International Journal of Information Technology & Decision Making
© World Scientific Publishing Company

DEFEASIBLE REASONING IN WEB-BASED FORMS THROUGH ARGUMENTATION

SERGIO ALEJANDRO GÓMEZ[*]

Department of Computer Science & Engineering, Universidad Nacional del Sur, Av. Alem 1253, (8000) Bahía Blanca, Argentina - Email: sag@cs.uns.edu.ar

CARLOS IVÁN CHESÑEVAR

Artificial Intelligence Research Group, Department of Computer Science, Universitat de Lleida, Campus Cappont, C/Jaume II 69, E-25001 Lleida, Spain, Email: cic@eps.udl.es

GUILLERMO RICARDO SIMARI

Department of Computer Science & Engineering, Universidad Nacional del Sur, Av. Alem 1253, (8000) Bahía Blanca, Argentina - Email: grs@cs.uns.edu.ar

Received (Day Month Year)
Revised (Day Month Year)

The notion of forms as a way of organizing and presenting data has been used since the beginning of the World Wide Web. Web-based forms have evolved together with the development of new markup languages, in which it is possible to provide validation scripts as part of the form code to test whether the intended meaning of the form is correct. However, for the form designer, part of this intended meaning frequently involves other features which are not constraints by themselves, but rather *attributes* emerging from the form, which provide plausible conclusions in the context of incomplete and potentially inconsistent information. As the value of such attributes may change in presence of new knowledge, we call them *defeasible attributes*. In this paper we propose extending traditional web-based forms to incorporate defeasible attributes as part of the knowledge that can be encoded by the form designer. The proposed extension allows the specification of scripts for reasoning about form fields using a defeasible knowledge base, expressed in terms of a Defeasible Logic Program.

Keywords: Defeasible reasoning; argumentation; web forms; markup languages.

1. Introduction

The notion of form as a way of organizing and presenting data is a well-known structural abstraction for data collection, storage, and information retrieval. Forms are important means to design and develop user-oriented information systems, and have long been used since the very beginning of the World Wide Web. Web-based

[*]Depto. de Cs. e Ing. de la Computación, Universidad Nacional del Sur, Av. Alem 1253, (8000) Bahía Blanca, Argentina

forms have evolved together with the introduction of new markup languages (*e.g.*, XML), in which it is possible to provide validation scripts as part of the form code in order to test whether the intensional meaning of the form is correct.[16]

Fulfilling the goals of the Semantic Web program [17] requires having tools capable of dealing with the potential inconsistencies and incompleteness of web data sources. A particularly important application domain is e-commerce technologies, which typically demand validation of user data (*e.g.*, credit card numbers) against a set of criteria for determining if a given user is eligible for certain prospective transaction. Performing validations on field values allows to determine whether the intended meaning of such fields is coherent according to some criteria established by the form designer. Such validations usually consist of a number of hard-coded decision criteria which are specified as a portion of imperative code in a script language. However, in many cases there are some emerging features which can be inferred as part of the "intended meaning" of the form without being field values themselves. Thus, in the case of a bank loan application, the notion of "reliable client" applied to some particular customer may be inferred as a plausible conclusion from knowing the annual income and banking records of that customer. Such features (or *attributes*) of the form are difficult to model in terms of pieces of imperative code, particularly in presence of incomplete and potentially contradictory information. The associated conclusions that could be inferred turn out to be *defeasible*, as they may change in the light of new information.

Our proposal is to extend traditional web-based forms to incorporate additional attributes as part of the declarative knowledge that can be encoded in a form. The value of such attributes will depend on the global consideration of incomplete or potentially inconsistent information. As these values may change in presence of new evidence or information, we call them *defeasible attributes.*

The proposed extension allows the specification of scripts for handling such defeasible attributes on the basis of a defeasible knowledge base associated with the form, expressed in terms of *Defeasible Logic Programming* (DeLP),[18] a particular formalization of defeasible argumentation [19] based on logic programming. We will show how this extension can be easily integrated with existing client-based approaches for handling forms, such as the use of JavaScript validation codes. The rest of this paper is structured as follows. In Section 2, we present the fundamentals of defeasible argumentation with an emphasis on Defeasible Logic Programming (DeLP), along with a worked example which will be used to illustrate the different concepts presented in the paper. Section 3 describes generic issues about web-based forms as well as the importance of introducing defeasible attributes. Section 4 introduces the notion of web-based form with defeasible attributes, or δ-*forms.* Section 5 presents X-DeLP, a script-like variant of DeLP oriented towards XHTML standards. We show how to encode δ-forms using X-DeLP, and illustrate how this scripting language can help to provide a first approach to formalizing argument-based ontologies. Section 6 analyzes implementation issues related to X-DeLP. Section 7 discusses related work and finally Section 8 concludes and outlines some future research directions.

2. Modeling Argumentation in DeLP

2.1. *Argumentation in AI: Background*

Artificial Intelligence (AI) is concerned with the challenge of modeling common-sense reasoning, which almost always occurs in the face of incomplete and potentially inconsistent information.[20,21,22] Logical models of common-sense reasoning demand the formalization of principles and criteria for characterizing valid patterns of inference. In this respect, classical logic has proven to be inadequate, since it behaves *monotonically* and cannot deal with inconsistencies at object level.[21]

When a rule supporting a conclusion may be defeated by new information, it is said that such reasoning is *defeasible.*[23,24,25,26,27] When defeasible reasons or rules are chained to reach a conclusion, we have *arguments* instead of proofs. Arguments may compete, rebutting each other, so that a *process* of argumentation is a natural result of the search for arguments. Adjudication of competing arguments must be performed, comparing arguments in order to determine what beliefs are ultimately accepted as *warranted* or *justified.* Preference among conflicting arguments is defined in terms of a *preference criterion* which establishes a relation " $\preceq$ " among possible arguments; thus, for two arguments $\mathcal{A}$ and $\mathcal{B}$ in conflict, it may be the case that $\mathcal{A}$ is strictly preferred over $\mathcal{B}$ ($\mathcal{A} \succ \mathcal{B}$), that $\mathcal{A}$ and $\mathcal{B}$ are equally preferable ($\mathcal{A} \succeq \mathcal{B}$ and $\mathcal{A} \preceq \mathcal{B}$) or that $\mathcal{A}$ and $\mathcal{B}$ are not comparable with each other. In the above setting, since we arrive at conclusions by building defeasible arguments, and since *logical argumentation* is usually referred to as *argumentation*, we sometimes call this kind of reasoning *defeasible argumentation.*

Let us consider a well-known problem of nonmonotonic reasoning in AI about the flying abilities of birds, recast in argumentative terms. Consider the following sentences:

(1) Birds usually fly.
(2) Penguins usually do not fly.
(3) Penguins are birds.

The first two sentences correspond to *defeasible rules* (rules which are subject to possible exceptions). The third sentence is a *strict rule*, where no exceptions are possible. Now, given the fact that *Tweety is a penguin* two different arguments can be constructed:

(1) Argument $\mathcal{A}$ (based on rules 1 & 3): Tweety is a penguin. Penguins are birds. Birds usually fly. So, Tweety flies.
(2) Argument $\mathcal{B}$ (based on rule 2): Tweety is a penguin. Penguins usually do not fly. So, Tweety does not fly.

In this particular situation, two arguments arise that cannot be accepted simultaneously (as they reach contradictory conclusions). Note that argument $\mathcal{B}$ seems rationally preferable over argument $\mathcal{A}$, as it is based on more *specific* information. As a matter of fact, specificity is commonly adopted as a syntax-based preference criterion among conflicting arguments, favoring those arguments which are *more informed* or *more direct*.[28,29] In this particular case, if we adopt specificity as a preference criterion, argument $\mathcal{B}$ is justified, whereas $\mathcal{A}$ is not (as it is defeated by $\mathcal{B}$). The above situation can easily become much more complex, as an argument may be defeated by a second argument, which in turn can be defeated by a third argument, *reinstating* the first one (for instance, we might learn that Tweety is actually a genetically altered penguin and might know as well that genetically altered penguins are able to fly).

The growing success of argumentation-based approaches has caused a rich crossbreeding with other disciplines, providing interesting results in different areas such as group decision making,[30] knowledge engineering,[31] legal reasoning,[32,33] and multiagent systems,[34,35,36] among others. During the last decade several defeasible argumentation frameworks have been developed, most of them on the basis of suitable extensions to logic programming (see [19,37,38]). *Defeasible logic programming* (DeLP) [18] is one of such formalisms, combining results from defeasible argumentation theory [27] and logic programming. DeLP is a suitable framework for building real-world applications which has proven to be particularly attractive in that context, such as clustering,[7] intelligent web search,[8,1] knowledge management,[4,39] multiagent systems[10] and natural language processing,[5] among others.

2.2. *Defeasible Logic Programming: Fundamentals*

2.2.1. *Knowledge Representation*

Next we will introduce the basic definitions to represent knowledge in Defeasible Logic Programming (DeLP). For an in-depth treatment, the interested reader is referred to.[18,40] In what follows, we assume that the reader has basic knowledge concerning fundamentals aspects of logic programming.[41,42]

Definition 2.1. (DeLP program $\mathcal{P}$) A defeasible logic program (delp) is a set $\mathcal{P} = (\Pi, \Delta)$ where Π and Δ stand for sets of *strict* and *defeasible* knowledge, respectively. The set Π of strict knowledge involves *strict rules* of the form $L \leftarrow Q_1, \ldots, Q_k$ and *facts* (strict rules with empty body), and it is assumed to be *non-contradictory*.[a] The set Δ of defeasible knowledge involves *defeasible rules* of the form $L \prec Q_1, \ldots, Q_k$, which stands for "*$Q_1, \ldots, Q_k$ provide tentative reasons to believe L*". Strict and defeasible rules in DeLP are defined using a finite number of literals. A literal is an atom (L), the strict negation of an atom ($\sim L$) or the default negation of an atom (*not* L).

[a]Contradiction stands for deriving two complementary literals w.r.t. strict negation (L and $\sim L$) or default negation (L and $\mathsf{not}\, L$).

The underlying logical language in DeLP is that of extended logic programming,[43,42] enriched with a special symbol " $\prec$ " to denote defeasible rules. Both default and classical negation are allowed (denoted *not* and $\sim$, respectively). Syntactically, the symbol " $\prec$ " is all what distinguishes a *defeasible* rule $L \prec Q_1, \ldots, Q_k$ from a *strict* (non-defeasible) rule $L \leftarrow Q_1, \ldots, Q_k$. DeLP rules are thus to be thought of as *inference rules* rather than implications in the object language. Analogously as in traditional logic programming, the *definition* of a predicate P in $\mathcal{P}$, denoted $P^{\mathcal{P}}$, is given by the set of all those (strict and defeasible) rules with head P and arity n in $\mathcal{P}$. If P is a predicate in $\mathcal{P}$, then $name(P)$ and $arity(P)$ will denote the predicate name and arity, resp. We will write $\mathsf{Pred}(\mathcal{P})$ to denote the set of all predicate names defined in a DeLP program $\mathcal{P}$.

Next we present an example in the banking domain which will help to illustrate how to encode knowledge corresponding to a real-world problem using DeLP.

Example 2.1. An international bank keeps track of its clients in order to determine which clients are potential candidates for getting loans. For every client the bank keeps name, country of origin, profession, average income per month, and family status of the client. The account manager of the bank has a number of criteria for granting loans, which allow to determine whether a person has a reasonable "profile," according to his personal records. Such criteria provide a way to arrive to tentative, *defeasible* conclusions (as they might be changed in the presence of new information).

Figure 1 shows a DeLP program $\mathcal{P}_{bank}$ for assessing the status of a loan application. Facts (1–3) are defined using a predicate $info$, and have the form *info*(*Name*, *Country*, *Profession*, *IncomePerMonth*), providing information about different customers —fact (1) indicates that John is a PhD student from a country named Krakosia and has an average income of \$400 a month; fact (2) says that Ajax is also a PhD student but from Greece and has an average income of \$350 a month, and from fact (3) it follows that Danae is from Greece with an income of $\$10,000$ a month and with no information regarding her profession. Facts (4–6) describe how much money has been requested by each customer to the bank, whereas facts (7–9) summarize the family records of the customers. Facts (10–11) establish that Krakosia and Greece are considered as trustworthy countries by the bank authorities, and finally fact (12) says that Peter is a millionaire.

Defeasible rules specify tentative criteria for granting loans. Thus, defeasible rules (13–14) express that a person P is typically a candidate for getting a loan granted if either P has the 'right profile' or if the requested loan is reasonable with respect to the income in the last months and P comes from a trustworthy country. Rule (15) says that a right profile is defined in terms of monthly income and country. Rule (16) establishes that usually all countries are trustworthy. Rule (17) says that a person P has a reasonable income if it is typically \$300 a month or higher. Rule (18) expresses that usually a person P who is not economically solvent does not have a reasonable income. Rules (19–20) say that usually PhD students are not solvent people unless they come from rich families. Finally, rule (21) says that families classified by the bank with a status "rich" are usually those with a bank record which supports such classification.

Program $\mathcal{P}_{bank}$ includes also a strict rule (22) which indicates that all millionaires are candidates for loans. Note that in this particular example we have

$$\begin{aligned}\mathsf{Pred}(\mathcal{P}_{bank}) \quad = \quad & \{info/4,\ family\_record/2,\ req\_loan/2,\\ & credible/1,\ candidate/1,\ profile\_ok/1,\ trustctry/2,\\ & goodincome/1,\ solvent/1,\ richfamily/1,\ millionaire/1\ \}\end{aligned}$$

Facts (user provided information):
(1) $info(john, krakosia, phdstudent, 400)$.
(2) $info(ajax, greece, phdstudent, 350)$.
(3) $info(danae, greece, none, 10000)$.
(4) $req\_loan(john, 2000)$.
(5) $req\_loan(ajax, 4500)$.
(6) $req\_loan(danae, 1000)$.

Facts (bank provided information):
(7) $family\_record(john, rich)$.
(8) $family\_record(ajax, unknown)$.
(9) $family\_record(danae, unknown)$.
(10) $credible(krakosia)$.
(11) $credible(greece)$.
(12) $millionaire(peter)$.

Defeasible rules:
(13) $candidate(P) \prec profile\_ok(P)$.
(14) $candidate(P) \prec info(P, \_, \_, Income), req\_loan(P, Amount),$
$\qquad Amount < Income * 10, trustctry(P, Ctry)$.
(15) $profile\_ok(P) \prec goodincome(P), trustctry(P, Ctry)$.
(16) $trustctry(P, Ctry) \prec info(P, Ctry, \_, \_), credible(Ctry)$.
(17) $goodincome(P) \prec info(P, \_, \_, Income), Income > 300$.
(18) $\sim goodincome(P) \prec \sim solvent(P)$.
(19) $\sim solvent(P) \prec info(P, \_, phdstudent, \_)$.
(20) $solvent(P) \prec info(\_, \_, phdstudent, \_), richfamily(P)$.
(21) $richfamily(P) \prec family\_record(P, rich)$.

Strict rules:
(22) $candidate(P) \leftarrow millionaire(P)$.

Fig. 1. Defeasible logic program $\mathcal{P}_{bank}$ with bank criteria for analyzing a loan application (Example 2.1)

2.2.2. *Argument, Counterargument, and Defeat*

Given a DeLP program $\mathcal{P} = (\Pi, \Delta)$, solving queries results in the construction of *arguments.* An argument $\mathcal{A}$ is a (possibly empty) set of ground defeasible rules that together with the set Π provides a logical proof for a given literal Q, satisfying the additional requirements of *non-contradiction* and *minimality.* Formally:

Definition 2.2. (Argument) Given a DeLP program $\mathcal{P}$, an *argument* $\mathcal{A}$ for a query Q, denoted $\langle \mathcal{A}, Q \rangle$, is a subset of ground instances of defeasible rules in $\mathcal{P}$, such that:

(1) there exists a *defeasible derivation* for Q from $\Pi \cup \mathcal{A}$;
(2) $\Pi \cup \mathcal{A}$ is non-contradictory (*i.e.*, $\Pi \cup \mathcal{A}$ does not entail two complementary literals L and $\sim L$ (or L and $\mathsf{not}\, L$)), and,
(3) $\mathcal{A}$ is minimal with respect to set inclusion (*i.e.*, there is no $\mathcal{A}' \subset \mathcal{A}$ such that there exists a defeasible derivation for Q from $\Pi \cup \mathcal{A}'$).

An argument $\langle \mathcal{A}_1, Q_1 \rangle$ is a *sub-argument* of another argument $\langle \mathcal{A}_2, Q_2 \rangle$ if $\mathcal{A}_1 \subseteq \mathcal{A}_2$. Given a DeLP program $\mathcal{P}$, $Args(\mathcal{P})$ denotes the set of all possible arguments that can be derived from $\mathcal{P}$.

The notion of defeasible derivation corresponds to the usual query-driven SLD derivation used in logic programming (see [41] for a textbook presentation), performed by backward chaining on both strict and defeasible rules; in this context a negated literal $\sim P$ is treated just as a new predicate name *no_P*. Minimality imposes a kind of "Occam's razor principle"[27] on argument construction. The non-contradiction requirement forbids the use of (ground instances of) defeasible rules in an argument $\mathcal{A}$ when $\Pi \cup \mathcal{A}$ entails two complementary literals. It should be noted that non-contradiction captures the two usual approaches to negation in logic programming (*viz.*, default negation and classical negation), both of which are present in DeLP and related to the notion of counterargument, as shown next.

Definition 2.3. (Counterargument) An argument $\langle \mathcal{A}_1, Q_1 \rangle$ is a *counterargument* for an argument $\langle \mathcal{A}_2, Q_2 \rangle$ iff

a) (subargument attack) there is a subargument $\langle \mathcal{A}, Q \rangle$ of $\langle \mathcal{A}_2, Q_2 \rangle$ (called *disagreement subargument*) such that the set $\Pi \cup \{Q_1, Q\}$ is contradictory, or
b) (default negation attack) a literal $\mathsf{not}\, Q_1$ is present in the body of some rule in $\mathcal{A}_2$.

The former notion of attack is borrowed from Simari-Loui's framework;[27] the latter one is related to Dung's argumentative approach to logic programming[44] as well as to other formalizations, such as in Prakken and Sartor's work[45] or Kowa and Toni's work.[46]

As in most argumentation frameworks, we will assume a *preference criterion* on conflicting arguments defined as a relation $\preceq$ which is a subset of the cartesian product $Args(\mathcal{P}) \times Args(\mathcal{P})$. This leads to the notion of *defeat* among arguments as a refinement of the notion of counterargument.[27] In particular, we will distinguish between two kinds of defeaters, *proper* and *blocking defeaters* as follows.

Definition 2.4. (Proper and blocking defeaters) An argument $\langle \mathcal{A}_1, Q_1 \rangle$ is a *proper defeater* for an argument $\langle \mathcal{A}_2, Q_2 \rangle$ if $\langle \mathcal{A}_1, Q_1 \rangle$ counterargues $\langle \mathcal{A}_2, Q_2 \rangle$ with a disagreement subargument $\langle \mathcal{A}, Q \rangle$ (subargument attack) and $\langle \mathcal{A}_1, Q_1 \rangle$ is strictly preferred over $\langle \mathcal{A}, Q \rangle$ w.r.t. $\preceq$.

An argument $\langle \mathcal{A}_1, Q_1 \rangle$ is a *blocking defeater* for an argument $\langle \mathcal{A}_2, Q_2 \rangle$ if $\langle \mathcal{A}_1, Q_1 \rangle$ counterargues $\langle \mathcal{A}_2, Q_2 \rangle$ and one of the following situations holds: (a) There is a disagreement subargument $\langle \mathcal{A}, Q \rangle$ for $\langle \mathcal{A}_2, Q_2 \rangle$, and $\langle \mathcal{A}_1, Q_1 \rangle$ and $\langle \mathcal{A}, Q \rangle$ are unrelated to each other w.r.t. $\preceq$; or (b) $\langle \mathcal{A}_1, Q_1 \rangle$ is a default negation attack on some literal $\mathsf{not}\, Q_1$ in $\langle \mathcal{A}_2, Q_2 \rangle$.

The term *defeater* will be used when referring indistinctly to proper or blocking defeaters.

Generalized specificity [27] is typically used as a syntax-based preference criterion among conflicting arguments, favoring those arguments which are *more informed* or *more direct*.[27,47] For the sake of example, let us consider three arguments $\langle \{a \prec b, c\}, a \rangle$, $\langle \{\sim a \prec b\}, \sim a \rangle$ and $\langle \{(a \prec b); (b \prec c)\}, a \rangle$ built on the basis of a program $\mathcal{P}{=}(\Pi, \Delta) = (\{b, c\}, \{b \prec c; a \prec b; a \prec b, c; \sim a \prec b\})$. When using generalized specificity as the comparison criterion between arguments, the argument $\langle \{a \prec b, c\}, a \rangle$ would be preferred over the argument $\langle \{\sim a \prec b\}, \sim a \rangle$ as the former is considered *more informed* (*i.e.*, it relies on more premises). However, argument $\langle \{\sim a \prec b\}, \sim a \rangle$ is preferred over $\langle \{(a \prec b); (b \prec c)\}, a \rangle$ as the former is regarded as *more direct* (*i.e.*, it is obtained from a shorter derivation). However, it must be remarked that besides specificity other alternative preference criteria could also be used; *e.g.*, considering numerical values corresponding to necessity measures attached to argument conclusions [6,3] or defining argument comparison using rule priorities. This last approach is used in D-PROLOG,[25] Defeasible Logic,[48] extensions of Defeasible Logic,[49,50] and logic programming without negation as failure.[51,52]

Example 2.2. Consider the DeLP program shown in Example 2.1. There exists an argument $\mathcal{A}$ supporting the defeasible conclusion that John is a candidate for a loan, *i.e.*, $\langle \mathcal{A}_1, candidate(john) \rangle$, where:[b]

$$\begin{aligned}\mathcal{A}_1 = \{&(candidate(john) \prec profile\_ok(john));\\ &(profile\_ok(john) \prec goodincome(john), trustctry(john, krakosia));\\ &(trustctry(john, krakosia) \prec info(john, krakosia, \_, \_), credible(krakosia));\\ &(goodincome(john) \prec info(john, \_, \_, 400), 400 > 300)\};\end{aligned}$$

[b]For the sake of clarity, we use parentheses to enclose defeasible rules in arguments, separated by semicolons, *i.e.* $\mathcal{A} = \{(rule_1) ; (rule_2) ; \ldots ; (rule_k)\}$.

Another argument $\langle \mathcal{A}_2, \sim goodincome(john) \rangle$ can be derived from $\mathcal{P}_{bank}$, supporting the conclusion that *John* does not have a reasonable income, with:

$$\begin{aligned}\mathcal{A}_2 = \{&(\sim goodincome(john) \prec \sim solvent(john));\\ &(\sim solvent(john) \prec info(john, \_, phdstudent, \_)\}.\end{aligned}$$

In this case, using generalized specificity [27] as the preference criterion among conflicting arguments, it turns out that the argument $\langle \mathcal{A}_2, \sim goodincome(john) \rangle$ is a *blocking* defeater for the argument $\langle \mathcal{A}_1, candidate(john) \rangle$.

2.2.3. *Computing Warrant through Dialectical Analysis*

Given an argument $\langle \mathcal{A}, Q \rangle$, there may exist different defeaters $\langle \mathcal{B}_1, Q_1 \rangle, \ldots, \langle \mathcal{B}_k, Q_k \rangle$, $k \geq 0$ for $\langle \mathcal{A}, Q \rangle$. Should the argument $\langle \mathcal{A}, Q \rangle$ be defeated, then it would be no longer supporting its conclusion Q. However, since defeaters are arguments, they may on their turn be defeated. That prompts for a complete recursive dialectical analysis to determine which arguments are ultimately defeated. To characterize this process we will introduce first some auxiliary notions.

An *argumentation line* starting in an argument $\langle \mathcal{A}_0, Q_0 \rangle$ (denoted $\lambda^{\langle \mathcal{A}_0, Q_0 \rangle}$) is a sequence $[\langle \mathcal{A}_0, Q_0 \rangle, \langle \mathcal{A}_1, Q_1 \rangle, \langle \mathcal{A}_2, Q_2 \rangle, \ldots, \langle \mathcal{A}_n, Q_n \rangle \ldots]$ that can be thought of as an exchange of arguments between two parties, a *proponent* (evenly-indexed arguments) and an *opponent* (oddly-indexed arguments). Each $\langle \mathcal{A}_i, Q_i \rangle$ is a defeater for the previous argument $\langle \mathcal{A}_{i-1}, Q_{i-1} \rangle$ in the sequence, $i > 0$. In order to avoid *fallacious* or ill-formed reasoning (*e.g.* infinite argumentation lines), dialectics imposes additional constraints on such an argument exchange to be considered rationally *acceptable.* Acceptable argumentation lines can be proven to be finite. An in-depth treatment of such acceptability constraints can be found in Garcia and Simari's work.[18] Given a DeLP program $\mathcal{P}$ and an initial argument $\langle \mathcal{A}_0, Q_0 \rangle$, the set of all acceptable argumentation lines starting in $\langle \mathcal{A}_0, Q_0 \rangle$ accounts for a whole dialectical analysis for $\langle \mathcal{A}_0, Q_0 \rangle$ (*i.e.*, all possible dialogues about $\langle \mathcal{A}_0, Q_0 \rangle$ between proponent and opponent), formalized as a *dialectical tree.*

Nodes in a dialectical tree $\mathcal{T}_{\langle \mathcal{A}_0, Q_0 \rangle}$ can be marked as *undefeated* and *defeated* nodes (U-nodes and D-nodes, resp.). A dialectical tree will be marked as an AND-OR tree: all leaves in $\mathcal{T}_{\langle \mathcal{A}_0, Q_0 \rangle}$ will be marked U-nodes (as they have no defeaters), and every inner node is to be marked as *D-node* iff it has at least one U-node as a child, and as *U-node* otherwise. An argument $\langle \mathcal{A}_0, Q_0 \rangle$ is ultimately accepted as valid (or *warranted*) w.r.t. a DeLP program $\mathcal{P}$ iff the root of its associated dialectical tree $\mathcal{T}_{\langle \mathcal{A}_0, Q_0 \rangle}$ is labeled as *U-node.*[c]

[c]An alternative definition for warrant can be provided on the basis of the *length* of argumentation lines: an argumentation line $[\langle \mathcal{A}_0, Q_0 \rangle, \langle \mathcal{A}_1, Q_1 \rangle \ldots, \langle \mathcal{A}_k, Q_k \rangle]$ is *won* iff k is odd, and lost otherwise. The root node of a dialectical tree is warranted iff every argumentation line in $\mathcal{T}_{\langle \mathcal{A}_0, Q_0 \rangle}$ is won.[53]

Given a DeLP program $\mathcal{P}$, solving a query Q w.r.t. $\mathcal{P}$ accounts for determining whether Q is supported by (at least) one warranted argument. Different doxastic attitudes can be distinguished as follows:

(1) *Yes*: accounts for believing Q iff there is at least one warranted argument supporting Q on the basis of $\mathcal{P}$.
(2) *No*: accounts for believing $\sim Q$ iff there is at least one warranted argument supporting $\sim Q$ on the basis of $\mathcal{P}$.
(3) *Undecided*: neither Q nor $\sim Q$ are warranted w.r.t. $\mathcal{P}$.
(4) *Unknown*: Q does not belong to the signature of $\mathcal{P}$.

Thus, according to DeLP semantics, given a program $\mathcal{P}$, solving a query Q — for any $Q \in \mathsf{Pred}(\mathcal{P})$— will result in a value belonging to the set $Ans = \{$*Yes*, *No*, *Undecided*$\}$.

Example 2.3. Consider the query $candidate(john)$ solved w.r.t. the program $\mathcal{P}_{bank}$ (see Fig. 1). As shown in Example 2.1, this query would start a search for arguments supporting $candidate(john)$, and argument $\langle \mathcal{A}_1, candidate(john)\rangle$ will be found. In order to determine whether this argument is warranted, its dialectical tree will be computed: as shown in Example 2.1, there is only one (blocking) defeater for $\langle \mathcal{A}_1, candidate(john)\rangle$, namely $\langle \mathcal{A}_2, \sim goodincome(john)\rangle$. This defeater, on its turn, has another (proper) defeater $\langle \mathcal{A}_3, solvent(john)\rangle$, with

$$\begin{aligned}\mathcal{A}_3 = \{&(solvent(john) \multimap info(john, \_, phdstudent, \_), richfamily(john));\\ &(richfamily(john) \multimap family\_record(john, rich))\}\end{aligned}$$

The resulting (marked) dialectical tree is depicted in Fig. 2(i). As the root node of $\mathcal{T}_{\langle \mathcal{A}_1, candidate(john)\rangle}$ is a U-node, the answer to $candidate(john)$ is Yes.

Consider now the query $candidate(ajax)$. As in John's case, there is an argument supporting this query, namely $\langle \mathcal{B}_1,\ candidate(ajax)\rangle$, with:

$$\begin{aligned}\mathcal{B}_1 = \{&(candidate(ajax) \multimap profile\_ok(ajax));\\ &(profile\_ok(ajax) \multimap goodincome(ajax), trustctry(ajax, greece));\\ &(trustctry(ajax, greece) \multimap info(ajax, greece, \_, \_), credible(greece));\\ &(goodincome(ajax) \multimap info(ajax, \_, \_, 350), 350 > 300)\};\end{aligned}$$

which is defeated by the argument $\langle \mathcal{B}_2, \sim goodincome(ajax)\rangle$, with

$$\begin{aligned}\mathcal{B}_2 = \{&(\sim goodincome(ajax) \multimap \ \sim solvent(ajax));\\ &(\sim solvent(ajax) \multimap info(ajax, \_, phdstudent, \_)\};\end{aligned}$$

Hence the associated dialectical tree for $candidate(ajax)$ has two nodes, with the root labeled as D-node (Fig. 2(ii)). The original argument for $candidate(ajax)$ is therefore not warranted.

Let us now consider the query $candidate(danae)$. In this case, there is an argument $\langle \mathcal{C}_1, candidate(danae)\rangle$ that has no defeaters (and hence it is warranted) for concluding that Danae has the right profile for the bank, with:

$$\mathcal{C}_1 = \{(candidate(danae) \multimap profile\_ok(danae));$$

$$(profile\_ok(danae) \prec goodincome(danae), trustctry(danae, greece));$$
$$(trustctry(greece) \prec info(danae, greece, \_, \_), credible(greece));$$
$$(goodincome(danae) \prec info(danae, \_, \_, 10000), 10000 > 300)\}$$

As the argument has no defeaters, the resulting dialectical tree will have a unique node, as depicted in Fig. 2(iii).

Note that there is also a second argument without defeaters supporting the query *candidate*(*danae*), namely $\langle \mathcal{C}_2, candidate(danae)\rangle$, with:

$$\mathcal{C}_2 = \{(candidate(danae) \prec info(danae, \_, \_, 10000), req\_loan(danae, 1000),$$
$$1000 < 10000 * 10, trustctry(danae, greece));$$
$$(trustctry(danae, greece) \prec info(danae, greece, \_, \_), credible(greece))\}.$$

For this particular argument, the same analysis as for argument $\mathcal{C}_1$ can be applied.

Finally, there exists an empty argument $\mathcal{D}_1$ for concluding *candidate*(*peter*) (*i.e.*, $\mathcal{D}_1 = \emptyset$). Note that this argument is empty as *candidate*(*peter*) follows from the strict knowledge in $\mathcal{P}_{bank}$ (*i.e.*, Peter is a candidate for a loan as he is a millionaire.[d]). As this argument is empty, made up of strict derivations, it has no defeaters, and consequently it is also warranted. The resulting dialectical tree is depicted in Fig. 2(iv).

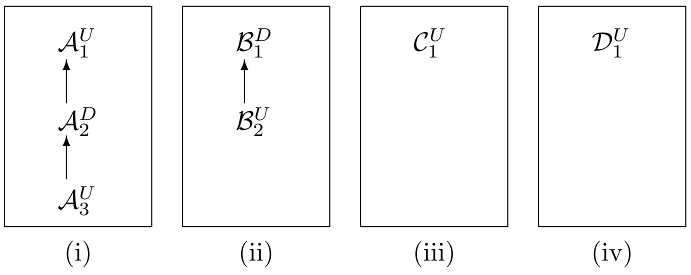


Fig. 2. Dialectical trees for queries: (i) *candidate*(*john*), (ii) *candidate*(*ajax*), (iii) *candidate*(*danae*), and (iv) *candidate*(*peter*) w.r.t. $\mathcal{P}_{bank}$

3. Web-based Forms: From HTML to XForms

The notion of form is a central structural abstraction for data collection, storage, and retrieval in information management systems. From the very beginning of the World Wide Web, HTML standards included the possibility of *form design* along with a number of ways for allowing interactive behavior by means of control techniques

[d]As this is a toy example, we assume that millonaires are granted loans independently of the amount requested (which is clearly not the case in a real-world situation).

(checkboxes, radio buttons, etc.). Forms provide a standard way of allowing the user to send information back to the server by means of different technologies to verify and validate data (*e.g.*, CGI scripting). A number of programming technologies —in particular along with the evolution of the Java programming language— were later developed, enabling the creation of interactive web applications which outperformed static web pages. Dynamic HTML (DHTML) favored the development of client-side, in-browser applications, by embedding pieces of programming code written in script languages (like JavaScript or VBScript). The growing popularity of e-commerce technologies as well as the envisioning of the Semantic Web motivated the specification of sophisticated standards for web-based forms, notably XForms.[54e]

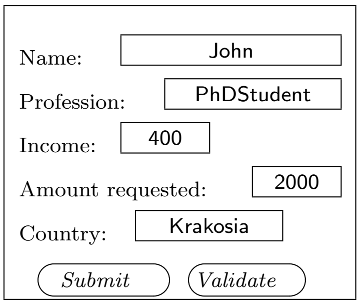
In spite of the evolution of web-based form technologies, most form designers perform validation of form fields by enforcing constraints (*e.g.*, numeric ranges) encoded as pieces of imperative code in a scripting language (*e.g.*, JavaScript). Thus, validation of data is done client-side, and the form data is finally processed by a program located in a remote server (usually accessing some sort of database). However, in many cases there are some emerging features which can be inferred as part of the "intended meaning" of the form without being field values themselves. Thus, in the case of a bank loan application such as the one discussed in the previous sections, a concept like *reliable client*, modelled on the basis of the field values for a particular customer, could prove useful for the form designer in order to codify decision making issues associated with form processing. To identify every relevant attributes needed to infer a concept like "reliable customer" using only imperative code may be a difficult task, as in complex situations such conclusions are defeasible (particularly in presence of incomplete and potentially inconsistent information).

To address the above problem, the concept of form can be suitably extended to formalize such complex scenarios on the basis of DeLP by means of *defeasible attributes*, as we will see in the next section. In order to do this, we will first provide a rather generic definition of the concept of form on the basis of the notions of *form schema* and *form instance*, which will prove useful for presenting our approach.

Definition 3.1. (Form Schema. Form Instance) A *form schema* is a pair $\mathcal{F}$=$\langle F, T\rangle$, where $F = [f_1,\, f_2,\, \ldots,\, f_n]$ is a list of *form fields* and $T = [T_1,\, \ldots,\, T_n]$ is a list of *types* (each of them consisting of a set of values). Given a form schema $\mathcal{F}$=$\langle F, T\rangle$ defined as above, a *form instance* based on $\mathcal{F}$ with value V (denoted $\mathcal{F}_V$) is a pair $\mathcal{F}_V = \langle F, V\rangle$, where $V = [v_1, \ldots, v_n]$ is a list of values such that every $v_i \in T_i$ is the associated value for $f_i \in F$.

Example 3.1. Let F = [*name*, $profession$, *income*, *amountreq*, *country*] and T = [*string*, *string*, *real*, *integer*, *string*], where *string*, *real*, and *integer* are type names with the usual meaning, then $\mathcal{F} = \langle F, T\rangle$ is a form schema. Let V = [*john*, *phdstudent*, 400, 2000, *krakosia*], then $\mathcal{F}_V = \langle F, V\rangle$ is a form instance based on $\mathcal{F}$.

[e]For an in-depth analysis, see `http://www.w3.org`.



Name:	John
Profession:	PhDStudent
Income:	400
Amount requested:	2000
Country:	Krakosia

Fig. 3. Form view for the loan application

Figure 3 shows the typical graphical appearance of a web-based form according to the form schema given in Ex. 3.1. Note that according to Def. 3.1, control actions associated with the form (*e.g.*, *submit*, *validate*, etc.) are not considered as elements in the form schema.

4. Integrating Forms with DeLP: Forms with Defeasible Attributes

In this section we will show how to extend traditional web-based forms to incorporate defeasible knowledge expressed in terms of a DeLP program, characterizing the notion of *forms with defeasible attributes* or δ-forms. Our goal is to provide a way to associate a DeLP program $\mathcal{P}$ with an arbitrary form schema (which will correspond to a number of different possible form instances). The program $\mathcal{P}$ is assumed to represent declarative knowledge associated with the problem domain of the form schema. Thus, as discussed in the previous example, a form schema corresponding to a bank application could have an associated DeLP program which represents tentative (and possibly conflicting) policies for granting loans.

Given a form schema $\mathcal{F}$=$\langle F, T\rangle$, and a particular form instance $\mathcal{F}_V$, we will capture the factual knowledge involved in $\mathcal{F}_V$ in terms of a set $facts(\mathcal{F}_V)$ of DeLP facts. Such facts will be obtained by introducing new predicate names associated with those field names in a form $\mathcal{F}$, and new constant names corresponding to the values present in V. Formally:

Definition 4.1. (Set of facts) Let $\mathcal{F}$= $\langle F, T\rangle$ be a form schema, with F=[f_1, …, f_n], and let $\mathcal{F}_V$ be a form instance. We define the set $facts(\mathcal{F}_V) =_{def} \{f_1(\mathcal{F}, v_1)$, $f_2(\mathcal{F}, v_2)$, …, $f_n(\mathcal{F}, v_n)\}$.

Example 4.1. Given the form instance $\mathcal{F}_V$ in Example 3.1, the corresponding set $facts(\mathcal{F}_V)$ is $\{name(\mathcal{F}, john)$, $profession(\mathcal{F},$ $phdstudent)$, $income(\mathcal{F}, 400)$, $amountreq(\mathcal{F}, 2000)$, $country(\mathcal{F},$ $krakosia)\}$.

Next we will show how field values can be integrated with an arbitrary DeLP program $\mathcal{P}$, characterizing thus the concept of δ-forms. Formally:

Definition 4.2. (Form schema with defeasible attributes. δ-form instance)

Let $\mathcal{F} = \langle F, T\rangle$ be a form schema, and $\mathcal{P} = (\Pi, \Delta)$ a DeLP program. A *form schema with defeasible attributes* (or *δ-form schema*) $\mathcal{D}$ is a pair $\langle \mathcal{F}, \mathcal{P}\rangle$. If V is a set of values for the form $\mathcal{F}$, a *δ-form instance* $\mathcal{D}_V$ is the pair $\langle \mathcal{F}_V, \mathcal{P}\rangle$. The set of *defeasible attributes* for $\mathcal{D}_V$ is defined as the set of predicates $\mathsf{Pred}(\Pi \cup facts(\mathcal{F}_V), \Delta)$.

Given a δ-form schema $\langle \mathcal{F}, \mathcal{P}\rangle$, the above definition aims at identifying features or attributes in the form $\mathcal{F}$ encoded by the form designer as distinguished predicates in the program $\mathcal{P}$. Such attributes are said to be *defeasible*, as their associated value will be determined by DeLP queries solved w.r.t. the DeLP program $(\Pi \cup facts(\mathcal{F}_V), \Delta)$. Hence, changing the field values in the form $\mathcal{F}$ or changing the underlying DeLP program $\mathcal{P}$ will result in changing the value for these attributes. As stated before, defeasible attributes will represent relevant features for the form designer, whose value depends on the DeLP program encoding relevant domain knowledge with the addition of particular facts which represent the field values for a given form instance.

Example 4.2. Let $\mathcal{F} = \langle F, T\rangle$ be the form schema given in Example 3.1, and consider the program $\mathcal{P}_{bank}{}' = \mathcal{P}_{bank} \backslash \{(1), \ldots, (6)\} \cup \{rule_1\} \cup \{rule_2\}$, where

$$\begin{array}{rcl} rule_1 & = & info(N, C, P, I) \leftarrow name(\mathcal{F}, N),\ country(\mathcal{F}, C), \\ & & \qquad\qquad profession(\mathcal{F}, P),\ income(\mathcal{F}, I) \\ rule_2 & = & req\_loan(N, A) \leftarrow name(\mathcal{F}, N),\ amountreq(\mathcal{F}, A); \end{array}$$

i.e., the program given in Fig. 1 excluding user-provided information (rules (1) to (6)), along with two additional strict rules $rule_1$ and $rule_2$ linking the form $\mathcal{F}$ with the DeLP program $\mathcal{P}_{bank}$.

Let $\mathcal{D} = \langle \mathcal{F}, \mathcal{P}_{bank}{}'\rangle$ be a δ-form schema. According to the DeLP program $\mathcal{P}_{bank}{}'$, one defeasible attribute in $\mathcal{D}$ is $candidate/1 \in \mathsf{Pred}(\mathcal{P}_{bank}{}')$. Suppose now that four different users John, Ajax, Danae and Peter fill in this δ-form according to the data given in Example 2.1 and following the form schema given in Example 3.1. Form fields in the form $\mathcal{F}$ would be filled in with different values, and for every particular user u a δ-form instance $\mathcal{D}_u$ would be obtained. Thus, when analyzing for example the query "$candidate(john)$", the δ-form instance $\mathcal{D}_{john}$ would involve providing all his particular user details, which will be present as a set of DeLP facts (Def. 4.1).

Along with the two additional strict rules given above, reasoning from $\mathcal{P}_{bank}{}' \cup facts(\mathcal{D}_{john})$ would result in the dialectical analysis shown in Example 2.3, determining that $candidate(john)$ is warranted. The same applies for the other three users with respect to $candidate(ajax)$, $candidate(danae)$, and $candidate(peter)$, respectively.

In the previous example, we have seen how form fields (Def. 3.1) can be referred as predicate names in DeLP. Similarly, the notion of δ-form (Def. 4.2) distinguish some predicate names as defeasible attributes (as in the case of $candidate/1$ mentioned before). In order to use such attributes in the context of web-based applications, we will encode DeLP programs in a suitable markup language called X-DeLP.

5. X-DeLP: encoding DeLP in Web-based forms

In this Section we will present the main features of X-DeLP, a scripting language for encoding DeLP programs in XML. XDeLP supports the representation of defeasible knowledge bases by augmenting XHTML with tags that allow to represent defeasible logic programs. XDeLP can be embedded directly in XHTML documents or used in XML documents. This design choice provides several advantages (as remarked by Heflin *et. al.*[55] in the context of SHOE): (1) web authors are more confortable with XML syntax as there are many commercial applications to edit it; (2) its knowledge contents can be used in other XML aware applications, and (3) the XSLT style sheet standard [56] can be used to render it in a way that is adequate for human understanding. In order to make this article self-contained, we will briefly summarize next the main elements of the XML technology. A more in-depth discussion can be found elsewhere.[57]

Markup languages based on XML consists of a set of *element types* which serve to define *types* of documents and are referred to as *Document Type Definitions* or DTDs for short. XML allows authors to create their own markup tags (*e.g.*, *<author>*). *Well-formed* XML documents are documents that meet the constraints in the XML specification, whereas *valid* XML documents are those which are well-formed and additionally meet all of the constraints specified in the DTD. XML provides start tags (*e.g.*, *<foo>*), end tags (*e.g.*, *</foo>*), and empty tags (*e.g.*, *<foo bar="baz"/>*. Empty tags can have attributes (*e.g.*, *bar*) that take a value (*e.g.*, *baz*). Elements that contain some mixture of markup/character data must have matching start- and end-tags (*e.g.*, *<country gov="democracy">Krakosia</country>*).

Element type declarations allow an XML application to constrain the elements that can occur in the document and to specify the order in which can occur. The expression *<!ELEMENT foo EMPTY>* declares *foo* elements to be empty, whereas the expression *<!ELEMENT foo (apple|orange|banana)>* declares that the element *foo* can contain exactly one element in the set {*apple*, *orange*, *banana*}. Additionally, XML allows to declare element types that can contain other elements. The expression *<!ELEMENT person (name, address, phone?)>* declares that a person has a name, an address, and optionally a phone number. Zero or more occurrences can be specified as in *<!ELEMENT books (book)*>*, one or more occurrences as in *<!ELEMENT books (book)+>*. Character data is denoted by the keyword *#PCDATA* as in *<!ELEMENT quotation #PCDATA>*. Attribute list declarations serve to specify the name, type, and optionally the default value of the attributes associated with an element. Thus, the expression *<!ATTLIST foo bar CDATA #REQUIRED>* means that the element *foo* has the attribute *bar* containing character data which will be ignored by the XML parser. The modifier *#REQUIRED* means that giving a value to the attribute is mandatory. Other modifiers such as *#IMPLIED* are possible meaning that the value for the attribute will be computed by an external application.

5.1. *XDeLP: a syntactic characterization in XML*

Next we will present a syntactic characterization of X-DeLP using XML in order to illustrate how DeLP features can be encoded into web-based forms. For the sake of example, our presentation will be based on the $\mathcal{P}_{bank}$ program presented in Section 2. First, an XML tag <*delp*> is required to define DeLP programs as distinguished entities in an XML document. In our case study the program is named $\mathcal{P}_{bank}$, so that the resulting XML representation would be as shown below.

```
<delp id="PBank" version="1.0">
```

Clearly, it may be desirable to annotate programs with comments, *e.g.*:

```
<comment>Program for defining criteria for granting
a loan application.</comment>
```

Within an XML setting, a DeLP program will be composed by definitions of rule-schemas and declarations of rule and fact instances. The corresponding DTD representation follows:

```
<!ELEMENT delp (comment?, definitions, declarations)>
<!ELEMENT comment PCDATA>
<!ATTLIST delp id CDATA #REQUIRED version CDATA #REQUIRED>
```

The definition of a program involves the specification of the atoms and rule schemas allowed. For example, to define an atom *info*(*Name*, *Country*, *Profession*, *Income*) the DTD representation would be as follows:

```
<def-atom name="info" arity="4">
          <def-arg pos="1" param="Name" type="string" />
          <def-arg pos="2" param="Country" type="string" />
          <def-arg pos="3" param="Profession" type="string" />
          <def-arg pos="4" param="Income" type="real" />
</def-atom>
```

This DTD is indicating that the atom name is *info* with arity 4. Besides, the name and type of each parameter for *info* is specified. The definition of atoms with arity 0 is also possible, accounting for propositional DeLP programs. The corresponding part of the DTD for atoms follows below:

```
<!ELEMENT definitions (def-language, def-rules)> <!ELEMENT def-language (def-
atom)*>
<!ELEMENT def-atom (def-arg)*>
<!ATTLIST def-atom name CDATA #REQUIRED arity CDATA #REQUIRED>
<!ELEMENT def-arg EMPTY>
<!ATTLIST def-arg pos CDATA #REQUIRED param CDATA #REQUIRED
          type CDATA #REQUIRED>
```

DeLP rules will have a unique ID and an associated type (*defeasible* or *strict*). In particular, arguments (parameters) in literals can be anonymous by using the *dash* qualifier.[f] The attribute *negated* could be associated with atoms in the program. A given atom A with predicate name L which appears as head of a rule in a program can take *yes* or *no* as possible values. In the first case, A stands for $\sim L$ (*i.e.*, L is preceded by strict negation), whereas in the second case it just stands for L. The value *no* is adopted by default. A similar analysis applies for those atoms present in the body of a rule. However, in this case the attribute *negated* can have three possible values (*no*, *yes*, and *default*), which stand for L, $\sim L$, and *not* L (default negation), respectively. The DTD definitions follow:

```
<!ELEMENT def-rules (def-rule)*>
<!ATTLIST def-rule id CDATA #REQUIRED
          type (defeasible | strict) #REQUIRED>
<!ELEMENT def-rule (comment?,def-head, def-body)>
<!ELEMENT def-head (arg)*>
<!ATTLIST def-head name CDATA #REQUIRED negated (no | yes) “no”>
<!ELEMENT arg EMPTY>
<!ATTLIST arg pos CDATA #REQUIRED value (CDATA|dash) #REQUIRED>
<!ELEMENT def-body (def-body-atom)+>
<!ELEMENT def-body-atom (arg)*>
<!ATTLIST def-body-atom name CDATA #REQUIRED
          negated (no | yes | default) “no”>
```

Example 5.1. Consider the program $\mathcal{P}_{bank}$ in Figure 1. The strict rule (22) $candidate(P) \leftarrow millionaire(P)$ can be expressed as follows:

```
<def-rule id=“22” type=“strict”>
          <def-head name=“candidate” negated=“no”>
                    <arg pos=“1” param=“P” type=“string” />
          </def-head>
          <def-body>
                    <def-body-atom name=“millionaire” negated=“no”>
                              <arg pos=“1” value=“P” />
                    </def-body-atom>
          </def-body>
</def-rule>
```

Analogously, rule (16) will be expressed as:

```
<def-rule id=“16” type=“defeasible”>
          <def-head name=“trustctry” negated=“no”>
```

[f]Note that this corresponds to the “underscore” anonymous variable in the PROLOG programming language.

```
                <arg pos="1" param="P" type="string" />
                <arg pos="2" param="Ctry" type="string" />
        </def-head>
        <def-body>
                <def-body-atom name="info" negated="no">
                        <arg pos="1" value="P" />
                        <arg pos="2" value="Ctry" />
                        <arg pos="3" value="dash" />
                        <arg pos="4" value="dash" />
                </def-body-atom>
                <def-body-atom name="credible" negated="no">
                        <arg pos="1" value="Ctry" />
                </def-body-atom>
        </def-body>
</def-rule>
```

Given a DeLP program $\mathcal{P}$, the previous DTD representation accounts for an XML-based formulation of $\mathcal{P}$. However, modeling the concept of argument requires the *ground instantiation* of rules. The DTD definitions follow below.

```
<!ELEMENT declarations (rule-instances, facts)>
<!ELEMENT rule-instances (rule-instance)*>
<!ATTLIST rule-instance id CDATA #REQUIRED>
<!ELEMENT rule-instance (subst)*>
<!ELEMENT subst EMPTY>
<!ATTLIST subst param CDATA #REQUIRED value CDATA #REQUIRED>
<!ELEMENT facts (fact)*>
<!ELEMENT fact (arg)*>
<!ATTLIST fact negated (yes | no) "no"
        type (fact | assumption) "fact" name CDATA #REQUIRED>
```

Example 5.2. Facts naturally correspond to ground information. Thus, using our DTD representation, the fact that John is PhDStudent from Krakosia and has an income of \$400 ($info(john, krakosia, phdstudent, 400)$) will be encoded as:

```
<fact negated="no" type="fact" name="info">
        <arg pos="1" value="john"/>
        <arg pos="2" value="krakosia"/>
        <arg pos="3" value="phdstudent"/>
        <arg pos="4" value="400"/>
</fact>
```

Analogously, in order to represent the rule instance:

$$trustctry(john, krakosia) \prec info(john, krakosia, \_, \_), credible(krakosia),$$

we will write:

```
<rule-instance id="16">
        <subst param="P" value="john" />
        <subst param="Ctry" value="krakosia" />
</rule-instance>
```

Finally, suitable annotation tags are provided to represent arguments, argumentation lines, and dialectical trees in XDeLP. An argument $\langle \mathcal{A}, H \rangle$ will be represented by a fact instance H and a set $\mathcal{A}$ of rule instances, namely:

```
<argument>
        <rule-instances>... XML representation for A... </rule-instances>
        <fact>... XML representation for H... </fact>
</argument>
```

A tag named <*complete-argument*> is also provided for representing the set of all rules (strict and defeasible) used for the derivation of an argument.[g] A tag named <*argument-derivation*> is also provided in order to represent the tree supporting the derivation of an argument. Finally, tags for representing argument lines and argument trees are provided. Each node of an argument tree has an associated epistemic status that can take either one of two values—defeated or undefeated. Next we present the corresponding DTD definitions:

```
<!ELEMENT argument (rule-instances, fact)>
<!ELEMENT complete-argument
        (rule-instances, facts, fact)>
   <!ELEMENT argument-derivation (fact, argument-derivation*)>
   <!ELEMENT argument-line (argument)*>
   <!ELEMENT argument-tree (argument, argument-tree*)>
<!ATTLIST argument-tree epistemic-status (defeated | undefeated) #IMPLIED>
```

5.2. *Redefining XDeLP Programs: Towards Argument-based Ontologies*

A common way to provide semantics to documents on the web is through the use of *ontologies*.[58] In computer science, ontologies establish a joint terminology between members of a community of interest (*e.g.*, among human or automated agents), which include machine-interpretable definitions of basic concepts in the domain and relations among them. Ontologies are usually expressed in logic-based languages, as they provide a declarative way of expressing knowledge along with inference capabilities to reason about the concepts being represented. In this context, XDeLP offers an interesting alternative for modeling ontologies (using predicate definitions) and performing inferences on the basis of the underlying argumentative engine. In an ontological setting, a DeLP program $\mathcal{P}$ (Def. 2.1) can be thought of as a set of

[g]Note that the definition of argument only requires to consider a finite set of ground instances of defeasible rules. However, for implementation purposes it may be useful to keep track of all the rules (strict and defeasible) used in constructing an argument.[2]

predicate definitions, *i.e.* $\mathcal{P} =_{def} \{P_1^{\mathcal{P}}, \ldots, P_k^{\mathcal{P}}\}$. Thus, in our example concerning bank loans, the program $\mathcal{P}_{bank}$ (Fig. 1) provides the definition of a number of predicates (*candidate*, *trustctry*, etc.). Recent research [59] has been oriented towards providing a first approach to formalizing *argument-based ontologies* in which XDeLP can be used for ontology interchange in the context of the Semantic Web.

Formalizing techniques for *merging* and *refining* ontologies [60,61] is an active area of research in several disciplines (such as deductive databases and information-integration). In particular, ontologies may be subject to *refinements* or changes in the light of new information. To illustrate our approach, we will show a naïve approach of how XDeLP can be used to model such changes by means of the notion of *program redefinition*, a concept borrowed from implementations of logic programming.[h] A redefinition of a program $\mathcal{P}_1$ w.r.t. another program $\mathcal{P}_2$ is a new DeLP program $\mathcal{P}$ that includes all predicate definitions in $\mathcal{P}_1$ and $\mathcal{P}_2$, except for those predicates in $\mathcal{P}_1$ which are also defined in $\mathcal{P}_2$. Formally:

Definition 5.1. (Redefinition) Let $\mathcal{P}_1, \mathcal{P}_2$ be two DeLP programs, such that $\mathcal{P}_1$ defines the predicates R_1, R_2, …, R_n, and $\mathcal{P}_2$ defines the predicates S_1, …, S_m. The *redefinition* of $\mathcal{P}_1$ w.r.t. $\mathcal{P}_2$, denoted $\mathcal{P}_1 \lhd \mathcal{P}_2$, is a new program $\mathcal{P}'$ defined as follows:
$\mathcal{P}' = \mathcal{P}_1 \lhd \mathcal{P}_2 =_{def} \{R_1^{\mathcal{P}_1}, \ldots, R_n^{\mathcal{P}_1}\} \cup \{S_1^{\mathcal{P}_2}, \ldots, S_m^{\mathcal{P}_2}\} \setminus \{\ R_i^{\mathcal{P}_1} \mid \exists S_j^{\mathcal{P}_2}$ in $\mathcal{P}_2$, with $name(R_i) = name(S_j)$, and $arity(R_i) = arity(S_j)\}$.

Thus, redefining a DeLP program basically involves providing new predicate definitions, which supersede already existing ones (if any).

Let us suppose that the bank gets a number of basic criteria from the Homeland Security Office (HSO) concerning how to assess trustworthiness of countries. Such criteria could be encoded by HSO programmers in a DeLP program $\mathcal{P}_{sec}$ as shown in Fig. 5: there is information about *greece* and *krakosia* being countries with democratic governments, and the fact that *krakosia* is a country at war. Defeasible rules provide tentative criteria for making a country credible: democratic countries are credible, unless they are at war or have corrupt governments. The bank authorities could therefore merge their knowledge base $\mathcal{P}_{bank}$ by considering the information given in $\mathcal{P}_{sec}$. The resulting redefined program $\mathcal{P}_{bank} \lhd \mathcal{P}_{sec}$ would consider a more detailed analysis for countries, as factors such as political system, political situation, etc. would be taken into account when granting loans, as shown in the following example.

Example 5.3. Consider the DeLP programs $\mathcal{P}_{bank}$={(1), …, (20)} and $\mathcal{P}_{sec}$={(1′), …, (6′)} from Fig. 1 and 5, resp. Computing $\mathcal{P}_{bank} \lhd \mathcal{P}_{sec}$ gives as a result a new DeLP program $\mathcal{P}'$ = {(1), …, (20)} ∪ {(1'), …, (6')} \ {(10),(11)}, in which the definition of *credible* provided by $\mathcal{P}_{bank}$ is replaced by the new de-

[h] In most PROLOG implementations, the result of the redefinition of the current program $\mathcal{P}_1$ w.r.t. another program $\mathcal{P}_2$ can be modelled by the command `reconsult(`$\mathcal{P}_2$`)`.[62]

finition given in $\mathcal{P}_{sec}$. Solving the query "$candidate(john)$" w.r.t. $\mathcal{P}'$ involves a search for arguments similar to the one perfomed in Example 2.3: an argument $\langle \mathcal{A}'_1, candidate(john)\rangle$ supports the query $candidate(john)$,[i] with:

$$\begin{aligned}\mathcal{A}'_1 = \{ & (candidate(john) \multimap profile\_ok(john));\\ & (profile\_ok(john) \multimap goodincome(john), trustctry(john, krakosia));\\ & (trustctry(krakosia) \multimap info(john, krakosia, \_, \_), credible(krakosia));\\ & (credible(krakosia) \multimap country(krakosia, democracy));\\ & (goodincome(john) \multimap info(john, \_, \_, 400), 400 > 300)\ \}.\end{aligned}$$

As in Example 2.3, this argument is defeated by another argument $\langle \mathcal{A}_2, \sim goodincome(john)\rangle$, which on its turn is defeated by another argument $\langle \mathcal{A}_3, solvent(john)\rangle$. In all these arguments, however, the redefined program allows a fourth argument to be inferred, namely $\langle \mathcal{A}_4, \sim credible(krakosia)\rangle$ with:

$$\begin{aligned}\mathcal{A}_4 = \{\sim credible(krakosia) \multimap\ & country(krakosia, democracy),\\ & country(krakosia, atwar)\}\end{aligned}$$

which is a proper defeater for $\langle \mathcal{A}'_1, candidate(john)\rangle$. As a result, the root of the dialectical tree for the query "$candidate(john)$" is marked as D-node, as shown in Fig. 4.

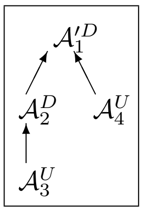


Fig. 4. Dialectical trees for query $candidate(john)$ w.r.t. $\mathcal{P}_{bank} \lhd \mathcal{P}_{sec}$

Note that redefining a program will usually result in providing more up-to-date information associated with particular predicates. Thus, arguments in a program $\mathcal{P}_1$ which had a particular epistemic status (e.g. warranted) might no longer keep it in a redefined version $\mathcal{P}_1 \lhd \mathcal{P}_2$.

6. XDeLP: implementation issues

6.1. *Embedding XDeLP in Client-side Code*

As already mentioned before, programmers usually perform some validations or pre-processing in data contained in a form by attaching some imperative, script-like language code to buttons (*e.g.* in Javascript). Such pieces of code allow typically

[i]Note that argument $\langle \mathcal{A}'_1, candidate(john)\rangle$ involves defeasible information about Krakosia coming from $\mathcal{P}_{sec}$, in contrast with the original argument $\langle \mathcal{A}_1, candidate(john)\rangle$.

(1′) $country(greece, democracy)$
(2′) $country(krakosia, democracy)$
(3′) $country(krakosia, atwar)$
(4′) $credible(Ctry) \prec country(Ctry, democracy).$
(5′) $\sim credible(Ctry) \prec country(Ctry, democracy), country(Ctry, atwar)$
(6′) $\sim credible(Ctry) \prec country(Ctry, democracy), country(Ctry, corruptgovt)$

Fig. 5. Defeasible logic program $\mathcal{P}_{sec}$ from the HSO

to validate field values, permitting to determine whether the intended meaning of such fields is coherent according to some criteria established by the form designer. However, as we have already seen in previous sections, there are many cases in which it may be useful to include as part of those validations some features which can be inferred as part of the "intended meaning" of the form without being field values themselves.

On the basis of the XDeLP formalization presented in Section 5, we propose to integrate the DeLP inference engine with a web-browser, extending the JavaScript programming[j] language with suitable primitives. The architecture for the proposed approach is depicted in Fig. 7. The extension to JavaScript consists of primitives for calling the DeLP engine services. This is implemented through specialized built-in boolean messages like *warranted*(*progid*, *query*), whose intended meaning is to determine if there exists a warranted literal *query* w.r.t. form $formid$. Similar functions are implemented for other possible values for defeasible attributes (*e.g.*, *undecided*). Next we show an example of how the proposed approach works in a JavaScript client-side script.

Example 6.1. Suppose $\mathcal{P}$ is a δ-form as described in Example 3.1, which in turn is written in XForms and has $form1$ as its identifier. Then, a JavaScript programmer would be capable of writing the code embedded in a handler function for the *Validate* button as shown in Fig. 6.

6.2. *Computing Arguments Efficiently in DeLP*

As stated before, XDeLP solves queries on the basis of the DeLP argumentative formalism. As performing defeasible argumentation is a computationally complex task, an abstract machine called JAM (Justification Abstract Machine) has been specially developed for an efficient implementation of DeLP.[18] JAM provides an argument-based extension of the traditional WAM (Warren's Abstract Machine) for PROLOG. A full-fledged implementation of DeLP is available online,[k] including facilities for visualizing arguments and dialectical trees. On the basis of this abstract machine a Java-based Integrated Development Environment (IDE) for DeLP

[j]For the sake of simplicity we restrict ourselves to the case of JavaScript for our analysis. The approach can be naturally extended to any other scripting language.
[k]See `http://lidia.cs.uns.edu.ar/DeLP`

```
<script language="JavaScript">
        function validate()
        {
                if( form1.warranted(pbank, candidate(form1.name.value)) )
                    alert( "The requested loan will be probably granted." +
                           "We will contact you in a week." );
                else
                    alert( "Your case will be analyzed and " +
                           "we will contact you in a month." );
        }
</script>
```

Fig. 6. An example of JavaScript function attached to a form button

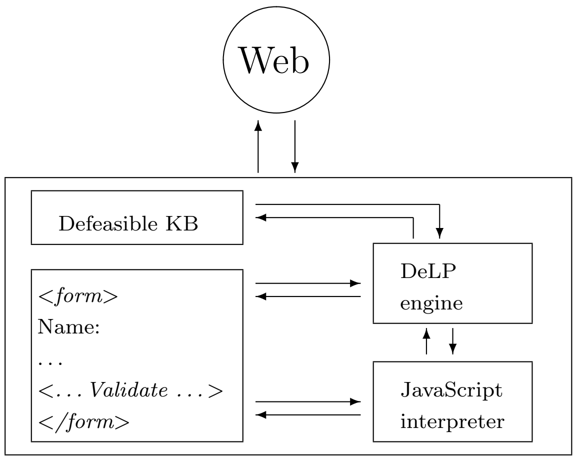


Fig. 7. A framework for embedding the DeLP inference engine in a browser application

has also been developed,[63] which was used in our experiments as a prototype for embedding XDELP in a web browser. This Java version of DeLP allows to compile DeLP code into JAM opcodes. A visual environment for interacting with DeLP programs is provided. Several features leading to efficient implementations of DeLP have also been recently studied, in particular those related to comparing conflicting arguments by specificity,[47] computing dialectical trees efficiently[2] and extending DeLP to incorporate possibilistic reasoning.[6,64,3] Equivalence results with other extensions of logic programming have also been established.[40]

Performing defeasible argumentation is a computationally complex task. Cecchi *et. al.*[65] have proved that the complexity of the decision problem "whether a set of defeasible rules is an argument for a literal under a defeasible logic logic program" is P-complete, and the existence of an argument for a literal to be in NP.

7. Related work

To the best of our knowledge there are no other works in the area of introducing defeasible knowledge in web-based forms as done in this paper. Recent research [16] has been focused on developing a methodology for designing form-based decision support systems, which uses factoring and synthesis to process knowledge involved in forms. The resulting framework allows flexible creation and modification of computer-generated forms useful for decision making and suited for simplifying the process of report generation. However, even though this approach exploits the semantics of the knowledge involved in forms, it does not provide any connection with web-based systems nor with handling defeasible knowledge.

A pioneering work in the area of ontology representation for the semantic web is SHOE.[55] As in our approach, SHOE combines features of markup languages, knowledge representation, and ontologies. Its basic structure consists of ontologies, which define rules that guide what kind of assertions may be made and what kind of inferences may be drawn on ground assertions, and instances that make assertions based on those rules. Unlike to our approach, SHOE only implements strict Horn-rules while we also provide capabilities for the representation of defeasible rules. Besides, SHOE is designed to eliminate the possibility of contradictions as it does not permit logical negation. Instead, our approach not only allows for the use of negations but it handles logical contradictions by means of a dialectical analysis, making it suitable for interaction among agents in the realm of semantic web applications.

In similar direction to our work, rule-based defeasible reasoning in the context of the Semantic Web has motivated the development of alternative systems such as DR-DEVICE,[13] which is capable of reasoning about RDF metadata over multiple Web sources using defeasible logic rules.[12,11] In contrast with our approach, this system is implemented on top of the CLIPS production rule system, whereas our proposal relies on the computation of warrant performed by the DeLP inference engine using logic programming techniques. Furthermore, comparison among rules in defeasible logic is performed on the basis of a superiority relationship, whereas our proposal relies on a modular comparison criterion among arguments. Besides, DeLP does not need to be supplied with defeater rules because the system will find all possible counterarguments automatically on the basis of the arguments it is able to build, and will decide on the defeat relation using the provided comparison criterion. Thus, a DeLP programmer does not need to encode exceptions explicitly.

The Rule Markup Initiative[66] constitutes a joint effort towards defining a shared *Rule Markup Language* (RuleML), permitting both forward (bottom-up) and backward (top-down) rules in XML for deduction, rewriting, and further inferential-transformational tasks. Our approach to encoding DeLP programs shares in part some of the goals of the RuleML initiative. Nevertheless, the current RuleML language does not provide tags for integrating defeasible implications, arguments, and argumentation trees, as presented here.

As discussed in Section 5.2, XDeLP offers a knowledge representation language that can be used for characterizing ontologies, using the underlying argumentative engine to perform inferences. In this respect, using argumentation for modeling ontology engineering has been recently focus of research from a social perspective, *i.e.*, defining a shared ontology as a social process. In this setting, different participants collaborate in developing a shared ontology. During the discussion, participants exchange arguments which may give support to or object certain ontology engineering decisions. Tempich *et. al.*[67] present an ontology which formalizes the main concepts which are used in an ontology engineering discussion and thus enables tracking arguments and allows for inconsistency detection. Their approach, however, differs from ours, in the sense that the focus is in providing an ontology for modeling argumentation, whereas our proposal focuses on using XDeLP language as a vehicle for modeling ontologies in general and using argument-based inference as a reasoning mechanism.

Wagner[68] points out the need of representing negative information in Semantic Web applications. In this regard, mainstream developments concerning reasoning in the web propose using languages based on very expressive Description Logics (DL)[69] (such as DAML,[70] OIL,[71] DAML+OIL,[72] and OWL[73]) for representing ontologies. Although those approaches combine successfully expressiveness and efficiency issues, they fail to deal with inconsistent knowledge bases as current systems (notably RACER[74]) only perform a consistency check while leaving the burden of eliminating inconsistencies to the programmer. For instance, consider the following DL (inconsistent) knowledge base $\mathcal{KB} = (TBox, ABox)$ where $TBox = \{(phdstudent \sqsubseteq \neg candidate); (phdstudent \sqcap rich \sqsubseteq candidate)\}$, and $ABox = \{phdstudent(john), rich(john)\}$ establishing that PhD students are not candidates for loans unless they are rich and that John is both a PhD student and rich. Clearly, both $candidate(john)$ and $\neg candidate(john)$ can be derived. In contrast to those languages, DeLP is also capable of handling inconsistencies in a natural transparent way as we show next.

Relating DL and Logic Programming, Grosof *et. al.*[75] propose an algorithm for translating a class of DL programs to Prolog. This class of programs is defined by the intersection of DL with Logic Programming, characterizing the so-called Description Logic Programming. Part of our research is currently focused on extending those results to DeLP. For instance, $\mathcal{KB}$ defined above could be translated as an equivalent defeasible logic program $\mathcal{P} = (\Pi, \Delta)$ where $\Pi = \{phdstudent(john), rich(john)\}$ and $\Delta = \{(\sim candidate(X) \prec phdstudent(X)); (candidate(X) \prec phstudent(X), rich(X))\}$. Two arguments can be derived w.r.t. program $\mathcal{P}$, namely $\langle \mathcal{A}_1, \sim candidate(john)\rangle$ and $\langle \mathcal{A}_2, candidate(john)\rangle$, where $\mathcal{A}_1 = \{\sim candidate(john) \prec phdstudent(john)\}$ and $\mathcal{A}_2 = \{candidate(john) \prec phdstudent(john), rich(john)\}$. Using specificity as the comparison criterion, argument $\mathcal{A}_2$ will result warranted, thus solving the contradicting situation. Note that while the output of the procedure proposed by Grosof *et. al.* (when a given knowledge base is used as an input) is expressive enough for turning standard Prolog rules into defeasible rules, it does not take into account the treatment of explicit negation, as required in DeLP.

8. Conclusions and Future Work

We have presented a novel argument-based approach for enriching traditional forms for web-based environments, which can be suitably adapted to existing markup language technologies like XHTML. As discussed in the introduction, our proposal involves providing the possibility of modeling inferences based on concepts which are part of the intended meaning of a form, which we have formalized as defeasible attributes.

We have shown that the use of an embedded DeLP interpreter on the client side allows the form designer to develop richer form schemas, in which the interaction of defeasible attributes is taken into account as part of the "behavior" of the form. Knowledge bases for forms are expressed in a declarative way, making easier to enrich a form by for example merging two existing knowledge bases. It must be noted that our approach assumes that all available information is encoded on the client side (browser application), which arises a number of privacy and security issues. In the case of the bank application, the bank would probably want to keep its criteria undisclosed, so that they could not be misused by the client or by third parties. In such cases client-side security considerations would be in order (e.g. by incorporating encryption techniques), which are currently an issue under research.

Implementing program redefinition as described in Section 5.2 is quite straightforward, and offers an attractive possibility for integrating defeasible knowledge bases from different sources (as $\mathcal{P}_{bank}$ and $\mathcal{P}_{sec}$). Clearly, additional ontological considerations (*e.g.* unique name assumption, etc.) are required for such merging operations; extending our formalization to handle such considerations is part of our current research work. The sample problem presented in this paper was encoded using a Java-based DeLP interpreter and solved successfully under the methodology we have described. However, our experiments regarding this approach only account as a "proof of concept" prototype, as we have not been able yet to carry out thorough evaluations in the context of real-world applications. Research in this direction is currently being pursued.

Acknowledgments

This research was funded by Agencia Nacional de Promoción Científica y Tecnológica (PICT 2002 No. 13.096, PICT 2003 No. 15.043, PAV 2004 076), by CONICET (Argentina), by projects TIC2003-00950 and TIN2004-07933-C03-03 (MCyT, Spain) and by Ramón y Cajal Program (MCyT, Spain).

References

1. C. Chesñevar, A. Maguitman, and G. Simari. Argument-based critics and recommenders: A qualitative perspective on user support systems. *Data & Knowledge En-*

gineering (in press), 2006.

2. C. Chesñevar, G. Simari, and L. Godo. Computing dialectical trees efficiently in possibilistic defeasible logic programming. *LNAI Springer Series Vol. 3662 (Proc. of the 8th Intl. Conference on Logic Programming and Nonmonotonic Reasoning LPNMR 2005)*, pages 158–171, September 2005.
3. C. Chesñevar, G. Simari, L. Godo, and T. Alsinet. Argument-based expansion operators in possibilistic defeasible logic programming: Characterization and logical properties. *LNAI/LNCS Springer Series, Vol. 3571 (Proc. of the 8th ECSQARU Intl. Conference, Barcelona, Spain)*, pages 353–365, September 2005.
4. C. Chesñevar, R. Brena, and J. Aguirre. Knowledge distribution in large organizations using defeasible logic programming. In *Proc. of the 18th Canadian Conference on AI (published in LNCS, Vol. 3501, Springer Verlag)*, pages 244–256. Springer Verlag, 2005.
5. C. Chesñevar and A. Maguitman. An Argumentative Approach to Assessing Natural Language Usage based on the Web Corpus. In *Proc. of the 16th ECAI Conf., Valencia, Spain*, pages 581–585, August 2004.
6. C. Chesñevar, G. Simari, T. Alsinet, and L. Godo. A Logic Programming Framework for Possibilistic Argumentation with Vague Knowledge. In *Proc. of the Intl. Conference in Uncertainty in Artificial Intelligence (UAI 2004). Banff, Canada*, pages 76–84, July 2004.
7. S. Gómez and C. Chesñevar. A Hybrid Approach to Pattern Classification Using Neural Networks and Defeasible Argumentation. In *Proc. of 17th Intl. FLAIRS Conference. Miami, Florida, USA*, pages 393–398. American Assoc. for Art. Intel., May 2004.
8. C. Chesñevar and A. Maguitman. ArgueNet: An Argument-Based Recommender System for Solving Web Search Queries. In *Proc. of the 2nd IEEE Intl. IS-2004 Conference. Varna, Bulgaria*, pages 282–287, June 2004.
9. M. Capobianco, C. Chesñevar, and G. Simari. An argument-based framework to model an agent's beliefs in a dynamic environment. In *Proc. of the First International Workshop on Argumentation in Multiagent Systems. AAMAS 2004 Conference, New York, USA*, pages 163–178, July 2004.
10. R. Brena, C. Chesñevar, and J. Aguirre. Argumentation-supported information distribution in a multiagent system for knowledge management. In *Proc. 2nd. Intl. Workshop on Argumentation in Multiagent Systems (ArgMAS). 4th Intl. AAMAS Conf., Utrecht, Holland. To appear in LNCS/LNAI Springer Series (in press)*, July 2005.
11. G. Antoniou, A. Bikakis, and G. Wagner. A system for nonmonotonic rules on the web. *LNCS 3323 (Proc. of RuleML2004)*, pages 23–26, 2004.
12. G. Antoniou, D. Billington, G. Governatori, and M. Maher. Representation results for defeasible logic. *ACM Trans. on Comp. Logic*, 2(2):255–287, 2001.
13. N. Bassiliades, G. Antoniou, and I. Vlahavas. A defeasible logic reasoner for the semantic web. In *Proc. of the Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 49–64, 2004.
14. N. Bassiliades, E. Kontopoulos, and G. Antoniou. A visual environment for developing defeasible rule bases for the semantic web. In S. Tabet A. Adi, S. Stoutenburg, editor, *Proc. International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML-2005). Galway, Ireland, 10-12 November, 2005 (in press)*, pages 49–64. Springer Verlag, 2005.
15. N. Bassiliades, G. Antoniou, and I. Vlahavas. A defeasible logic reasoner for the semantic web. *International Journal on Semantic Web and Information Systems (to appear)*, 2005.

16. J. Wu, H. Doong, C. Lee, T. Hsia, and T. Liang. A methodology for designing form-based decision support systems. *Decision Support Systems*, (36):313–335, 2004.
17. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scient. American*, 2001.
18. A. García and G. Simari. Defeasible Logic Programming an Argumentative Approach. *Theory and Prac. of Logic Program.*, 4(1):95–138, 2004.
19. C. Chesñevar, A. Maguitman, and R. Loui. Logical Models of Argument. *ACM Computing Surveys*, 32(4):337–383, December 2000.
20. John McCarthy and Patrick J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.
21. Raymond Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13(1,2):81–132, April 1980.
22. John McCarthy. *Formalizing Common Sense: Papers by John McCarthy.* Ablex Publishing Corporation, 1990.
23. John Pollock. *Knowledge and Justification.* Princeton, 1974.
24. John L. Pollock. Defeasible Reasoning. *Cognitive Science*, 11:481–518, 1987.
25. Donald Nute. Defeasible Reasoning. In James H. Fetzer, editor, *Aspects of Artificial Intelligence*, pages 251–288. Kluwer Academic Publishers, Norwell, MA, 1988.
26. J. L. Pollock. *Cognitive Carpentry: a blueprint for how to build a person.* Bradford/MIT Press, 1995.
27. G. Simari and R. Loui. A Mathematical Treatment of Defeasible Reasoning and its Implementation. *Artificial Intelligence*, 53:125–157, 1992.
28. 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.
29. David Poole. Explanation and Prediction: an Architecture for Default and Abductive Reasoning. *Computational Intelligence*, 5:97–110, 1989.
30. P. Zhang, J. Sun, and H. Chen. Frame-based argumentation for group decision task generation and identification. *Decision Support Systems*, 39:643–659, 2005.
31. Daniela Carbogim, David Robertson, and John Lee. Argument-based applications to knowledge engineering. *The Knowledge Engineering Review*, 15(2):119–149, 2000.
32. H. Prakken and G. Sartor. The role of logic in computational models of legal argument - a critical survey. In A. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond*, pages 342–380. Springer, 2002.
33. Bart Verheij. *Virtual Arguments. On the Design of Argument Assistants for Lawyers and Other Arguers.* Asser Press, The Hague, 2005.
34. S. Parsons, C. Sierrra, and N. Jennings. Agents that Reason and Negotiate by Arguing. *Journal of Logic and Computation*, 8:261–292, 1998.
35. Carles Sierra and Pablo Noriega. Agent-mediated interaction. from auctions to negotiation and argumentation. In *Foundations and Applications of Multi-Agent Systems – In LNCS Series, Vol. 2403*, pages 27–48. Springer, 2002.
36. Iyad Rahwan, Sarvapali D. Ramchurn, Nicholas R. Jennings, Peter Mcburney, Simon Parsons, and Liz Sonenberg. Argumentation-based negotiation. *Knowl. Eng. Rev.*, 18(4):343–375, 2003.
37. Henry Prakken and Gerard Vreeswijk. Logical Systems for Defeasible Argumentation. In D. Gabbay and F.Guenther, editors, *Handbook of Philosophical Logic*, pages 219–318. Kluwer Academic Publishers, 2002.
38. Antonios C. Kakas and Francesca Toni. Computing argumentation in logic programming. *Journal of Logic and Computation*, 9(4):515–562, 1999.

39. C. Chesñevar, R. Brena, and J. Aguirre. Modelling power and trust for knowledge distribution: an argumentative approach. *LNAI Springer Series (Proc. of the 3rd Mexican International Conference on Artificial Intelligence – MICAI 2005)*, 3789:98–108, November 2005.
40. Carlos I. Chesñevar, Jürgen Dix, Frieder Stolzenburg, and Guillermo R. Simari. Relating Defeasible and Normal Logic Programming through Transformation Properties. *Theoretical Computer Science*, 290(1):499–529, 2003.
41. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, New York, second edition edition, 1987.
42. Vladimir Lifschitz. Foundations of logic programming. In *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, 1996.
43. Michael Gelfond and Vladimir Lifschitz. Logic programs with classical negation. In *Proceedings of the 7th International Conference on Logic Programming. Jerusalem*, June 1990.
44. Phan M. Dung. On the Acceptability of Arguments and its Fundamental Role in Nomonotonic Reasoning and Logic Programming. In *Proc. of the 13th. International Joint Conference in Artificial Intelligence (IJCAI), Chambéry, Francia*, pages 321–357, 1993.
45. Henry Prakken and Giovani Sartor. Argument-based extended logic programming with defeasible priorities. *Journal of Applied Non-classical Logics*, 7:25–75, 1997.
46. Robert Kowalski and Francesca Toni. Abstract Argumentation. *Artificial Intelligence and Law*, 4(3-4):275–296, 1996.
47. F. Stolzenburg, A. García, C. Chesñevar, and G. Simari. Computing Generalized Specificity. *J. of N.Classical Logics*, 13(1):87–113, 2003.
48. Donald Nute. Basic defeasible logic. In Luis Fariñas del Cerro, editor, *Intensional logics for programming.* Oxford: Claredon Press, 1992.
49. Grigoris Antoniou, Michael J. Maher, and David Billington. Defeasible logic versus logic programming without negation as failure. *Journal of logic programming*, 42:47–57, 2000.
50. Grigoris Antoniou, David Billington, and Michael Maher. Normal forms for defeasible logic. In *Proceedings of international joint conference and symposium on logic programming*, pages 160–174. MIT Press, 1998.
51. Antonis C. Kakas, Paolo Mancarella, and Phan M. Dung. The acceptability semantics for logic programs. In *Proceedings of the 11th. international conference on logic programming. Santa Margherita, Italy*, pages 504–519. MIT Press, 1994.
52. Yannis Dimopoulos and Antonis Kakas. Logic programming without negation as failure. In J. Lloyd, editor, *Logic Programming*, pages 369–383. MIT Press, Cambridge, MA, 1995.
53. Henry Prakken. Relating protocols for dynamic dispute with logics for defeasible argumentation. *Synthese*, 127:187–219, 2001.
54. M. Dubinko, L. Klotz, R. Merrick, and T.V. Raman. XForms 1.0 - W3C Recomm. 14 Oct. 2003, 2003.
55. J. Heflin, J. Hendler, and S. Luke. SHOE: A Blueprint for the Semantic Web. pages 29–63, 2003.
56. J. Clark. XSL Transformations (XSLT) Version 1.0. W3C Recommendation 16 Nov. 1999, 1999.
57. Sean McGrath. *XML by example. Building e-commerce applications.* Prentice Hall, 1998.
58. Steffen Staab and Rudi Studer, editors. *Handbook on Ontologies.* International Handbooks on Information Systems. Springer, 2004.

59. S. Gómez, C. Chesñevar, and G. Simari. A language for ontology interchange among intelligent agents on the semantic web based on defeasible argumentation. In *Workshop on Agents and Intelligent Systems. Proc. of the XI Argentinean Conference in Computer Science (to appear). Universidad de Entre Ríos, Argentina*, 2005.
60. Natalya F. Noy. Semantic integration: a survey of ontology-based approaches. *SIGMOD Rec.*, 33(4):65–70, 2004.
61. Dejing Dou, Drew V. McDermott, and Peishen Qi. Ontology translation on the semantic web. *Journal of Data Semantics*, 2:35–57, 2005.
62. Ivan Bratko. *Prolog Programming for Artificial Intelligence (2nd. Ed).* Addison Wesley, 1990.
63. A. Stankevicius, A. Garcia, and G. Simari. Compilation techniques for defeasible logic programs. In *Proc. of the 6th Intl. Congress on Informatics Engineering*, pages 1530–1541. Univ. de Buenos Aires, Bs. Aires, Argentina, Ed. Fiuba, April 2000.
64. C. Chesñevar, G. Simari, T. Alsinet, and L. Godo. Modelling agent reasoning in a logic programming framework for possibilistic argumentation. In *Proc. of 2nd European Workshop on Multiagent Systems. Barcelona, Spain*, pages 135–142, December 2004.
65. Laura A. Cecchi, Pablo R. Fillottrani, and Guillermo R. Simari. On Complexity of DeLP through Game Semantics. *Eleventh International Workshop on Non-Monotonic Reasoning (in press)*, 2006.
66. Harold Boley, Mike Dean, Benjamin Grosof, Michael Sintek, Bruce Spencer, Said Tabet, and Gerd Wagner. FOL RuleML: The First-Order Logic Web Language, 2004.
67. Christoph Tempich, Helena Sofia Pinto, York Sure, and Steffen Staab. An argumentation ontology for distributed, loosely-controlled and evolving engineering processes of ontologies (diligent). In Asunción Gómez-Pérez and Jérôme Euzenat, editors, *ESWC*, volume 3532 of *Lecture Notes in Computer Science*, pages 241–256. Springer, 2005.
68. Gerd Wagner. Web Rules Need Two Kinds of Negation. In N. Henze F. Bry and J. Maluszynski, editors, *Proc. of the 1st International Workshop, PPSW3 '03. Springer-Verlag LNCS 2901*, 2003.
69. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook – Theory, Implementation and Applications.* Cambridge University Press, 2003.
70. Deborah McGuiness, Richard Fikes, Lynn Andrea Stein, and James Hendler. DAML-ONT: An Ontology Language for the Semantic Web. In Dieter Fensel, James Hendler, Henry Lieberman, and Wolfgang Wahlster, editors, *Spinning the Semantic Web*, pages 65–93. The MIT Press, 2003.
71. S. Decker, D. Fensel, F. van Harmelen, I. Horrocks, S. Melnik, M. Klein, and J. Broekstra. Knowledge representation on the web. *Proc. of the 2000 Description Logic Workshop (DL 2000)*, pages 89–97, 2000.
72. Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, and Lynn Andrea Stein. DAML+OIL (March 2001) Reference Description, 2001. http://www.w3.org/TR/daml+oil-reference.
73. Deborah L. McGuiness and Frank van Harmelen. OWL Web Ontology Language Overview, 2004. http://www.w3.org/TR/owl-features/.
74. Volker Haarslev and Ralf Möller. RACER System Description. Technical report, University of Hamburg, Computer Science Department, 2001.
75. Benjamin N. Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description Logic Programs: Combining Logic Programs with Description Logics. *WWW2003, May 20-24, Budapest, Hungary*, 2003.