Parallelism in Abductive 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

June 1993

e-mail: {antonis,george}@jupiter.cca.ucy.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. In particular, we examine the exploitation of various forms of parallelism (OR-parallelism, independent as well as dependent AND-parallelism) in an abductive logic program. The purpose of this work is twofold: i) to propose a model for parallel computation of abduction in ALP and to derive execution strategies that are more effective than sequential ones, ii) to find ways to use the existing technology developed over the years in parallelising deductive logic programming in the framework of abductive logic programming. The ideas described in the paper have been tested by means of an interpreter built over a concurrent logic language and the results are promising.

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. Furukawa ([5]) has argued that abductive inference and its parallel realisation should be one of the future research themes, following up the FGCS project.

In this paper we study the problems of parallelisation of abduction by concentrating in one framework, that of Abductive Logic Programming (ALP). Many of the ideas are applicable, more generally, to other frameworks of abduction or hypothetical reasoning. The framework of ALP that we will adopt is the one originally proposed by Eshghi and Kowalski ([3,4]) and developed further by Kakas and Mancarella ([8,9]). Thus we consider an abductive logic program to be 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 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.

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 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. The various execution strategies that are presented have been tested on a meta interpreter written in a concurrent logic language that simulates a parallel environment; first results show that the introduction of parallelism in many cases enhances the efficiency of abductive logic programs.

The main aim of our investigation is to derive a parallel execution model for abductive logic programming. Then we can map this onto some, possibly existing, parallel model for deductive logic programming (independent AND, Andorra, hybrid AND/OR or otherwise), importing as much of the existing technology as it is possible in the parallel abductive logic programming framework.

2. The Abductive Proof Procedure for Logic Programming

In the abductive proof procedure for logic programming [4,8] (see also [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 integrity. Consider the following example:

```
P:   p <- a, q, r.   s <- s1, s2.        I:   <- a,s.
     p <- b.         s <- s3.                 <- b,v.
                                              <- b,t.

     q <- true.      v <- not w.
```

```
r <- a.          w <- c.
                 t <- c.
```

where a, b, c are abducibles and '<-' in **I** denotes negation (for example, '<- a,s' means '~(a∧s)' Assuming a Prolog-like evaluation order, the query <- p will reduce using the first clause to <- a, q, 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 prove their failure and hence the satisfaction of the constraint. Assuming that this is the case, q will then get reduced followed by r. The latter needs the abducible a which is already part of the hypothesis **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 will 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**={b,c}. However, the evaluation of the second constraint <- b,t requires the absence of c and thus the second rule for p yields no further solutions to the query.

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**. Two points are of interest here: first, 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; second, that the reuse of hypotheses in **Δ** can be done more effectively in a parallel computation.

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 the sequel we will describe the various sources of parallelism which can exist in ALP.

The process oriented behaviour of the model allows the exploitation of a synchronised form of AND-parallelism, *a là* "stream parallelism", during the parallel evaluation of a conjunction of literals that appear in an abductive phase. When a literal has been abduced, this information is made available to all other processes running in parallel. We recall from the previous section that in the computational model of abduction as defined 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 a structure **E**, similar to **Δ**, those abducibles that during a consistency phase are assumed to be absent. This suggests the support of a communication model similar to Linda's tuple space. The benefit here is twofold: i) if some other process in the conjunction requires the abduction of a literal already in **Δ**, this process can avoid entering a consistency phase since it knows that this is already being performed; ii) if some other process requires the absence of the abduced literal, this incosistency can be detected early and the computation can be abandoned saving unnecessary work. The first case is illustrated in the evaluation of the first clause for `p` where `r` requires the abduction of `a`, and the second case arises in the evaluation of the integrity constraints for `b`. Thus, it is possible to reuse and share hypotheses as well as detect incosistencies faster. The synchronisation of processes through **Δ** and **E**, as described above is an important aspect of parallelism that appears in ALP on which any parallel implementation of abduction must concentrate.

The consistency phase can give rise to both AND- and OR-parallelism; both sources of parallelism are illustrated in the integrity constraint for the abducible `a`. 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 the set of hypotheses **Δ** formed is common to both rules for `s` and should therefore be consistent. Note that, in practice, we may not have to deal with this kind of parallelism very often; due to the restriction we have imposed that goals which can be abduced or are involved in a constraint evaluation are ground, it is unlikely that more than one rule will match. However, the OR-parallelism that can be exploited in the consistency phase arises more often. This is illustrated in the first rule for `s` 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.

Other forms of AND- and OR-parallelism can be exploited, similar to the ones found in deductive logic programming. There is 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). The groundness restriction we have imposed, coupled with a data flow analysis using Condition Graph Expressions ([6]), may also lead to the exploitation of independent AND-parallelism. In the fifth section we discuss further the interaction of AND- and OR-parallelism during the interleaving between the abductive and consistency phases.

4. An Illustration of the Benefits of Parallelism in ALP

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 negation as failure. Eshghi and Kowalski ([4]) show how negation as failure can be understood through abduction, where a negative literal `not p` is considered as a primitive

abducible `p*` with the constraints `<- p,p*` and `<- pvp*`. Consider, for instance, the following Prolog definition of a number being a multiple of 4:

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

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

This is a typical case where dependent AND-parallelism can be exploited. As the `even` processes execute in parallel, the literals that are abduced as well as the ones that must be absent are recorded in Δ and $\mathbf{E}$ which are accessible to all processes for reading and/or writing (a kind of blackboard). For instance, while evaluating `mult4(24)`, the process `even(24)` will eventually need to abduce `not even(11),` which will have already been abduced by the other `even(12)` process running concurrently.

Dependent AND-parallelism can also help in detecting incosistencies faster. Consider the following (rather artificial!) definition of `occurs_only_in_the_first(E,X,Y)` which succeeds if `E` is a member of the list `X` but not a member of the list `Y`:

```
occurs_only_in_the_first(E,X,Y) <- not absent(E,X),
                                   not member(E,Y).
absent(E,X) <- not member(E,X).
```

If `occurs_only_in_the_first` is called with both the parameters `X` and `Y` instantiated to the same list `L`, the parallel abduction on the negative goals will detect the inconsistency earlier than a sequential evaluation by noticing that we require the presence of `not member(e,L)` in both Δ and $\mathbf{E}$.

5. A Parallel Model of Computation

The ideas that have been described so far have led to the design of an initial parallel model of computation. The model is based on a "lazy" non deterministic execution strategy that tries first to reduce in parallel all deterministic goals before adhering to OR-parallelism. A goal involved in an abductive derivation is non deterministic if it is defined by more than one rule in the program $\mathbf{P}$. A goal involved in a consistency derivation is non deterministic if it is the conjunction of more than one abductive literal not present in Δ.

The computation interleaves between a deterministic reduction phase and a nondeterministic 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 has either terminated (successfully) or suspended. If there are processes which are suspended this means that there are potentially many different 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 Δ and $\mathbf{E}$ environments of the parent deterministic derivation.

The following example illustrates the behaviour of our model.

```
p <- p1, p2, p3.
p <- p4, p5.

p1 <- not p11.          p2 <- …        p3 <- not p31.
p11 <- not p121.        p2 <- …        p31 <- p32, not p34.
p11 <- not p122.                       p32 <- not p11.
p121 <- true.                          p33 <- not p34.
p122 <- true.                          p34 <- true.
```

Figure 1 below shows the state of the computation when the first deterministic phase has ended and the computation is about to split to the various OR branches that can be formed. We follow the usual convention that boxed computations refer to consistency phases and unboxed ones refer to abductive phases. In addition, thick lines show OR-parallelism while the thinner ones refer to AND-parallelism. The partial recordings in Δ and $\mathbf{E}$ are also shown. Note that the two rules for `p11` are executed as AND- rather than OR-parallelism since they are involved in a consistency phase. The reduction of `p31` gives rise to two different possible ways of ensuring the consistency, namely by failing at `not p11` or `not p33`. This results in a non deterministic split with the two OR branches, shown in the figure, one where we enter an abductive phase with `p11` and another with `p33`.

The suspension of a non deterministic consistency phase provides a solution to 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 `p32` been allowed to proceed it would have created an inconsistency since it would have attempted to record in $\mathbf{E}$ the absence of `not p11` while `p1` would try to record it in Δ. 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 non-deterministic allowing the computation to proceed in different OR branches, that cover both possibilities.

As a more concrete illustration of the above problem, consider the following recursive version of `mult4(X)`:

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

where `even(X)` is defined as above. For a ground query of `rec_mult4` (eg. `rec_mult4(4)` as in figure 2), abductive derivations for `even` and `not rec_mult4` start in parallel. The abductive and consistency derivations associated with `even` are all deterministic and they are allowed to proceed to completion. However, the abduction of `not rec_mult4(X-2)` leads 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) by splitting into two abductive OR branches of `even(1)` and `rec_mult4(0)`. The first one will detect the inconsistency generated by its attempt to record `not even(1)` in **E** (which has already been recorded in **Δ**) and fail, while the second branch will succeed updating **E** with `rec_mult4(0)`.

The model described above has been tested on a meta-interpreter written in the concurrent logic programming language Parlog simulating an AND/OR-parallel environment. **Δ** and **E** are represented as lists managed by corresponding monitor processes. Access to them by the processes running concurrently is done via stream channels (using mergers where required). These common stores are used to synchronise the execution of the processes, thus saving unnecessary recomputations. A limitation of the interpreter is that currently it handles only ground literals even for purely deductive goals.

The results of running a number of example programs have been quite promising. For instance, the abductive version of `mult4` runs nearly twice as fast than the usual negation as failure version. This suggests that the enhancement of the abductive logic programming framework with parallelism can lead to a more efficient execution for many classes of programs.

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 take into consideration the special characteristics of

abductive logic programs. We have concentrated mainly on the usual types of parallelism that can be found in a deductive logic program, but other novel types of parallelism ([1]) can also be considered.

Regarding implementation, we are currently rewriting the interpreter in AKL 0.8, trying in the process to take advantage of the facilities offered in that environment (Δ and $\mathbf{E}$, for instance, are implemented using AKL's monitors and ports). We are also paying particular attention to the combination of dependent AND- and OR-parallelism offered by the Andorra model which can be used to model the two phases (deterministic and non deterministic) our model comprises. Another interesting issue of study is to replace the OR-parallelism in the model with a sequential execution of the different choices of suspension, exploiting work on intelligent backtracking ([2]).

References

[1] Brogi A. and Ciancarini P., The Concurrent Language, Shared Prolog, *ACM Transactions on Programming Languages and Systems*, 13(1), Jan. 1991, pp. 99.123.

[2] Codognet C. and Codognet P., Non deterministic Stream AND-Parallelism Based on Intelligent Backtracking, *6th ICLP*, Lisbon, 1989, pp. 63-79.

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

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

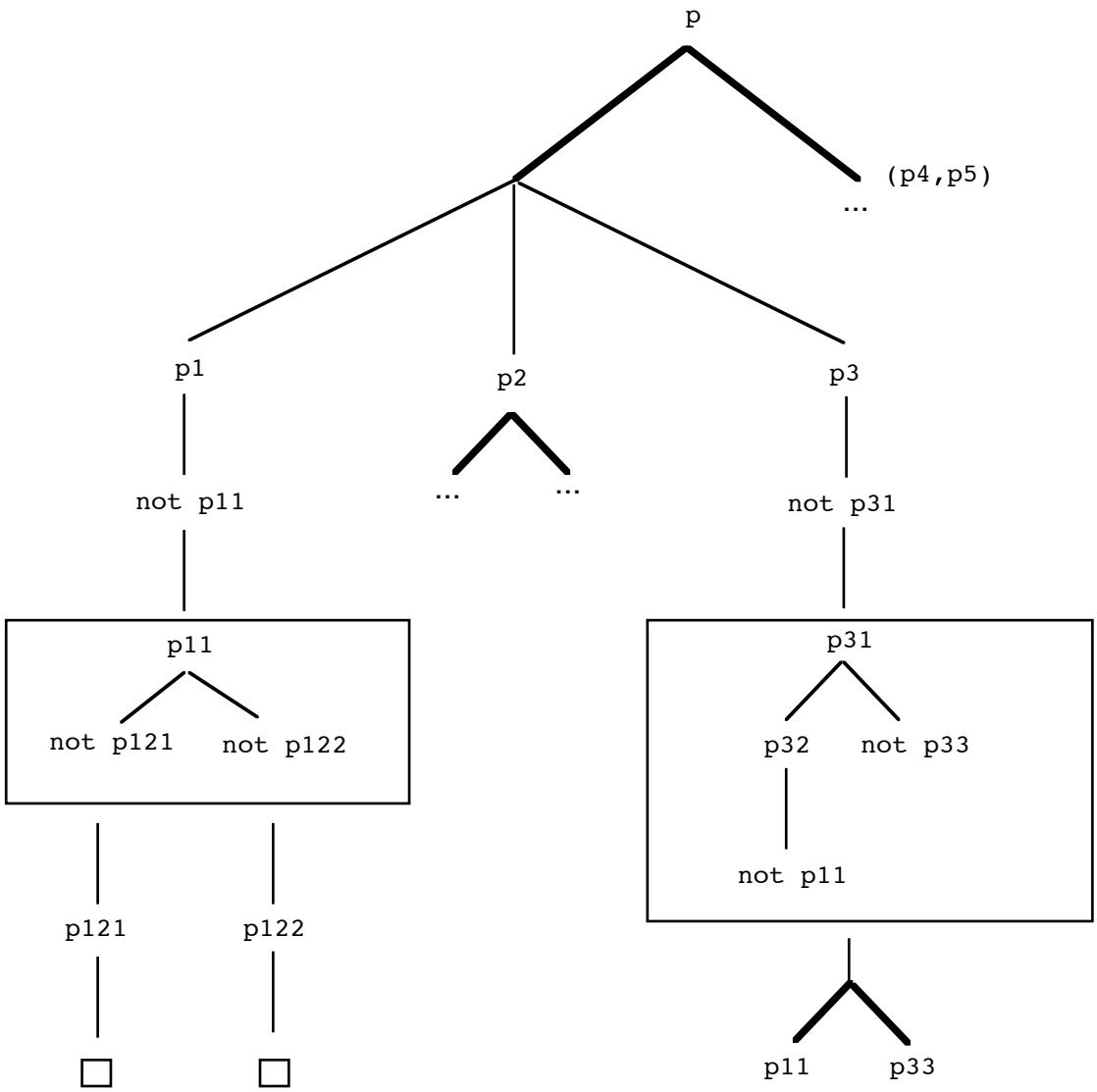
[5] Furukawa K., contribution to The Fifth Generation Project: Personal Perspectives, ed. E. Shapiro and D. H. D. Warren, *Communications of the ACM*, March 1993, pp. 48-101.

[6] Hermenegildo M. V., *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*, Ph.D. Thesis, University of Texas at Austin, August 1986.

[7] Kakas A. C., Kowalski R. A. and Toni F., Abductive Logic Programming, *Journal of Logic and Computation*, 1993, (to appear).

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

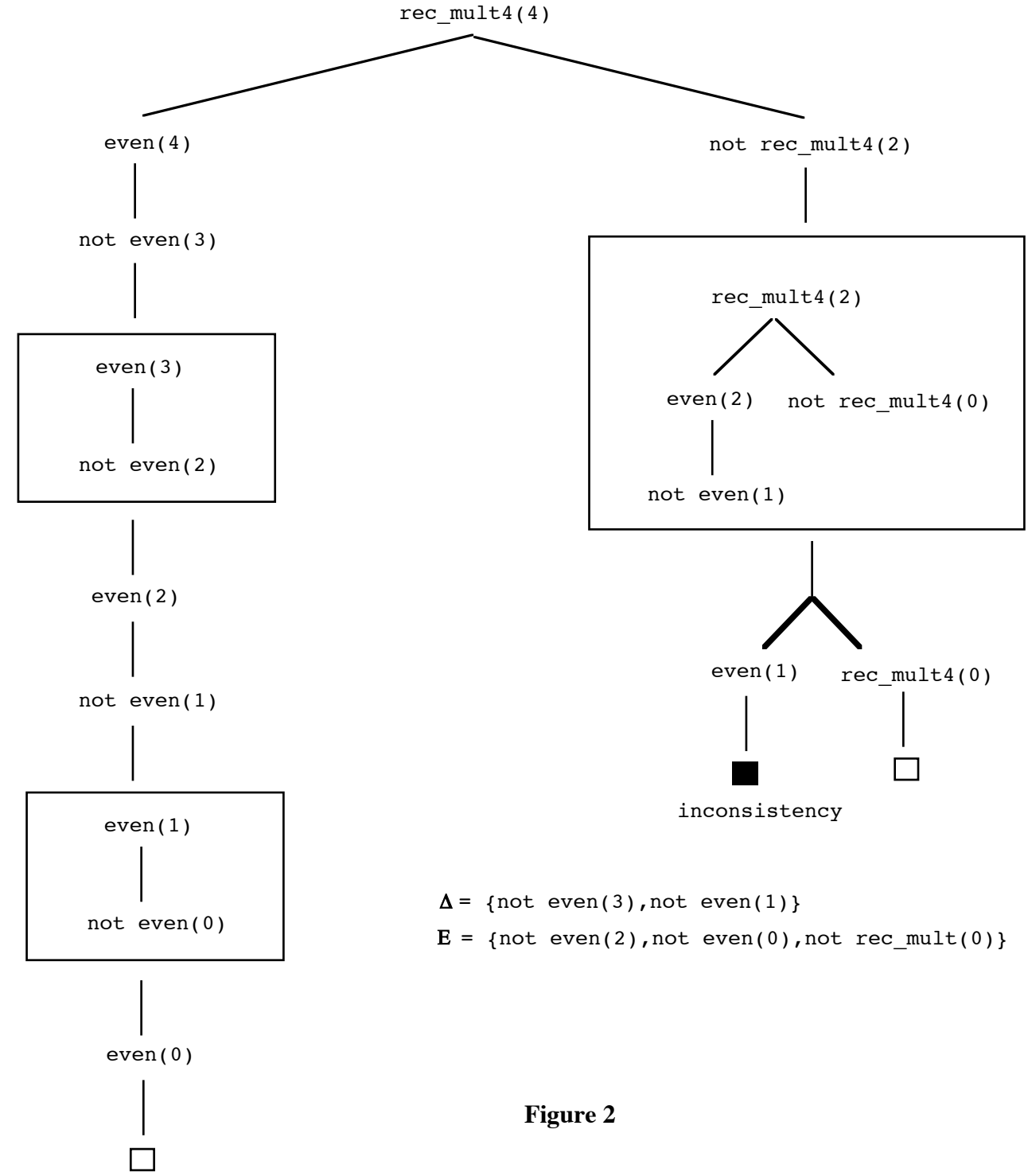
[9] Kakas A. C. and Mancarella P., On The Relationship Between Truth Maintance and Abduction, *Proc. 1st PRICAI*, Nagoya, Japan, 1990.



Δ = {not p11,not p31,not p34,…}

E = {not p121,not p122,…}

Figure 1



Δ = {not even(3),not even(1)}

E = {not even(2),not even(0),not rec_mult(0)}

Figure 2