Sources of Parallelism in Defeasible Argumentation[1]

Alejandro J. García[2] Guillermo R. Simari

Grupo de Investigación en Inteligencia Artificial (GIIA)
Instituto de Ciencias e Ingenieria de la Computación (ICIC)
Departamento de Ciencias de la Computación,
UNIVERSIDAD NACIONAL DEL SUR
Av.Alem 1253 – (8000) Bahía Blanca, ARGENTINA. FAX: (54) (91) 595136
e-mail: ccgarcia@criba.edu.ar grs@criba.edu.ar

KEYWORDS: Defeasible Reasoning, Argumentation, Parallelism.

Abstract

In a defeasible argumentation formalism, an *argument* is used as a defeasible reason for supporting conclusions. A conclusion q will be considered valid only when the argument that supports it becomes a *justification*. Building a justification involves the construction of a non-defeated argument $\mathcal{A}$ for q. In order to establish that $\mathcal{A}$ is a non-defeated argument, the system looks for *counterarguments* that could be *defeaters* for $\mathcal{A}$. Since defeaters are arguments, there may exist defeaters for the defeaters, and so on, thus requiring a complete dialectical analysis. The language of *Defeasible Logic Programming* (an extension of logic programming) provides a knowledge representation language for defeasible argumentation.

In Logic Programming, different kinds of parallelism have been studied, OR-parallelism, independent and dependent AND-parallelism, and also unification parallelism. All of these types of parallelism are at the language level.

In this work we introduce different kinds of parallelism that could be exploited at different levels in a defeasible argumentation system. At the language level, all types of parallelism identified for logic programming can be used. Besides, several arguments for a conclusion q can be constructed in parallel. Once an argument $\mathcal{A}$ for q is found, defeaters for $\mathcal{A}$ could be searched in parallel. Finally several argumentation lines in the dialectical analysis between arguments and defeaters, could be explored in parallel.

[1]This work was partially supported by the Secretaría de Ciencia y Técnica, Universidad Nacional del Sur.

[2]Fellow of Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), República Argentina

Sources of Parallelism in Defeasible Argumentation

1 Introduction

In this work, implicitly exploitable parallelism for a defeasible argumentation system will be studied. We will use the defeasible argumentation formalism developed in [24, 22, 7], and the knowledge representation language of *Defeasible Logic Programming* [5].

In a defeasible argumentation formalism [24, 4, 19], an *argument* is used as a defeasible reason for supporting conclusions. A conclusion q will be considered valid only when the argument that supports it becomes a *justification*. Building a justification [7], involves the construction of a non-defeated argument $\mathcal{A}$ for q. In order to establish that $\mathcal{A}$ is a non-defeated argument, the system looks for *counter-arguments* that could be *defeaters* for $\mathcal{A}$. Since defeaters are arguments, there may exist defeaters for the defeaters, and so on, thus requiring a complete dialectical analysis. In [5], an extension of logic programming called *Defeasible Logic Programming* (DLP) was defined. DLP uses defeasible argumentation for capturing common sense reasoning features that are difficult to express in conventional logic programming.

In Logic Programming, different kinds of parallelism have been studied [11], OR-parallelism, independent and dependent AND-parallelism, and also unification parallelism. Since DLP is an extension of logic programming, these four kinds of parallelism can be exploited. However, there are new sources of parallelism that can be implicitly exploited in a defeasible argumentation system: (1) several arguments for a conclusion q can be constructed in parallel, (2) once an argument $\mathcal{A}$ for q is found, defeaters for $\mathcal{A}$ can be searched in parallel, and (3) several argumentation lines in the dialectical analysis between arguments and defeaters, can be explored in parallel.

2 Building Arguments in Parallel

We present here a knowledge representation language for defeasible argumentation as an extension of Logic Programming. The language has two different negations: *strong negation*, which is represented by the symbol "$\sim$" used for representing contradictory knowledge; and *weak negation* (*negation as failure*), represented by the symbol "`not`" used for representing incomplete information.

Two types of program clauses are used for representing defeasible and non-defeasible rules:

- *extended program clauses* [3] (EPC): l `<-` $q_1, \ldots, q_n$.
- *defeasible program clauses* [4] (DPC): l `-<` $q_1, \ldots, q_n$.

[3] We use this terminology following [8]. However, the system presented here, can accommodate inconsistency whereas the language reported in [8] cannot.

[4] Given the similarity in use and syntax, we use the term 'clause' for this construction, even though it is not properly a clause, but a meta relation between the head and body of the rule.

In both kinds of clauses l is a literal (*i.e.*, a predicate "p" or a negated predicate "$\sim p$"), and each q_i $(0 \leq i \leq n)$ is a literal or a literal preceded by the symbol `not`. Thus, strong negation is allowed in the consequent of a clause, and negation as failure over literals is allowed in the antecedent. If $n = 0$, an EPC becomes "$l \leftarrow$ `true`." (or simply "l.") and is called a *fact*, whereas a DPC becomes "$l \prec$ `true`.", and is called a *presumption*.

We will use the usual PROLOG typographic conventions for program clauses, except that for an EPC we will write "`head <- body.`" rather than "`head :- body.`"; and "`head -< body.`" for a defeasible clause. An EPC is used to represent sound (*i.e.*, non-defeasible) information such as: `bird(X) <- penguin(X).` which expresses that "all penguins are birds", whereas a DPC is used to represent defeasible knowledge such as: `fly(X) -< bird(X).` which expresses that "presumably, a bird can fly" or "usually, a bird can fly."

Since the knowledge representation language is an extension of the language of Logic Programming, a finite set of EPCs and DPCs, will be called a *defeasible logic program* (DLP). Given a DLP $\mathcal{P}$, a *defeasible derivation* for a literal l is the finite set of EPC and DPC obtained by backward chaining from `q` as in a PROLOG program, using the clauses in the order specified by the DLP. When a defeasible derivation for a literal "$\sim$`p`" is started, the symbol "$\sim$" is considered as part of the name of the predicate `p`.

Example 2.1 : Here follows a DLP that will be referred to in other examples:

```
fly(X) -< bird(X).                bird(X) <- chicken(X).
~fly(X) -< chicken(X).            bird(X) <- duck(X).
fly(X) -< chicken(X),scared(X).   bird(X) <- penguin(X).
chicken(koko) -< true.            ~fly(X) <- penguin(X).
penguin(tweety) -< true.          penguin(chilly).
duck(mark).                       scared(koko).
chicken(pipi).                    duck(tim).
```

Using this DLP, there are defeasible derivations for each of the following literals: "`~fly(tweety)`", "`fly(tweety)`", "`fly(koko)`" and "`~fly(koko).`" □

As Example 2.1 shows, the notion of defeasible derivation does not forbid inferring two complementary literals from a given DLP $\mathcal{P}$. However, the defeasible argumentation formalism uses a dialectical analysis of arguments and counter-arguments in order to justify the conclusions. This will be the subject of the next section, but first we will show how to obtain defeasible derivations in parallel.

2.1 Parallel Defeasible Derivations

The knowledge representation language defined above is an extension of the language of logic programming. Therefore, all sources of parallelism identified for logic programming can be exploited for defeasible derivations.

As pointed by Gopal Gupta *et al.* in [11], there are many proposals for extending a logic programming language with *explicit* constructs for concurrency, as Delta

Prolog, CS-Prolog, Shared prolog, Parlog, GHC, and Concurrent Prolog. However, parallel execution of logic programs can also be done implicitly, *i.e.*, the parallelism can be exploited by the evaluator at run-time itself. In this work, we will consider only the approaches that exploit parallelism from logic program implicitly, *i.e.*, without intervention of the programmer.

Four main forms of parallelism (implicitly exploitable) can be identified in logic programs [10]:

1. Unification parallelism.
2. Or-parallelism.
3. Independent And-parallelism.
4. Dependent And-parallelism.

Unification parallelism arises when different argument terms (also subterms) can be unified in parallel. *Or-parallelism* arises when a subgoal can unify with heads of more that one clause, then the subgoals in the bodies of these clauses can be executed in parallel. *And-parallelism* arises when multiple subgoals in a query or in the body of a clause are executed in parallel. When the runtime bindings for the variables in these subgoals have non-intersecting sets of variables (*i.e.*, are independent), then parallel execution of such subgoals gives rise to *independent and-parallelism*. *Dependent and-parallelism* arises when two or more subgoals have a common variable and are executed in parallel.

Gopal Gupta points in [10] that these four kinds of parallelism are orthogonal to each other, *i.e.*, each one can be exploited without affecting the exploitation of the other. Thus, it is possible to exploit the four of them simultaneously. However, no efficient parallel system has been built yet that achieves this, and such an efficient parallel system that exploits maximal parallelism remains the ultimate goal of researchers in parallel logic programming.

These four kinds of parallelism can be exploited in DLPs when making defeasible derivations. And also many of the implementations developed for logic programming can be applied directly to DLPs. However, as we will show in the next sections there are new sources of parallelism that can be implicitly exploited in a defeasible argumentation system.

2.2 Arguments

Given a DLP, the defeasible derivation notion does not forbid inferring two complementary literals (see Example 2.1). Therefore, in order to allow only one of two complementary goals to be accepted as a sensible possibility, we need a criterion for choosing between the two. In defeasible argumentation, answers to queries must be supported by *arguments*. Let $\mathcal{P}$ be a DLP; then, we will distinguish the subset $\mathcal{S}$ of EPC in $\mathcal{P}$, and the subset $\mathcal{D}$ of DPC in $\mathcal{P}$.

Definition 2.1 (Argument) *An* argument $\mathcal{A}$ *for a query* h*, also denoted* $\langle \mathcal{A}, h \rangle$*, is a subset of* ground instances *of* DPC*s of* $\mathcal{D}$*, such that: (1) There exists a defeasible derivation for* h *from* $\mathcal{S} \cup \mathcal{A}$*, (2)* $\mathcal{S} \cup \mathcal{A}$ *is consistent, and (3)* $\mathcal{A}$ *is minimal with respect to set inclusion.*

Definition 2.2 (Consistency) *A set of program clauses $\mathcal{A}$ is* consistent *if there is no defeasible derivation for any pair of complementary literals (with respect to strong negation "$\sim$"). Conversely, a set of program clauses $\mathcal{A}$ is* inconsistent *if there are defeasible derivations for a pair of complementary literals.*

Given a DLP $\mathcal{P}$, the set $\mathcal{S}$ must be consistent, although the set $\mathcal{D}$, and hence $\mathcal{P}$ itself (*i.e.*, $\mathcal{S} \cup \mathcal{D}$) may be inconsistent. It is only in this form that a DLP may contain potentially inconsistent information.

Example 2.2 : Consider the DLP of example 2.1.
The query " `-< ~fly(koko).`" has the argument:

$$\mathcal{A}_1 = \left\{ \begin{array}{l} \texttt{\sim fly(koko) -< chicken(koko).} \\ \texttt{chicken(koko) -< true.} \end{array} \right\}$$

whereas, the query " `-< fly(koko).`" has two arguments:

$$\mathcal{A}_2 = \left\{ \begin{array}{l} \texttt{fly(koko) -< bird(koko).} \\ \texttt{chicken(koko) -< true.} \end{array} \right\}$$

$$\mathcal{A}_3 = \left\{ \begin{array}{l} \texttt{fly(koko) -< chicken(koko),scared(koko).} \\ \texttt{chicken(koko) -< true.} \end{array} \right\}$$

The query " `-< ~fly(tweety).`" has the argument:

$$\mathcal{A}_4 = \left\{ \texttt{penguin(tweety) -< true.} \right\}$$

but, there is no argument for " `-< fly(tweety).`" because its defeasible derivation is inconsistent with respect to $\mathcal{S}$. □

2.3 Parallel Consistency Checking

In order to obtain an argument for a literal h, the consistency checking will be done simultaneously with the defeasible derivation. Every time the body of a program clause P is derived, it will be tested whether the head h of P (with its current variable bindings) is consistent. A head h will be inconsistent when its complement $\overline{h}$ [5] can be derived with $\mathcal{S} \cup \mathcal{A}$. On the other hand, if a head h is certified as consistent, then will be remembered as a *temporary fact*. These temporary facts will be used for avoiding re-derivation of goals, and for checking consistency of the following derived sub-queries. We call them temporary facts, because they are erased once the main query is resolved. This approach was developed for the implementation of Defeasible Logic Programming, and we refer the interested reader to [6, 5] for the details of this implementation.

A defeasible derivation is carried out in a top-down fashion, whereas consistency-checking and the generation of temporary facts are done in a bottom-up manner. Thus, the consistency of a literal l is checked once the literal l has been derived. This is necessary because the goal being checked for consistency must be instantiated.

[5] The symbol " $\overline{\ }$ " will be used to denote the complement of a literal with respect to strong negation (*i.e.*, $\overline{a} = \sim a$, and $\overline{\sim a} = a$).

Indeed, if consistency of a goal were checked before the goal is proved, it could be instantiated with wrong terms and this would lead to unexpected results. For instance, in Example 2.1, if the query " `-< fly(X).`" is submitted, and consistency is checked before the variable `X` is instantiated, then there is no consistent defeasible derivation for any instance, because the strict derivation for "$\sim$`fly(chilly)`" would invalidate them. However, there is a consistent defeasible derivation with `X = koko` and `X = mark`.

Our implementation for consistency checking consists of adding an extra query at the end of every clause. The added query is the complement of the head of the clause. For example the EPC `p(X)` $\leftarrow$ `q(X)` is transformed to `p(X)` $\leftarrow$ `q(X) #`$\sim$`p(X)`, and the DPC $\sim$`r(X)` $\prec\!\!-$ $\sim$`s(X)` is transformed to $\sim$`r(X)` $\prec\!\!-$ $\sim$`s(X) #r(X)`. The symbol "`#`" must be read as " and cannot be strongly proved that".

Since consistency checking is done trying to derive grounded goals, *i.e.*, goals with no free variables, then independent and-parallelism between the consistency check and the derivation of the rest of the argument can be performed in an easily controlled way.

2.4 Computing Arguments in Parallel

Or-parallelism could be exploited in a defeasible argumentation system in order to obtain all the possible arguments for a given query. In a sequential system, when a subgoal can be unified with heads of more than one clause, then more than one argument could be obtained by backtracking. If or-parallelism is used, assuming an unlimited number of processors, every solution could be computed in parallel and backtracking would not be necessary. However, in practice a limited number of processors will be available, so backtracking is needed.

Or-parallelism is related to breadth-first search and tends to be more complete than PROLOG, because OR branches of the AND-OR tree are traversed concurrently. Therefore, even solutions to the right of an infinite branch can be computed.

Example 2.3 : If the query "`fly(X)`" is submitted to the program of Example 2.1 then the following arguments may be obtained (in parallel):

{ `fly(koko)` $\leftarrow$ `bird(koko) chicken(koko) -< true.` }
{ `fly(koko)` $\leftarrow$ `chicken(koko),scared(koko) chicken(koko) -< true.` }
{ `fly(mark)` $\leftarrow$ `bird(mark)` } □
{ `fly(pipi)` $\leftarrow$ `bird(pipi)` }
{ `fly(tim)` $\leftarrow$ `bird(tim)` }

It is interesting to note that, obtaining all possible consistent defeasible derivations for a given literal would help to obtain minimal arguments: every time a proper argument is found, the superset is discarded.

3 Parallel Argumentation

An argument $\mathcal{A}$ is a consistent defeasible derivation that supports a literal h. However, there may exist arguments that *disagree* with $\mathcal{A}$. Given an argument $\langle \mathcal{A}, h \rangle$,

every inner literal will define a subargument that could be counter-argued. Formally, an argument $\langle \mathcal{B}, q \rangle$ is a *sub-argument* of $\langle \mathcal{A}, h \rangle$ if $\mathcal{B} \subseteq \mathcal{A}$.

An argument $\langle \mathcal{A}_1, h_1 \rangle$ *counter-argues* $\langle \mathcal{A}_2, h_2 \rangle$ at a literal h, if there exists a sub-argument $\langle \mathcal{A}, h \rangle$ of $\langle \mathcal{A}_2, h_2 \rangle$ such that the set $\mathcal{S} \cup \{h_1, h\}$ is inconsistent. The literal h is called the counter-argument point, and $\langle \mathcal{A}, h \rangle$ the disagreement subargument. Thus, every argument has a set of potential counter-argument points that could be 'attacked' by others arguments. Note that here there is also an underlying consistency check procedure necessary to discover counter-arguments.

Example 3.1 : Continuing with Example 2.2, the argument $\mathcal{A}_1$ is a counter-argument for both $\mathcal{A}_2$ and $\mathcal{A}_3$ (at `fly(koko)`), and also $\mathcal{A}_2$ and $\mathcal{A}_3$ are counter-arguments for $\mathcal{A}_1$ (both at $\sim$`fly(koko)`). Note that the argument $\mathcal{A}_4$ has no counter-arguments. □

3.1 Finding Counter-arguments in Parallel

In order to find a counter-argument for a given argument $\mathcal{A}$, we need to find an argument $\mathcal{C}$ that disagree with some literal `p` in $\mathcal{A}$. Let $Co(\mathcal{A})$ be the set of consequents of EPC and DPC used for building $\mathcal{A}$ (excluding facts). Each element of $Co(\mathcal{A})$ is a potential counter-argument point. Actually, there could be more counter-argument points that do not belong to $Co(\mathcal{A})$, but this has been resolved using "inverted EPCs" (see [6] for a complete description of this solution). Therefore, in order to find a counter-argument for an argument $\mathcal{A}$, we will see if there exist an argument for the complement of an element of $Co(\mathcal{A})$.

Thus, once the argument $\mathcal{A}$ as been built, we can compute the set $Co(\mathcal{A})$, and then compute in parallel all the possible counter-arguments for $\mathcal{A}$ just trying to build an argument for the complement of each element of $Co(\mathcal{A})$. Observe that the derivation of each counter-argument will be independent of the others.

3.2 Defeaters and Justifications

In Defeasible Logic Programming a query `q` will succeed if the supporting argument for it is not defeated; it then becomes a *justification*. In order to verify whether an argument is non-defeated, its associated counter-arguments $\mathcal{B}_1$, $\mathcal{B}_2$, ... $\mathcal{B}_k$ are examined, each of them being a potential (defeasible) reason for rejecting $\mathcal{A}$. If any $\mathcal{B}_i$ is better than (or unrelated to) $\mathcal{A}$, then $\mathcal{B}_i$ is a candidate for defeating $\mathcal{A}$. In this work, we will use a formal criterion called *specificity* which allows to discriminate between two conflicting arguments. However, the notion of *defeating argument* can be formulated independently of which particular argument-comparison criterion is used. Namely, if some $\mathcal{B}_i$ is better (*i.e.*, more specific, in our case) than $\mathcal{A}$, then $\mathcal{B}_i$ is called a *proper defeater* for $\mathcal{A}$; if neither argument is better than the other, a blocking situation occurs, and we will say that $\mathcal{B}_i$ is a *blocking defeater* for $\mathcal{A}$.

Definition 3.1 (Defeating argument) *An argument* $\langle \mathcal{A}_1, h_1 \rangle$ defeats *an argument* $\langle \mathcal{A}_2, h_2 \rangle$ *at literal* h*, if and only if there exists a sub-argument* $\langle \mathcal{A}, h \rangle$ *of* $\langle \mathcal{A}_2, h_2 \rangle$ *such that* $\langle \mathcal{A}_1, h_1 \rangle$ *counter-argues* $\langle \mathcal{A}_2, h_2 \rangle$ *at* h*, and either:*

(1) $\langle \mathcal{A}_1, h_1 \rangle$ is strictly more specific than $\langle \mathcal{A}, h \rangle$ (then $\langle \mathcal{A}_1, h_1 \rangle$ is a proper *defeater of $\langle \mathcal{A}_2, h_2 \rangle$); or*
(2) $\langle \mathcal{A}_1, h_1 \rangle$ is unrelated by specificity to $\langle \mathcal{A}, h \rangle$ (then $\langle \mathcal{A}_1, h_1 \rangle$ is a blocking *defeater of $\langle \mathcal{A}_2, h_2 \rangle$).*

As defined above, a defeater $\mathcal{B}$ for $\mathcal{A}$ is a counter-argument that attacks a sub-argument $\mathcal{C}$ of $\mathcal{A}$, provided that $\mathcal{C}$ is not better than $\mathcal{B}$. Therefore, defeaters for $\mathcal{A}$ can be searched in parallel, computing counter-arguments in parallel, and using the comparison criterion once every counter-argument is obtained.

The next definition characterizes specificity as defined in [16, 22] (adapted to fit in this framework). Intuitively, this notion of specificity favors two aspects in an argument: it favors an argument (1) with more information content and (2) with shorter derivations. This notion is made formally precise in the next definition. We use the symbol "$|\sim$ " to denote a defeasible derivation,*i.e.*, $\mathcal{P} |\sim\ h$ means that h has a defeasible derivation from $\mathcal{P}$. Let $\mathcal{S}_G$ be the maximal subset of $\mathcal{S}$ that does not contain facts. Let $\mathcal{F}$ be the set of literals in $\mathcal{P}$ that have a defeasible derivation.

Definition 3.2 (Specificity)
An argument $\mathcal{A}_1$ for h_1 is strictly more specific than *an argument $\mathcal{A}_2$ for h_2 (denoted $\langle \mathcal{A}_1, h_1 \rangle \succ \langle \mathcal{A}_2, h_2 \rangle$) if and only if:*
(1) For all $G \subseteq \mathcal{F}$: if $\mathcal{S}_G \cup G \cup \mathcal{A}_1 |\sim\ h_1$ and $\mathcal{S}_G \cup G \not|\sim\ h_1$, then $\mathcal{S}_G \cup G \cup \mathcal{A}_2 |\sim\ h_2$.
(2) There exists $G' \subseteq \mathcal{F}$ such that $\mathcal{S}_G \cup G' \cup \mathcal{A}_2 |\sim\ h_2$ and $\mathcal{S}_G \cup G' \not|\sim\ h_2$ and $\mathcal{S}_G \cup G' \cup \mathcal{A}_1 \not|\sim\ h_1$.

Example 3.2 : Continuing with Example 2.2, argument $\mathcal{A}_1$ is strictly more specific than argument $\mathcal{A}_2$ because $\mathcal{A}_1$ does not use the EPC '`bird(X) <- chicken(X).`' and therefore is a more direct argument. However, argument $\mathcal{A}_3$ is strictly more specific than $\mathcal{A}_1$, because it contains more information. Then, $\mathcal{A}_1$ is a proper defeater for $\mathcal{A}_2$, and $\mathcal{A}_3$ is a proper defeater for $\mathcal{A}_1$. □

Since defeaters are arguments, there may exist defeaters for defeaters, and so on. In order to obtain a justification for a given query, a dialectical analysis is needed. The PROLOG program of Figure 1 shows the specification of this analysis (`\+` stands for PROLOG's negation as failure).

```
justify(Q) :- find_argument(Q,A), \+ defeated(A).
defeated(A) :- find_defeater(A,D), \+ defeated(D).
```

Figure 1: Justification specification

The specification of Figure 1 leads in a natural way to the use of trees to organize our dialectical analysis. In order to accept an argument $\mathcal{A}$ as a justification for q, a tree structure can be generated. The root of the tree will correspond to the argument $\mathcal{A}$ and every inner node will represent a defeater (proper or blocking) of its father. Leaves in this tree will correspond to non-defeated arguments. This structure is called a *dialectical tree*.

Definition 3.3 (Marking of a dialectical tree) *Let $\mathcal{A}$ be an argument for a literal h, and $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ be its associated dialectical tree. Nodes in $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ are recursively marked as* defeated *or* undefeated *nodes (D-nodes and U-nodes respectively) as follows.*

1. *Leaves of $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ are* U-nodes.
2. *Let $\langle \mathcal{B}, q \rangle$ be an inner node of $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$. Then $\langle \mathcal{B}, q \rangle$ will be an* U-node *iff every child of $\langle \mathcal{B}, q \rangle$ is a* D-node. *The node $\langle \mathcal{B}, q \rangle$ will be a* D-node *iff it has at least an* U-node *as a child.*

Definition 3.4 (Justification) *Let $\mathcal{A}$ be an argument for a literal h, and let $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ be its associated dialectical tree. The argument $\mathcal{A}$ will be a justification for a literal h if the root of $\mathcal{T}_{\langle \mathcal{A}, h \rangle}$ is a U-node.*

In order to explain how the current (sequential) system decides if an argument is a justification, it is useful to see a dialectical tree as a set of *argumentation lines*. Let $\langle \mathcal{A}_1, h_1 \rangle$ be an argument structure, and let $\mathcal{T}_{\langle \mathcal{A}_1, h_1 \rangle}$ be its associated dialectical tree. Every path λ in $\mathcal{T}_{\langle \mathcal{A}_1, h_1 \rangle}$ from the root $\langle \mathcal{A}_1, h_1 \rangle$ to a leaf $\langle \mathcal{A}_n, h_n \rangle$, denoted $\lambda = [\langle \mathcal{A}_1, h_1 \rangle, \langle \mathcal{A}_2, h_2 \rangle, \ldots, \langle \mathcal{A}_n, h_n \rangle]$, constitutes an *argumentation line* for $\langle \mathcal{A}_1, h_1 \rangle$. In each argumentation line, every argument $\langle \mathcal{A}_i, h_i \rangle$ defeats its predecessor $\langle \mathcal{A}_{i-1}, h_{i-1} \rangle$. The argument $\langle \mathcal{A}_1, h_1 \rangle$ supports the main query h_1, $\langle \mathcal{A}_2, h_2 \rangle$ interferes the justification of h_1 and so on. Therefore, an argumentation line can be split in two disjoint sets: λ_S of supporting arguments, and λ_I of interfering arguments.

In the current (sequential) implementation, given a query `q` the system will first try to generate an argument $\mathcal{A}_1$ for `q`. Then a dialectical tree will be generated in depth-first order, considering (from left to right) every argumentation line in a sequential order. First, the system will try to build a defeater $\mathcal{A}_2$ for some counter-argument point in $\mathcal{A}_1$ (see Figure 2). If such defeater exists, then it will try to build a defeater $\mathcal{A}_3$ for $\mathcal{A}_2$, and so on, building in this form an argumentation line. In a dialectical tree there are as many argumentation lines as leaves in the tree, and each of them could finish in a supporting or an interfering argument. It is interesting to note, that although the procedure of Figure 1 describes an exhaustive analysis, pruning it is also described by the semantics of PROLOG's negation as failure.

Example 3.1 Consider Figure 2 (i), if an argumentation line ends with a supporting argument $\mathcal{A}_5$, then the defeater $\mathcal{A}_2$ will be defeated (*i.e.*, will be a `D`-node), and therefore $\mathcal{A}_1$ will be –up to this point– a `U`-node. Although there could be more defeaters for $\mathcal{A}_4$, looking for them is useless because $\mathcal{A}_4$ is already defeated, hence, the dialectical tree can be pruned on this node. Since the system performs an exhaustive analysis, it will look for any other possible defeater $\mathcal{A}_4'$ for $\mathcal{A}_3$, creating a new argumentation line. Note that if an undefeated node is found for $\mathcal{A}_3$, the label for $\mathcal{A}_1$ will change from `U`-node, to `D`-node.

Consider now Figure 2 (ii), if the argumentation line ends with an interfering argument as $\mathcal{A}_4'$, then the defeater $\mathcal{A}_2$ will be undefeated (*i.e.*, a `U`-node), and $\mathcal{A}_1$ will be defeated. Again, there could be more defeaters for $\mathcal{A}_3$, but looking for them is also useless. Since the system is trying to prove that $\mathcal{A}_1$ is a justification, the

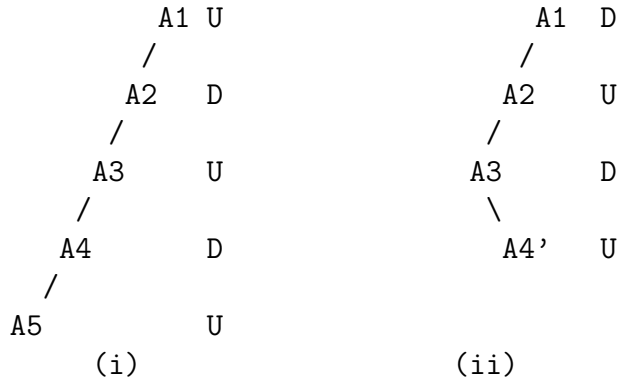

Figure 2: Argumentation lines of different length

dialectical tree is pruned in $\mathcal{A}_3$, and any other defeater $\mathcal{A}_3'$ for $\mathcal{A}_2$ will be sought, creating a new argumentation line. □

Before analyzing a parallel implementation for the dialectical tree, we will introduce some conditions in order to avoid infinite argumentation lines. A DLP is a finite set of program clauses, and therefore there is a finite number of arguments that may be involved in a dialectical tree. However, we need to impose conditions in order to avoid cycles in this tree. In [22], a detailed analysis exposes different kinds of fallacies: reciprocal defeaters, contradictory argumentation, and circular argumentation. These three situations are averted by requiring that all the argumentation lines of a dialectical tree to be *acceptable*. Let $\mathcal{P}$ be a DLP, where $\mathcal{S}$ is the set of EPC. Let $\lambda = [\langle \mathcal{A}_0, h_0 \rangle, \langle \mathcal{A}_1, h_1 \rangle, \ldots, \langle \mathcal{A}_i, h_i \rangle, \ldots, \langle \mathcal{A}_n, h_n \rangle]$ be an argumentation line.

Definition 3.5 (Acceptable argumentation line) *λ is an* acceptable argumentation line *iff:*

1. *For every defeater $\langle \mathcal{A}_i, h_i \rangle$ and every proper subargument $\langle \mathcal{B}, q \rangle$ of $\langle \mathcal{A}_i, h_i \rangle$, $\langle \mathcal{B}, q \rangle$ and $\langle \mathcal{A}_{i-1}, h_{i-1} \rangle$ are concordant; that is, $\mathcal{S} \cup \{q, h_{i-1}\}$ must be consistent.*
2. *The sets λ_S of supporting arguments, and λ_I of interfering arguments, of λ must each be concordant sets of arguments (i.e., $\mathcal{S} \cup \bigcup_{i=1}^{n} \mathcal{A}_i$ is consistent).*
3. *No argument $\langle \mathcal{A}_k, h_k \rangle$ in λ is a subargument of an earlier argument $\langle \mathcal{A}_i, h_i \rangle$ of λ ($i < k$).*

3.3 Finding Justifications in Parallel

Consider the dialectical tree of Figure 3. The root node $\mathcal{A}_1$ is a D-node, because there are two undefeated defeaters for it: $\mathcal{A}_3$ and $\mathcal{A}_5$. If depth-first search would be used over this tree, then the search would be stopped in node $\mathcal{A}_{12}$. However, in a breadth-first search the process would be stopped in node $\mathcal{A}_5$.

In the current (sequential) system the dialectical tree is explored in depth-first order. However, as shown above, given an argument $\mathcal{A}$, its defeaters can be computed in parallel. Therefore, the dialectical tree can be computed in parallel, if every time a node (argument) is obtained, its children (defeaters) are computed in parallel. This parallel process gives to the search tree a breadth-first flavor.

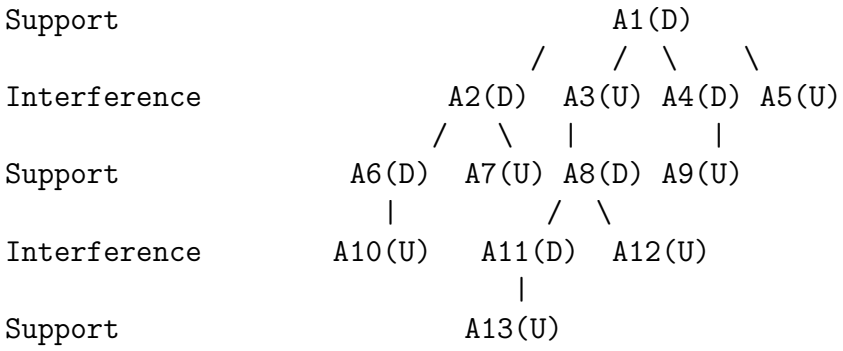

Figure 3: A complete dialectical tree

If the dialectical tree is computed in parallel, then the control of the justification process will be distributed over the nodes. Thus, the marking procedure of D-nodes and U-nodes will be done by message passing between the nodes of the dialectical tree. We will show next the interaction among the nodes, separating the construction of the dialectical tree from the marking procedure of nodes.

a) Parallel construction of the dialectical tree. After constructing $\mathcal{A}$, the following actions will be carried out:

1. The set $Co(\mathcal{A})$ of potential counter-argument points will be obtained.
2. For every $l \in Co(\mathcal{A})$ it will be called (in parallel) the construction of arguments for $\overline{l}$. These arguments (potential defeaters) will be the children of $\mathcal{A}$.
3. If $\mathcal{A}$ receives from a child $\mathcal{B}$ the message that its derivation fails, then the status of $\mathcal{B}$ will be dead.
4. If $\mathcal{A}$ receives from a child $\mathcal{B}$ (counter-argument) the message that its derivation succeeds, then $\mathcal{B}$ will be compared with the disagreement sub-argument $\mathcal{C}$ of $\mathcal{A}$.
   If $\mathcal{C}$ is better than $\mathcal{B}$, then the status of $\mathcal{B}$ will be dead.
   Otherwise, if $\mathcal{B}$ is better than $\mathcal{C}$ then $\mathcal{B}$ will be a defeater for $\mathcal{A}$, and $\mathcal{A}$ will send a message to the child $\mathcal{B}$ for generating its own defeaters (children). In the latter case the status of $\mathcal{B}$ will be potential defeater

The previous algorithm indicates how to build the dialectical tree. However, we also need to mark every node in order to know whether the root node is a U-node.

b) Parallel marking procedure of a dialectical tree. In order to mark a node as U-node or D-node, the following criterion is used:

1. If a node $\mathcal{B}$ (argument) has no defeaters then $\mathcal{B}$ sends a message to its father indicating it is a U-node.
2. If a node $\mathcal{A}$ receives from one of its children $\mathcal{B}$ the message that $\mathcal{B}$ is a U-node, then (as $\mathcal{B}$ defeats $\mathcal{A}$): (1) the node $\mathcal{A}$ becomes a D-node, (2) $\mathcal{A}$ sends a message to its father to inform that it is now a D-node, and (3) in order to prune the dialectical tree, $\mathcal{A}$ sends a message to its children that are still alive, to abort their dialectical process.

3. If a node $\mathcal{A}$ receives a message from one of its children $\mathcal{B}$, indicating that $\mathcal{B}$ becomes a D-node, then $\mathcal{B}$ status will be defeated defeater.

4. If every 'potential' defeater of $\mathcal{A}$ becomes a 'defeated' defeater, then $\mathcal{A}$ becomes a U-node, and sends a message to its father indicating this.

Given a query h and an argument $\mathcal{A}$ for h, the dialectical process will finish when the argument $\mathcal{A}$ becomes a U-node (*i.e.*, a justification for h), or when at least a defeater for $\mathcal{A}$ becomes a U-node and therefore $\mathcal{A}$ becomes a D-node.

If a query h has a justification, then it is considered 'justified', and the answer to the query will be YES. Nevertheless, there are several reasons for an argument not to be a justification: there may exist a non-defeated proper defeater, or a non-defeated blocking defeater, or there may be no argument at all. Therefore, in a DLP there are four possible answers for a query " $\prec$ h":

- YES, if there is a justification $\mathcal{A}$ for h.
- NO, if for each possible argument $\mathcal{A}$ for h, there exists at least one proper defeater for $\mathcal{A}$ marked as U-node.
- UNDECIDED, if for each possible argument $\mathcal{A}$ for h, there are no proper defeaters for $\mathcal{A}$ marked U-node, but there exists at least one blocking defeater for $\mathcal{A}$ marked U-node.
- UNKNOWN, if there exists no argument for h.

4 Related Work

In this work we define a parallel argumentation system based in the defeasible argumentation formalism developed in [24, 22, 23, 5]. However, there are other formalisms for defeasible argumentation. In [4] P. Dung has proposed a very abstract and general argument-based framework, where he completely abstracts from the notions of argument and defeat. H. Prakken and G. Sartor [20, 21] have developed an argumentation system inspired by legal reasoning. Like us, they use the language of extended logic programming, but they introduce a *dialectical proof theory* for an argumentation framework that fits the abstract format developed by Dung, Kowalski *et al.* [4, 2]. Later, Prakken [18] generalized the system to default logic's language. R. Kowalski and F. Toni [12] have outlined a formal theory of argumentation, in which defeasibility is stated in terms of non-provability claims. Other related works are those by Vreeswijk [25], Bondarenko [1], Pollock [15], Loui [13], and Nute [14].

The interested reader is referred to the following surveys in defeasible argumentation: Prakken & Vreeswijk [17], and Chesñevar *et al.* [3]. A complete bibliography of parallelism in logic programming can be found in [10] and [9]. Gupta *et al.* [11] and [10] are good surveys of implementation of parallelism in logic programming.

5 Conclusions

Different sources of parallelism for a defeasible argumentation system were studied. We considered only those forms of parallelism that could be exploited implicitly. We showed that all types of parallelism identified for logic programming can be used for obtaining consistent defeasible derivations. In particular, or-parallelism could be exploited in order to obtain all the possible arguments for a given query.

Once an argument $\mathcal{A}$ for q is found, defeaters for $\mathcal{A}$ can be sought in parallel. Thus the dialectical tree can be computed in parallel, giving to the search process a breadth-first flavor. A distributed process for finding justifications was developed. This process builds the dialectical tree and marks every node as D-node or U-node.

Although much work in defeasible argumentation, and also in parallel logic programming it is being pursued, to our knowledge, this work is the first approach to parallel defeasible argumentation.

References

[1] A. Bondarenko, P.M. Dung, R.A. Kowalski, and F. Toni. An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intelligence*, 93:63–101, 1997.

[2] A. Bondarenko, F. Toni, and R.A. Kowalski. An assumption-based framework for non-monotonic reasoning. *Proc. 2nd. International Workshop on Logic Programming and Non-monotonic Reasoning*, pages 171–189, 1993.

[3] C. I. Chesñevar, A. Maguitman, and R.P.Loui. Logical models of arguments. *submitted to ACM Computing Surveys*, 1998.

[4] Phan M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning and logic programming and n-person games. *Artificial Intelligence*, 77:321–357, 1995.

[5] Alejandro J. García. *Defeasible Logic Programming: Definition and Implementation* (MSc Thesis). Departamento de Cs. de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina, July 1997.

[6] Alejandro J. García and Guillermo R. Simari. Una extensión de la máquina abstracta de Warren para la argumentación rebatible. In *Proceedings of the III Congreso Argentino en Ciencias de la Computación*, October 1997.

[7] Alejandro J. García, Guillermo R. Simari, and Carlos I. Chesñevar. An argumentative framework for reasoning with inconsistent and incomplete information. In *Workshop on Practical Reasoning and Rationality*. 13th biennial European Conference on Artificial Intelligence (ECAI-98), August 1998.

[8] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In D. Warren and P. Szeredi, editors, *Proc. ICLP*, pages 579–597. MIT Press, 1990.

[9] Steve Gregory. *Parallel Logic Programming in PARLOG. The language and its implementation.* Addison-Wesley, 1987.

[10] Gopal Gupta. *Multiprocessor Execution of Logic Programs.* Kluwer Academic Publishers, 1994.

[11] Gopal Gupta, Khayri, A.M. Ali, Manuel Hermenegildo, and Mats Carlsson. Parallel execution of prolog programs: A survey. Technical report, Department of Computer Science, New Mexico State University. http://www.cs.nmsu.edu/lldap/pub_para/survey.html.

[12] Robert A. Kowalski and Francesca Toni. Abstract argumentation. *Artificial Intelligence and Law*, 4(3-4):275–296, 1996.

[13] Ronald P. Loui, Jeff Norman, Joe Altepeter, Dan Pinkard, Dan Craven, Jessica Lindsay, and Mark Foltz. Progress on room 5: A testbed for public interactive semi-formal legal argumentation. In *Proc. of the 6th. International Conference on Artifcial Intelligence and Law*, July 1997.

[14] Donald Nute. Basic defeasible logic. In Luis Fari nas del Cerro, editor, *Intensional Logics for Programming.* Claredon Press, Oxford, 1992.

[15] John L. Pollock. *Cognitive Carpentry: A Blueprint for How to Build a Person.* Massachusetts Institute of Technology, 1995.

[16] David L. Poole. On the Comparison of Theories: Preferring the Most Specific Explanation. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 144–147. IJCAI, 1985.

[17] H. Prakken and G. Vreeswijk. Logical systems for defeasible argumentation (to appear). In Gabbay, editor, *Handbook of Philosophical Logic, second edition.* 1998.

[18] Henry Prakken. *Logical Tools for Modelling Legal Argument. A Study of Defeasible Reasoning in Law.* Kluwer Law and Philosophy Library, 1997.

[19] Henry Prakken. Dialectical proof theory for defeasible argumentation with defeasible priorities (preliminary report). In *Proceedings of the 4th Modelage Workshop on Formal Models of Agents. Springer Lecture Notes on AI.* Springer Verlag, Berlin, 1998.

[20] Henry Prakken and Giovanni Sartor. A system for defeasible argumentation, with defeasible priorities. In *Proc. of the International Conference on Formal Aspects of Practical Reasoning, Bonn, Germany.* Springer Verlag, 1996.

[21] Henry Prakken and Giovanni Sartor. Argument-based logic programming with defeasible priorities. *Journal of Applied Non-classical Logics*, 7(25-75), 1997.

[22] Guillermo R. Simari, Carlos I. Chesñevar, and Alejandro J. García. The role of dialectics in defeasible argumentation. In *Anales de la XIV Conferencia Internacional de la Sociedad Chilena para Ciencias de la Computación.* Universidad de Concepción, Concepción (Chile), November 1994.

[23] Guillermo R. Simari and Alejandro J. García. A knowledge representación language for defeasible argumentation. In *CLEI'95, Canela, Brasil*, August 1995.

[24] Guillermo R. Simari and Ronald P. Loui. A Mathematical Treatment of Defeasible Reasoning and its Implementation. *Artificial Intelligence*, 53:125–157, 1992.

[25] Gerard A.W. Vreeswijk. Abstract argumentation systems. *Artificial Intelligence*, 90:225–279, 1997.