Parallel Abduction in Logic Programming[†]

Antonis C. Kakas and George A. Papadopoulos

Department of Computer Science
University of Cyprus
75 Kallipoleos Str.
Nicosia, T.T. 134, P.O. Box 537
CYPRUS
e-mail: {antonis,george}@turing.cs.ucy.ac.cy

Abstract: Logic Programming has been recently extended to include abduction as an inference mechanism leading to the development of Abductive Logic Programming (ALP). In this work we study the introduction of parallelism in the operational behaviour of an abductive logic program over and above the parallelism found in an ordinary logic program. In particular, we examine the exploitation of various forms of parallelism (OR-parallelism, dependent AND-parallelism) in an abductive logic program the purpose of which being twofold: i) to propose a model for parallel computation of abduction in ALP, and ii) to find ways to use the existing technology developed over the years in parallelising deductive logic programming in the framework of ALP. The ideas described in the paper have led to the design of a parallel model, particularly suited to a shared-memory architecture, which has been mapped onto the Andorra model as expressed by the concurrent logic language AKL.

Keywords: Abductive Logic Programming; Parallelism; Andorra Model.

[†] 1st International Symposium on Parallel Symbolic Computation (PASCO), Linz, Austria, 26-28 September 1994, World Scientific Publishing, pp. 214-224.

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 and elsewhere such as in the field of databases. As a result, it is useful to study ways for making the computation of abduction more effective. One such way which this paper addresses is the parallelisation of abduction.

It has been argued ([6]) that abductive inference and its parallel realisation should be one of the future research themes in parallel logic programming. The problem of parallel abduction has been studied in [8] using a model generation theorem prover. This approach is based mainly on a bottom-up computation of models corresponding to abductive explanations. The approach taken in this paper will be based on a top-down computational model for generating abductive explanations and testing their satisfiability with respect to integrity constraints.

In this work we study the problems of parallelisation of abduction by concentrating in one framework, that of Abductive Logic Programming (ALP). Many of the ideas however are applicable, more generally, to other frameworks of abduction or hypothetical reasoning. The framework of ALP that we will adopt was originally proposed by Eshghi and Kowalski ([4,5]) and developed further by Kakas and Mancarella ([12,13]).

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 ordinary (deductive) logic programming, abductive inference mechanisms have several sources of parallelism of many forms (OR-parallelism, independent and dependent AND-parallelism). In this work 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. A parallel computational model for abductive logic programming is defined as an extension of the basic Andorra model featuring new sources of parallelism (and non determinism) stemming from abduction and the integrity checking that it contains. We illustrate how our parallel computational model for abduction can be implemented in AKL and we present the basics of such an implementation.

The work in this paper provides a first step in parallelising top-down abductive reasoning. Based on this we aim to apply these ideas to more general forms of abduction, such as constructive abduction (eg. [3]) and integrated forms of abductive and constraint programming ([2,14]).

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

An example of this semantics for abduction is the generalised stable model semantics ([12]). In this Δ is an abductive explanation iff there exists a stable model $M(\Delta)$ of $P \cup \Delta$ such that: i) $M(\Delta) \models G$ and ii) $M(\Delta) \models I$ where $\models$ denotes truth in the model $M(\Delta)$. For simplicity we assume the usual restrictions: there are no rules for abducible atoms, integrity constraints have been compiled into denials with at least one abducible and the hypotheses generated are variable free.

In the abductive proof procedure for logic programming [5,12] (see also [11] 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. Consider the following example where a, b, c are abducibles and '$<-$' in **I** denotes negation; so for instance, '$<- a, s$' means '$\neg(a \wedge s)$'.

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

I:    <- a, s.
      <- b, v.
      <- b, t.
```

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 Δ=`{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`.

In the above scenario note the synchronisation performed implicitly by the processes involved in an abductive or consistency phase via the updating of the hypotheses set Δ. The main point of interest here is that inconsistencies can potentially be detected earlier in a parallel realisation of the model especially in view of the fact that the model allows the recording of abducibles in Δ at the beginning of their consistency phase.

3 Sources of Parallelism in an Abductive Logic Program

The above example demonstrates the high potential of parallelism that exists in an abductive logic program. In this section we will describe the various sources of parallelism which can exist in ALP.

As in deductive logic programming one place where we can introduce parallelism is in the evaluation of a conjunction in an abductive phase. The process oriented behaviour of the model allows the exploitation of a synchronised form of AND-parallelism during the parallel evaluation of these conjunctions. More importantly, this parallelism can, in fact, be combined and interleaved with the parallelism that can be exploited in the consistency phase leading to a number of interesting and important optimisations.

The consistency phase can give rise to both AND- and OR-parallelism; both sources of parallelism are illustrated in the integrity checking for the abducible `a` of the example in the previous section. In particular, while trying to satisfy the constraint `<- a, s` the two different clauses for `s` can be tried in parallel. This is, in fact, AND- rather than OR-parallelism since both clauses must end in failure and the set of hypotheses Δ formed is common to both rules and processes for `s` and should therefore be jointly consistent. Note that this AND-parallelism is combined with the AND-parallelism in the abductive phase.

The OR-parallelism that can appear in the consistency phase can be illustrated by considering the first rule for `s` (in the same example) where the failure of *either* `s1` or `s2` suffices to fail `s`. Hence, as far as the consistency phase is concerned, the evaluation of a conjunction of literals in parallel is OR-parallelism. This form of OR-parallelism in the consistency phase can, in fact, appear quite often and gives rise to different possible explanations.

Another new aspect of parallel abductive computation is the issue of communication and synchronisation of processes through the common set of abducibles in Δ. When a literal has been abduced, this information can be made available to all other processes running in parallel. We recall from the previous section that in the computational model of abduction used in this paper, the hypotheses generated during an abductive phase can be recorded *immediately* in Δ without waiting for the successful termination of the associated consistency phase. The sequential implementation uses this fact but in a limited way. A parallel realisation of the model can exploit it much more effectively.

We can also record in Δ those abducibles that during a consistency phase are assumed to be absent. This, together with the recording of the presence of hypotheses in Δ and the fact that we can have in parallel abductive and consistency processes, leads to the following important optimisation: if some other process requires the absence of the abduced literal, this inconsistency can be detected early and the computation can be abandoned saving unnecessary work. In the example of the previous section, this case arises in the evaluation of the integrity constraints for `b` (especially if `u` represents a big computation), where the inconsistency due to the needed presence and absence of `c` can be detected early.

This synchronisation of processes through the hypotheses set Δ is an important aspect of parallelism that appears in ALP on which any

parallel implementation of abduction must concentrate.

Other forms of AND- and OR-parallelism can be exploited, similar to the ones found in deductive logic programming. There is, for instance, the usual kind of OR-parallelism where different parts of the search space can be explored concurrently (as is the case of resolving `p` in an abductive phase with both its rules defining it). Thus, an abductive logic program contains both a don't know non deterministic as well as a process oriented part, the last one arising due to the "dependent" (because of the synchronisation of the concurrently executing processes via the common hypotheses set Δ) AND-parallelism.

Negation as Failure Through Parallel Abduction

Before closing this section it is worth mentioning that the parallel framework of abductive logic programming can, in fact, be used to enhance the computational efficiency of ordinary logic programming by applying it to the computation of NAF. Eshghi and Kowalski ([4]) show how NAF can be understood through abduction, where a negative literal `not p` is considered as a primitive abducible p^* with the constraint $\leftarrow p, p^*$. We also have a disjunctive integrity constraint $p \vee p^*$ which has the effect that whenever we require the absence of the abducible p^* in a consistency phase, we must enter a new abductive phase to prove the goal `p` (cf. the execution of goals with nested NAF in Prolog).

Effectively this means that we can i) compute in *parallel* all the possible ways of failing to prove `p` needed for the computation of `not p` and ii) detect faster the incompatibility of requiring for the same goal the success of the subgoals `q` and `not q`. In the following example

```
p <- not q, not r.
r <- s.
r <- not q.
q <- …
s <- …
```

the query `p` will require Δ={`not q, not r`} and subsequently we will compute in parallel all possible ways of proving `q` and `r` to show that each one cannot succeed. One of these ways for proving `r` (coming from the second rule for `r`) will require the absence of `not q` and hence the whole computation will fail early by noticing the incompatibility of needing both the presence and the absence of `not q`.

Although in this special case of abduction for NAF this can be seen as a form of tabulation of NAF calls, for general abduction in ALP with other integrity constraints and especially when we have both NAF and abduction this is not the case and a more general model like the one proposed in this paper is required.

4 A Parallel Model for ALP

4.1 Description of the model

The ideas that have been described so far lead to the design of a model of parallelising the execution of an abductive logic program. This design must take into consideration the following issues: i) the benefits accrued from letting all concurrently executing agents know at all times the contents of the hypotheses set, and ii) the need for a mechanism to handle the incompatible requests that may arise during the parallel execution of abductive and consistency phases.

Our model combines *eager* abduction with a *lazy* non deterministic execution strategy that tries first to reduce in parallel all deterministic goals before adhering to don't know OR-parallelism. We need to explain here what we mean by "non deterministic goal" in the context of abductive logic programming.

- A goal is non deterministic if it can affect the hypotheses set in a (don't know) non deterministic way. We can characterise a goal as being non deterministic if either i) it is involved in an abductive derivation and is defined by more than one rule in a program **P**, or ii) it is involved in a consistency derivation and is a conjunction involving at least one abductive literal *not present* in Δ.

The hypotheses set Δ could be viewed as a structure shared by all processes involved in a computation. Every such process should be able to either enhance Δ with additional information recording the presence or (need for) absence of an abducible or, alternatively, check whether some information relevant to what it is about to do (abduce a literal or commence a consistency phase) has already been recorded in Δ by some other process. Note that once some information is recorded it is never retracted since, as we will explain below, conflicts are handled by creating new hypotheses sets. Note also that access to Δ should be an atomic operation.

Accessing Δ for reading or writing can thus be understood as operations over the constraint system of bags ([15]) where we are particularly interested in the following two operations (slightly modified to serve our purposes): i) `inform(Δ,t)` which succeeds if some new information `t` can be added to the bag Δ and fails otherwise, and ii) `check(Δ,t)` which succeeds if some information `t` is present in the bag Δ and fails otherwise. In particular, `t` is of the form `abd(present)` or `abd(absent)` where the labels `present` and `absent` denote that the information recorded in the hypotheses set Δ is that the literal `abd` has been abduced or its absence has been assumed respectively. So `inform(Δ,abd(present))` (or `inform(Δ,abd(absent))` for that matter) will

fail if the relevant information regarding `abd` is already recorded in Δ.

We now describe in more detail the way an abducible literal involved in an abductive or consistency phase is handled by our model. First we examine how abducibles are handled locally within an abductive or a consistency phase and then describe how different abductive and consistency phases are put together to form a complete computation.

In an abductive phase a literal `abd` is handled by an operation `abduce(Δ,abd)` which:

— terminates immediately successfully if `abd(present)` is already recorded in Δ,
— fails immediately if `abd(absent)` is already recorded in Δ,
— otherwise updates Δ with `abd(present)` and enters a consistency phase to check whether `abd`'s constraints are satisfied.

In a consistency phase a literal `abd` is handled by an operation `consistency(Δ,abd)` which:

— terminates immediately successfully if `abd(absent)` is already recorded in Δ,
— fails immediately if `abd(present)` is already recorded in Δ,
— otherwise updates Δ with `abd(absent)`.

So far we have concentrated on how processes in an abductive or consistency phase execute locally. These processes form an AND-parallel process tree which expands as long as the processes are able to perform deterministic reductions. However, we recall for the case when non deterministic goals are involved, that if an `abduce` process is executed in the context of a predicate where more than one of its defining rules are applicable, or, alternatively, a `consistency` process is executed in the context of a rule including at least one abducible not already present in Δ, separate OR-branches must be created to compute the different explanation sets.

We now turn our attention to this issue of non determinism explaining also how the concepts of *stability* and *non determinate promotion*, two important properties of the Andorra family of languages, can be exploited by the model we propose to handle this nature of abduction. Using AKL terminology again, we notice that the `inform` operation is the only one that can be "noisy" (i.e. affect its external environment) and may therefore be a need for it to suspend. Such a need arises if `inform` is executed in a choice-box. An `inform` operation is executed in a choice-box if the abducible involved in it is part of a non deterministic (in the sense defined above) goal. The example beside helps to clarify the way the deterministic part of the computation interacts with the non deterministic one.

Figure 1 shows the configuration of the query `<- t` when the deterministic part of the computation has completed execution and the non deterministic part has suspended. Note here that `abd`* is a shorthand for `abd(absent)`. Note also that the thicker lines denote potential OR-parallelism (referred to as CHOICE-branches in AKL terminology).

P:
```
t <- p, q, s.
p <- a.
p <- b.
q <- c.
r <- c, e, f.
s <- d.
u <- g.
```

I:
```
<- a, c.
<- a, v.
<- d, r.
<- b, u.
```

where all the literals from `a` to `g` denote abducible literals.

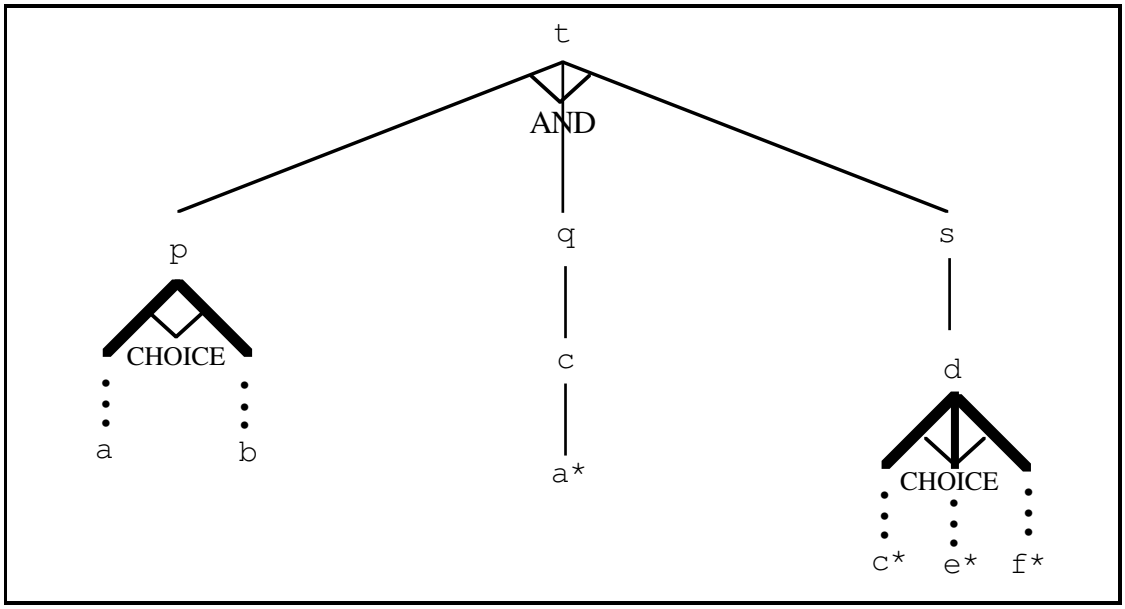

Fig 1 — and/or configuration - reduction of the deterministic part

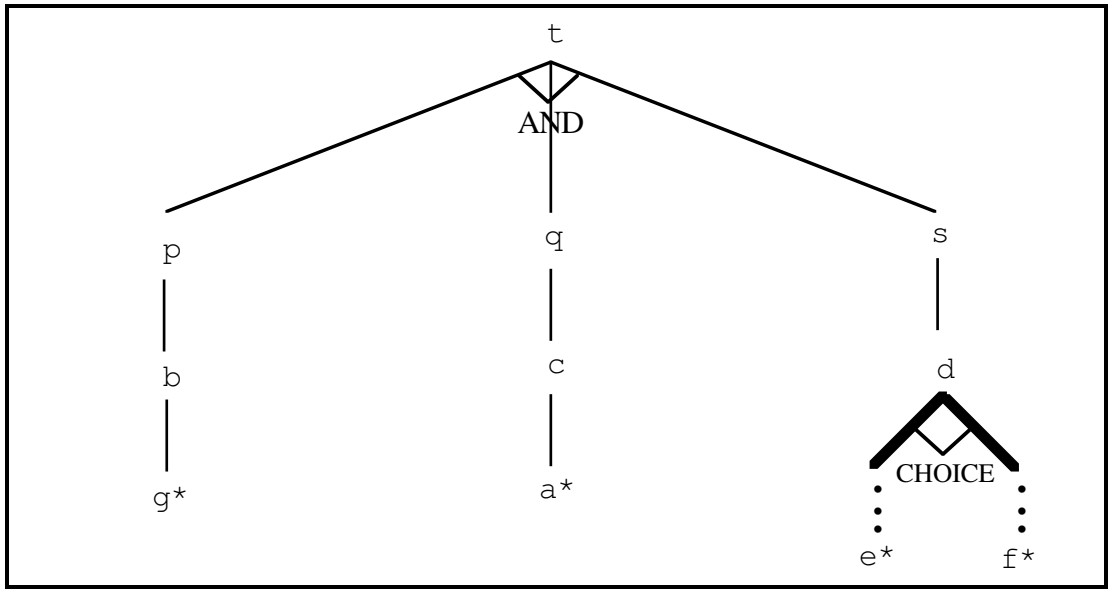

Fig 2 — and/or configuration - fully-fledged state of the deterministic part

Figure 2 presents an AND-conjunction where the reduction of p, q and s is done as an AND-parallel computation. Dotted lines show parts that can potentially get suspended because of the presence of non determinism. Assuming that the hypotheses set Δ starts empty, the reduction of the above literals causes the abduction of c and d, their inclusion in Δ and the commencing of a corresponding number of consistency phases for c and d. Note, however, that the reduction of p suspends initially, because either of its two rules could be used, resulting in the possibility of different hypotheses sets. Since the abduction of c can only be consistent with the absence of a, the computation along this particular AND-branch (which contains an abductive and a consistency phase) is allowed to complete enhancing Δ with c and a^*. On the other hand, since d can be made consistent with the failure of r and hence either with the failure of c or e or f, the consistency phase for d gives rise to a non deterministic choice and hence its execution suspends.

Now the successful abduction of c along the middle AND-branch, allows the deterministic promotion of the CHOICE-branch associated with p since now the only consistent way to reduce it is with the abduction of b. This enhances Δ further with b itself and, eventually, also with the absence of g. In addition, one of the CHOICE-branches that comprise the computation associated with the consistency phase for d (in particular the one corresponding to the absence of c) can now be removed. At this stage the deterministic phase ends since the system quiescences with those computations that are still possible and require the "noisy" application of the inform operation being suspended. Figure 2 shows the state of the fully-fledged deterministic part of the computation.

The computation will now split into two OR-branches, each one inheriting the hypotheses set that has been computed so far, namely Δ = {c, d, a^*, b, g^*} and extended with e^* for one branch and f^* for the other, to eventually give the two solutions Δ1 = {Δ, e^*} and Δ2 = {Δ, f^*} respectively.

To summarise, the computation interleaves between a deterministic reduction phase and a non deterministic splitting phase. During the deterministic phase all abductive and consistency derivations that are deterministic are done in parallel; non deterministic derivations are suspended. At the end of the deterministic phase, any one process which has not failed, has either terminated (successfully) or suspended. If there are processes which are suspended this means that there are potentially more than one explanation sets Δ for the original query. We thus have a non deterministic stage where the computation splits into independent OR-branches, for every different combination of choices in the suspended goals, which can be explored in parallel. Every such derivation inherits the explanation set Δ of the parent deterministic derivation.

The suspension of a non deterministic consistency phase provides a way of handling the problem of *resolution of conflict* when a process wants to record the abduction of some abducible while some other wants to record its absence. In the current example, had the consistency phase corresponding to the abducible d been allowed to proceed this would have resulted in an inconsistency since it would have attempted to record in Δ the absence of c while the reduction of q would try to record its presence. In such a case, it is not possible (without extra information) to determine on whose favour the conflict must be resolved and so we treat this situation as a non deterministic one allowing

the computation to proceed in different OR-branches, thus covering all possibilities.

As a final but more concrete illustration of the above discussion, consider the following recursive definition of a multiple of 4, where again we treat NAF as a primitive abducible.

```
rec_mult4(0).
rec_mult4(X) :-
   even(X), not rec_mult4(X-2).

even(0).
even(X) :- not even(X-1).
```

Given the query `rec_mult4(4)` the abductive derivations for `even(4)` and `not rec_mult4(2)` start in parallel. The abductive and consistency derivations associated with the first goal `even(4)` are all deterministic and they are allowed to proceed to completion. However, the abduction of `not rec_mult4(2)` leads, initially, to a non deterministic consistency derivation where the failure of either one of `not even(1)` or `not rec_mult4(0)` alone is sufficient for the success of the derivation. Hence the consistency phase suspends and will resume when the whole computation has reached quiescence (i.e. the computation of `even(4)` has terminated). It will then detect immediately the inconsistency generated by its attempt to record the absence of `even(1)` in Δ (which has already been recorded in Δ as present by the process `even(4)`) and allow the deterministic promotion of the second CHOICE-branch which will succeed updating Δ with the absence of `rec_mult4(0)`. The final solution is therefore Δ = {`not even(3)`, `not even(1)`, `not rec_mult4(2)`} where only those abducibles that must be present are shown here.

4.2 Rewrite Rules

We now provide a more formal description of the model in terms of a set of operational rewrite rules. This set of rewrite rules should be seen as an enhancement and extension of those for AKL ([9]). Where possible we refrain from including information particular to AKL *per se*. For example, the treatment of variable bindings is done in exactly the same way as in AKL. Note though that abducible goals remain suspended until they are fully ground. Also for simplicity of presentation we will not include here the rules required for negation as failure. Using the abductive treatment of NAF ([5]) these extra rules are analogous to the ones presented below together with an additional rule that generates a new abductive phase within a consistency phase.

An important new characteristic of our model is the fact that the constraint satisfaction, namely the integrity checking of the abducibles in the consistency phases, is not assumed to be external to the parallel model. Instead this is incorporated into the model so that this task can itself be parallelised and also be carried out in parallel to the usual task of goal reduction and constraint (namely abducible) generation. Thus apart from the usual **`and`**, **`choice`** and **`or`** boxes of AKL we also have a new box, denoted by $\mathbf{and}_{\mathbf{con}}$, to capture the AND-parallelism inside the consistency phases where the integrity checking is performed.

Computation starts with a top level query which is enclosed in an **`AND`** box whose purpose is to define the context of the current Δ. All the deterministic (with respect to abduction) part of a computation is performed within the context of such an **`AND`** box. In the sequel `A` will always stand for a ground abducible.

Top level initiation of computation

`P =>` **`AND`**`(`**`and`**`(P);`Δ`)`

The rest of rewrite rules can be split into two groups corresponding to the abductive and consistency phases respectively.

Abductive phase – ordinary local forking

`and``(S,P,T) =>`
  `and``(S,`**`choice`**`(`**`and`**`(Q1),…,`**`and`**`(Qn)),`
    `T)`

where `Pi :- Qi`, `i`=1…n, are the defining clauses for the non abducible literal `P`.

Abductive phase – simplification

`and``(S,A,T) =>` **`and`**`(S,T)` if `A` $\in \Delta$

Abductive phase – propagation of failure

`and``(S,A,T) =>` **`fail`** if $A^* \in \Delta$

Abductive phase – deterministic promotion

`and``(S,`**`choice`**`(`**`and`**`(P)),T) =>`
  `and``(S,P,T)`

Abductive phase – abducing a literal

`and``(S,A,T) =>`
  `and``(S,`**`con`**`(A),T),` $\Delta \cup$ `A`
if `A`, $A^* \notin \Delta$

This rule is applied *only* if the left hand side is not inside some choice box. When it is applied note that it modifies the environment Δ of its parent **`AND`** box.

Linking rule

`con``(A) =>`
  $\mathbf{and}_{\mathbf{con}}$`(`**`and`***`(Q1),…,`**`and`***`(Qn))`

where $\neg$`(A,Qi)`, `i`=1…n, are the integrity constraints involving the abducible literal `A`.

The purpose of the new box **and*** is, like in an **and** box, to capture the local unification environment of each separate branch of the consistency phase $\mathbf{and}_{\mathbf{con}}$ box. Note that an **and*** box never rewrites by itself.

The rewrite rules for $\mathbf{and}_{\mathbf{con}}$ are as follows.

Consistency phase – local expansion

$\mathbf{and}_{\mathbf{con}}$(R,**and***(S,P,T),U) =>
  $\mathbf{and}_{\mathbf{con}}$(R,**and***(S1,Q1,T1),…,
    and*(Sn,Qn,Tn),U)

where Pi :- Qi, i=1…n, are the defining rules for the predicate P and Si, Qi, Ti, is the *non empty* resolvent of P with the corresponding clause.

Consistency phase – simplification

$\mathbf{and}_{\mathbf{con}}$(R,**and***(S,A,T),U) =>
  $\mathbf{and}_{\mathbf{con}}$(R,**and***(S,T),U)

if $A \in \Delta$ and *at least one* of S or T is non empty.

Consistency phase – local forking, first case

$\mathbf{and}_{\mathbf{con}}$(R,**and***(S,A,T),U) =>
  choice($\mathbf{and}_{\mathbf{con}}$(R,U),
    $\mathbf{and}_{\mathbf{con}}$(R,**and***(S,T),U)
    if $A^* \in \Delta$

Consistency phase – local forking, second case

$\mathbf{and}_{\mathbf{con}}$(R,**and***(S,A,T),U) =>
  choice($\mathbf{and}_{\mathbf{con}}$(R,**and***(A),U),
    $\mathbf{and}_{\mathbf{con}}$(R,**and***(S,T),U)

if $A, A^* \notin \Delta$ and *at least one* of S or T is non empty.

Consistency phase – abducing literal's absence

$\mathbf{and}_{\mathbf{con}}$(S,**and***(A),T) =>
  $\mathbf{and}_{\mathbf{con}}$(S,T), $\Delta \cup A^*$
    if $A, A^* \notin \Delta$

As in the abductive phase this rule is applied *only* when the left hand side $\mathbf{and}_{\mathbf{con}}$ is not inside some **choice** box. When it is applied note that it modifies the environment Δ of its parent **AND** box.

Consistency phase – deterministic promotion

choice($\mathbf{and}_{\mathbf{con}}$(P)) => $\mathbf{and}_{\mathbf{con}}$(P)

Finally, we have the rule for the non deterministic promotion which is the same for both the abductive and consistency phases.

Non deterministic promotion

AND(**and**(S,**choice**(P,Q),T);Δ) =>
  OR(**AND**(**and**(S,P,T);Δ),
    AND(**and**(S,Q,T);Δ)

where each **AND** box inherits a copy of Δ.

In addition to the above rules there are a number of trivial rules for promotion which are not mentioned here.

Any successful computation using these rules can easily be mapped onto a successful derivation (or a set of them) of the sequential computational model for abductive logic programming as defined in [5, 13] by following the order in which the abducible hypotheses have been added into the set Δ. The converse is also trivially true. Hence the parallel model is correct whenever the corresponding sequential model is so and can compute any solution that the sequential model is able to compute.

5 Implementation Issues

The model described in this paper was initially tested on a meta-interpreter written in the concurrent logic programming language Parlog using, for the case of don't know non determinism, the machinery that was developed for its extension Pandora ([1]). Δ was implemented as a list managed by a corresponding monitor process. Access to it by the processes running concurrently was done via stream channels (using mergers where required). This common store was used to synchronise the execution of the processes, thus detecting inconsistencies earlier and saving unnecessary recomputations. The results of running a number of example programs on that interpreter were quite promising. For instance, the non recursive abductive version of a multiple of 4 defined by the following rule

```
mult4(X) :- even(X), even(X/2).
```

where even is defined as before run nearly twice as fast than the usual NAF version.

We are currently rewriting the interpeter in AKL ([9]), exploiting in the process the characteristics of the language. Access to Δ is done by means of ports ([10]), which provide efficient many-to-one communication, coupled with a monitoring process which effectively implements the operations check and inform. This first attempt serves more to provide a means of evaluating the model rather than being tuned for performance.

Another interesting issue regarding the implementation of the hypotheses set Δ is whether

the updating of information via the `inform` operation is done atomically or not. Here, we stress the fact that the correctness of our model does not depend on whether the abduction of some literal (or the need for its absence) is recorded in Δ as an atomic action or as an eventual publication. However, it is a factor that could influence the performance since a process may or may not be able to use results that have already been computed by some other process. For the case of a shared-memory based implementation, which is what we are aiming for in this paper, we believe that the benefits of atomic updating outweigh its limitations (see also relevant discussion in [7]).

A set of clauses representing a predicate definition in an abductive logic program together with the associated set of constraints for the abducibles invoked in its clauses is compiled as two sets of AKL rewrite rules, one for the case of using it in the abductive phase, and the other one for the case of using it in the consistency phase (we recall that in the consistency phase AND-parallelism is reversed to OR-parallelism and vice versa). Suitable guard operators, available in the AKL system, are used to express the required execution behaviour of the code. As an example, and using only the most important of the AKL rules generated for a procedure, we present below those AKL rules generated for some of the predicates of the first example program in section 4.

```
% Code for procedure p
p_abd :- abduce(Δ,a) ? true.
p_abd :- abduce(Δ,b) ? true.

p_con :- consistency(Δ,a),
         consistency(Δ,b).

% Code for procedure q
q_abd :- abduce(Δ,c).

q_con :- consistency(Δ,c) ? true.

% Code for procedure r
r_abd :- abduce(Δ,c), abduce(Δ,e),
         abduce(Δ,f).

r_con :- consistency(Δ,c) ? true.
r_con :- consistency(Δ,e) ? true.
r_con :- consistency(Δ,f) ? true.
```

Process `abduce` can be defined as follows:

```
abduce(Δ,abd) :-
   abduce(abd,Ans)@Δ,
   abduce1(Δ,Ans,abd).

abduce1(Δ,recorded,abd) :-
   | constraints(abd,C),
     consistency(Δ,C).
abduce1(Δ,present,abd) :- | true.
abduce1(Δ,absent,abd) :- | fail.
```

where we assume the existence of a predicate `constraints(abd,C)` which returns in `C` (in an appropriate format) the constraints associated with the abducible `abd`, i.e. those denials which contain a condition with the same abducible predicate as that of `abd`. Note the use of ports in this (indirect) implementation of the operations `inform` and `check`. In particular, a monitor process controls access to Δ and responds with the message `recorded` if `abd(present)` succeeded in being informed, and `present` or `absent` if `abd(present)` or `abd(absent)` respectively have already been recorded.

Furthermore, process `consistency` can be defined as follows:

```
consistency(Δ,abd) :-
   consistent(abd,Ans)@Δ,
   consistency1(Δ,Ans,abd).

consistency1(Δ,recorded,abd) :-
   | true.
consistency1(Δ,present,abd) :-
   | fail.
consistency1(Δ,absent,abd) :-
   | true.
```

Regarding the first rule for `consistency1`, we should note that the absence of `abd` cannot cause a violation of the integrity constraints for the particular denial form of integrity constraints we have considered and so there is no need for any other action. There are, however, cases where there is a need to call an abductive phase. For example, if we have disjunctive integrity constraints such as $p \vee abd$, the absence of `abd` requires the proof of `p`, i.e. the invocation of an abductive phase for `p`. Such a case arises, for instance, with NAF where we effectively have integrity constraints of the form $p \vee p^*$. For these cases this rule takes the following form:

```
consistency1(Δ,recorded,abd) :- |
   constraints(abd,C), abduce(Δ,C).
```

6 Conclusions and Further Work

In this paper we have proposed and studied a parallel computational model for abduction in logic programming. The exploitation of the different forms of AND/OR-parallelism available in an abductive logic program has many benefits. Among others, it reduces the search space, causes a faster detection of inconsistencies, and allows the sharing and reuse of hypotheses.

Although it is not possible to exploit all forms of parallelism efficiently, the initial aim is to identify the best possible combination of AND/OR-parallelism on top of existing parallel models of logic programming, that also take into consideration the special needs of abductive logic programs. Along these lines we have concentrated on the Andorra model which, we believe, offers a number of characteristics suitable to our purpose.

Our first attempt to design and implement such a parallel model of abductive logic programming concentrates on the use of the AKL system which offers the potential we require ranging from issues of programming expressiveness (such as ability to interface non deterministic computations with reactive ones - an important property of our model) to implementation ones (such as a good programming and developing environment).

A number of issues need further consideration. The effectiveness and efficiency of the model should be evaluated further by means of more application programs (ranging from simple benchmarks to real life applications). We intend to evaluate the effectiveness of our model compared with the sequential model and with the bottom up model of [8]. Also, the model should be extended to handle non ground abducible literals along the lines of constructive abduction in logic programming as defined for example in [3]. It will then be necessary to incorporate in the model an enhanced unification constraint solver for equality and disequality constraints. More generally, the integration of the model with Constraint Logic Programming and the generalisation of the types of integrity constraints that are supported need further investigation and are topics of future research. In particular, it is useful to compare our parallel model of abduction where the task of integrity checking is itself parallelised and incorporated explicitly into the model, with an alternative parallel model where the integrity checking is performed by a separate specialised module tightly coupled with the generation of the abductive hypotheses.

Acknowledgements

We are grateful to Reem Bahgat for many useful comments and corrections in an earlier version of the paper.

References

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

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

[3] Denecker M and De Schreye D., SLDNFA: An Abductive Procedure for Normal Abductive Programs, *JICSLP'92*, Washington D. C., MIT Press, pp. 686-700.

[4] Eshghi K. and Kowalski R. A., Abduction Through Deduction, Technical Report, Department of Computer Science, Imperial College, 1988.

[5] Eshghi K. and Kowalski R. A., Abduction Compared With Negation By Failure, *6th ICLP,* Lisbon, 1989, MIT Press, pp. 234-255.

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

[7] Hummel S. F. and Kelly R., A Rationale for Massively Parallel Programming with Sets, *Journal of Programming Languages*, 1993, Vol. 1, pp. 187-207.

[8] Inoue K., Ohta Y., Hasegawa R. and Nakashima M, Bottom Up Abduction by Model Generation, *IJCAI'93.*

[9] Janson S. and Haridi S, Programming Paradigms of the AKL, *ISLP'91,* San Diego, 1991, MIT Press, pp. 167-183.

[10] Janson S., Montelius J. and Haridi S., Ports for Objects in Concurrent Logic Programs, *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.

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

[12] Kakas A. C. and Mancarella P., Generalised Stable Models: a Semantics for Abduction, *9th ECAI*, Stockholm, 1990, pp. 385-391.

[13] Kakas A. C. and Mancarella P., On The Relationship Between Truth Maintenance and Abduction, *1st PRICAI*, Nagoya, Japan, 1990.

[14] Kakas A. C. and Michael A., Integrating Abduction and Constraint Logic Programming, Internal Report, 1994.

[15] Saraswat V. A., *Concurrent Constraint Programming*, Ph.D. Thesis, Carnegie-Mellon University, January 1989, appeared as ACM Doctoral Dissertation Award, MIT Press series on Logic Programming, 1993.