International Journal on Artificial Intelligence Tools
Vol. 9, No. 4 (2000)

CONCURRENT ABDUCTIVE LOGIC PROGRAMMING IN PANDORA

REEM BAHGAT, OSAMA MOSTAFA
Department of Computer Science, Faculty of Computers and Information
Cairo University, 5 Tharwat Street, Orman, Dokki, Cairo - Egypt
E-mail: rbahgat@ritsec1.com.eg

GEORGE A. PAPADOPOULOS
Department of Computer Science, University of Cyprus
75 Kallipoleos Street, P.O. Box 20537, CY-1678, Nicosia, Cyprus
E-mail: george@cs.ucy.ac.cy

The extension of logic programming with abduction (ALP) allows a form of hypothetical reasoning. The advantages of abduction lie in the ability to reason with incomplete information and the enhancement of the declarative representation of problems. On the other hand, concurrent logic programming is a framework which explores AND-parallelism and/or OR-parallelism in logic programs in order to efficiently execute them on multi-processor / distributed machines.

The aim of our work is to study a way to model abduction within the framework of concurrent logic programming, thus taking advantage of the latter's potential for parallel and/or distributed execution. In particular, we describe Abductive Pandora, a syntactic sugar on top of the concurrent logic programming language Pandora, which provides the user with an abductive behavior for a concurrent logic program. Abductive Pandora programs are then transformed into Pandora programs which support the concurrent abductive behavior through a simple programming technique while at the same time taking advantage of the underlying Pandora machine infrastructure.

Keywords: Abduction, Distributed and Parallel Implementations, Non-monotonic and Hypothetical Reasoning, Concurrent Logic Programming Languages.

1. Introduction

Abduction was introduced by the philosopher Pierce as one of the three main forms of reasoning (the other two being deduction and induction). Recently, the importance of abductive reasoning has been demonstrated in many areas of Artificial Intelligence such as diagnosis, temporal reasoning, planning and semantic networks but also elsewhere such as in the field of databases, linguistics, and in the intelligent retrieval of information in the Web. In logic programming, in particular, abduction is achieved by means of finding conditional answers to queries. As a

result, it is useful to study ways for making the computation of abduction more effective. It has been argued[1] that abductive inference and its parallel realization should be one of the future research themes in parallel/distributed logic programming.

The development of an abductive framework in logic programming has been proposed[2,3] and further developed, among others,[4,5,6]. An abductive logic program is a triple **<P,A,I>** where **P** is a general logic program, **A** is a set of abducible atoms and **I** a set of constraints. For simplicity a number of restrictions are usually imposed: there are no rules for abducible atoms, integrity constraints are compiled into denials with at least one abducible and the hypotheses generated are variable free. In the abductive proof procedure for logic programming (see[5] for a review of the main ideas), the computation interleaves between *abductive* phases that generate and collect abductive hypotheses with *consistency* phases that incrementally check these hypotheses for consistency with respect to the integrity constraints. The operational semantics for sequential execution of abduction is well defined within this framework and has been used in building meta-interpreters on top of Prolog systems. However, as in usual deductive logic programming, abductive inference mechanisms have several sources of parallelism of many forms (OR-parallelism, independent and dependent AND-parallelism). The introduction of these forms of parallelism into an abductive logic program are examined,[7] the operational behavior of such a program enhanced with parallelism are studied, and its effect on the efficiency of execution compared with the corresponding sequential version are highlighted.

An alternative idea however has been proposed[8] within the general context of concurrent constraint programming,[9] which encompasses concurrent logic and constraint programming. The fundamental move is that in the case of a concurrent constraint program deadlocking, this is interpreted as a need to generate some hypothetical values for the benefit of the suspended agents, some of which should be able to resume execution. This generation of hypothetical values can then correspond to an abductive phase whereas the execution of agents in the ordinary way can correspond to the deductive phase. The possibility of modeling this framework using the concurrent language PARLOG[10] is explored[11]. The experience we have gained in that work has led to the development of the present framework that is described in this paper.

The rest of the paper is organized as follows: the next two sections set the stage by briefly introducing the reader to abduction and the concurrent logic language Pandora. The main part of the paper, comprising the following two sections, describes in detail the framework of modeling abduction in Pandora and relationship with related work. The paper ends with some conclusions and plans for future research.

2. Abductive Logic Programming

Abduction is reasoning to explanations for an observation using a known background theory about the domain of the observations. In many cases, together with the background theory, we also have a set of integrity constraints whose purpose is to restrict the possible abductive explanations. For example, the observation that "the grass is wet" can be explained either by the hypothesis "it rained" or by the hypothesis that "the sprinkler is on" using the known theory that "rain or sprinkler causes the grass to be wet". If in addition we know that "the ground is dry" then the associated integrity constraint that "raining implies that everything must be wet" renders the first explanation unacceptable.

In ALP an abductive logic program is a triple **<P,A,I>** where **P** (the known theory) is a general logic program, **A** (the hypotheses) is a set of abducible atoms and **I** a set of integrity constraints. The definition of abduction is then given as follows:

> Given an abductive logic program **<P,A,I>** and a goal (or observation) **G**, then $\Delta \subseteq \mathbf{A}$ is an abductive explanation for **G** iff: i) $\mathbf{P} \cup \Delta \models \mathbf{G}$ and $\mathbf{P} \cup \Delta$ satisfies the integrity constraints **I** where '|=' is given by some chosen underlying semantics for logic programming, and ii) $\mathbf{P} \cup \Delta$ satisfies **I** under some notion of integrity constraint satisfaction.

In the abductive proof procedure for logic programming,[2,6,5] the computation interleaves between *abductive* phases that generate and collect abductive hypotheses with *consistency* phases that incrementally check these hypotheses for consistency with respect to the integrity constraints. Consider the following example where **a**, **b**, **c** are abducibles and '**<-**' in **I** denotes negation; so for instance, '**<- a, s**' means '$\neg(\mathbf{a} \wedge \mathbf{s})$'.

P:	**p <- a, r.**	**I:**	**<- a, s.**
	p <- u, b.		**<- b, v.**
	r <- a.		**<- b, t.**
	s <- s1, s2.		
	s <- s3.		
	v <- not w.		
	w <- c.		
	t <- c.		

Assuming a Prolog-like evaluation order, the query <- **p** will reduce using the first clause to **<- a**, **r**. Consequently, **a** will be abduced and the computation will enter a consistency phase to satisfy the constraint **<- a**, **s**. During the consistency phase all rules for **s** will be tried with the aim to show their failure and hence the satisfaction of the constraint. Assuming that this is the case, **r** will then be reduced to the abducible **a**. This is already part of the hypotheses set Δ that we are trying to construct. The computation will thus end with $\Delta = \{\mathbf{a}\}$.

On backtracking, the second clause for **p** will be tried which, assuming that **u** will be evaluated successfully, it will then cause the abduction of **b** and the

commencing of a consistency phase for it. The constraint **<- b**, **v** requires the failure of **v**, and subsequently of **not w**, which causes the abduction of **c** and hence the extension of Δ to {**b**,**c**}. However, the evaluation of the second constraint **<- b**, **t** requires the absence of **c**, resulting in an overall failure to generate another solution (explanation) to the query **<- p** using the second rule for **p**.

3. Pandora

Pandora[12,13] is a non-deterministic parallel logic programming language, which provides stream AND-parallelism, committed-choice non-determinism, and don't-know non-determinism. Pandora belongs to the family of those logic languages that adhere to the Andorra model of computation[14] in that it tries to combine in an effective way the benefits of don't-care stream parallelism as advocated by the family of concurrent constraint languages[9] with the search capabilities of ordinary (sequential) Prolog. Other members of the Andorra family of logic languages include Andorra-I[15,16] and AKL.[17]

3.1. *Syntax of Pandora programs*

There are two types of relation in a Pandora program: don't-care relations and don't-know relations, corresponding to the kinds of OR non-determinism in logic programming.[12] Similar to Parlog relations,[10] a *don't-care relation* is defined by a procedure that comprises a sequence of universally quantified guarded Horn clauses, each of the form:

Head ← Guard | Body.

Head is the head of the clause, **Guard** and **Body** are conjunction of goals forming the guard and body of the clause, respectively. '←' denotes the implication operator, '**,**' is the parallel conjunction operator, and '|' is the commit operator. If **Guard** is an empty conjunction, the commit operator is omitted. If **Guard** is non-empty, then an empty conjunction in the body is denoted by the goal **true**. The implication operator is omitted when both **Guard** and **Body** are empty conjunctions.

Each procedure is preceded by a mode declaration denoting its input (**?**) and output arguments (**^**). An example of a don't-care relation definition follows below, where **list1**, and **list2** are input arguments, while **merge_list** is an output argument.

```
mode merge(list1 ?, list2 ?, merge_list ^).

merge([H1 | T1], [H2 | T2], [H1 | R]) ← H1 =< H2 : merge(T1, [H2 | T2], R).
merge([H1 | T1], [H2 | T2], [H2 | R]) ← H1 > H2 : merge([H1 | T1], T2, R).
merge([ ], Y, Y).
merge(X, [ ], X).
```

A *don't-know* relation is defined by a procedure that comprises a sequence of universally quantified guarded Horn clauses. The syntax of these clauses is similar to the clauses in a don't-care relation procedure, except for the implication operator which is denoted by '**:-**' (as in a Prolog procedure) instead of '←'. Moreover, the procedure defining a don't-know relation is not preceded by any declaration. The following piece of code defines the **between/4** don't-know relation:

between(X, Y, Z, [ ]) :- X > Z : true.
between(X, Y, Z, [Y]) :- X =< Y, Y =< Z : true.

3.2. *The Computational model of Pandora programs*

The Pandora computational model is related to Warren's Andorra model.[14] In Pandora, a non-deterministic choice is made after all deterministic computation, as well as committed-choice non-deterministic computation take place. Therefore, the computation alternates between two phases: the *AND-parallel phase* when goals are evaluated concurrently except for the don't-know non-deterministic ones which are delayed, and the *deadlock phase* in which an "arbitrary" non-deterministic goal can be reduced. A new AND-parallel phase then begins for each search branch of the non-deterministic choice.[12] A more detailed explanation follows.

(i) *Evaluating a Goal for a Don't-Care Relation*

A clause in a don't-care relation procedure is said to be a *candidate clause* for a goal if and only if the input unification and the guard evaluation succeed. Input unification is a one-way transfer of data, from a goal to a clause, whereas output unification is a transfer in the other direction. Input unification is applied on input arguments of the relation while output unification is applied on output arguments. Both input unification and guard evaluation must not result in instantiating unbound input arguments in a goal and would suspend until those arguments are sufficiently instantiated to match the corresponding arguments in the clause head. If the input unification and/or the guard evaluation fails, the clause is a *non-candidate clause*. Moreover, the clause is said to be *suspended* if neither the input unification nor the evaluation of the guard has failed and at least one of them is suspended.[12] Assuming the use of an OR-parallel search operator among the clauses of a relation, then the evaluation of a don't-care relation goal starts with an OR-parallel search for a candidate clause, and one of the following cases results:

(a) **all clauses in the procedure are non-candidates:**
The goal fails.

(b) **at least one clause is a candidate:**
The goal is reduced with an arbitrary candidate clause. That is, the goal is replaced in the current conjunction of goals by the goals in the body of the selected clause and the output arguments in the head of the clause are unified with the corresponding arguments of the goal.

(c) **there are no candidate clauses but there is at least one suspended clause:**
The goal suspends until it is reduced with a clause that becomes a candidate, or fails if all the suspended clauses become non-candidates.

(ii) *Evaluating a Goal for a Don't-Know Relation*

A clause in a don't-know relation procedure is said to be a *non-candidate clause* for a goal if and only if its head is not unifiable with the goal and/or it has a false (unsatisfiable) guard. In the AND-parallel phase, the evaluation of a don't-know relation goal starts with testing whether the goal is deterministic, i.e. it has at most one candidate clause, and one of the following cases results:

(a) **all clauses in the goal's procedure are non-candidates:**
The goal fails.

(b) **all clauses except one, say *Cj*, are non-candidates:**
The goal is reduced with clause *Cj*.

(c) **the goal is non-deterministic:**
It suspends until it becomes deterministic, in which case either case (a) or case (b) will be satisfied, or the deadlock phase is begun.

If the computation deadlocks, the default behavior is to select an arbitrary goal for a don't-know relation and create its choice point. For each search branch, a new AND-parallel phase is begun after reducing the goal with one of the clauses in its procedure.[12]

(iii) *Rewrite Rules*

Following the approach used for other Andorra based languages such as AKL,[17] it is possible to represent the execution of a Pandora program as a series of rewrites of goal expressions. These rewrites are defined by means of a set of rewrite rules, which for the case of Pandora is the one shown below. We use **A** to denote a single sub-goal, **G** and **B** to denote a group of sub-goals acting as guard and body of some clause respectively, **R**, **S** and **T** to denote AND-conjunctions, and **C** to denote a guard **G** after unification of the goal to be reduced with the head of the clause having **G** as a guard has taken place. Furthermore, **V** and **W** refer to sets of variable bindings created during unification.

Local forking

A => choice(and(G1)$_{V1}$|B1,…,and(Gn)$_{Vn}$|Bn)

This rule applies for a goal **A** to be reduced in a don't-care fashion when a number of candidate clauses are available.

Failure Propagation

and(R,fail,S) => fail

Choice Elimination

choice(R,fail|B,S) => choice(R,S)

Deterministic Promotion

and(R,choice(C_v|B),S)$_W$ => and(R,C,B,S)$_{V \cup W}$

Commit_Promotion

choice(R,C_v|B,S) => choice(C $_V$|B)
if $\mathbf{C}_v$ is satisfiable and quiet (i.e. it produces no output bindings).

Guard Distribution

choice(R,or(G,S)|B,T) => choice(R,G|B,or(S)|B,T)

Non-Deterministic Promotion

and(T1,choice(R, S),T2)$_W$ => or(and(T1,R,T2) $_W$,and(T1,choice(S),T2) $_W$)

The description of Pandora's execution in terms of the above set of rewrite rules will assist us in presenting the abductive Pandora framework in a more formal than free text description fashion by extending it with the machinery required to realize abduction, as the latter is perceived in the following section.

4. Abduction in Pandora

Abduction is Pandora can be realized by exploring the idea previously suggested[8] of giving a different interpretation to the case of a concurrent logic program reaching a deadlocked state. In ordinary concurrent logic languages, where programs execute in a don't-care committed choice fashion, a deadlocked state is considered as a failure to derive a solution. In the Andorra model, a deadlocked state is interpreted as the end of the current deterministic phase of the computation, to be followed by a non-deterministic one where different branches of the OR-tree are to be explored in parallel. This is also true for Pandora, being a member of the Andorra family of languages. In the abductive Pandora framework the deadlocked state is interpreted as the end of the current consistency (or deductive) phase to be followed by an abductive phase. In particular, one or more (input) arguments of some suspended process will be abduced (i.e. will be instantiated to the values that they had expected to receive during the ordinary deductive phase), thus allowing the computation to be resumed. The abductive phase effectively assumes that the selected abducibles have indeed been instantiated to their indicated values, thus activating all the processes that are suspended on them. This new framework is realized in a variant of Pandora which we will call Abductive Pandora.

4.1. *Abductive Pandora*

Abductive Pandora[18] is a syntactic sugar for the concurrent logic programming language Pandora, which allows the user to easily write an abductive logic program that will be executed concurrently. The program is then transformed to a corresponding Pandora program which supports the concurrent abductive behavior.

By structuring the program into some conceptually meaningful form (abducibles, constraints, and logic program), each such unit of information becomes easily accessible in the program. Ideally, all the details of implementation should be invisible to the user of the language - the programmer can just concentrate on the objects and the relations between them. Let us discuss the way of carrying out this principle in Abductive Pandora. The user writes his/her program, classifying it to three sections as the following:

Section 1 : **abducibles([a_1, a_2, …, a_n])**.
Section 2 : **constraints([[a_1, c_1], [a_2, c_2], …, [a_n, c_n]])**.
Section 3 : **ordinary logic programming relation definitions (in Prolog, or Parlog, or Pandora)**.

The first section contains a list of abducibles $a_1, a_2, \ldots, a_n$ which represents the set of abducible hypotheses that can be abduced when needed. The second section is a list of constraints which is represented by a set of sub-lists, each of them representing a constraint on one abducible hypothesis, that is in the head of the sub-list. For instance, the sub-list:

$[a_1, c_{11}, c_{12}, \ldots, c_{1n}] \quad n \geq 0$.

is equivalent to the integrity constraint:

$\leftarrow a_1, c_{11}, c_{12}, \ldots, c_{1n}$

The tail of the sub-list $c_{11}, c_{12}, \ldots, c_{1n}$ represents a conjunction of goals that cannot be true if a_1 is to be abduced. If there is more than one integrity constraint on the same abducible atom, then more than one sub-list would be present in the list of constraints, whose heads are the same abducible atom. The last section contains ordinary logic program clauses that can be written in either Prolog, or Parlog, or Pandora itself.

Example:

Let us consider the example of finding the mode of locomotion of certain animals or birds. We can write this program in Abductive Pandora, as follows:

```
abducibles = [animal(fifi), bird(fifi), ostrich(fifi)].
constraints = [[animal(fifi),human(fifi)], [bird(fifi), snake(fifi)],
               [ostrich(fifi), not bird(fifi)]].
locomotion(X, fly) :- bird(X), not ostrich(X).
locomotion(X, walk) :- animal(X), not snake(X).
locomotion(X, walk) :- ostrich(X).
locomotion(X, crawl) :- snake(X).
```

The program states that birds fly except ostriches and animals walk except snakes. It also states that ostriches walk and snakes crawl. It also includes the set of abducibles together with integrity constraints on them. The abductive behavior of this program is explained later on in this section.

4.2. *Mapping Abductive Pandora to Pandora*

As being previously explained, Abductive Pandora is a syntactic sugar on top of Pandora in order to simplify to the programmer his/her definition of an abductive

logic program. Programs in Abductive Pandora are then automatically transformed to Pandora programs that provide the expected concurrent abductive behavior.

In order to implement a concurrent abductive behavior using Pandora, we introduce the following programming technique:

(i) The abducible hypotheses will be represented as don't-know relations. This is because:
  (a) when the selected abducible hypothesis fails to satisfy its integrity constraints, the computation may not fail if an alternative abducible hypothesis is successfully checked with respect to its integrity constraints;
  (b) to make the system more complete, in other words, to produce all solutions (explanations) for the observation.

(ii) The integrity constraints will appear in the bodies of abducible clauses, and hence must be satisfied in order for the abducible atom in the head of the clause to be abduced.

(iii) The remaining relations in the problem will be implemented either as don't-care relationships if they are deterministic, or don't-care non-deterministic, or as don't-know relations if they are don't-know non-deterministic.

Taking the previous Abductive Pandora program as an example, our implementation would automatically transform it into the Pandora program that follows below. The last section of the Abductive Pandora program is copied as it is in the Pandora program. In this example, it only includes don't-know (Prolog) relations. We must note that the abducible hypotheses that appear in the abducible list and don't have constraints will not appear in the constraint list and they will appear in the corresponding Pandora program as facts.

```
animal(fifi) :- not human(fifi).                    (1)
bird(fifi) :- not snake(fifi).                      (2)
ostrich(fifi) :- not(not bird(fifi)).               (3)
locomotion(X, fly) :- bird(X), not ostrich(X).      (4)
locomotion(X, walk) :- animal(X), not snake(X).     (5)
locomotion(X, walk) :- ostrich(X).                  (6)
locomotion(X, crawl) :- snake(X).                   (7)
```

4.3. *Concurrent abductive behavior in Pandora*

Here we show in detail how an Abductive Pandora program can exhibit a concurrent abductive behavior in a way that can be supported by the underlying ordinary deductive implementation of Pandora:

(i) *Deterministic Abduction*

In the AND-parallel phase, if an abducible goal has only one clause unifiable with it, then abduction takes place deterministically during the AND-parallel phase, in

parallel with the reduction of other goals. The integrity constraints (if any) in the body of the abducible clause will then be executed in parallel with other goals.

(ii) *Non-Deterministic Abduction*

In the deadlock phase, an arbitrary suspended goal for a don't-know relation is selected for reduction. In case it is an abducible goal, then abduction takes place, and a set of alternative branches result. For each search branch, a new AND-parallel phase is begun, running all deterministic don't-know relation goals as well as reducible goals for don't-care relations until they are exhausted. As a result, abduction can either take place concurrently with the reduction of other goals (in case it is deterministic), or takes place separately in the deadlock phase. Integrity constraints are always checked during the AND-parallel phase.

If we set the goal **locomotion(fifi,Y)** with the previous abductive framework of the respective program, we can get the search tree as in figure 1. An Abductive Pandora system will evaluate the goal **locomotion(fifi,Y)** by first suspending during the AND-parallel phase. Then it will non-deterministically get reduced in the deadlock phase using clauses 4, 5, 6, and 7. For each search branch, an AND-parallel phase will begin in which all deterministic don't-know relation goals will run, as well as reducible goals for don't-care relations. The first branch contains a conjunction of two deterministic goals, which run in parallel, terminating in failure. The second branch consists of two deterministic goals, in which the first goal **animal(fifi)** is an abducible goal, then abduction takes place deterministically in the AND-parallel phase and the computation enters a consistency phase to satisfy the constraint **not human(fifi)**. The consistency phase will succeed and hence abduce the hypothesis **animal(fifi)**. The evaluation of the second goal **not snake(fifi)** will also succeed, resulting in overall success of this branch, producing the answer **Y = walk**. The third branch contains a deterministic abducible goal, which will be successfully abduced, with the answer **Y = walk**, while the last branch will fail. Thus we get the answer **Y** = **walk** based on the assumption that **animal(fifi)**, or **Y = walk** based on the assumption that **ostrich(fifi)**, and **bird(fifi)**.

(iii) *Rewrite Rules*

We now extend the set of rewrite rules that was used in the previous section to describe the Pandora computational model with the machinery required to model abduction. Effectively, the binding environment of each rewrite rule is enhanced with two additional items: a set of variables Δ, whose eventual set of bindings represents the answer sought, and a set of integrity constraints **IC** that must be satisfied for a potential rewrite to be valid. Here, instead of using $\mathbf{C}_v$ for representing the guard **G** after the unification of a goal **A** with **G**'s head, we use directly the derived binding θ.

Local forking

A => choice(and(G1)$_{V1}$|B1,…,and(Gn)$_{Vn}$|Bn ; Δ,IC)

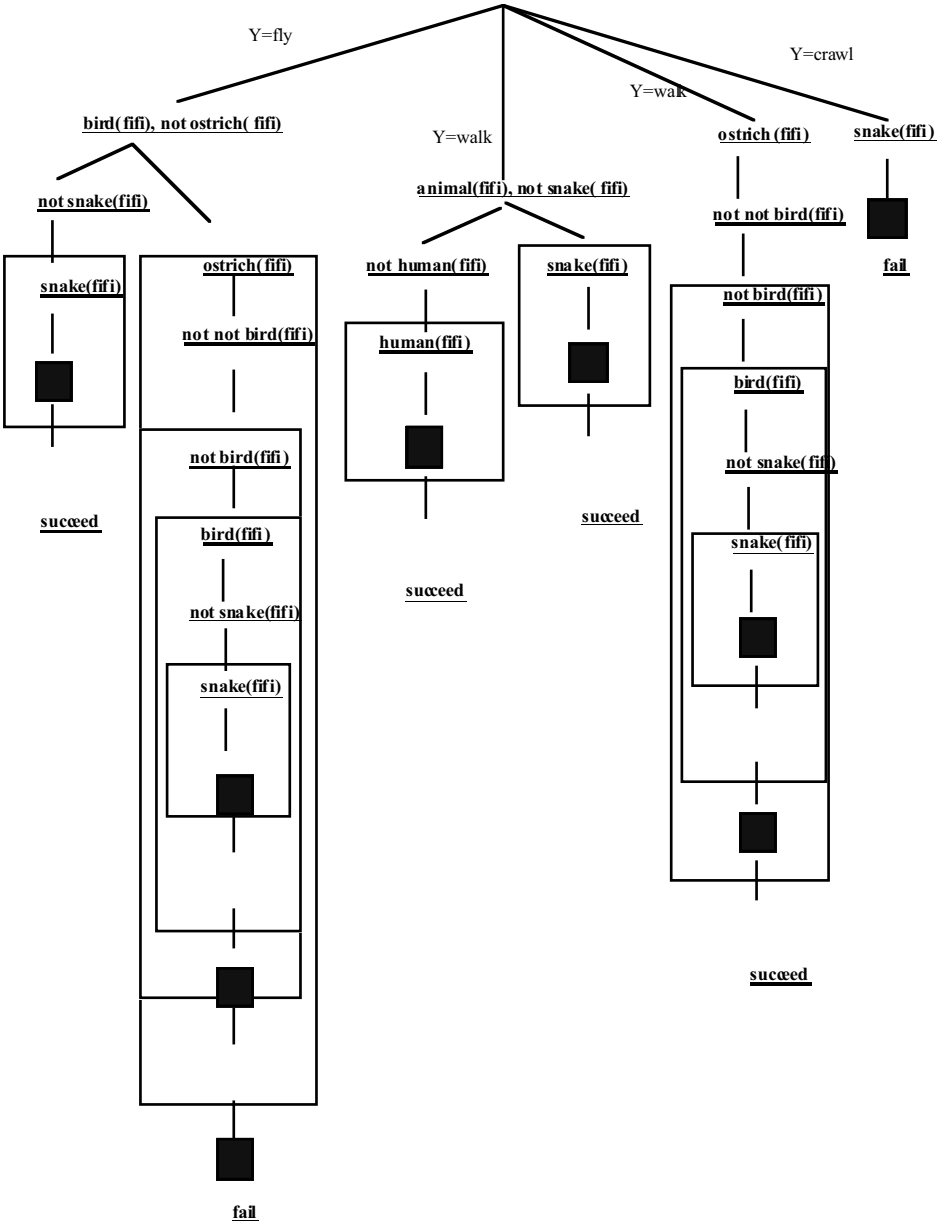

Fig. 1 Concurrent Abductive Behavior in Pandora

Failure Propagation

and(R,fail,S ; Δ,IC) => fail

Choice Elimination

choice(R,fail|B,S ; Δ,IC) => choice(R,S ; Δ,IC)

Deterministic Promotion

and(R,choice(θ$_v$|B),S ; Δ,IC)$_W$ => and(R,B,S ; Δ∧θ,IC∧θ)$_{V \cup W}$, if **IC**∧θ is satisfiable

Commit Promotion

choice(R, θ|B),S ; Δ,IC) => choice(B ; Δ∧θ,IC∧θ) , if **IC**∧θ is satisfiable

Guard Distribution

choice(R,or(G,S)|B,T ; Δ,IC) => choice(R,G|B,or(S)|B,T ; Δ,IC)

Non-Deterministic Promotion

and(T1,choice(R,S),T2 ; Δ,IC)$_W$ => or(and(T1,R,T2 ; Δ,IC)$_W$,
and(T1,choice(S),T2 ; Δ,IC)$_W$)

4.4. *Another example: "electrical circuits"*

Figure 2 presents an electrical circuit whose corresponding Abductive Pandora program follows below. This abductive application example was initially expressed in the general framework of concurrent constraint programming.[3,11] The electrical circuit includes a battery (**S**), three bulbs, and wires connecting them. The aim is that, based on observing the status of the bulbs (on or off), the program can abduce the possible explanations for such observation. Note that **battery**, **wire**, and **bulb** are abducible atoms, whose argument values may be assumed if necessary. Also for this example, there are no integrity constraints.

The operational abductive meaning of **wire**, for instance, is that if the computation includes the goal **wire(ok, plus, Out)**, then it can be deterministically abduced during the AND-parallel phase, with the variable **Out** being instantiated to the value of the second parameter (**plus**), whereas the goal **wire(broken, plus, Out)** is abduced with the variable **Out** being instantiated to **0**. On the other hand, the goal **wire(State, plus, Out)** will initially suspend. In the deadlock phase, this goal can be non-deterministically abduced either with the variables **State**, and **Out** being instantiated to the values **ok**, and **plus** respectively, or instantiated to the values **broken**, and **0** respectively.

The procedure **circuit** is, in fact, the one forming the actual circuit configuration where the last three arguments correspond to the observations that are made

regarding the state of the three bulbs ("on" or "off") and the rest denote the explanations that must be generated.

```
abducibles ([battery(empty, 0, 0), battery(ok, plus, plus),
             wire(ok, Connect, Connect), wire(broken, _, 0),
              bulb(ok, on, plus, plus), bulb(ok, off, 0, _),
             bulb(ok, off, _, 0), bulb(damaged, off, _, _)]).
constraints([ ]).

mode circuit(^, ^, ^, ^, ^, ^, ^, ^, ^, ^, ?, ?, ?).
circuit(S, B1, B2, B3, W1, W2, W3, W4, W5, W6, L1, L2, L3) ←
        battery(S, SL, SR),
        wire(W1, SL, B1L), wire(W2, SR, B1R), wire(W3, B1L, B2L),
        wire(W4, B1R, B2R), wire(W5, B2L, B3L), wire(W6, B2R, B3R),
        bulb(B1, L1, B1L, B1R), bulb(B2, L2, B2L, B2R),
        bulb(B3, L3, B3L, B3R).
```

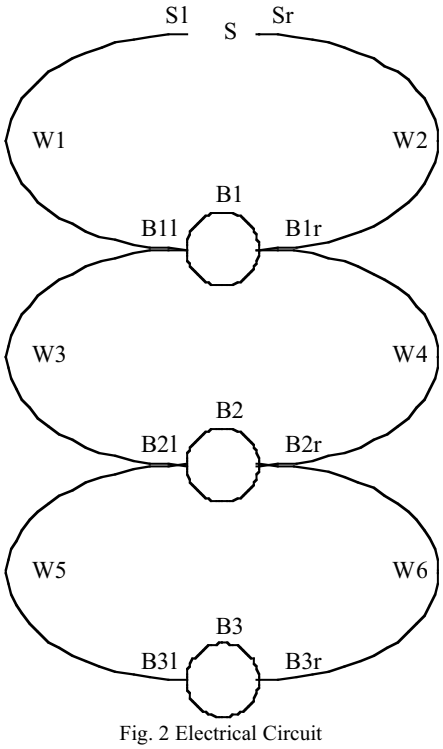

Fig. 2 Electrical Circuit

When we compile this program by our system, it will transform it to the following Pandora program:

battery(empty, 0, 0).
battery(ok, plus, plus).

wire(ok, Connect, Connect).
wire(broken, _, 0).

bulb(ok, on, plus, plus).
bulb(ok, off, 0, _).
bulb(ok, off, _, 0).
bulb(damaged, off, _, _).

mode circuit(^, ^, ^, ^, ^, ^, ^, ^, ^, ^, ?, ?, ?).
circuit(S, B1, **B**2, **B**3, **W**1, **W**2, **W**3, **W**4, **W**5, **W**6, **L**1, **L**2, **L**3) ←
battery(S, SL, **S**R**),**
wire(W1, **S**L, **B**1L**), wire(W**2, **S**R, **B**1R**), wire(W**3, **B**1L, **B**2L**),**
wire(W4, **B**1R, **B**2R**), wire(W**5, **B**2L, **B**3L**), wire(W**6, **B**2R, **B**3R**),**
bulb(B1, **L**1, **B**1L, **B**1R**), bulb(B**2, **L**2, **B**2L, **B**2R**),**
bulb(B3, **L**3, **B**3L, **B**3R**).**

Now, consider the following goal:
circuit(S, B1, **B**2, **B**3, **W**1, **W**2, **W**3, **W**4, **W**5, **W**6, **on, on, on)**.

This query will produce the single explanation:

[ok, ok, ok, ok, ok, ok, ok, ok, ok, ok].

Whereas, the following query:

circuit(S, B1, **B**2, **B**3, **W**1, **W**2, **W**3, **W**4, **W**5, **W**6, **off, off, off)**.
has multiple explanations such as:

[empty, ok, ok, ok, ok, ok, ok, ok, ok, ok],
[ok, damaged, ok, damaged, ok, ok, ok, broken, ok, ok],
[empty, damaged, damaged, damaged, broken, broken, ok, ok, ok, ok],
etc.

4.5. *A Semi-formal translation scheme*

Below we present a semi-formal algorithmic translation scheme of an Abductive Pandora program to its equivalent Pandora program:

Let **Abd** = list of abducibles defined in the Abductive Pandora program.
Let **Cons** = list of constraints lists defined in the Abductive Pandora program.

For each **A** ∈ **Abd** do:
get **C** = { $\mathbf{C_i}$ | $\mathbf{C_i}$ ∈ **Cons** s.t. $\mathbf{C_i}$ = **[A | Tail$_i$]**}
if **C** = [] then *% no integrity constraints on A*
produce_clause (**A**)
else
For each $\mathbf{C_i}$ ∈ **C**

produce_clause(**A :- Tail$_i$**)

For each clause $\mathbf{C_l}$ in the Abductive Pandora program copy $\mathbf{C_l}$ to the resulting Pandora program.

4.6. *Optimizing the abductive phase*

When a choice is given as to which predicate or argument of a predicate can be abduced, the decision that will be taken more often than not has a great effect on the efficiency of the execution. Finding the best combination of abducibles to abduce among the set of candidate ones is a popular area of current research in the field of abductive reasoning and its solution usually involves the use of heuristics. In our model we try to minimize the guesses taken and we choose the following simple heuristic: the process chosen to have its abducible parameters abduced is the one with the smallest number of abducible parameters that can still be abduced. We have enhanced our Pandora implementation with this policy that is invoked when the non-deterministic phase of the computation commences.

Furthermore, it is possible, for instance, to associate with each head argument in an abducible position an attribute indicating an assumption cost[19] or a probability value[20] as it is illustrated by the following piece of code:

abducibles ([battery(empty, 0.3:0, 0), battery(ok, 0.7:plus, plus), …])

The interpretation here is that in the goal **battery(empty, 0.3:0, 0)**, the second argument can be abduced with an assumption cost (or hypothesis probability) of 0.3 whereas in the goal **battery(ok, 0.7:plus, plus),** that argument's assumption cost is 0.7. The Pandora system can then use these attributes during the examination of the suspended state of the computation to decide which abducible(s) to abduce.

Last but not least the Pandora system also supports a user-defined meta-relation, called the *deadlock handler*, which gives the programmer a powerful way to handle deadlock according to the need of the application in hand. One way in which it can be used is in combination with the non-deterministic fork, to implement a heuristic search. This is a way to reduce the search space by intelligently selecting a specific don't-know non-deterministic goal to execute, when several ones are candidates. Another way is by deleting some suspended goals and adding new goals to the existing computation; a new AND-parallel phase is then started.

The use of the above techniques may help to alleviate a problem that most abductive models, including the one described in this paper, suffer from: the difficulty in computing minimal explanations and avoiding computing the same explanation multiple times.

5. Comparison With Related Work

As we have said at the beginning of the paper, although sequential abduction has received some attention, this is not also the case for parallel abduction. In a previous publication,[2] we examined the potential of introducing parallelism into abduction by exploiting all types of AND/OR parallelism as they are found in a typical parallel Prolog execution framework. We then presented an implementation framework for parallel abduction on top of the AKL system[17] and, as we did in this paper, we formalize the abductive execution mechanism by extending the rewrite rules system of the AKL execution framework with the machinery required to handle the

abductive phase. Both AKL and Pandora are members of the family of languages based on the Basic Andorra Model,[14] in which execution comprises of two alternating phases: the and-parallel phase where all deterministic and don't-care non-deterministic goals are eagerly evaluated, followed by a non-deterministic phase where a don't-know non-deterministic goal is evaluated by creating its choice point. Unlike other frameworks for integrating AND/OR parallelism in logic programming,[21,22,23,24] the model supports all types of AND and OR parallelism and suits very well the abductive behavior in which a literal is non-deterministically abduced only when the deductive phase and any deterministic abduction have terminated. Hence, unnecessary abduction is dramatically reduced. The fundamental difference between our work on top of AKL and the present one is in the way the abductive hypotheses set Δ is handled. In another publication,[7] Δ is empty at the beginning and it is gradually extended with abductive hypotheses. Once a certain hypothesis is added to the set, all concurrently executing processes should become aware of Δ's change and, if applicable, take immediately advantage of it. For instance, if Δ is extended by some process with the abductive hypothesis **a** then any other concurrently executing process which finds itself requiring the absence of **a** (i.e. the satisfaction of **not a**) can immediately terminate execution, thus stopping the system from performing unnecessary computations. Furthermore, if another process also wants to abduce **a**, then once it realizes that **a** has already been abduced it can carry on its execution without having to perform a consistency checking on the integrity constraints associated with **a** (since the process which has already abduced **a** will already have started such a consistency phase). Thus, the model implements an "eager" abduction policy coupled with an immediate notification of any changes in the hypotheses set. This tight synchronization between concurrently executing processes via the shared hypotheses set Δ saves a lot of unnecessary computation; however, we essentially rely on an atomic updating of the shared space Δ and we therefore base our model on Atomic AKL[17] whose sensible execution environment has to be one using shared memory. Recently,[25] an attempt has been made to map this model to a distributed architecture where Δ is viewed as a Linda-like shared data-space and is updated by means of Linda's **out**, **in** and **rd** primitives. However, in order to ensure consistency between the possibly conflicting hypotheses requirements generated by the concurrently executing processes, a high amount of communication must be performed between them, amounting to a potentially high overhead.

The current work is based on the proposal by Codognet and Saraswat[8] who introduce abduction to the concurrent constraint framework by re-interpreting a deadlocked situation as being a need to abduce some (input) constraints (rather than treating deadlock as an erroneous state as it is the case for ordinary concurrent constraint programming). Compared to the framework described in the previous paragraph, here a "lazy" abductive policy is used, in that we resort to the abductive phase only if the deductive phase has not been successful in reaching a conclusion. Furthermore, the hypotheses set Δ is predefined, comprising a set of (input) variables whose eventual bindings (either during the deductive or the abductive phase) constitute the required answer. We introduced these ideas to the concurrent logic language Parlog,[11] where we develop an abductive framework on top of that language. In that framework, a Parlog program is allowed to proceed with its

ordinary execution until a deadlocked state is reached. Then, an abductive phase commences during which a monitoring process collects all the frozen suspended states of each one of the concurrently (and now deadlocked) executing processes. Based on criteria essentially similar to the ones mentioned in section 4.6 above, it decides which input variables to "abduce" (i.e. instantiate them to the values that were expected by the processes that got suspended on them). This then leads to the re-commencing of the (ordinary deductive) computation until the next deadlocked phase is reached, and so forth. In the case where different choices are possible, computation splits into different OR-branches as required. In order for the frozen suspended states of the processes involved in some computation to be collected and examined, all processes are connected in a "left-hand-with-right-hand" manner using the well known short-circuit technique. The frozen state of some process is then passed to the monitoring process via the intermediate processes through the short circuit, the latter realized by a set of variables shared in twos between a pair of processes. Incidentally, another such short circuit is used to detect the presence of deadlock. This apparatus has the benefit of being implementable in a distributed system since the above mentioned activities are by nature decentralized and are performed by each process separately. Furthermore, the model does not depend on any specific concurrent logic language and can be implemented on top of almost all concurrent logic languages. However, the model is essentially interpretive; our 500 lines of Parlog code interpreter translates the original program to one enhanced by the short circuits apparatus and run under a meta abductive compiler, itself running on top of Parlog. The derived program is quite different from the original one and thus difficult to be understood and debugged. For instance, in the electrical circuits example of section 4.4, the 3-lines **wire** predicate would be translated in this model to the following code:

```
wire(LSC,RSC,m(L,R,ok),ConnectIn,ConnectOut)
        <- L=R, ConnectOut=ConnectIn.
wire(LSC,RSC,m(L,R,broken),_,ConnectOut) <- ConnectOut=m(L,R,0).

wire(LSC,RSC,m(L,R,ok),ConnectIn,ConnectOut)
         <- LSC=RSC, L=R, ConnectOut=ConnectIn.

wire(LSC,RSC,m(L,R,broken),_,ConnectOut)
         <- LSC=RSC, ConnectOut=m(L,R,0).

wire([collect_states(S)|LSC1],RSC,State,ConIn,ConOut) <-
     RSC=[collect_states(wire(LSC,RSC1,abd(State),ConIn,ConOut)|S])|RSC1],
        wire(LSC1,RSC1,State,ConIn,ConOut).
```

The work presented in this paper is clearly based on an approach initially implemented in a distributed fashion,[8,11] although the supported syntax is reminiscent more of how we would model abduction in an ordinary Prolog program. Moreover, here we use a semi-interpretive approach where the original concurrent logic program is compiled down to Pandora code, itself running on top of Parlog. The benefits of this approach is that the original program is left intact and that the abductive apparatus is completely handled by the Pandora system which has been

optimized to run on top of Parlog. Furthermore, the abductive phase, which is effectively handled by the Pandora run-time system, can potentially be compiled down to WAM code, thus enhancing further its efficiency. The potential of compiling Pandora code to WAM is further described in another publication.[12] Additionally, the Pandora system supports a user-defined meta relation which can be used to get feedback from the user in optimizing further the execution of an abductive program (see also discussion in section 4.6). Finally, the Pandora system offers the opportunity of introducing (and thus supporting) abduction in three languages, namely Pandora itself but also Parlog and ordinary Prolog.

6. Conclusions and Future Work

We presented a possible way to model abduction within the framework of concurrent logic programming, and presented Abductive Pandora, a syntactic sugar to the concurrent logic programming language Pandora, which allows a user to define an abductive concurrent logic program in an easy manner. We have concentrated on the Pandora language, which we believe, offers a number of characteristics suitable to our purpose, with its two types of relation and its novel execution model of alternating between two phases.

However, the user does not need to learn Pandora in order to use Abductive Pandora; he/she can program either in Prolog, Parlog, or Pandora, extended with a list of abducibles and a list of integrity constraints. Then our system will transform this program into the corresponding Pandora program. So, Abductive Pandora saves the user from learning the Pandora language to get the benefits of abduction, but, without loss of principle bases, the user can write his/her program with the language which he/she knows, but in a specific design. Moreover, even Prolog programs will run using the Pandora execution model, transparently extracting AND-parallelism whenever possible among deterministic goals while delaying non-deterministic ones until they become deterministic or until the deadlock phase.

As part of our future plans, the proposed model will be extended to handle non-ground abducible literals along the line of constructive abduction in logic programming.[5] It will then be necessary to incorporate in the language an enhanced deep-guard. Also the integration of the model with constraint logic programming[12,26] and the generalization of the types of integrity constraints that are supported needs further investigation and are topics of future research. We are also looking into the issue of compiling the Pandora abductive framework down to WAM code, thus enhancing further its efficiency. Finally, we are planning to test the effectiveness of our approach by using it to implement a number of applications that require abductive reasoning.

References

[1] K. Furukawa, contribution to *The Fifth Generation Project: Personal Perspectives*, eds. E. Shapiro and D. H. D. Warren, Communications of the ACM, (March 1993) 48-01.

[2] K. Eshghi and R. A. Kowalski, *Abduction through Deduction*, Technical Report, Department of Computer Science, Imperial College, London (1988).

[3] K. Eshghi and R. A. Kowalski, *Abduction Compared with Negation by Failure*, Proc. 6th International Conference on Logic Programming, Portugal, MIT Press (1989) 234-255.

[4] W. Chen and D. S. Warren, *Abductive Logic Programming*, Technical Report Department of Computer Science, State University of New York at Stony Brook (1989).

[5] A. C. Kakas, R. A. Kowalski and F. Toni, *Abductive Logic Programming*, Journal of Logic and Computation, Vol. 2 (6) (1992) 719-770.

[6] A. C. Kakas and P. Mancarella, *Generalised Stable Models: a Semantics for Abduction*, Proc. 9th European Conference on Artificial Intelligence, Stockholm, Sweden (1990) 385-391.

[7] A. C. Kakas and G. A. Papadopoulos, *Parallel Abduction in Logic Programming*, Proc. 1st International Symposium on Parallel Symbolic Computation, Linz, World Scientific Publishing (Sept. 1994) 214-224.

[8] P. Codognet and V. A. Saraswat, *Abduction in Concurrent Constraint Programming*, Technical Report, Xerox Palo Alto Research Center (1992).

[9] V. A. Saraswat, *Concurrent Constraint Programming*, ACM Doctoral Dissertation Award, MIT Press series on Logic Programming (1993).

[10] S. Gregory, *Parallel Logic Programming in PARLOG*, Addison-Wesley Publishing Company (1987).

[11] G. A. Papadopoulos, *Abductive Behaviour of Concurrent Logic Programs*, Proc. Third Golden West Conference on Intelligent Systems, ACM SIGART, Las Vegas, Kluwer Academic Publishers, Vol. 1 (1994) 27-38.

[12] R. M. Bahgat *Non-deterministic Concurrent Logic Programming in PANDORA*, World Scientific Series in Computer Science, Vol. 37, Singapore (1993).

[13] R. M. Bahgat and S. Gregory, *Pandora: Non-deterministic Parallel Logic Programming*, Proc. 6th International Conference on Logic Programming, Lisbon (1989) 471-486.

[14] D. H. D. Warren, *The Andorra Principle*, Gigalips Workshop, University of Bristol, Bristol, England (1987).

[15] R. Yang, *Programming in Andorra-I*, Technical Report, Department of Computer Science, University of Bristol, Bristol, England (1988).

[16] R. Yang, *Solving Simple Substitution Ciphers in Andorra-I*, Proc. 6th International Conference on Logic Programming, Lisbon, Cambridge, Mass., MIT Press (1989) 113-128.

[17] S. Janson and S. Haridi, *Programming Paradigms of the Andorra Kernel Language*, Proc. International Symposium in Logic Programming, San Diego, MIT Press (1991) 167-183.

[18] O. M. M. Ismail, *Concurrent Abductive Logic Programming based on the Andorra Family of Logic Programming*, M.Sc. Thesis, Institute of Statistical Studies and Research, Department of Computing & Information Sciences, Cairo University, Egypt (1997).

[19] A. Wearn, *Reactive Abduction*, Proc. 10th European Conference on Artificial Intelligence, Vienna, Austria (1992) 159-163.

[20] D. Poole, *Probabilistic Horn Abduction and Bayesian Networks*, Artificial Intelligence 64 (1993) 81-129.

[21] U. Baron, J. C. De Kergommeaux, et al., *The Parallel ECRC Prolog System PEPSys: An overview and evaluation results*, Proc. Int. Conf. Fifth Generation Computer Systems, Tokyo, Japan 1988 841-850.

[22] B. Ramkumar and L. V. Kale, *Machine Independent And and OR Parallel Execution of Logic Programs: Part I – the Binding Environment, Part II – Compiled Execution*, IEEE Trans. on Parallel and Distributed Systems, Vol. 5, No. 2, February 1994.

[23] J. S. Conery, *The OPAL Machine*, Implementations of Distributed Prolog, ed. Peter Kacsuck and Michael J. Wise, Wiley 1992 159-182.

[24] C. Voliotis, A. Thanos, N. Sgouros, and G. Papakonstantinou, *Daffodil: A Framework for Integrating AND/OR Parallelism*, Proc. 5th Hellenic Conference on Informatics, Athens, Greece (1995) http://www.sena.gr/epy

[25] A. Ciampolini, E. Lamma, P. Mello and C. Stefanelli, *Abductive Coordination for Logic Agents*, Proc. 14th ACM Symposium on Applied Computing (SAC'99), San Antonio, Texas, ACM Press (1999) 134-140.

[26] A. C. Kakas and A. Michael, *Integrating Abductive and Constraint Logic Programming*, Proc. Twelfth International Conference on Logic Programming, MIT Press (1995) 399-417.