© Third Golden West International Conference on Intelligent Systems (GWIC'94), *ACM SIGART, Las Vegas, USA, 6-8 June, 1994, Kluwer Academic Publishers, Vol. 1, pp. 27-38.*

ABDUCTIVE BEHAVIOUR OF CONCURRENT LOGIC PROGRAMS

GEORGE A. PAPADOPOULOS
Department of Computer Science, University of Cyprus, 75 Kallipoleos Str., Nicosia, T.T. 134, P.O. Box 537, Cyprus. EMAIL: george@turing.cs.ucy.ac.cy

Abstract

We present a possible way to model abduction within the framework of concurrent logic programming. In particular, we describe an extension to the concurrent logic programming language PARLOG and an associated computational model which allows a user to specify an abductive behaviour for a concurrent logic program. The proposed model therefore exploits the inherent parallelism of concurrent logic programs. We discuss some of the problems we faced and the solutions we adopted. A prototype implementation of the model has been built and is also discussed.

Keywords. Abductive Logic Programming, Concurrent Logic Programming.

1 Abduction

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 and linguistics. 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 ([5]) that abductive inference and its parallel realisation should be one of the future research themes in parallel logic programming.

The development of an abductive framework in logic programming has been proposed in [4] and further developed, among others, in [7,8]. 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 [7] 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). In [9] we examine the introduction of these forms of parallelism into an abductive logic program, we study the operational behaviour of such a program enhanced with parallelism and we highlight its effect on the efficiency of execution compared with the corresponding sequential version. An alternative idea however has been proposed in [2] within the general context of concurrent constraint programming 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.

In this paper we apply this idea to the class of the so called concurrent logic languages ([15]) and in particular the language PARLOG ([6]). More to the point, we extend PARLOG with suitable annotations to indicate abductive behaviour and we show how a concurrent logic program can exhibit such an abductive behaviour in a way that can be supported by the underlying "ordinary" deductive implementation. The rest of the paper is organised as follows: the next section introduces *abductive PARLOG* and shows how abductive PARLOG programs can be coded up in this extended language whereas the third section discusses a prototype implementation of the extended language on top of an ordinary PARLOG system; the final section comprises the conclusions and suggestions for further work.

2 Abductive behaviour of a concurrent logic program

Concurrent logic languages ([15]) have enjoyed a widespread use and have undergone major development over the past decade. A concurrent logic program is a set of guarded Horn clauses of the form

```
H :- G1,…,Gm | B1,…,Bn          m,n ≥ 0
```

where '`H`' is the head, '`|`' is the commit operator, '`G1,…,Gm`' is the guard part and '`B1,…,Bn`' is the body part. Declaratively, the meaning of the above clause is that `H` is true if both `G1,…,Gm` and `B1,…,Bn` are true. Operationally, the guard calls `G1` to `Gm` are evaluated first in parallel and upon successful termination the computation commits to the body of the clause. The head `H` is of the form `p(t1,…,tn)` where `p/n` is a predicate name of arity `n` and `t1,…,tn` are its arguments. There may be more than one rule with the same name `p` and arity `n`, in which case they form a group definition of the process `p`.

Computation starts with a set of cooperating processes (goals) executing in parallel and communicating by means of shared variables. The clauses of a program specify the behaviour and the various transitions possible for each process. Provided that for a certain goal to be reduced there are more than one candidate clauses to select from, the first one to perform head unification and solve its guard successfully will be chosen; the computations in the head or guard of the other candidate clauses will then be abandoned. Thus concurrent logic languages incorporate the concept of committed choice "don't care" non determinism from CSP.

In this paper we show how the notion of abduction can be modelled in the framework of concurrent logic programming by applying the ideas proposed in [2] to the case of the concurrent logic programming language PARLOG ([6]). In PARLOG the clauses that form a procedure are associated with a mode declaration that states which arguments are input (i.e. the values specified in the head of one of the procedure's clauses have to be present for that clause to be candidate for selection) and which are output (i.e. the corresponding values will be produced upon commitment to that clause). We now extend the language with a third mode annotation, '`@`', which states that the corresponding argument can be abduced. In particular, a PARLOG clause for a procedure `p/n` now takes the form

```
p(i1?,…,ik?,ab1@,…,abm@,o1^,…,on^) :- Guard | Body
```

where the arguments `i1` to `ik` are input arguments, `o1` to `on` are output arguments and `ab1` to `abm` are abducible arguments whose value *can be assumed* if the need arises (i.e. if they are not present and the clause cannot be reduced). As an example the arguments of the following procedure

```
equipment(ok@,Signals^) :- produce_signals(Signals).
equipment(not_ok@,Signals^) :- error_condition(Signals).
```

comprise an abducible parameter and an output one.

As for the case of the traditional abductive logic programming framework, computation interleaves between an ordinary deductive phase and an abductive one. For the case of a PARLOG program, the deductive phase corresponds to the performing of the input unifications, guard evaluations and reductions to the bodies of the selected clauses. The need for an abductive phase emerges when the whole computation has suspended with no remaining process able to reduce. In this case an "abductive" phase can be initiated whose purpose is to abduce one or more of the abducible arguments of some suspended process so that the computation can resume. The abductive phase effectively assumes that the selected abducibles have indeed been instantiated to their indicated values, thus activating all the processes that are suspending on them. Considering again the above example, assume the existence of the clause

```
monitor_equipment(ok?)  :-  …
monitor_equipment(not_ok?)  :-  …
```

and the following suspended AND-conjunction

```
…,  monitor_equipment(Status),  equipment(Status,Signals),  …
```

After the suspension of the computation, the abductive phase should select the suspended literal `equipment` and assume either of the two values `ok` or `not_ok` by instantiating the variable `Status` to either of them. This will now activate the process `monitor_equipment` and ordinary deductive computation will resume. If it is discovered in retrospect that the assumption was wrong, the second of the two values should be tried. Computation will either succeed if an assumed value succeeds in causing the successful termination of all processes involved in the computation or fail if all assumptions lead to failed derivations.

The following example, taken from [2] and adapted to the syntax of abductive PARLOG, illustrates a configuration comprising three light bulbs connected in sequence to a battery. Note that the first argument of the procedures `battery`, `wire` and `bulb` that denote the state of the corresponding component are abducible arguments whose value may be assumed if necessary. The operational meaning of the procedure `wire`, for instance, is that the call `wire(ok,plus,Out)` is reduced with the variable `Out` being instantiated to the value of the second parameter (`plus`) whereas the call `wire(broken,plus,Out)` is reduced with the variable `Out` being instantiated to `0`. On the other hand the call `wire(State,In,Out)` will initially suspend; if however the computation eventually enters an abductive phase, it is possible to abduce the first argument, i.e. instantiate `State` to either `ok` or `broken` and continue from that point onwards. The procedure `circuit` is, in fact, the one forming the actual circuit configuration where the last three arguments correspond to the observations that were made regarding the state of the three bulbs and the rest denote the explanations that must be generated.

```
mode  battery(charging_level@,left_wire^,right_wire^).
battery(empty,0,0).
battery(ok,plus,plus).

mode  wire(state@,left_connection?,right_connection^).
wire(ok,Connect,Connect).
wire(broken,_,0).

mode  bulb(condition@,light^,left_wire?,right_wire?).
bulb(ok,on,plus,plus).
bulb(ok,off,0,_).
bulb(ok,off,_,0).
bulb(damaged,off,_,_).
```

```
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).
```

The way a query is formulated in abductive PARLOG is illustrated below.

```
<- abductive_parlog(Goal,Constraints),
   Goal=[circuit(S,B1,B2,B3,W1,W2,W3,W4,W5,W6,on,on,on),
         print([S,B1,B2,B3])],
   Constraints=[…].
```

where the role of `Constraints` will be explained later on. Incidentally, note that the above query produces the single explanation `[ok,ok,ok,ok]` whereas the following one

```
<- abductive_parlog(Goal,Constraints),
   Goal=[circuit(S,B1,B2,B3,W1,W2,W3,W4,W5,W6,off,off,off),
         print([S,B1,B2,B3])],
   Constraints=[…].
```

has multiple explanations such as `[empty,ok,ok,ok]`, `[empty,damaged,broken,ok]`, `[ok,damaged,broken,broken]`, etc.

Note that by extending PARLOG with the abductive mode annotation '`@`', we have allowed a programmer to specify in a procedure which arguments, if any, can be abduced, thus controlling the extent to which abduction will be performed and consequently reducing the search space. Later on we discuss the possibility of enhancing an abductive mode annotation with extra information that will help in selecting during the abductive phase the best candidate for abduction from the suspended processes. Note also that a procedure mode declaration without any abductive annotations cannot possibly take part in the abductive phase.

3 Implementation of abductive PARLOG

In implementing the abductive extensions to PARLOG, the following problems must be solved:

— Detection of a globally suspended state of computation that indicates the end of the current deductive phase. Note that in general the processes involved in a computation are scattered around in a parallel system, or even a distributed one over a number of machines. Note also that for an ordinary concurrent logic program a globally suspended state is, in fact, an erroneous situation because it indicates deadlock.

— Upon detection of a suspended state, a decision must be taken as to which abducibles will be abduced. The decision is critical in computing efficiently the solutions since "bad guesses" will lead to unnecessary computations.

— If the abduction of some abducible(s) proves to be unsuccessful in deriving a solution (or more solutions are sought) the computation should backtrack to the point before the abduction and try a different path. Note here that an ordinary concurrent logic language implementation supporting only committed choice "don't care" parallelism has no machinery to backtrack.

In the sequel we discuss the solutions we adopted for each one of the above points, showing in the process how the original program is transformed into one having the additional functionality required for supporting an abductive behaviour. We describe also the

most important parts of the implementation itself resorting to the use of '…' to hide details which could obscure the presentation of the model.

3.1 DETECTION OF DEADLOCK

In solving the first problem we are interested in a solution which requires no extra machinery in the underlying PARLOG implementation (namely a suitably modified WAM) and which can work even in the case where an abductive PARLOG program may be running in a distributed environment. The solution we have adopted is based on an *all-declarative* approach using the power of short circuits, a programming technique first introduced by Takeuchi and used, among other applications, for detecting deadlock and termination or getting snapshots of systems of processes running concurrently ([14,15]).

In particular, all messages produced by a concurrently executing system of processes are connected by means of a short circuit, the ends of which are held by a monitoring process. Upon receiving and consuming a signal, a process unifies the left and right switches of the circuit for the particular message. If a process produces more messages that what it consumes, it extends the circuit accordingly. Deadlock is detected when the monitoring process observes that the two ends of the circuit that it holds have been unified, i.e. no more messages are in transit ([15]). We illustrate how an abductive PARLOG procedure can support this functionality by showing the transformed version of `wire`.

```
mode  wire(state@,left_connection?,right_connection^).
wire(m(L,R,ok),ConnectIn,ConnectOut)  <-
   L=R,  ConnectOut=ConnectIn.
wire(m(L,R,broken),_,ConnectOut)  <-  ConnectOut=m(L,R,0).
```

Note that while in the first clause we close the switch because we absorb a message (`ok`), in the second we simply propagate the switch from the consumed message (`broken`) to the produced one (`0`). In general, for every abductive or input argument of a procedure there exists a clause enhanced with the functionality just described. The monitoring process is of the form

```
mode  monitor(messSCL?,messSCR?,…).
monitor(MessSC,MessSC,…)  <-  resolve deadlock
```

where its exact functionality will be described later on.

3.2 ABDUCTIVE PHASE

The second phase comprises two parts: first, the deadlocked global state of the computation must be examined and second, a decision must be taken as to which abducible argument(s) must be abduced.

3.2.1 Examining the Deadlocked Global State of the Computation. The global state of a set of concurrently executing processes can be examined by using again techniques based on short circuits, in particular the one for collecting snapshots of the executing processes ([14]). More to the point, all processes involved in a computation are connected by means of left and right switches which, collectively, form a short circuit. When the monitoring process detects deadlock it sends the message `collect_states([])` to all the deadlocked processes using one of the ends of the short circuit and then it waits for the message to reappear at the other end. Each deadlocked process that receives the message via one of its local switches propagates it to the rest of the processes using the other switch after enhancing it with the current values of its arguments (i.e. its state). Note that because this mechanism is initiated after the detection of deadlock it is not possible for a process to change its state after it has reported it to the monitoring process by means of the `collect_states` message. The exact way a process's state is represented within a `collect_states` message is

shown below for the case of the procedure `wire` which, when transformed, must now be extended accordingly to handle this new functionality.

```
mode wire(leftS?,rightS?,state@,l_connect?,r_connect^).
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(LSC1,RSC1,abd(State),
                           ConIn,ConOut)|S])|RSC1],
  wire(LSC1,RSC1,State,ConIn,ConOut).
```

Each abducible argument is indicated by means of enclosing it into the structure `abd` whereas the rest of the arguments (input and output) are included as they are. Note that upon terminating, a process closes both circuits (the one for messages and the one for processes); had it been reduced to a number of processes it would have splitted the corresponding circuit accordingly. In general, a transformed procedure comprises two additional arguments (the left and right local switches of the processes' short circuit and an extra clause for reporting its state to the monitoring process. We now describe the monitoring process in more detail noting its enhancement with the second short circuit.

```
mode monitor(procSCL?,procSCR?,procmessSCL?,messSCR?,…).
monitor(PSC,PSC,_,_,…) <- end computation
monitor(PSCL,PSCR,MessSC,MessSC,…) <-
  PSCL=[[collect_states([])|PSCL1],
  monitor_wait(PSCL1,PSCR,…).

mode monitor_wait(procSCL?,procSCR?,…).
monitor_wait(PSCL,[collect_states(States)|PSCR],…) <-
  resolve_deadlock(States,NStateAbd,NStateRest),
  next_deductive_phase(PSCL,PSCR,NStateAbd,NStateRest,…).
```

The first clause detects the end of the computation by checking whether the left and right ends of the short circuit for processes have been unified. Similarly, the second detects deadlock, i.e. the end of the current deductive phase, and invokes the mechanism for examining the current state of the computation by sending the triggering mechanism `collect_states([])` to all suspended processes and then invoking the process `monitor_wait` which waits for the message `collect_states` to reappear with a snapshot of the system of deadlocked processes. The snapshot is then passed to the process `resolve_deadlock` which examines the state of the computation and decides which abducibles to abduce. This will lead to the instantiation of some abducible variables and the formation of a new state of computation which is passed to `next_deductive_phase` for the third phase of the computation.

3.2.2 Deciding What to Abduce. The top level definition of `resolve_deadlock` is shown below.

```
mode resolve_deadlock(state?,selected_abd^,n_state_rest^).
resolve_deadlock(State,SelectAbds,NStateRest) <-
  filter_states(State,CandidateAbds,RestState),
  abduce(CandidateAbds,RestState,Select_Abds,NStateRest).

mode filter_states(state?,selected_abds^,rest_of_state^).
filter_states(…) <- ….
```

```
mode  abduce(list_abds?,rest_goals?,sel_abd^,n_state_rest^).
abduce(CandAbds,RestState,Sel_Abds,NStateRest)  <-  ….
```

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 minimise 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. Furthermore, if a number of processes satisfy the above heuristics, those for which reduction is deterministic (i.e. only one of a process's defining clauses can be selected to reduce that process) are given preference. The ability to use more sophisticated heuristics is discussed later on.

More to the point, the process `filter_states` is responsible for filtering from the AND-conjunction of suspended goals those that cannot be abduced. A goal that cannot be abduced is one comprising non-abducible parameters or ground abducible parameters (namely parameters that have already been abduced). Here the structure `abd` mentioned earlier on assists in the filtering process. That part of the suspended AND-conjunction comprising the literals that can be abduced is then passed to the process `abduce` which forms the new state. This latter process is responsible for selecting those abducibles which will actually be abduced using the heuristics discussed already. As an example, if the snapshot of the deadlocked state is

```
States=[p(abd(X),3),q(abd(Y),Z),r(X,W),s(abd(2),Z,W)]
```

then `filter_states` will instantiate its two output arguments as follows.

```
CandidateAbs=[p(abd(X),3),q(abd(Y),Z)],
RestState=[r(X,W),s(abd(2),Z,W)]
```

Assuming that the reduction of `p` is deterministic, `abduce` will instantiate its two output arguments as follows.

```
SelectedAbs=[p(abd(X),3)],
NewStateRest=[q(abd(Y),Z),r(X,W),s(abd(2),Z,W)]
```

3.3 GENERATING NEW DEDUCTIVE PHASES

The process `next_deductive_phase` is responsible for implementing the third phase, i.e. the initiation of new deductive phases corresponding to the abduction of the selected abducibles. There may be more than one such deductive phase attributed to the possibility of producing multiple explanations. In order to be able to produce multiple explanations every new deductive phase must be executed as a different OR-branch. The top level machinery to do that is shown below and it is an adaptation of well known techniques for implementing OR-parallel interpreters in concurrent logic languages ([15]) using operations such as *copy*, *freeze* and *melt*. In particular, for each different set of selected abducibles a new deductive phase commences with a copy of the generated new state of computation whereas the old suspended state is saved. Upon failure (or need for further explanations), the system backtracks to the saved state and another copy is constructed and executed, this time with a different set of selected abducibles.

```
mode  next_deductive_phase(prSCL?,prSCR?,abds?,rest?,…).
next_deductive_phase(SCL,SCR,NewStateAbd,NStateRest,…)  <-
  get_clauses(NewStateAbd,Clauses),
  generate_explanations(SCL,SCR,NewStateAbd,
                        Clauses,NStateRest,…).
```

```
mode  generate_explanations(pSCL?,pSCR?,sel_abs?,
                            cls?,rest_state?,…).
generate_explanations(SCL,SCR,Abd,[Cl|Cls],RestState,…)  <-
  generate_explanation(SCL,SCR,Abd,Cl,RestState,…),
  generate_explanations(SCL,SCR,Abd,Cls,RestState,…).

mode  generate_explanation(pSCL?,pSCR?,sel_abs?,
                           clause?,rest_state?,…)  <-
generate_explanation(SCL,SCR,Abds,Clause,RestState,…)  <-
  form_query(SCL,SCR,Abds,Clause,RestState,…,Query),
  run_query(Query,…).

mode  form_query(pSCL?,pSCR?,sel_abs?,cls?,
                 rest_state?,query^…).
form_query(SCL,SCR,Abds,Clause,RestState,…,Query)  <-  ….

mode  run_query(query?,…).
run_query([ProcSCL,ProcSCR,MessSCL,MessSCR,Query],…)  <-
  Query,  monitor(ProcSCL,ProcSCR,MessSCL,MessSCR,…).
```

Note that `form_query` is responsible for setting up the mechanisms for detecting deadlock and examining the global state of the computation (i.e. forming new short-circuits for messages and processes).

3.4 TOP-LEVEL QUERY AND USE OF CONSTRAINTS

The top-level predicate `abductive_parlog(Goal,Constraints)` commences the computation and it is defined as follows.

```
mode  abductive_parlog(query?,constraints?).
abductive_parlog(Query,Constraints)  <-
  run_query([PSCL,PSCR,MessSCL,MessSCR,Query,Constraints]
            ,…).
```

The parameter `Constraints` which is effectively added to the first AND-conjunction is used to restrict even more the extent to which the system resorts to abduction in breaking the deadlock. This parameter states the conditions under which abductive hypotheses can be generated. The simplest form of such *integrity constraints* is negation combined with the restriction that hypotheses should be variable free ([7,8]). As an example, consider the following query.

```
<-  abductive_parlog(Goal,Constraints),
     Goal=[circuit(S,B1,B2,B3,W1,W2,W3,W4,W5,W6,off,on,off),
           print([S,B1,B2,B3])],
     Constraints=[not((S=empty,B1=on)),not((S=empty,B2=on)),
                  not((S=empty,B3=on))].
```

The user has imposed the constraint that if at least one bulb is observed to be working then it is not possible for the battery to be empty. The inclusion of these constraints into the computation reduces considerably the search space since those abducible values that do not satisfy the constraints will not be tried.

3.5 OTHER HEURISTIC MECHANISMS

It is important to note that, if desired, the model can be enhanced with other heuristics. It is possible, for instance, to associate with each head argument in an abducible position an attribute indicating an assumption cost ([16]) or a probability value ([12]) as it is illustrated by the following piece of code.

```
p(?,@,…).
p(1,a:0.3,…)  <-  ….
p(2,b:0.7,…)  <-  ….
```

The interpretation here is that in the goal `p(1,X,…),X` can be abduced with an assumption cost (or hypothesis probability) of 0.3 whereas in the goal `p(2,X,…),X`'s assumption cost is 0.7. The process `resolve_deadlock` can then use these attributes during the examination of the suspended state of the computation to decide which abducible(s) to abduce.

The extension of the model with heuristics such as the ones just described 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.

4 Discussion. Related and further work

The relationship between abduction and constraint programming is well known and has been explored in a number of papers (eg. [2,10,11]). In constraint programming the computation interleaves between a constraint generation phase where a variable is instantiated to one of a set of possible values and a constraint satisfaction phase where it is checked whether the chosen value satisfies the imposed constraints; if it does not then another one must be tried. In abduction the computation interleaves between a hypotheses generation phase and a consistency phase that checks whether a newly generated hypothesis is consistent with the rest of the existing hypotheses or imposed constraints. In fact [10] claims that constraint solving is a subcase of abduction. Hence, many of the techniques devised for constraint satisfaction are applicable also to the case of abduction.

In particular, the extension of PARLOG with an abductive mechanism along the lines of the model proposed in [2] leads to a framework similar in many respects to the Pandora model ([1]) which combines stream AND-parallelism with OR-search (parallel or sequential) and is particularly suited to constraint satisfaction problems. However, there are some important differences between the two models especially with regard to the detection and resolving of deadlock. Pandora relies heavily on the use of PARLOG's (extended) metacall whereas our technique using short circuits is effectively independent of any particular underlying implementation mechanism; in fact our high-level transformation techniques could well be applied easily to programs written in other similar concurrent logic languages such as GHC or FCP ([15]). Nevertheless, Pandora could be used as a basis for developing an efficient sequential implementation of abductive PARLOG with minimal effort.

As we have stated already we were particularly interested in developing an abductive framework based on concurrent logic programming especially suited to distributed environments. The kind of applications we have in mind is multi agent systems, distributed expert and medical diagnosis systems ([3]) and diagnosis systems requiring reactive properties ([16]). We are currently extending the definition of `resolve_deadlock` to include the functionality described in the paragraph 3.5.

Regarding the *abductive policy* our model is adopting there are two issues involved ([10]): i) which relations are allowed to have abducible instances, and ii) which instances of an abducible relation can in fact be abducible. In abductive PARLOG a relation is abducible if it contains at least one abducible mode declaration. Furthermore, all instances of that relation (along its abducible arguments) are abducible. If desired however, the model could

be extended to restrict abduction to particular instances of a clause by dispensing with the abductive mode declaration and moving the abducible operator '@' to the head of particular clauses of the relation. In the following relation, for instance,

```
mode  p(?,?,…).
p(@X,3,…)  <-  ….
p(X,4)  <-  ….
```

the first argument of p can be abduced only if the second one is instantiated to 3. The use of constraints in the top-level call abductive_parlog however effectively provides this functionality. Note here that the abductive mechanism we have described subsumes the (extended) four arguments abductive operator '@' proposed in [10].

5 Conclusions

We have presented a way to model abductive behaviour within the framework of concurrent logic programming by advocating the idea, first proposed for the general framework of concurrent constraint programming languages, of abducing arguments rather than predicates. The model has been tested by means of a meta interpreter written on top of the language PARLOG; however any other typical concurrent logic language could be used instead. The model inherits the high degree of concurrency enjoyed by concurrent logic languages and it is particularly suited to distributed environments.

Acknowledgements

I am grateful to Antonis Kakas for introducing me to the field of abductive reasoning and explaining its central concepts, and to Reem Bahgat for pointing out the similarities in concurrent logic languages between abductive behaviour and constraint satisfaction and explaining some key properties of the Pandora model.

References

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

[2] P. Codogent and V. A. Saraswat (1992), 'Abduction in Concurrent Constraint Programming', Technical Report, Xerox Palo Alto Research Centre.

[3] A. Eliëns (1991), 'Distributed Logic Programming for Artificial Intelligence', *AI Communications*, Vol. 4, No. 1, pp. 11-21.

[4] K. Eshghi and R. A. Kowalski (1989), 'Abduction Compared With Negation By Failure', *Proc. 6th International Conference on Logic Programming,* Lisbon, Portugal, pp. 234-255.

[5] K. Furukawa (1993), contribution to 'The Fifth Generation Project: Personal Perspectives', eds. E. Shapiro and D. H. D. Warren, *Communications of the ACM*, March, pp. 48-101.

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

[7] A. C. Kakas, R. A. Kowalski and F. Toni (1992), 'Abductive Logic Programming', *Journal of Logic and Computation*, Vol. 2, No 6, pp. 719-770.

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

[9] A. C. Kakas and G. A. Papadopoulos (1994), 'Parallel Abduction in Logic Programming', *Proc. 1st International Symposium on Parallel Symbolic Computation*, Linz, Austria, (to appear).
[10] E. Maïm (1992), 'Abduction and Constraint Logic Programming', *Proc. 10th European Conference on Artificial Intelligence*, Vienna, Austria, pp. 149-153.
[11] A. Michael and A. C. Kakas (1994), 'Integrating Abduction and Constraint Logic Programming', Internal Report, Department of Computer Science, Univ. of Cyprus.
[12] A. Pool (1992), 'Logic Programming, Abduction and Probability', *Proc. International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, pp. 530-538.
[13] V. A. Saraswat (1993), '*Concurrent Constraint Programming*', ACM Doctoral Dissertation Award, MIT Press series on Logic Programming.
[14] V. A. Saraswat, D. Weinbaum, K. Kahn and E. Y. Shapiro (1988), 'Detecting Stable Properties of Networks in Concurrent Logic Programming Languages', *Proc. 7th ACM Symposium on Principles of Distributed Computing*, pp. 210-222.
[15] E. Y. Shapiro (1989), 'The Family of Concurrent Logic Programming Languages', *Computing Surveys*, Vol. 21 (3), pp. 412-510.
[16] A. Wærn (1992), 'Reactive Abduction', *Proc. 10th European Conference on Artificial Intelligence*, Vienna, Austria, pp. 159-163.