An Argumentative Framework for Reasoning with Inconsistent and Incomplete Information

Alejandro J. García1 and Guillermo R. Simari1 and Carlos I. Chesñevar1

Abstract. We present here a knowledge representation language, where defeasible and non-defeasible rules can be expressed. The language has two different negations: classical negation, which is represented by the symbol “∼” used for representing contradictory knowledge; and negation as failure, represented by the symbol “not” used for representing incomplete information. Defeasible reasoning is done using an argumentation formalism. Thus, systems for acting in a dynamic domain, that properly handle contradictory and/or incomplete information can be developed with this language.

An argument is used as a defeasible reason for supporting conclusions. A conclusion q will be considered justified only when the argument that supports it becomes a justification. Building a justification involves the construction of a non-defeated argument A for q. In order to establish that A is a non-defeated argument, the system looks for counterarguments that could be defeaters for A. Since defeaters are arguments, there may exist defeaters for the defeaters, and so on, thus requiring a complete dialectical analysis. The system also detects, avoids, circular argumentation. The language was implemented using an abstract machine defined and developed as an extension of the Warren Abstract Machine (WAM).

1 The language

Our language is defined in terms of two types of program clauses:

- extended program clauses \( \text{(epc)} \): \( l \leftarrow q_1, \ldots, q_n \).
- defeasible program clauses \( \text{(dpc)} \): \( l \leftarrow \neg q_1, \ldots, \neg q_n \).

There are two different negations: classical negation, which is represented by the symbol “∼” used for representing contradictory knowledge; and negation as failure, represented by the symbol “not” used for representing incomplete information. In both kinds of clauses l is a literal (i.e., a predicate “p” or a negated predicate “∼p”), and each \( q_i \) (\( n \geq 0 \)) is a literal, or a literal preceded by the symbol not. Thus, classical 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 “\( l \leftarrow \text{true} \)” (or simply “\( l \)”.) and is called a fact, whereas a DPC becomes “\( l \leftarrow \neg \text{true} \)” and is called a presumption.

We will use the usual PROLOG typographic conventions for an EPC, except that we will write “head \( \leftarrow \) body” rather than “head := body”; and “head \( \leftarrow \) body” for a defeasible clause. An EPC is used to represent sound (i.e., not defeasible) information such as: bird(X) \( \leftarrow \) penguin(X) which expresses that “all penguins are birds” whereas a DPC is used to represent defeasible knowledge such as: fly(X) \( \neg \) bird(X) which expresses that “presumably, a bird can fly” or “usually, a bird can fly.”

As mentioned earlier, program clauses can contain two types of negation. Classical negation (“∼”) can be used in clauses such as:

- ∼guilty(X) \( \leftarrow \) innocent(X).
- ∼free(X) \( \leftarrow \) ∼innocent(X).

to express that “an innocent is not guilty”, and that “usually, if someone is not innocent, then it is not free.” Negation as failure (not) may also be used in clauses such as:

innocent(X) \( \leftarrow \) not guilty(X).

∼cross-railway-tracks \( \leftarrow \) not ∼train-is-coming.

to express that “assume that someone is innocent whenever it has not been proven that s/he is guilty” and “generally, do not cross railroad tracks if it cannot be proven that no train is coming.” This kind of rules could not be written with only one type of negation.

Operationally, the difference between the two negations is as follows: a query “∼q” succeeds when there exists a proof for “q.” On the other hand a query “not q” will succeed when no proof can be found for “q”. In fact, proving a negated literal “∼p” is carried out just as if the “∼” symbol were syntactically part of the predicate name, thereby treating “∼p” as an atomic predicate name. In this system, a proof is a formal derivation called an argument, which justifies a query q. Arguments and justifications will be introduced in the following section.

With two types of negations, the Closed World Assumption (CWA) of a predicate p could be expressed within the language, writing the clause

\[ ∼p(X) \leftarrow \text{not } p(X). \]

i.e., “∼p(X) will be derived whenever the proof of p(X) fails.” Also, new forms of CWA can be written with the obvious
interpretation:

\[ p(X) \leftarrow \text{not } \sim p(X). \]
\[ p(X) \leftarrow \text{not } p(X). \]
\[ \sim p(X) \leftarrow \text{not } \sim p(X). \]

Nevertheless, if a defeasible clause is used, a defeasible notion of CWA could be represented:

\[ \sim p(X) \leftarrow \text{not } p(X). \]

which expresses that “the failure of the proof of \( p(X) \) is a good reason to assume \( \sim p(X) \)”. Thus, the following clauses can be written:

\[ \sim \text{dead}(X) \leftarrow \text{not } \text{dead}(X). \]
\[ \text{dangerous}(X) \leftarrow \text{not } \text{dangerous}(X). \]

**Definition 1.1** A defeasible logic program (DLP) is a finite set of EPCs and DPCs.

Let \( P \) be a DLP; then, we will distinguish the subset \( S \) of EPC in \( P \), and the subset \( D \) of DPC in \( P \).

**Example 1.1** : Here follows a DLP that will be referred to in other examples:

- \( \text{fly}(X) \leftarrow \text{bird}(X). \)
- \( \sim \text{fly}(X) \leftarrow \text{chicken}(X). \)
- \( \text{fly}(X) \leftarrow \text{chicken}(X), \text{scared}(X). \)
- \( \text{chicken}(\text{coco}) \leftarrow \text{true}. \)
- \( \text{penguin}(\text{petete}) \leftarrow \text{true}. \)
- \( \sim \text{dangerous}(X) \leftarrow \text{pet}(X). \)
- \( \text{dangerous}(X) \leftarrow \text{tiger}(X). \)
- \( \text{bird}(X) \leftarrow \text{chicken}(X). \)
- \( \text{scared}(\text{coco}). \)
- \( \text{bird}(X) \leftarrow \text{penguin}(X). \)
- \( \sim \text{fly}(X) \leftarrow \text{penguin}(X). \)
- \( \text{pet}(\text{kitty}). \)
- \( \text{tiger}(\text{kitty}). \)

Given a DLP \( P \), a defeasible derivation for a query \( \sim \neg q \) is a finite set of EPC and DPC obtained by backward chaining from \( q \) as in a Prolog program, using both strict and defeasible rules in the order specified by the DLP. The symbol \( \sim \) is considered as part of the predicate when generating a defeasible derivation.

**Example 1.2** : Using the DLP of example 1.1, there are defeasible derivations for each of the following queries:

- \( \neg \sim \text{fly}(\text{petete}) \)
- \( \neg \text{fly}(\text{petete}) \)
- \( \neg \text{fly}(\text{coco}) \)
- \( \neg \sim \text{fly}(\text{coco}) \)

It can be seen from Example 1.2, that the defeasible derivation notion does not forbid inferring two complementary literals from a given DLP \( P \). 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 the next section the notion of *argument* will be introduced to allow the defeasible argumentation formalism developed in [20, 19] which will allow us to define an inference scheme for the language. But first, we need to explicate when a set of clauses is deemed consistent.

**Definition 1.2 (Consistency)** A set of program clauses \( A \) is consistent if there is no defeasible derivation for any pair of complementary literals (with respect to classical negation \( \sim \)). Conversely, a set of program clauses \( A \) is inconsistent if there are defeasible derivations for a pair of complementary literals.

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

**Example 1.3** : Here follows a set of rules that supports inconsistent conclusions, because \( \text{fly}(\text{petete}) \) and \( \sim \text{fly}(\text{petete}) \) can be derived.

\[ \text{penguin}(\text{petete}) \leftarrow \text{true}. \]
\[ \text{bird}(X) \leftarrow \text{penguin}(X). \]
\[ \text{fly}(X) \leftarrow \text{bird}(X). \]
\[ \sim \text{fly}(X) \leftarrow \text{penguin}(X). \]

Observe also, that the DLP of example 1.1 is an inconsistent set of program clauses, although its associated set \( S \) is a consistent one.

## 2 Arguments, Rebuttals and Defeaters

In this framework, answers to queries must be supported by *arguments*. However, arguments may be defeated by other arguments. A query \( q \) will succeed if the supporting argument for it is not defeated; it then becomes a justification. Before defining formally the notion of justification, we define arguments, counterarguments and defeaters.

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

The above definition is adapted from [20] to fit in this framework. It states that an argument is a consistent defeasible derivation for a given query \( h \), using a minimal set of rules. Note that EPCs are not part of the argument.

**Example 2.1** : Consider the DLP of example 1.1.

The query \( \neg \sim \text{fly}(\text{coco}) \) has the argument:

\[ A_1 = \left\{ \begin{array}{l}
\sim \text{fly}(\text{coco}) \leftarrow \text{chicken}(\text{coco}). \\
\text{chicken}(\text{coco}) \leftarrow \text{true}.
\end{array} \right\} \]

However, the query \( \neg \sim \text{fly}(\text{coco}) \) has two arguments:

\[ A_2 = \left\{ \begin{array}{l}
\text{fly}(\text{coco}) \leftarrow \text{bird}(\text{coco}). \\
\text{chicken}(\text{coco}) \leftarrow \text{true}.
\end{array} \right\} \]

\[ A_3 = \left\{ \begin{array}{l}
\text{fly}(\text{coco}) \leftarrow \text{chicken}(\text{coco}), \text{scared}(\text{coco}). \\
\text{chicken}(\text{coco}) \leftarrow \text{true}.
\end{array} \right\} \]
The query \(-\lnot \text{fly}(\text{petete})\) has the argument:
\[
A_4 = \{ \text{penguin}(\text{petete}) \lnot \text{true.} \}\]

Finally, there is no argument for \(-\lnot \text{fly}(\text{petete})\) because its defeasible derivation is inconsistent with respect to \(S\) (see example 1.3). \(\square\)

Definition 2.2 (Sub-argument) An argument \(\langle B, q \rangle\) is a sub-argument of \(\langle A, h \rangle\) if \(B \subseteq A\).

Definition 2.3 (Counterargument or rebuttal) We say that \(\langle A_1, h_1 \rangle\) counterargues or rebuts \(\langle A_2, h_2 \rangle\) at literal \(h\), if and only if there exists a sub-argument \(\langle A, h \rangle\) of \(\langle A_2, h_2 \rangle\) such that the set \(S \cup \{h_1, h\}\) is inconsistent.

An argument is in fact a proof tree, involving rules from \(S\) and \(D\). Hence, arguments will be depicted as triangles abstracting a tree shape [19]. The upper vertex of the triangle will be labeled with the argument’s conclusion, and the argument will be associated with the triangle itself. Sub-arguments will be represented as smaller triangles inside a big one, which corresponds to the main argument at issue. Figure 1 shows the graphical representation of an argument \(\langle A_2, h_2 \rangle\) with one sub-argument \(\langle A, h \rangle\), and a counterargument \(\langle A_1, h_1 \rangle\).

Example 2.2: Continuing with Example 2.1, the argument \(A_1\) is a counterargument for both \(A_2\) and \(A_3\) (at \(\text{fly}(\text{coco})\)), and also \(A_2\) and \(A_3\) are counterarguments for \(A_1\) (both at \(\lnot \text{fly}(\text{coco})\)). Note that the argument \(A_4\) has no counterarguments. \(\square\)

As we mentioned before, the justification process for proving \(q\) involves the construction of a non-defeated argument \(A\) for \(q\). In order to verify whether an argument is non-defeated, its associated counterarguments \(B_1, B_2, \ldots, B_k\) are examined, each of them being a potential (defeasible) reason for rejecting \(A\). If any \(B_i\) is better than (or unrelated to) \(A\), then \(B_i\) is a candidate for defeating \(A\). If the argument \(A\) is better than \(B_i\), then \(B_i\) is not taken into account.

So we must clarify what makes an argument “better” than an other one. In this work, as a particular example, we will define 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 \(B_i\) is better (i.e., more specific, in our case) than \(A\), then \(B_i\) is called a proper defector for \(A\); if neither argument is better than the other, a blocking situation occurs, and we will say that \(B_i\) is a blocking defector for \(A\). This is a skeptical criterion that could be changed producing a different formal decision procedure.

Definition 2.4 (Defeating argument) An argument \(\langle A_1, h_1 \rangle\) defeats an argument \(\langle A_2, h_2 \rangle\) at literal \(h\), if and only if there exists a sub-argument \(\langle A, h \rangle\) of \(\langle A_2, h_2 \rangle\) such that \(\langle A_1, h_1 \rangle\) counterargues \(\langle A_2, h_2 \rangle\) at \(h\), and either:

1. \(\langle A_1, h_1 \rangle\) is strictly more specific than \(\langle A, h \rangle\) (then \(\langle A_1, h_1 \rangle\) is a proper defector of \(\langle A_2, h_2 \rangle\)); or
2. \(\langle A_1, h_1 \rangle\) is unrelated by specificity to \(\langle A, h \rangle\) (then \(\langle A_1, h_1 \rangle\) is a blocking defector of \(\langle A_2, h_2 \rangle\)).

The next definition characterizes specificity as defined in [14, 19] (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. In other words, an argument is deemed better that another if it is more precise or more concise. This notion is made formally precise in the next definition. We use the symbol \(\lnot \rightarrow\) to denote a defeasible derivation (i.e., \(\mathcal{P} \lnot \rightarrow h\) means that \(h\) has a defeasible derivation from \(\mathcal{P}\)), and the symbol \(\rightarrow\) to denote a derivation where only EPCs are used. Let \(S_G\) be the maximal subset of \(S\) that does not contain facts. Let \(F\) be the set of literals in \(\mathcal{P}\) that have a defeasible derivation.

Definition 2.5 (Specificity) An argument \(A_1\) for \(h_1\) is strictly more specific than an argument \(A_2\) for \(h_2\) (denoted \(\langle A_1, h_1 \rangle \rightarrow \langle A_2, h_2 \rangle\)) if and only if:

1. For all \(G \subseteq F\) : if \(S_G \cup G \cup A_1 \lnot \sim h_1\) and \(S_G \cup G \not\vdash h_1\), then \(S_G \cup G \cup A_2 \lnot \sim h_2\).
2. There exists \(G' \subseteq F\) such that \(S_G \cup G' \cup A_2 \lnot \sim h_2\) and \(S_G \cup G' \not\vdash h_2\) and \(S_G \cup G' \cup A_1 \not\vdash h_1\).

Example 2.3: Continuing with Example 2.1, argument \(A_1\) is strictly more specific than the argument \(A_2\) because \(A_1\) does not use the EPC \(\text{bird}(X) \Leftarrow \text{chicken}(X)\) and therefore is a more direct argument. However, argument \(A_3\) is strictly more specific than \(A_1\), because it contains more information. Then, \(A_1\) is a proper defector for \(A_2\), and \(A_3\) is a proper defector for \(A_1\). \(\square\)

3 Dialectical Trees and Justifications

Since defeaters are arguments, there may exist defeaters for defeaters, and so on. This means that it is necessary to pursue argument supports to ascertain their well-foundedness. This justification process is called a dialectical analysis. It can be specified in the context of Logic Programming in the following way (here \(\\vdash\) stands for PROLOG’s negation as failure):

\[
\text{justify}(Q) :- \text{find_argument}(Q, A), \not\vdash \text{defeated}(A) \quad \text{defeated}(A) :- \text{find_defeater}(A, D), \not\vdash \text{defeated}(D)
\]

The above description leads in a natural way to the use of trees to organize our dialectical analysis. In order to accept an argument \(A\) as a justification for \(q\), a tree structure can be generated. The root of the tree will correspond to the
argument $A$ 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 3.1 (Dialectical tree)** [19]
Let $A$ be an argument for $h$. A dialectical tree for $(A, h)$, denoted $T_{(A, h)}$, is recursively defined as follows:
1. A single node labeled with an argument $(A, h)$ with no defeaters (proper or blocking) is by itself a dialectical tree for $(A, h)$. This node is also the root of the tree.
2. Suppose that $(A, h)$ is an argument with defeaters (proper or blocking) $(A_1, h_1), (A_2, h_2), \ldots, (A_n, h_n)$. We construct the dialectical tree for $(A, h)$, $T_{(A, h)}$, by labeling the root node of with $(A, h)$ and by making this node the parent node of the roots of the dialectic trees for $(A_1, h_1), (A_2, h_2), \ldots, (A_n, h_n)$, i.e., $T_{(A_1, h_1)}, T_{(A_2, h_2)}, \ldots, T_{(A_n, h_n)}$.

Nodes in the dialectical tree can be recursively marked as defeated or undefeated nodes (D-nodes and U-nodes respectively). Let $A$ be an argument for a literal $h$, and $T_{(A, h)}$ be its associated dialectical tree.

**Definition 3.2 (Marking of a dialectical tree)**
(1) Leaves of $T_{(A, h)}$ are U-nodes.
(2) Let $(B, q)$ be an inner node of $T_{(A, h)}$. Then $(B, q)$ will be an U-node if every child of $(B, q)$ is a D-node. The node $(B, q)$ will be a D-node if it has at least an U-node as a child.

This definition suggests a bottom-up marking procedure, through which we are able to determine if the root of a dialectical tree turns out to be marked as defeated or undefeated.

![Dialectical tree for example 3.1](image)

**Example 3.1**: Consider the following DLP:

- $a \leftarrow b$.
- $\neg e \leftarrow l$.
- $c \leftarrow true$.
- $b \leftarrow c$.
- $\neg h \leftarrow k$.
- $f \leftarrow true$.
- $\neg b \leftarrow e$.
- $\neg b \leftarrow i$.
- $i \leftarrow true$.
- $e \leftarrow f$.
- $k \leftarrow l$.
- $l \leftarrow true$.
- $\neg h \leftarrow c$.
- $n \leftarrow true$.
- $\neg e \leftarrow h$.
- $\neg h \leftarrow l$.
- $h \leftarrow i$.
- $\neg k \leftarrow n, l$.

Here the argument $A = \{a \leftarrow b, b \leftarrow c\}$ for $a$ can be built. Argument $A$ has three defeaters attacking the literal $b$: $B_1 = \{\neg b \leftarrow e, e \leftarrow f\}$ and $B_2 = \{\neg b \leftarrow i\}$. $B_2$ is a proper defeater for $A$, the other two are blocking defeaters. Argument $B_1$ has also two blocking defeaters attacking the literal $e$: $C_1 = \{\neg e \leftarrow h, h \leftarrow i\}$ and $C_2 = \{\neg e \leftarrow i\}$. Argument $C_1$ has three blocking defeaters: $E_1, E_2$ and $E_3$, and finally $E_1$ has one proper defeater $F_1$.

![Marked dialectical tree for Figure 2 with pruning](image)

**Definition 3.3 (Justification)** Let $A$ be an argument for a literal $h$, and let $T_{(A, h)}$ be its associated dialectical tree. The argument $A$ will be a justification for a literal $h$ if the root of $T_{(A, h)}$ is an U-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 $a \leftarrow \neg h$:

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

Finally, an example that shows all the concepts defined above is presented.

**Example 3.2**: Given the DLP of Example 1.1, the following arguments can be built:
Example 4.1: Consider the following DLP:

\[
p(X) \leftarrow \neg \text{not q}(X).
\]
\[
q(a).
\]
\[
q(X) \leftarrow \neg r(X).
\]
\[
r(b).
\]
\[
r(c).
\]
\[
\neg q(X) \leftarrow r(X), s(X).
\]
\[
s(c).
\]

Here the query \(\neg q(c)\) succeeds because it has a justification, however, there is no justification for \(\neg q(b)\) (actually there is no argument for this query). The query \(\text{not q}(d)\) succeeds because there is no justification for \(q(d)\). However, since there is a justification for \(q(b)\), then the query \(\text{not q}(b)\) fails. Note that the query \(\text{not q}(c)\) succeeds, because although there is an argument for \(q(c)\), there is a defeater for it, so there is no justification for \(q(c)\). Thus, the queries \(p(c)\) and \(p(d)\) succeed, whereas \(p(a)\) and \(p(b)\) fail. □

5 Avoiding circular argumentation

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. Nevertheless, we need to impose conditions in order to avoid cycles in this tree. In [19], it is shown that circular argumentation is a particular case of fallacious argumentation. There, a detailed analysis exposes different kinds of undesirable situations, and solutions are proposed accordingly. We next briefly present the problems and their solutions and refer to [19] for details.

In order to analyze fallacious argumentation, it is useful to see a dialectical tree as a set of argumentation lines. Following [19], this notion is formally defined as follows. Let \(\langle A_0, h_0 \rangle\) be an argument, and let \(T_{A_0, h_0}\) be its associated dialectical tree.

**Definition 5.1 (Argumentation line)** Every path \(\lambda\) from the root \(\langle A_0, h_0 \rangle\) to a leaf \(\langle A_n, h_n \rangle\) in \(T_{A_0, h_0}\), denoted \(\lambda = [(A_0, h_0), (A_1, h_1), (A_2, h_2), \ldots, (A_n, h_n)]\), is called an argumentation line for \(h_0\).

In each argumentation line \(\lambda = [(A_0, h_0), (A_1, h_1), \ldots, (A_n, h_n)]\), the argument \(A_0, h_0\) is supporting the main query \(h_0\), and every argument \(A_i, h_i\) defeats its predecessor \((A_{i-1}, h_{i-1})\). Therefore, for \(k \geq 0\), \((A_{2k}, h_{2k})\) is a supporting argument for \(h_0\) and \((A_{2k+1}, h_{2k+1})\) 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: \(\lambda_S\) of supporting arguments, and \(\lambda_I\) of interfering arguments. Thus, an argumentation line \(\lambda\) can be construed as an alternate sequence of supporting and interfering arguments as in any ordered debate. In a dialectical tree, there are as many argumentation lines as leaves in the tree, and each can end up in a supporting or an interfering argument.

We next present three types of fallacious argumentation, and establish necessary and sufficient conditions to avert them.

An Argumentative Framework ... 17

\[\text{\textbullet} A_1 \rightarrow \neg b \rightarrow d \rightarrow B \]

\[\text{\textbullet} \neg b \rightarrow \sim b \rightarrow A \rightarrow \sim b \rightarrow d \]

**Figure 4.** Reciprocal defeaters

The first problematic situation is shown in Figure 4. This happens when a pair of arguments defeat each other. In this
case, \( (A_1, d) \) defeats \( (A_2, b) \), attacking the subargument \( (B, \sim d) \), but \( (A_2, b) \) also defeats \( (A_1, d) \) attacking the subargument \( (A, \sim b) \). Clearly, this situation is nonsensical as it leads to an infinite argumentation line. The first condition expressed in Definition 5.3 prevents this situation.

\[
\begin{array}{c}
\sim r \\
A \\
\sim p \\
X \\
\sim q \\
Y \\
\sim r \\
A
\end{array}
\]

Figure 5. Contradictory argumentation line

Figure 5 shows a case where the same argument \( (A) \) becomes both a supporting and an interfering argument of itself. This too is a nonsensical situation as it arises because the supporting argument \( C \) has a subargument \( Z \) for the literal \( r \), which is contradictory with the purpose of arguing in favor of \( \sim r \) (argument \( A \)). An argument like \( C \) ought to be avoided in a sound argumentation line. Clearly, there should be agreement among supporting arguments (respectively interfering) in any argumentation line. This underlies the second condition in Definition 5.3. This is expressed formally based on the notion of argument concordance as proposed in [19] and recalled next. Supporting (respectively interfering) arguments should be mutually concordant so as to make the dialectical process of argumentation coherent.

**Definition 5.2 (Concordance)**

Two arguments \( (A_1, h_1) \) and \( (A_2, h_2) \) are concordant iff the set \( S \cup A_1 \cup A_2 \) is consistent. More generally, a set of arguments \( \{(A_i, h_i)\}_{i=1}^{n} \) is concordant iff \( S \cup \bigcup_{i=1}^{n} A_i \) is consistent.

\[
\begin{array}{c}
\sim r \\
W \\
\sim p \\
X \\
\sim q \\
Y \\
\sim s \\
Z \\
\sim r \\
W \\
\sim p \\
X
\end{array}
\]

Figure 6. Circular argumentation

Finally, Figure 6 shows an example of circular argumentation, where the same argument \( A \) is introduced again in the line as a supporting argument. This is avoided by the third condition in Definition 5.3, which disallows the more general problematic situation where a subargument of an earlier argument is reintroduced further down the argumentation line.

These three situations are averted by requiring that all argumentation be addressed be acceptable as defined below:

**Definition 5.3 (Acceptable argumentation line) [19]**

Let \( \lambda = [(A_0, h_0), (A_1, h_1), \ldots, (A_i, h_i), \ldots, (A_n, h_n)] \) be an argumentation line, \( \lambda \) is an acceptable argumentation line iff:

1. For every defeater \( (A_i, h_i) \) and every proper subargument \( (B, q) \) of \( (A_i, h_i) \), \( (B, q) \) and \( (A_{i-1}, h_{i-1}) \) are concordant; that is, \( S \cup \{q, h_{i-1}\} \) must be consistent.
2. The sets \( \lambda \) of supporting arguments, and \( \lambda \) of interfering arguments of \( \lambda \) must each be concordant sets of arguments.
3. No argument \( (A_k, h_k) \) in \( \lambda \) is a subargument of an earlier argument \( (A_i, h_i) \) of \( \lambda \) \((i < k)\).

Hence, with these conditions averted undesirable situations, an acceptable dialectical tree is a dialectical tree where every argumentation line is acceptable. Then the notion of justification can be properly defined as follows [19].

**Definition 5.4 (Justification)**

The argument \( (A, h) \) is a justification for \( h \) iff its associated dialectical tree is acceptable and the root node of \( T_{(A, h)} \) is a U-node.

6 Implementation

In order to develop an efficient defeasible argumentation system, an abstract machine called \( \text{JAM} \) (Justification Abstract Machine) [7] has been designed as an extension of the Warren’s abstract machine \( \text{WAM} \) [1]. The \( \text{JAM} \) architecture has an instruction set, a memory structure and a set of registers for building arguments, counterarguments, and in this form obtaining justifications for queries. A compiler for defeasible logic programs was developed. It takes a defeasible logic program as its input and produces a sequence of \( \text{JAM} \) instructions as its output. The \( \text{JAM} \) was built as a virtual machine, and an interpreter for defeasible logic programs was developed over this machine.

7 Related Work

Other formalisms for defeasible argumentation have been separately developed. In [6] P. Dung has proposed a very abstract and general argument-based framework, where he completely abstracts from the notions of argument and defeat. In contrast we have defined an object language for representing knowledge and a concrete notion of argument and defeat, Dung’s approach assumes the existence of a set of arguments ordered by a binary relation of defeat. However, he defines various notions of ‘argument extensions’, which are intended to capture various types of defeasible consequence.

Inspired by legal reasoning, H. Prakken and G. Sartor [17, 18] have developed an argumentation system that like us, uses the language of extended logic programming. They introduce a dialectical proof theory for an argumentation framework fitting the abstract format developed by Dung, Kowalski et al. [6, 4]. However, since they are inspired by legal reasoning, the protocol for dispute is rather different from our dialectical approach. A proof of a formula takes the form of a dialogue tree, where each branch of the tree is a dialogue between a proponent and an opponent. Proponent and opponent have different rules for introducing arguments, leading to an asymmetric dialogue. Later, Prakken [16] generalized the system to default logic’s language.

R. Kowalski and F. Toni [9] have outlined a formal theory of argumentation, in which defeasibility is stated in terms of
non-provability claims. They argue that defeasible reasoning with rules of the form $P \text{ if } Q$ can be understood as "exact" reasoning with rules of the form $P \text{ if } Q \text{ and } S$ cannot be shown, where $S$ stands for one or more defeasible "non-provability claims".

Other related works are those by Vreeswijk [21], Bondarenko [3], Pollock [13], Loui [10], and Nute [11, 12]. The interested reader is referred to the following surveys in defeasible argumentation: Prakken & Vreeswijk [15], and Chesñevar et al. [5].

8 Conclusions

We have presented a knowledge representation language, that uses an argumentation formalism for defeasible reasoning. Defeasible rules allow to represent tentative knowledge, but also strong rules can be used. Since classical negation and negation as failure are both available in the language, contradictory and incomplete information can be represented. Several forms of CWA can be represented directly within the language.

Conclusions are supported by arguments, but when contradictory information is reached during a derivation, the defeasible argumentation formalism provides a way for deciding between competing arguments. If new information arises, then new arguments could be constructed and previous conclusions could be withdrawn. Thus, a correct behavior for a dynamic domain is obtained.

In order to develop efficient defeasible systems, the language was implemented using an abstract machine defined and implemented as an extension of the Warren Abstract Machine (WAM).

Acknowledgments

Alejandro J. García wishes to thank especially Hassan Ait-Kaci for many helpful discussions and suggestions. The authors are also grateful to Henry Prakken for helpful comments, and the anonymous referees for their suggestions. This work was partially supported by CONICET and Secretaría de Ciencia y Técnica UNS.

REFERENCES