Parallel Defeasible Argumentation[1]

Alejandro J. García   Guillermo R. Simari

Grupo de Investigación en Inteligencia Artificial (GIIA)
Departamento de Ciencias de la Computación,
UNIVERSIDAD NACIONAL DEL SUR
Av.Alem 1253 – (8000) Bahía Blanca, ARGENTINA
FAX: (++54)(291)4595136
e-mail: `ccgarcia@criba.edu.ar`   `grs@criba.edu.ar`

Abstract

Implicitly exploitable parallelism for Logic Programming has received ample attention. *Defeasible Argumentation* is specially apt for this optimizing technique. *Defeasible Logic Programming* (DLP), which is based on a defeasible argumentation formalism, could take full advantage of this type of parallel evaluation to improve the computational response of its proof procedure. In a defeasible argumentation formalism, a conclusion q is accepted only when the argument $\mathcal{A}$ that supports q becomes a *justification*. To decide if an argument $\mathcal{A}$ is a justification a dialectical analysis is performed. This analysis considers arguments and counter-arguments. DLP extends conventional Logic Programming, capturing common sense reasoning features, and providing a knowledge representation language for defeasible argumentation.

Since DLP is an extension of Logic Programming, the different types of parallelism studied for Logic Programming could be applied. We propose new sources of parallelism that can be implicitly exploited in the defeasible argumentation formalism implemented through DLP. Both the argumentation process and the dialectical analysis benefit from exploiting those sources of parallelism.

Keywords: Defeasible Reasoning, Argumentation, Parallel Logic Programming

1 Introduction

Implicitly exploitable parallelism for Logic Programming has received ample attention. *Defeasible Argumentation* is specially apt for this optimizing technique. *Defeasible Logic Programming* (DLP) [5, 7], which is based on a defeasible argumentation formalism [32, 30, 9], could take full advantage of this type of parallel evaluation to improve the computational response of its proof procedure.

In a defeasible argumentation formalism, such as [32, 4, 26], a conclusion q is accepted only when the argument $\mathcal{A}$ that supports q becomes a *justification*. To decide if an argument $\mathcal{A}$ is a justification a dialectical analysis is performed. This analysis considers arguments and counter-arguments. DLP extends conventional Logic Programming, capturing common sense reasoning features, and providing a knowledge representation language for defeasible argumentation.

To verify whether an argument $\mathcal{A}$ is a justification, 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 $\mathcal{A}$. Each $\mathcal{B}_i$ that is better than (or unrelated to) $\mathcal{A}$, is a candidate for defeating $\mathcal{A}$, and it is called a *defeater* for $\mathcal{A}$. Since each defeater $\mathcal{B}_i$ is an argument, there may exist defeaters for the defeaters, and so on, thus requiring a complete dialectical analysis. The defeasible argumentation formalism proposed in [32, 30, 9], performs this dialectical analysis generating a *dialectical tree*: a tree of arguments where every node is defeater of its father. The analysis of this tree establishes whether the root argument $\mathcal{A}$ becomes a justification for the conclusion that $\mathcal{A}$ supports.

In [5], an extension of Logic Programming called *Defeasible Logic Programming* (DLP) was defined. DLP uses the language of Extended Logic Programming and provides the possibility of adding more

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

information, in the form of *weak rules.* These weak rules are the key element for introducing *defeasibility* [22] and they will be used to represent a relation between pieces of knowledge that could be *defeated* after all things are considered. In DLP, the argumentation formalism developed in [32, 30, 9], is used for deciding between contradictory goals through a dialectical analysis. Thus DLP extends conventional Logic Programming, capturing common sense reasoning features, and also provides a knowledge representation language for defeasible argumentation.

Since DLP is an extension of Logic Programming, the different types of parallelism studied for Logic Programming could be applied. In this work we propose new sources of parallelism that can be implicitly exploited in the defeasible argumentation formalism implemented through DLP. Both the argumentation process and the dialectical analysis benefit from exploiting those sources of parallelism.

In Logic Programming, OR-parallelism, AND-parallelism, and also unification parallelism can be implicitly performed [12, 15]. As we will show next, these three kinds of parallelism can be exploited directly by DLP. However, there are new sources of parallelism that can be implicitly exploited in a defeasible argumentation formalism:

1. several arguments for a conclusion q can be constructed in parallel,
2. once an argument $\mathcal{A}$ for q is found, defeaters $\mathcal{B}_1$, $\mathcal{B}_2$, $\ldots$ $\mathcal{B}_k$ for $\mathcal{A}$ can be searched in parallel, and
3. several argumentation lines of the dialectical tree can be explored in parallel.

All these sources of parallelism for defeasible argumentation provide both a form of speeding up the dialectical analysis and a form of distributing the process of argumentation.

This work is organized as follows: proposed parallelism for Logic Programming will be first introduced. Then, the language of DLP will be described, showing how current parallelism can be applied. In later sections new sources of parallelism will be studied. We will consider only those forms of parallelism that could be exploited implicitly, *i.e.*, without the intervention of the programmer. In particular, OR-parallelism can be exploited in the process of obtaining all the possible arguments for a given query, and in the distributed process for finding justifications.

Although much work in the areas of Defeasible Argumentation and Parallel Logic Programming it is being pursued, to our knowledge, the approach presented here is the first considering Parallel Defeasible Argumentation.

2 Parallelism in Defeasible Logic Programs

Defeasible Logic Programming is an extension of Logic Programming. Naturally, the starting point for studying parallelism in DLP will be the currently proposed parallelism for conventional Logic Programming. In this section we will briefly introduce different types of parallelism studied for Logic Programming, and then we will show how these techniques can be applied in DLP.

2.1 Parallelism in Logic Programming

Logic Programming is a paradigm based on a subset of First-Order Predicate Logic, (we refer the interested reader to the following sources in Logic Programming [19, 18]). Before considering the issues involved in introducing parallelism for Logic Programs we will recall the language of Definite Logic Programming. A *Definite Program Rule* is an ordered pair, conveniently denoted '$Head \leftarrow Body$', whose first component, *Head*, is an atom, and whose second component, *Body*, is a finite set of atoms. A rule with the head A and body $\{Q_1, \ldots, Q_m\}$ can also be written as: $A \leftarrow Q_1, \ldots, Q_n$. As usual, if the body is empty, then a rule becomes " $A \leftarrow true$" (or simply 'A') and it is called a *fact.* A *definite goal* or a *query* for a logic program consist of a rule with empty head '$\leftarrow Q_1, \ldots, Q_n$'. A *Definite Logic Program* is a set of definite program rules.

To find answers to queries, an *evaluator* of the program that uses a resolution algorithm is needed. In Logic Programming there are many refutation procedures based on the resolution inference rule, which are refinements of the Robinson's original procedure [29]. We include in this section the definition of SLD-resolution[2]. for definite rules given in [19].

[2]SLD-resolution stands for Linear resolution whit Selection function for Definite rules

Definition 2.1 *Let G be a definite goal $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ and C be definite program rule $A \leftarrow B_1, \ldots, B_q$. Then G' is* derived *from G and C using the most general unifier θ if the following conditions holds:*

- *A_m is an atom, called the selected atom in G.*
- *θ is a most general unifier for A_m and A.*
- *G' is the query $\leftarrow A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k$*

In resolution terminology, G' is called a resolvent *for G and C.*

Definition 2.2 *Let P be a definite program and G a definite goal. An SLD-derivation consists of a sequence (finite or infinite) $G_0{=}G, G_1, \ldots$ of goals and a sequence $C_1, C_2, \ldots$ of variants of program rules of P and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that each G_{i+1} is derived from G_1 and C_{i+1} using θ_{i+1}.*

Definition 2.3 *An SLD-refutation of $P \cup \{G\}$ is a finite SLD-derivation of $P \cup \{G\}$ which has the empty rule as the last goal in the derivation. If G_n is the empty rule, then we say that the refutation has length n.*

Thus, the computation process of the evaluator consists of selecting a subgoal G in the body of a query, and searching for the head of a rule $H \leftarrow B$ in the program that unifies with G. If unification succeeds, then the selected subgoal G is replaced by the subgoals B of the rule and the unifying substitution is applied to the resulting collection of subgoals. The process of subgoal selection and reduction is repeated until no more subgoals remain, in which case a solution to the query is said to be found.

The evaluator of a logic program has considerable freedom (nondeterminism) in selecting which reduction paths to follow in order to solve a query [15, 12]. To solve a query "$\leftarrow Q_1, \ldots, Q_n$" it has the freedom of selecting which atom Q_i to solve first. Also, several rules in the program may unify with the selected atom Q_i, and the evaluator has freedom (again) in selecting any of these rules. Therefore, if a chosen rule does not lead to a solution, other rules may be tried until either a solution is found or all rules are exhausted. This freedom is the origin of two types of non-determinism:

1. nondeterminism-1: if several rule heads unify with the selected goal, it is non determinate which of them is to be selected. The policy used by the evaluator (called the *search rule*) results in two (sub)types of non-determinism:
   "don't care" non-determinism: once a choice is made the system commits to that choice.
   "don't know" non-determinism: more than one of the possible choices may eventually be tried in the search for a solution.
2. nondeterminism-2: if the current query has several goals, it is non determinate which of them will be selected next for execution. The policy used by the program evaluator is called the *computation rule.*

The relation between Logic Programming and parallelism [15] is based on this "freedom" in choosing execution paths, because a possibility which remains open in the formulation of resolution *is executing several of those paths in parallel.* The two types of non-determinism introduced above are the origins of the two main sources of parallelism in Logic Programs: *Or-parallelism* and *And-parallelism.*

1. *Or-parallelism* arises when a subgoal unifies with the heads of more that one rule, and the subgoals in the bodies of these rules are executed in parallel.
2. *And-parallelism* arises when multiple subgoals in a query or in the body of a rule are executed in parallel.

As remarked by Gopal Gupta in [12], 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. In these systems the programmer has to think explicitly in terms of parallel processes, thus sacrificing declarativeness. 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.

We will consider only approaches that implicitly exploit parallelism, *i.e.*, without intervention of the programmer. Four main forms of parallelism (implicitly exploitable) can be identified in logic programs [12]:

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

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 that are to the right of an infinite branch can be computed. As explained above *And-parallelism* arises when multiple subgoals in a query or in the body of a rule 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. *Unification parallelism* arises when different argument terms (also subterms) can be unified in parallel.

Gopal Gupta points out in [12] 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.

Finally, other low-levels types of parallelism are also possible [15]: *Stream Parallelism*, when several processes can evaluate complex data structures incrementally, in parallel with the process which is producing them; and *Search Parallelism*, when the program itself can be divided into disjoint sets of rules that several processes can search for rules whose heads unify with a given goal in parallel, each of them working in a different set.

2.2 Defeasible Logic Programming

Research in Logic Programming has striven to capture nonmonotonic features seeking to develop a more powerful tool for knowledge representation and common sense reasoning. Defeasible Logic Programming [5, 7] advances on this problem by handling contradictory programs and by providing the possibility of adding information in the form of *weak rules*. These weak rules are the key element for introducing *defeasibility* [22]. In our opinion, defeat should be the result of a global analysis of the corpus of knowledge of the agent performing the inference. Thus, in Defeasible Logic Programming (*DLP*), an argumentation formalism is used for deciding between contradictory goals through a dialectical analysis.

Following Gelfond and Lifschitz's definition for *Extended Logic Programs* in [10] literals will be allowed in the head and the body of a program rule, and default negation just in the body. In DLP a *literal* "L" is an atom "A" or a negated atom "$\sim A$", and an *extended literal* X is a literal possibly preceded by a default negation "*not*". Two kinds of rules will be part of the language, *Extended Strong Rules* and *Extended Defeasible Rules* according to the definitions bellow.

Definition 2.4 (Extended Strong Rule) *An Extended Strong Rule is an ordered pair, conveniently denoted Head $\leftarrow$ Body, whose first component, Head, is a literal, and whose second component, Body, is a finite set of extended literals. An extended strong rule with the head L and body* $\{X_1, \ldots, X_m\}$ *can also be written as:* $L \leftarrow X_1, \ldots, X_n$. *As usual, if the body is empty, then an extended strong rule becomes "$L \leftarrow true$" (or simply "L") and it is called a* fact.

Extended Defeasible Rules, to be defined below, will add a new representational capability for expressing a weaker link between the head and the body in a rule. An extended defeasible rule "*Head* $-\!\!<$ *Body*" is understood as expressing that "*reasons to believe in the antecedent Body provide reasons to believe in the consequent Head*" [32]. Extended defeasible rules also allow the use of default negation in the body of the rule with the obvious meaning.

Definition 2.5 (Extended Defeasible Rule) *An Extended Defeasible Rule is an ordered pair conveniently denoted Head* $\prec$ *Body, whose first component, Head, is a literal, and whose second component, Body, is a finite set of extended literals. An extended defeasible rule with head L and body* $\{X_1, \ldots, X_n\}$ *can also be written as:* $L \prec X_1, \ldots, X_n$*. If the body is empty, we will write "* $L \prec true$*" and we will call it a* presumption.

Syntactically, the symbol "$\prec$" is all that distinguishes a defeasible rule from a strong rule. Pragmatically, a defeasible rule is used to represent defeasible knowledge, *i.e.*, tentative information that may be used if nothing could be posed against it. Thus, whereas a strong rule is used to represent non-defeasible information such as "$bird(X) \leftarrow penguin(X)$", which expresses that *"all penguins are birds."*, a defeasible rule is used to represent defeasible knowledge such as "$flies(X) \prec bird(X)$" which expresses that *"usually, a bird can fly."*

An (Extended) *Defeasible Logic Program* (DLP) is a finite set of extended strong and defeasible rules. If $\mathcal{P}$ is a DLP, we will distinguish the subset Π of extended strong rules and the subset Δ of extended defeasible rules in $\mathcal{P}$. When required we will denote $\mathcal{P}$ as (Π, Δ). Whenever possible we will drop the "extended" qualifier for rules and programs.

A *defeasible query* (or simply a query) is a defeasible rule with empty consequent denoted "$\prec Q_1, \ldots, Q_n$", where each Q_i $(1 \leq i \leq n)$ is an extended literal.

Here is an example of a DLP program:

Example 2.1 :

$fly(X) \prec bird(X)$	$bird(X) \leftarrow chicken(X)$
$\sim fly(X) \prec chicken(X)$	$bird(X) \leftarrow duck(X)$
$fly(X) \prec chicken(X), scared(X)$	$bird(X) \leftarrow penguin(X)$
$chicken(tina) \prec true$	$\sim fly(X) \leftarrow penguin(X)$
$penguin(tweety) \prec true$	$penguin(chilly)$
$duck(mark)$	$scared(tina)$
$chicken(jean)$	$duck(tim)$

□

Given a DLP $\mathcal{P}$ and a defeasible query Q, an *SL-Defeasible Refutation* of $\mathcal{P} \cup \{Q\}$ is a finite sequence $C_1, C_2, \ldots, C_n$ of variants of strong or defeasible rules of $\mathcal{P}$, provided there exists a sequence $Q{=}Q_0, Q_1, \ldots, Q_n$ of defeasible queries and a sequence $\theta_1, \theta_2, \ldots, \theta_n$ of mgu's such that each Q_{i+1} is derived from Q_i and C_{i+1} using θ_{i+1}, and Q_n is the empty rule. If there exist an SL-defeasible refutation for Q, then the finite set of rules used in the refutation constitutes the *defeasible derivation* for Q. A defeasible derivation for a negated literal "$\sim p$" is carried out just as if the "$\sim$" symbol were a part, syntactically, of the predicate name, thereby treating "$\sim p$" as an atomic predicate name.

Using the program of example 2.1 there are defeasible derivations for each of the following literals: "$\sim fly(tweety)$", "$fly(tweety)$", "$fly(tina)$" and "$\sim fly(tina)$." Therefore, the notion of defeasible derivation does not forbid the inference of 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.

2.3 Parallel Defeasible Derivations

The knowledge representation language defined above is an extension of the language of logic programming. Moreover, the defeasible derivation notion is defined exactly as the SLD-refutation procedure, without distinguishing between strong and defeasible rules, and considering the symbol "$\sim$" as part of predicate names.

Therefore, an evaluator for defeasible logic programs, that computes defeasible derivations, will have the same "freedom" (nondeterminism) that it has in Logic Programming for selecting which reduction paths to follow in order to solve a query. Thus, all sources of parallelism identified (and to be developed) for logic programming can be exploited directly for performing defeasible derivations. And also many of the implementations developed for logic programming can be applied directly to DLP.

However, in a defeasible argumentation formalism, defeasible derivations are only one step in the process of justifying queries through a dialectical analysis. Therefore, we are interested in studying new sources of parallelism that can be implicitly exploited in defeasible argumentation. This will be done in the following sections.

3 Arguments in Parallel

A defeasible logic program $\mathcal{P}$ is *contradictory* if and only if, it is possible to defeasibly derive from $\mathcal{P}$ a pair of contradictory literals. Two literals are contradictory if they are complementary with respect to strong negation. Thus, *"flies(tina)"* and *"∼flies(tina)"* are contradictory literals. Observe that the program $\mathcal{P}$ of Example 2.1 is a contradictory program, because defeasible derivations for the contradictory literals mentioned above can be obtained.

Logic programming with strong negation was introduced in Extended Logic Programming (ELP) [10]. However, in ELP, when a pair of contradictory literals can be derived, the set *Lit* of all literals can be derived without considering any further analysis. In our opinion, when contradictory goals could be defeasibly derived, there should be a formal criterion for deciding between them. DLP uses a defeasible argumentation formalism in order to perform such a task.

The use of strong negation in program rules enriches language expressiveness, and also allows the representation of contradictory knowledge. However, if $\mathcal{P} = (\Pi, \Delta)$ is a DLP, the set Π of strong rules is used to represent non-defeasible information, so it must express certain internal coherence. Therefore, from now on, we will assume that in every DLP $\mathcal{P}$ the set Π is non-contradictory. If a contradictory set Π is used in a DLP then the same problems of Extended Logic Programming [10] will appear. Although the set Π must be non-contradictory, the set Δ, and hence $\mathcal{P}$ itself (*i.e.*, $\Pi \cup \Delta$), may be contradictory. It is only in this form that a DLP may contain contradictory information. Observe that the DLP of example 2.1 is a contradictory program, but its set Π is not.

In defeasible argumentation, answers to queries must be supported by *arguments*. An argument is a minimal and non contradictory set of rules used to derive a conclusion.

Definition 3.1 (Argument) *An* argument $\mathcal{A}$ *for a query* h*, also denoted* $\langle \mathcal{A}, h\rangle$*, is a subset of* ground instances *of* DPCs *of* Δ*, such that:*

1. *There exists a defeasible derivation for* h *from* $\Pi \cup \mathcal{A}$*,*
2. $\Pi \cup \mathcal{A}$ *is non contradictory, and*
3. $\mathcal{A}$ *is minimal with respect to set inclusion.*

Example 3.1 : Consider the DLP of example 2.1.
The query "$\sim fly(tina)$" has the argument:

$$\mathcal{A}_1 = \left\{ \begin{array}{l} \sim fly(tina) \prec chicken(tina) \\ chicken(tina) \prec true \end{array} \right\}$$

whereas, the query "$fly(tina)$" has two arguments:

$$\mathcal{A}_2 = \left\{ \begin{array}{l} fly(tina) \prec bird(tina) \\ chicken(tina) \prec true \end{array} \right\}$$

$$\mathcal{A}_3 = \left\{ \begin{array}{l} fly(tina) \prec chicken(tina), scared(tina) \\ chicken(tina) \prec true \end{array} \right\}$$

The query "$\sim fly(tweety)$" has the argument:

$$\mathcal{A}_4 = \left\{ \; penguin(tweety) \prec true \; \right\}$$

but, there is no argument for "$fly(tweety)$" because its defeasible derivation is contradictory with respect to Π. □

3.1 Checking for Contradictions in Parallel

To obtain an argument for a literal h, contradictions may be checked simultaneously with the defeasible derivation. Every time the body of a program rule R is derived, it will be tested whether the head h of R (with its current variable bindings) is non contradictory. A head h will be contradictory when its complement $\overline{h}$ [3] can be derived from $\Pi \cup \mathcal{A}$. On the other hand, if a head h is certified as non contradictory, then it will be 'remembered' as a *temporary fact*. These temporary facts will be used for checking for contradictions in the following derived sub-queries, and also for avoiding re-derivation of previous derived goals. We call them temporary facts, because they do not remain 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].

A defeasible derivation is carried out in a top-down fashion, whereas checking for contradictions and the generation of temporary facts are done in a bottom-up manner. Thus, a literal L is checked for contradictions once the literal L has been derived. This is necessary because the goal being checked must be instantiated. Indeed, if 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 is checked for contradictions before the variable X is instantiated, then there is no argument for any instance, because there is a derivation for "$\sim fly(chilly)$" (that uses only strong rules) that would invalidate them. However, there is a non contradictory defeasible derivation with the substitutions $X = tina$ or $X = mark$.

Our implementation for checking for contradictions consists of adding an extra query at the end of every rule (see [6, 5] for details). The added query is the complement of the head of the rule. For example, the strong rule $p(X) \leftarrow q(X)$ is transformed into $p(X) \leftarrow q(X)\#{\sim}p(X)$, and the defeasible rule ${\sim}r(X) \prec {\sim}s(X)$ is transformed into ${\sim}r(X) \prec {\sim}s(X)\#r(X)$. The symbol "#" must be read as "and cannot be strongly derived that".

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

3.2 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 rule, 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.

Example 3.2 : If the query "$fly(X)$" is submitted to the program of Example 2.1 then the following arguments may be obtained (in parallel):

$\{\ (fly(tina) \prec bird(tina)),\ (chicken(tina) \prec true)\ \}$
$\{\ (fly(tina) \prec chicken(tina), scared(tina)),\ (chicken(tina) \prec true)\ \}$
$\{\ fly(mark) \prec bird(mark)\ \}$
$\{\ fly(jean) \prec bird(jean)\ \}$
$\{\ fly(tim) \prec bird(tim)\ \}$
$\square$

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

4 Finding Defeaters in Parallel

An argument $\mathcal{A}$ is a non contradictory 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

[3]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 $\overline{{\sim}a} = a$).

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 $\Pi \cup \{h_1, h\}$ is contradictory. 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 procedure for checking for contradictions, necessary to discover counter-arguments.

Example 4.1 : Continuing with Example 3.1, the argument $\mathcal{A}_1$ is a counter-argument for both $\mathcal{A}_2$ and $\mathcal{A}_3$ (at the point $fly(tina)$), and also $\mathcal{A}_2$ and $\mathcal{A}_3$ are counter-arguments for $\mathcal{A}_1$ (both at `~fly(tina)`). Note that the argument $\mathcal{A}_4$ has no counter-arguments. □

To find a counter-argument for a given argument $\mathcal{A}$, we need to find an argument $\mathcal{C}$ that disagrees with some literal p in $\mathcal{A}$. Let $Co(\mathcal{A})$ be the set of consequents of he strong rules and defeasible rules 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 exists an argument for the complement of an element of $Co(\mathcal{A})$.

Thus, once the argument $\mathcal{A}$ has 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.

The defeasible argumentation formalism uses a formal criterion called *specificity* which let us to discriminate between two conflicting arguments. However, the notion of *defeating argument* can be independently formulated 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 4.1 (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 for* $\langle \mathcal{A}_2, h_2 \rangle$*); 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 for* $\langle \mathcal{A}_2, h_2 \rangle$*).*

As defined above, a defeater $\mathcal{B}$ for $\mathcal{A}$ is a counter-argument that attacks a subargument $\mathcal{C}$ 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.

We have formally defined a particular criterion called *specificity* (see [23, 32]) which allows discrimination between two conflicting arguments. Intuitively, this notion of specificity favors two aspects in an argument: it prefers an argument (1) with greater information content and/or (2) with less use of defeasible rules. In other words, an argument is deemed better than another if it is *more precise* and/or *more concise.* This notion is made formally precise in the next definition. We use the symbol "$\mid\!\sim$ " to denote a defeasible derivation (*i.e.*, $\mathcal{P} \mid\!\sim h$ means that h has a defeasible derivation from $\mathcal{P}$), and the symbol "$\vdash$" to denote a derivation where only strong rules are used.

Definition 4.2 (Specificity) *Let* $\mathcal{P}$ *be* (Π, Δ). *Let* Π_G *be the maximal subset of* Π *that does not contain facts. Let* $\mathcal{F}$ *be the set of literals that have a defeasible derivation from* $\mathcal{P}$. *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 $H \subseteq \mathcal{F}$*: if* $\Pi_G \cup H \cup \mathcal{A}_1 \mid\!\sim h_1$ *and* $\Pi_G \cup H \not\vdash h_1$,
then $\Pi_G \cup H \cup \mathcal{A}_2 \mid\!\sim h_2$
2. There exists $H' \subseteq \mathcal{F}$ *such that* $\Pi_G \cup H' \cup \mathcal{A}_2 \mid\!\sim h_2$ *and* $\Pi_G \cup H' \not\vdash h_2$
and $\Pi_G \cup H' \cup \mathcal{A}_1 \not\mid\!\sim h_1$

Example 4.2 : Continuing with Example 3.1, argument $\mathcal{A}_1$ is strictly more specific than argument $\mathcal{A}_2$ because $\mathcal{A}_1$ does not use the strong rule '$bird(X) \leftarrow chicken(X)$' 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 $\mathcal{A}_1$. □

5 Parallel Dialectical Analysis

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 naturally leads 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 called a *dialectical tree* 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. The formal definition follows:

Definition 5.1 (Dialectical tree) *Let $\mathcal{A}$ be an argument for h. A* dialectical tree *for $\langle\mathcal{A}, h\rangle$, denoted $\mathcal{T}_{\langle\mathcal{A},\,h\rangle}$, is recursively defined as follows:*
1. A single node labeled with an argument $\langle\mathcal{A}, h\rangle$ with no defeaters (proper or blocking) is by itself the dialectical tree for $\langle\mathcal{A}, h\rangle$.
2. Let $\langle\mathcal{A}_1, h_1\rangle$, $\langle\mathcal{A}_2, h_2\rangle$, ..., $\langle\mathcal{A}_n, h_n\rangle$ be all the defeaters (proper or blocking) for $\langle\mathcal{A}, h\rangle$. We construct the dialectical tree for $\langle\mathcal{A}, h\rangle$, $\mathcal{T}_{\langle\mathcal{A},\,h\rangle}$, by labeling the root node with $\langle\mathcal{A}, h\rangle$ and by making this node the parent node of the roots of the dialectical trees for $\langle\mathcal{A}_1, h_1\rangle$, $\langle\mathcal{A}_2, h_2\rangle$, ..., $\langle\mathcal{A}_n, h_n\rangle$.

Certain conditions are required in order to avoid the occurrence of cycles in the dialectical tree. It has been shown elsewhere[4] that circular argumentation is a particular case of *fallacious argumentation*. In order to avoid fallacious argumentation, certain conditions will be established over every *argumentation line* of the dialectical tree. Following [30], this notion is formally defined as follows. Let $\langle\mathcal{A}_0, h_0\rangle$ be an argument, and let $\mathcal{T}_{\langle\mathcal{A}_0,\,h_0\rangle}$ be its associated dialectical tree, every path λ from the root $\langle\mathcal{A}_0, h_0\rangle$ to a leaf $\langle\mathcal{A}_n, h_n\rangle$ in $\mathcal{T}_{\langle\mathcal{A}_0,\,h_0\rangle}$, denoted $\lambda = [\langle\mathcal{A}_0, h_0\rangle, \langle\mathcal{A}_1, h_1\rangle, \langle\mathcal{A}_2, h_2\rangle, \ldots, \langle\mathcal{A}_n, h_n\rangle]$, is called an *argumentation line* for h_0.

In each argumentation line $\lambda = [\langle\mathcal{A}_0, h_0\rangle, \langle\mathcal{A}_1, h_1\rangle, \ldots, \langle\mathcal{A}_i, h_i\rangle, \ldots, \langle\mathcal{A}_n, h_n\rangle]$, the argument $\langle\mathcal{A}_0, h_0\rangle$ is supporting the main query h_0, and every argument $\langle\mathcal{A}_i, h_i\rangle$ defeats its predecessor $\langle\mathcal{A}_{i-1}, h_{i-1}\rangle$. Therefore, for $k \geq 0$, $\langle\mathcal{A}_{2k}, h_{2k}\rangle$ is a *supporting* argument for h_0 and $\langle\mathcal{A}_{2k-1}, h_{2k-1}\rangle$ is an *interfering* argument for h_0. In other words, every argument in the line either supports h_0's justification or interferes with it. Therefore, an argumentation line can be split in two disjoint sets: λ_S of supporting arguments, and λ_I of interfering arguments.

Fallacious argumentation is avoided by requiring that all argumentation lines be *acceptable*. 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, λ is an *acceptable argumentation line* iff: (1) The sets λ_S of supporting arguments, and λ_I of interfering arguments of λ must each be non-contradictory sets of arguments. (2) 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 $(i < k)$. Hence, with these conditions averting undesirable situations, an *acceptable dialectical tree* is a dialectical tree where every argumentation line is acceptable.

Definition 5.2 (Marking of a dialectical tree) *Let $\langle\mathcal{A}, h\rangle$ be an argument and $\mathcal{T}_{\langle\mathcal{A},\,h\rangle}$ its acceptable dialectical tree, then:*
1. All the leaves in $\mathcal{T}_{\langle\mathcal{A},\,h\rangle}$ are marked as 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 a U-node iff every child of $\langle\mathcal{B}, q\rangle$ is a D-node. The node $\langle\mathcal{B}, q\rangle$ will be marked as a D-node iff it has at least a child marked as a U-node.

Thus, the notion of *justification* can be properly defined as follows:

Definition 5.3 (Justification) *Let $\mathcal{A}$ be an argument for a literal h, and let $\mathcal{T}_{\langle\mathcal{A},\,h\rangle}$ be its associated acceptable dialectical tree. The argument $\mathcal{A}$ for a literal h will be a justification iff the root of $\mathcal{T}_{\langle\mathcal{A},\,h\rangle}$ is a U-node.*

[4]We refer the interested reader to [30] where a detailed analysis of ill-formed reasoning dialogues are analyzed and definitions are introduced in order to avoid them.

If a query h has a justification, then it is considered 'justified', and the answer to the query will be YES. Nevertheless, there are other possible outcomes: 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, in the dialectical tree $\mathcal{T}_{\langle \mathcal{A},\, h\rangle}$ there exists at least one proper defeater for $\mathcal{A}$ marked as U-node.
- UNDECIDED, if there is no justification for h, and there exists an argument $\mathcal{A}$ for h, such that in its dialectical tree $\mathcal{T}_{\langle \mathcal{A},\, h\rangle}$ 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.

As described by the specification of Figure 1, in the current (sequential) implementation, given a query q the justification procedure will first try to generate an argument $\mathcal{A}_1$ for q. If no argument is found, the answer to q will be UNKNOWN. But if an argument $\mathcal{A}_1$ for q is found, then the justification procedure will try to build a defeater $\mathcal{A}_2$ for some counter-argument point in $\mathcal{A}_1$ (see the example below). If such defeater exists, it will try to build a defeater $\mathcal{A}_3$ for $\mathcal{A}_2$, and so on, building in this form an argumentation line.

Thus, a dialectical tree will be generated in depth-first order, considering (from left to right) every argumentation line. 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.

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. The following example shows how an argumentation line is constructed, considering supporting and interfering arguments, and how the marking procedure of U-nodes and D-nodes behaves with pruning.

$$\begin{tikzcd}
 & & & & \langle \mathcal{A}_1, h_1\rangle^U \arrow[dl, dash] \\
 & & & \langle \mathcal{A}_2, h_2\rangle^D \arrow[dl, dash] & \\
 & & \langle \mathcal{A}_3, h_3\rangle^U \arrow[dl, dash] & & \\
 & \langle \mathcal{A}_4, h_4\rangle^D \arrow[dl, dash] & & & \\
\langle \mathcal{A}_5, h_5\rangle^U & & & &
\end{tikzcd}$$

(i)

$$\begin{tikzcd}
 & & \langle \mathcal{A}_1, h_1\rangle^D \arrow[dl, dash] \\
 & \langle \mathcal{A}_2, h_2\rangle^U \arrow[dl, dash] & \\
\langle \mathcal{A}_3, h_3\rangle^D \arrow[dr, dash] & & \\
 & \langle \mathcal{A}'_4, h'_4\rangle^U &
\end{tikzcd}$$

(ii)

Figure 2: Argumentation lines of different length

Example 5.1 Suppose that, in order to find a justification for h_1, the argument $\mathcal{A}_1$ was found, and the argumentation line [$\langle \mathcal{A}_1, h_1\rangle$, $\langle \mathcal{A}_2, h_2\rangle$, $\langle \mathcal{A}_3, h_3\rangle$, $\langle \mathcal{A}_4, h_4\rangle$, $\langle \mathcal{A}_5, h_5\rangle$] was built (see Figure 2 (i)). In this situation, the argumentation line ends with the supporting argument $\mathcal{A}_5$, so the marking procedure establishes that $\langle \mathcal{A}_1, h_1\rangle$ is –up to this point– a U-node. However, the justification process cannot finish there because there could be more defeaters to consider. Therefore, the process will continue expanding other argumentation lines.

First note that there could be more defeaters for $\mathcal{A}_4$. However, $\mathcal{A}_4$ will continue to be a D-node because of $\mathcal{A}_5$, and will not change its status with the new defeaters. Therefore the tree can be pruned there without considering further defeaters for $\mathcal{A}_4$.

However, the previous analysis does not apply to $\mathcal{A}_3$, because if an undefeated defeater is found for it, the mark of $\mathcal{A}_3$ could change. It is for this reason that any other possible defeater $\mathcal{A}_4'$ for $\mathcal{A}_3$ is searched, creating a new argumentation line (see Figure 2 (ii)).

If a defeater $\mathcal{A}_4'$ is found (with no defeaters), then the argumentation line will end with an interfering argument, and therefore $\mathcal{A}_1$ will be a D-node (see Figure 2 (ii)). Again, pruning could be effected, because although there could be more defeaters for $\mathcal{A}_3$, they cannot modify the status of $\mathcal{A}_3$. However, there might be another defeater $\mathcal{A}_3'$ for $\mathcal{A}_2$, creating, in that case, a new argumentation line. □

5.1 Finding Justifications in Parallel

Consider the acceptable dialectical tree of Figure 3. The root node $\mathcal{A}$ is marked as D-node, because there is an undefeated defeater for it: $\mathcal{B}_3$. If depth-first search were used over this tree, then the whole tree would be explored. However, in a breadth-first search the dialectical process would be stopped after considering node $\mathcal{B}_3$ in the second level of the tree.

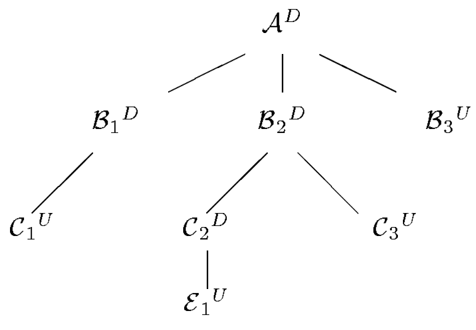

Figure 3: Marked dialectical tree

In the current (sequential) implementation 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 the search tree a breadth-first flavor.

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. Next, we will introduce a model for building justifications in parallel. For the sake of clarity we will separate the construction of the dialectical tree, from the marking procedure of the nodes. Our design goal is to delegate as much work as possible to the children process.

a) Parallel construction of the dialectical tree:
After the construction of an argument $\mathcal{A}$ by a local processor P, the following actions will be carried out:

1. The set $Co(\mathcal{A})$ of potential counter-argument points will be obtained by the local processor P.
2. For every $l \in Co(\mathcal{A})$ the construction of arguments for $\bar{l}$ will be called in parallel. These arguments are potential defeaters, and some of them could be the children of $\mathcal{A}$.
3. Once, the parallel calls are fired, the local processor P could also execute one of these parallel calls in order to avoid an idle wait.
4. When a remote processor finishes its execution, it tells its father about its status: failure (no argument or D-node), or success (U-node) (see below).
5. Once all the children processors have finished, the local processor can evaluate the status of the argument $\mathcal{A}$ that has been built, and then tells this status to its father processor (see below).

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 criteria are used:

1. If a node $\mathcal{B}$ (argument) has no defeaters then $\mathcal{B}$ sends a message to its father indicating its success (it becomes 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 it 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 every 'potential' defeater of $\mathcal{A}$ has failed, (no argument could be constructed or the argument was defeated) then $\mathcal{A}$ becomes a U-node, and it sends a message to its father indicating this.

6 Implementation and Future Work

During the last 20 years, the Warren's Abstract Machine (WAM) [36, 14] has become a *de facto* standard for implementing logic programming. Since then, several models for exploiting parallelism in Logic Programming have been developed [13], and several parallel extension for the WAM have been proposed [12]. In [15], an efficient parallel execution model for logic programs was proposed, and the RAPWAM, an abstract machine for restricted and-parallel execution of logic programs was designed as an extension of the WAM (see also [16]).

In order to develop an efficient implementation of DLP [5], an abstract machine called *Justification Abstract Machine* (JAM) has been designed as an extension of the WAM [36, 14]. The JAM architecture has an instruction set, a memory structure and a set of registers expecially designed for building arguments, counterarguments and dialectical trees.

The JAM inherits the WAM sequential architecture and the dialectical analysis is done sequentially, building the dialectical tree in a depth-first approach. However, all the parallel extensions proposed for the WAM may be used for extending the JAM. Some work is being done in this area [8].

7 Related Work

We have defined a parallel argumentation system based in the defeasible argumentation formalism developed in [32, 30, 31, 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 [27, 28] have developed an argumentation system inspired by legal reasoning. As we have done here, 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 [25] generalized the system using Default Logic's language. R. Kowalski and F. Toni [17] have outlined a formal theory of argumentation, in which defeasibility is stated in terms of non-provability claims. Other related works are those by Verheij [33, 34], Vreeswijk [35], Bondarenko [1], Pollock [22], Loui [20], and Nute [21].

The interested reader is referred to the following surveys in defeasible argumentation: Prakken & Vreeswijk [24], and Chesñevar *et al.* [3]. A complete bibliography of parallelism in logic programming can be found in [12] and [11]. Gupta *et al.* [13] and [12] are good surveys of implementation of parallelism in logic programming.

8 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 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 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. Master's thesis, Dep. de Ciencias 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 and Guillermo R. Simari. Defeasible Argumentation for Extended Logic Programming. Submitted to the 5th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR'99), December 1999.

[8] Alejandro J. García and Guillermo R. Simari. Parallel conctruction of dialectical trees for defeasible logic programming. In *Proceedings of the VI Workshop of Aspectos Teóricos de la Inteligencia Artificial.* Universidad Nacional de San Juan, May 1999.

[9] 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.

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

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

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

[13] 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, 1994. http://www.cs.nmsu.edu/lldap/pub_para/survey.html.

[14] Aït-Kaci Hassan. *Warren's abstract machine, a tutorial reconstruction.* MIT Press, 1991.

[15] M. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[16] M. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

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

[18] Vladimir Lifschitz. Foundations of logic programs. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 69–128. CSLI Pub., 1996.

[19] John W. Lloyd. *Foundations of Logic Programmming.* Springer-Verlag, 1987.

[20] Ronald P. Loui. et al. 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.

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

[22] John Pollock. *Cognitive Carpentry: A Blueprint for How to Build a Person.* MIT Press, 1995.

[23] David L. Poole. On the Comparison of Theories: Preferring the Most Specific Explanation. In *Proc. 9th IJCAI*, pages 144–147. IJCAI, 1985.

[24] H. Prakken and G. Vreeswijk. Logical systems for defeasible argumentation (to appear). In D.Gabbay, editor, *Handbook of Philosophical Logic, 2nd ed.* Kluwer Academic Pub.

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

[26] 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.

[27] 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.

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

[29] J. A. Robinson. A machine-oriented logic based on the resolution principle. In *Journal ACM 12*, pages 23–41, January 1965.

[30] Guillermo R. Simari, Carlos I. Chesñevar, and Alejandro J. García. The role of dialectics in defeasible argumentation. In *XIV International Conference of the Chilenean Computer Science Society*, November 1994.

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

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

[33] Bart Verheij. *Rules, Reasons, Arguments: formal studies of argumentation and defeat.* PhD thesis, Maastricht University, Holland, December 1996.

[34] Bart Verheij. Argue! an implemented system for computer-mediated argumentation. In *Proc. of the 10th Netherlands/Belgium Conference on Artificial Intelligence*, pages 57–66. CWI, Amsterdam, 1998.

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

[36] David Warren. An abstract prolog instruction set. Technical report, SRI International, Menlo Park, CA, October 1983. Technical Note 309.