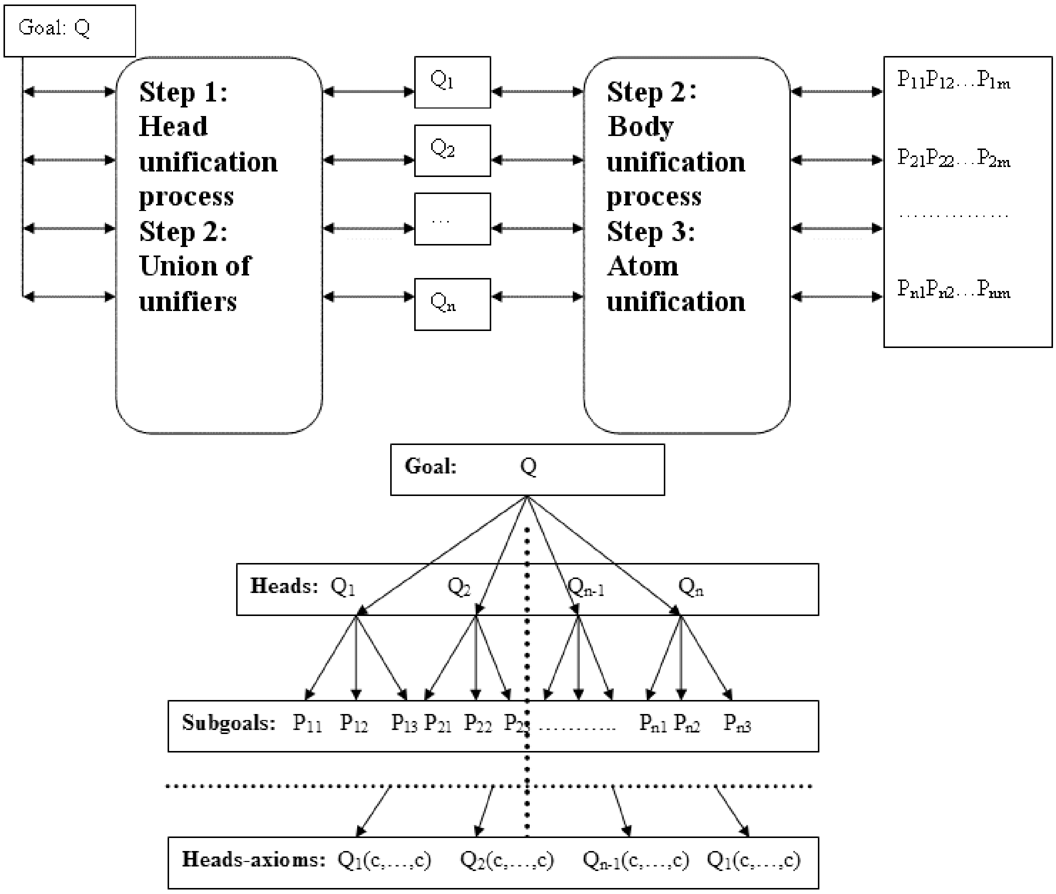

Figure 1: The general scheme of the deduction process and of the processes interaction

2. Membranes which stay for clause's heads representation. They perform *steps 2* and *3*. For this type of membranes there are possible two cases:

   The tail is not empty: the membranes complete the term unification of the head and goal, create submembranes which stay for the subgoals, and perform term unification with all subgoals.

   The tail is empty: the membranes complete the term unification.

All these ideas need to be specified, formalized, and developed, and we hope to return to this topic in a forthcoming research.

4 Final Remarks

In this work-in-progress paper we describe some preliminary ideas born from discussions about this topic during the Second Brainstorming Week. This is only the beginning and a lot of work have to be done. A first step is to fix the backwards chaining formalism that we want to study in P systems. *Function-free* clauses, i.e., clauses which contains only variables as terms can be a good starting points, but to handle relevant information *Datalog* [11] clauses can be more suitable. Datalog clauses are definite clauses that contains no functions symbols of non-zero arity. As we have mentioned, we have a long path to walk.

Acknowledgment. Miguel A. Gutiérrez Naranjo is partially supported by the project TIC2002-04220-C03-01 of the Ministerio de Ciencia y Tecnología of Spain, cofinanced by FEDER funds, while Vladimir Rogozhin is supported by "MolCoNet" project IST-2001-32008.

References

[1] Apt, K.R.: Logic Programming, *Handbook of Theoretical Computer Science.* Elsevier Science Publishers B.V., 1990

[2] Bratko, I.: *PROLOG Programming for Artificial Intelligence*, Third edition. Addison-Wesley, 2001.

[3] Doets K.: *From Logic to Logic Programming.* The MIT Press, 1994.

[4] Dzeroski, S., Lavrac N.: *An Introduction to Inductive Logic Programming* in Relational Data Mining, Springer, Berlin 2001. pp.:48-73

[5] Krishna S.N., Rama R.: A Variant of P Systems with Active Membranes: Solving NP-Complete Problems. *Romanian Journal of Information Science and Technology*, 2, 4 (1999), 357–367.

[6] Lloyd J.W.: *Foundations of Logic Programming.* (2nd ed.) Springer, Berlin, 1987

[7] Metakides G., Nerode A.: Principles of Logic and Logic Programming, *Studies in Computer Science nd Artificial Intelligence*, 1996.

[8] Nienhuys-Cheng S.H., de Wolf R.: *Foundations of Inductive Logic Programming*, LNAI 1228. Springer 1997.

[9] Păun, Gh.: Computing with Membranes, *Journal of Computer and System Sciences*, **61**, 1 (2000), 108–143.

[10] Logic Programming: `http://www.afm.sbu.ac.uk/logic-prog/`

[11] Deductive Databases: The DATALOG Approach:
`http://goanna.cs.rmit.edu.au/ zahirt/Teaching/subj-datalog.html`