The Uses of Computational Argumentation:
Papers from the AAAI Fall Symposium (FS-09-06)

Incorporating Classical Logic Argumentation into Policy-based Inconsistency Management in Relational Databases

Maria Vanina Martinez
University of Maryland College Park
College Park, MD 20742, USA
mvm@cs.umd.edu

Anthony Hunter
University College London
London WC1E 6BT, UK
a.hunter@cs.ucl.ac.uk

Abstract

Inconsistency management policies allow a relational database user to express customized ways for managing inconsistency according to his need. For each functional dependency, a user has a library of applicable policies, each of them with constraints, requirements, and preferences for their application, that can contradict each other. The problem that we address in this work is that of determining a subset of these policies that are suitable for application w.r.t. the set of constraints and user preferences. We propose a classical logic argumentation-based solution, which is a natural approach given that integrity constraints in databases and data instances are, in general, expressed in first order logic (FOL). An automatic argumentation-based selection process allows to retain some of the characteristics of the kind of reasoning that a human would perform in this situation.

1. Introduction

Inconsistency Management Policies (IMPs for short) are presented in (Martinez et al. 2008) as a framework that allows relational database users to specify how to handle inconsistency regarding a set of functional dependencies, making use of their expertise to provide customized ways of dealing with conflicts. Functional dependencies are a special kind of integrity constraint that express that the equality on the values of a set of attributes determines the equality of the values of another set of attributes, *e.g.*, if two tuples in a relation have the same value for attribute SSN, then the value for attribute $Name$ should be the same.

In (Martinez et al. 2008), it is assumed that given a set of functional dependencies $\mathcal{F}$, there is only one single-dependency policy associated with each *fd* in $\mathcal{F}$. This is a somewhat restrictive assumption since, in general, it is very likely that a whole set of candidate policies exists. Given the nature of IMPs, it is reasonable to allow the users to express constraints or requirements for the usage of policies, as well as preferences over them. The problem that arises then, is that constraints and user preferences can contradict each other, and therefore not all the candidate policies are applicable in all situations. Given a set of policies, users need a mechanism to select the one(s) whose properties and output(s) satisfies their application needs, constraints, and preferences. We will consider this problem assuming that the decision is made *one functional dependency at a time*.

If a user is presented with a relatively small number of policies, and conflicting constraints and preferences, then it is likely that he will weigh the different pieces of information and determine a preference on which policies he should use in that particular situation. When the amount of data grows, it becomes impossible for a human user to analyze it. We propose to address this problem in an automatic way, but retaining some of the characteristics of the kind of reasoning that humans perform in such situations; for instance, we would like to have the capacity to explain why a certain choice was made over another. In this work we adopt an argumentation-based framework that uses classical logic as the underlying formalism, as proposed in (Besnard and Hunter 2001). Integrity constraints and relational data are already expressed in the language of first order logic and their semantics corresponds directly to that of classical logic; it seems appropriate and natural then to choose a classical logic based approach. Policies, together with their requirements and constraints, as well as user preferences, can be encoded as FOL formulas and then be used in the process of policy selection. The main idea is to automatically generate arguments and counterarguments for and against the application of particular policies, and through a dialectical process choose the most suitable ones.

2. IMPs: Preliminaries

The concept of *policies* for managing inconsistency in databases violating a given set of functional dependencies was introduced in (Martinez et al. 2008), with the intention of obtaining a lower *degree* of inconsistency. We now briefly recall the principal concepts underlying the framework.

We assume the existence of relational schemas of the form $S(A_1, \ldots, A_n)$ where the A_i's are attributes. Each attribute A_i has an associated domain, $dom(A_i)$. A *tuple over* S is a member of $dom(A_1) \times \cdots \times dom(A_n)$, and a set R of such tuples is called a *relation*. We use *Attr*(S) to denote the set of all attributes in S. Moreover, we use $t[A_i]$ to denote the value of A_i attribute in tuple t.

Given the relational schema $S(A_1, \ldots, A_n)$, a functional dependency *fd* over S is an expression of the form $A'_1 \cdots A'_k \rightarrow A'_{k+1} \cdots A'_m$ where $\{A'_1, \ldots, A'_m\} \subseteq$ *Attr*(S). Relation R over schema

S satisfies the above functional dependency iff $\forall t_1, t_2 \in R$, $t_1[\{A'_1, \ldots, A'_k\}] = t_2[\{A'_1, \ldots, A'_k\}] \Rightarrow t_1[\{A'_{k+1}, \ldots, A'_m\}] = t_2[\{A'_{k+1}, \ldots, A'_m\}]$. Finally, we say that *fd* is defined over R.

The notions of *culprits* and *clusters* are the basic structures over which IMPs are defined. In the following we recall their formal definitions.

Definition 1 (Culprits and clusters) *Let R be a relation and $\mathcal{F}$ a set of functional dependencies. A* culprit *is a minimal set $c \subseteq R$ not satisfying $\mathcal{F}$. Moreover, given two culprits c, $c' \in culprits(R, \mathcal{F})$, we write $c \bigtriangleup c'$ iff $c \cap c' \neq \emptyset$. Let $\bigtriangleup^*$ be the reflexive transitive closure of relation $\bigtriangleup$; a* cluster *is a set $cl = \bigcup_{c \in e} c$ where e is an equivalence class of $\bigtriangleup^*$.*

We denote the set of all clusters in R w.r.t. $\mathcal{F}$ with *clusters*$(R, \mathcal{F})$.

Informally, culprits are minimal inconsistent subsets, and clusters group together tuples that are inconsistent among each other w.r.t. a functional dependency.

An *inconsistency management policy* (IMP for short) for a relation R w.r.t. a functional dependency *fd* over R is a function γ_{fd} from R to a relation $R' = \gamma_{fd}(R)$ that satisfies certain axioms, which we review informally here:

- Tuples that do not belong to any culprit cannot be eliminated or changed by γ_{fd}.
- Every tuple t' in R' is either in R or there exists a tuple t in R such that t and t' coincide in all values of attributes that do not appear in fd.
- Applying γ_{fd} cannot increase the number of culprits.
- Applying γ_{fd} cannot increase the number of tuples.

IMPs seek to reduce the *degree* of inconsistency in the relation, according to some appropriate measure of inconsistency (Lozinskii 1994; Hunter and Konieczny 2005; Grant and Hunter 2006; Martinez et al. 2007). In (Martinez et al. 2008) three types of policies are described: *tuple-based*, *value-based*, and *interval-based* policies. Informally, tuple-based policies delete tuples from a cluster, while value-based and interval-based ones allow to change the values of the attributes on the right hand side of the dependency; value-based policies only allow changes to values that are already present in the cluster, while interval-based policies allow values within the interval defined by the values in the cluster. The following example shows an instance of a relation and two value-based policies that are going to be used as the running example in the rest of the paper.

Example 1 *Assume the relation* Employee *in which the salaries are uniquely determined by the names. Therefore, the only functional dependency is fd : $Name \rightarrow Salary$.*

	Name	*Salary*	*Tax_bracket*
t_1	$John$	$70K$	20
t_2	$John$	$80K$	20
t_3	$John$	$75K$	20
t_4	$Mary$	$90K$	30

An IMP for $fd \in \mathcal{F}$ is applied to a relation by subsequently applying it to each cluster in that relation w.r.t. fd. In this example there is only one cluster w.r.t. fd, namely $\{t_1, t_2, t_3\}$ and there are two culprits. Suppose that the user of this database has the following two policies associated with fd:

- *Policy γ_1 says: for each cluster $c \in$ clusters$(Employee, fd)$, and $c = \{t_1, \ldots t_n\}$, for each tuple $t_i \in c$, set $t_i[salary] = \min(\{t_1[Salary], \ldots, t_n[Salary]\})$.*
- *Policy γ_2 says: for each cluster $c \in$ clusters$(Employee, fd)$, and $c = \{t_1, \ldots t_n\}$, for each tuple $t_i \in c$, if there exist k tuples in c (different from t_i) that have the same value for $Salary$ as t_i, then $t_i[salary]$ does not change; otherwise set $t_i[Salary] = \min(\{t_1[Salary], \ldots, t_n[Salary]\})$.*

Different policies produce different results (sets of tuples), each reducing the number of culprits or clusters to some extent by deleting or modifying tuples that satisfy a certain condition, etc. Furthermore, different users have different requirements depending on the task they seek to perform. Users can also have preferences over the application of policies, *e.g.*, they might require certain type of data to be included in the results, they might need to impose resource constraints such as computation time, etc. Suppose a user needs the results very quickly, and the two applicable policies return the same answers, or almost the same answer; in this case, the user might prefer to apply the policy that takes less time to compute, even if the results are just an approximation or a part of what he wanted in the first place.

Example 2 *Consider the setting of Example 1. If n is the maximum cardinality of clusters, and there are m clusters, then if γ_1 is applied to this relation, all salary values are changed to $70K$. The resulting number of clusters and culprits are both 0 after the application. The worst case running time for γ_1 is $O(m*n)$. The most efficient way to compute γ_2 requires to sort the tuples in each cluster and then perform binary searches to find k tuples with the same value for $Salary$. Assuming $k = 1$, the worst case running time for γ_2 is $O(m*n \log n)$. In this case, $t_2[Salary]$ changes to $70K$; the resulting number of culprits and clusters w.r.t. fd are both 0.*

Clearly, from the point of view of degree of inconsistency left, neither policy is more desirable than the other, since both completely remove the inconsistency w.r.t. fd. However, depending on the number and size of the clusters, the application of γ_2 can take much more time than γ_1. If the user has time constraints, then he could be more inclined to select γ_1. On the other hand, if the results of applying one or another are different (which is not the case for our simplified example), the user might be willing to sacrifice efficiency selecting γ_2 instead of performing a more thorough processing.

3. Knowledge Representation for IMPs

In order to make decisions regarding which policy to apply given a relation, a user of that relation, and a functional dependency, different kinds of information are needed in order to make an informed choice. Given a functional dependency *fd*, our framework contains the following levels of knowledge: data level, meta-data level, policy level, and meta-policy level. A system that automatically selects which policies are suitable for application must also have these different levels of knowledge available for the decisions and recommendations to be meaningful. In the rest of this section we define the contents of each level of information and provide a formal representation for them using FOL formulas.

3.1 Data Level Knowledge

One of the basic elements needed in the process of policy selection is the set of data to which the policies will be applied. In order to make an appropriate selection, this has to be done in context, since different policies might behave differently when applied to different data. Furthermore, it is likely that users have different preferences and requirements for different data sets. Assuming there is a set of relations which users are interested in, automatic policy selection requires encoding the contents of these relations in the underlying formalism, in this case FOL formulas.

From now on, we will denote a relation within the logic with a constant symbol corresponding to the name of the relation in lowercase. For instance, relation *Employee* will be denoted with *employee*, etc.

Definition 2 (Data term) *Given relation R over schema $S(A_1, \dots A_n)$, the ground term $f_R(c_1, \dots, c_k)$ denotes the tuple $t[\mathcal{A}] \in R$, where f_R is a function symbol of arity k, the c_i's are constant symbols, $\mathcal{A}$ is the ordered set $\{A'_1, \dots, A'_k\} \subseteq Attr(S)$, and $t[A'_i] = c_i$ for $1 \leq i \leq k$.*

Note that we assume that values for attributes within the relation are constant symbols; each tuple is therefore denoted as a ground term.

Definition 3 *Given relation R over schema $S(A_1, \dots, A_n)$, the set $K_{T(R)} = \{In(f_R(c_1, \dots, c_k), R) \mid \exists t \in R \; s.t. \; t[A_1] = c_1, \dots,$ and $t[A_k] = c_k\}$ denotes the tuples in relation R, where In is a predicate symbol of arity 2.*

The following example shows relation *Employee* from Example 1 represented as a set of ground atoms.

Example 3 *Assume we have relation* Employee *from Example 1 and let $f_{Employee}$ be a function symbol of arity 3. The set $K_{T(Employee)}$ is defined as follows:*

$d_1 : In(f_{Employee}(john, 70, 20), employee)$
$d_2 : In(f_{Employee}(john, 80, 20), employee)$
$d_3 : In(f_{Employee}(john, 75, 20), employee)$
$d_4 : In(f_{Employee}(mary, 90, 30), employee)$

3.2 Meta-data Level Knowledge

In a policy selection process it is important to consider not only the data from the relation but also information regarding inconsistency in the data w.r.t. a given functional dependency. In particular, we need to explicitly state the composition of the cluster structures. We assume the existence of predicate symbols *Cluster* and *InCluster*, of arity 3 and 2, respectively. Making use of these predicate symbols it is possible to represent the cluster structure as a set of atoms.

Definition 4 *Given relation R and a functional dependency fd, $K_{C(R,fd)} = \{Cluster(cl, r, fd) \mid cl \in clusters(R, fd)\} \cup \{InCluster(f_R(c_1, \dots, c_k), cl) \mid \exists t \in cl \; s.t. \; t[A_1] = c_1, \dots, t[A_k] = c_k\}$.*

Example 4 *For our running example, we have that $K_{C(Employee,fd)} =$*

$d_5 : Cluster(c_1, employee, fd)$
$d_6 : InCluster(f_{Employee}(john, 70, 20), c_1)$
$d_7 : InCluster(f_{Employee}(john, 80, 20), c_1)$
$d_8 : InCluster(f_{Employee}(john, 75, 20), c_1)$

Given a relation R and a functional dependency fd of interest, the set $K_{Data(R,fd)} = K_{T(R)} \cup K_{C(R,fd)}$ contains all the data terms that denote tuples in relation R together with the information regarding clusters w.r.t. fd.

3.3 Policy Level Knowledge

Given a relation R and a set of functional dependencies $\mathcal{F}$, we will denote with $policies(fd)$ the set of single-dependency policies associated with $fd \in \mathcal{F}$ that are available. These policies can come from very different sources; some of them might be imposed on the users by superiors, they can be legacy from other users or other applications, they could have been learned in a case by case fashion by the users (or a system), etc.

As mentioned before, given a relation R and a functional dependency fd, a policy associated with fd is applied to each of the clusters in the set of clusters determined by R and fd. We could say that each tuple in the cluster is analyzed, and the policy decides whether the tuple, as is or a modified version of it, will be included or not in the outcome. Based on this intuition, we present a formal representation of a policy and its outcome using first order logic.

We assume that $ResultSet$ is a predicate symbol of arity 2. Let t be a data term, γ denote a policy, and r denote relation R to which γ is applied; the atom $ResultSet(t, r, \gamma)$ represents the fact that the tuple corresponding to t will be part of the output of applying γ to relation R. An example of a result set atom is $ResultSet(f_{Employee}(john, 70, 20), employee, \gamma_1)$ which states that tuple t_1 is part of the result of applying policy γ_1 to relation *Employee*.

Definition 5 *Given relation R and functional dependency fd, a policy γ associated with fd can be written as a first order formula of the form: $\forall X.[Cluster(X, r, fd) \wedge \phi]$, where ϕ is the formula that describes the tuples that result from the application of the policy to a cluster.*

The following example shows the first order logic encoding of policy γ_1 from Example 1.

Example 5 *Policy γ_1 from Example 1 can be expressed as:*

$$\begin{array}{l} \forall X.[Cluster(X, employee, fd) \; wedge \\ \quad (\forall Z_1, Z_2, Z_3.(InCluster(f_R(Z_1, Z_2, Z_3), X) \rightarrow \\ \qquad (\exists Z_4, M, Z_5.InCluster(f_R(Z_4, M, Z_5), X) \wedge \\ \qquad (\forall Z_6, N, Z_7.InCluster(f_R(Z_6, N, Z_7), X) \wedge M \leq N) \\ \qquad\quad \rightarrow ResultSet(f_R(Z_1, M, Z_3), employee, \gamma_1)))] \end{array}$$

In the following we will denote the set of first order formulas that express the policies for a functional dependency fd with $folPolicies(fd)$.

3.4 Meta-policy Level Knowledge

In this framework we allow users to express constraints and preferences over policies. For instance, suppose two different users are working with the instance of relation *Employee* from Example 1. When more than one record of salary for a given employee exists, each user might need to apply a different policy depending on his application. A bank manager that has a list of employees that are asking for a loan may decide that, for safety, he will approve the loan if and only if the minimum salary on record for an employee is above a certain threshold. This gives rise to a policy that when applied will, for instance, delete for each employee all conflicting tuples except the one that yields the minimum value for salary. On the other hand, a tax inspector analyzing this same relation might need for each employee the average salary unless the variance is greater than a certain threshold, in which case all records should be kept.

These two users have different requirements for the same data; therefore, the process of deciding which (if any) of the available policies should be applied must make use of this information. Given a relation R and a functional dependency fd, we will use $K_{P(R,fd)}$ to denote the set of user requirements as a set of first order formulas.

Example 6 *Consider a setting similar to the one described above, where a user that is a bank manager is required to use policy γ_{bank}, and a tax inspector is required to use either policy γ_{tax1} or γ_{tax2}. The following formula expresses the requirements for tax inspectors.*
$\forall U.[User(U, employee) \wedge TaxInspector(U) \rightarrow$
$Use(U, employee, \gamma_{tax1}) \vee Use(U, employee, \gamma_{tax2})]$
Requirements for bank managers can be defined similarly.

This kind of formulas can be defined for each user and relation. In the following we will refer to the set $K_{P(R,fd)}$ as the set of *meta-policy axioms*. These axioms can also make use of other levels of knowledge. The following example shows how data and meta-data level knowledge can be used.

Example 7 *Consider the following meta-policy axiom for dependency fd : $Name \rightarrow Salary$ and relation* Employee *expressing that if an employee has more than one salary but all tax_bracket records are consistent, then if the user is a junior tax inspector, policy γ_{tax2} cannot be applied.*
$\forall U.[User(U, employee) \wedge TaxInspector(U) \wedge$
$Rank(U, junior) \wedge (\exists C.Cluster(C, employee, fd) \wedge$
$\forall X, Y, Y', Z_1, Z_2.[InCluster(f_{Employee}(X, Y, Z_1), C) \wedge$
$InCluster(f_{Employee}(X, Y', Z_2), C) \rightarrow Z_1 \neq Z_2) \rightarrow$
$\neg Use(U, employee, \gamma_{Tax2})]]$

Preferences about policies can also be defined based on how long they take to compute, how much inconsistency they remove, or various characteristics of the data that result from their application. The following first order formula expresses that for relation *Employee*, user Joe likes policies that remove more than 80% of the inconsistency from R:
$\exists P.[User(joe, employee) \wedge RemoveInc(P, employee, I) \wedge$
$(I > 80) \rightarrow Likes(joe, P)]$

Preferences can be directly specified by the user or learned by the system that implements the decision process.

In order to be able to have preferences that refer to particular properties of policies, we assume the existence of a set of atoms $K_{PProp(fd)}$ that contain properties of the policies in $folPolicies(fd)$. This set contains facts such as $RemoveInc(\gamma_1, employee, 50)$ or $MinTimeReq(\gamma_1, employee, 10000)$, which assert that policy γ_1 removes 50% of the inconsistency, and it requires a minimum of 10,000 seconds when applied to relation *Employee*, respectively. These facts can also be inferred by the system. For instance, having the specification of the policy and/or statistics of the behavior of the policy, it is possible to estimate the minimum time required by the application of the policy over certain sets of tuples.

Such a system can put together users' preferences and policies' properties to make decisions regarding which policies can or should be used in different circumstances. Suppose that the user told the system that he can wait a maximum of 20 minutes and tolerate no more than 20% of inconsistency. The system can use this knowledge to support the application of policies that satisfy these preferences and discard those that do not.

4. Argumentation-Based IMP Selection

In this section we discuss how to use classical logic argumentation to address the problem of selecting the set of policies that are suitable for application given a body of knowledge that encodes the data, the policies, and requirements and preferences over the policies, as described above.

Given a relation R and a functional dependency of interest to the user, the knowledgebase for the argumentation system is a fixed set $\Delta = folPolicies(fd) \cup K_{Data(R,fd)} \cup K_{P(R,fd)}$, which contains the set of formulas that combine the information from all levels described in the previous section. Arguments in this setting will be constructed for or against using particular policies. We use the conventional definition of argument from (Besnard and Hunter 2001): an argument is a pair $\langle \Phi, \alpha \rangle$, where Φ (the support) is a minimal consistent subset of Δ that logically entails α (the claim), which in general is not an element of Δ.

The meta-policy axioms defined in the previous section form a particularly important set of formulas to be used in the construction of arguments for using a particular policy. The following example shows how to build an argument for using a particular policy using meta-policy axioms in the user context.

Definition 6 *An argument $\langle \Phi, \alpha \rangle$ supports the use of a policy γ by user U in relation R iff α is a formula of the form $Use(u, r, \gamma)$. We will call these arguments* policy support arguments*; a counterargument for a policy support argument is a* policy counterargument.

Policy support arguments and policy counterarguments can in general be built from users' requirements and/or preferences. As shown in the previous section, users' requirements and preferences are expressed in meta-policy axioms, which specify which policies satisfy these criteria and which do not. Given a policy support argument, a policy counterargument for it negates either the claim of a policy support argument, or part(s) of its support.

$$\begin{tikzcd}
\langle \{f_1, f_5, f_8, f_{11}\},\ Use(tom,\ employee,\ \gamma_{tax1}) \rangle \\
\langle \{f_1, f_3, f_{10}\},\ \Diamond \rangle \arrow[u] \\
\langle \{f_1, f_3, f_{12}, f_{13}, d_5, d_6, d_7, d_8\},\ \Diamond \rangle \arrow[u]
\end{tikzcd}$$

Figure 1: Argument tree for policy argument $\langle \{f_1, f_5, f_8, f_{11}\}, Use(tom, employee, \gamma_{tax1}) \rangle$

Example 8 *Let δ be the meta-policy axiom:*

$$\forall X.[User(X, employee) \land BankManager(X) \rightarrow Use(X, employee, \gamma_{bank})]$$

Let Φ be the set

$$\{User(joe, employee), BankManager(joe), \delta\}$$

Then $\langle \Phi, Use(joe, employee, \gamma_{bank}) \rangle$ is a policy argument for γ_{bank}. Let α be the formula

$$\forall X.\exists P.[User(X, employee) \land Rank(X, junior) \land Restricted(P) \rightarrow \neg Use(X, employee, P)]$$

The argument $\langle \Psi, \neg Use(joe, employee, \gamma_{bank}) \rangle$ is a policy counterargument for $\langle \Phi, Use(joe, employee, \gamma_{bank}) \rangle$, with

$$\Psi = \{\alpha, User(joe, employee), Rank(joe, junior), Restricted(\gamma_{bank})\}$$

In this example, user Joe *is a junior bank manager. As a bank manager he can use policy γ_{bank} over database* Employee*, but because of his rank, he is not allowed to use restricted policies. Policies in this setting could be restricted, for instance, because they hide a lot of data; junior bank employees might not have enough experience to evaluate the pitfalls of hiding too much of the inconsistency.*

Example 9 *Suppose the user inputs the following facts and preferences to the system:*

(f_1) $User(tom, employee)$

(f_2) $TaxInspector(tom)$

(f_3) $MinTimeRequired(\gamma_{tax1}, employee, 10000)$

(f_4) $RemoveInc(\gamma_{tax2}, employee, 50)$

(f_5) $RemoveInc(\gamma_{tax1}, employee, 80)$

(f_6) $\forall X.[User(X, employee) \land TaxInspector(X) \rightarrow Use(X, employee, \gamma_{tax1})]$

(f_7) $\forall X.[User(X, employee) \land TaxInspector(X) \rightarrow Use(X, employee, \gamma_{tax2})]$

(f_8) $\exists P.[User(tom, employee) \land Likes(tom, employee, P) \rightarrow Use(tom, employee, P)]$

(f_9) $\exists P_1, P_2, D.[\neg Likes(tom, D, P_1) \land Likes(tom, D, P_2) \land Use(tom, D, P_2) \rightarrow \neg Use(tom, D, P_1)]$

(f_{10}) $\exists P, T.[MinTimeRequired(P, employee, T) \land User(tom, employee) \land (T > 6000) \rightarrow \neg Likes(tom, employee, P)]$

(f_{11}) $\exists P.[RemoveInc(P, employee, I) \land (I > 75) \land User(tom, employee) \rightarrow Likes(tom, employee, P)]$

In the following we refer to the formulas presented above by using their labels instead of the formulas themselves.

Consider our running example instance of relation Employee *and fd : $Name \rightarrow Salary$. The data and meta-data level description is composed of $K_{Data(Employee,fd)} = \{d_1, \ldots, d_8\}$ where the d_i's are formulas from Examples 3 and 4. Let f_{12} be an instance of meta-policy axiom from Example 7 for employee* Tom*, f_{13} be $Rank(tom, junior)$, and $\Delta = \{f_1, \ldots, f_{11}, f_{12}, f_{13}\} \cup K_{Data(Employee,fd)}$ be our knowledgebase.*

Figure 1 shows argumentation tree T for policy argument $\langle \{f_1, f_5, f_8, f_{11}\}, Use(tom, employee, \gamma_{tax1}) \rangle$. In order to evaluate whether or not the root claim is warranted, we use the judgement function *presented in (Besnard and Hunter 2001), which is an adaptation to classical logic argumentation of the concept of* dialectical tree marking *(García and Simari 2004). In this example $Judge(T) = Warranted$, since all its attackers are* defeated.

The previous example uses a specific warrant criterion defined for classical logic argumentation. Note that more complex ways of comparing and analyzing arguments can be defined; (Besnard and Hunter 2001) presents alternative judgement functions that consider intrinsic properties of arguments, instead of just the structure of the tree.

Let $\gamma_1, \ldots, \gamma_n$ be the candidate policies w.r.t. a functional dependency fd that can be applied to relation R by user U, and let $\Delta = folPolicies(fd) \cup K_{Data(r,fd)} \cup K_{P(r,fd)}$ be the corresponding knowledgebase. Given a finite knowledgebase Δ and a formula α, then, as shown in (Besnard and Hunter 2008), there is only a finite set of complete argument trees that can be built from Δ with the root being an argument whose claim is α, and each of these trees have finite branching and depth. Based on this result, for each γ_i the argument structure $\langle \mathcal{P}, \mathcal{C} \rangle$ for $\alpha_i = Use(u, r, \gamma_i)$ can be constructed, where $\mathcal{P}$ is the set of complete argument trees for α_i and $\mathcal{C}$ is the set of argument trees for $\neg\alpha_i$. These policy arguments and argument trees can be computed using a series of algorithms presented in (Besnard and Hunter 2008). The basic idea is to compile the knowledgebase Δ based on the set of minimal inconsistent subsets of Δ, and then generate arguments from this compilation. Since the cost of computing a compilation of Δ can be sometimes prohibitive, (Besnard and Hunter 2008) develops techniques that can be used to build approximate arguments.

Once we have the argument structures for all candidate policies, we have to decide whether or not each policy is suitable for application, given the knowledge contained in Δ. We want to characterize policies whose application is *supported* by Δ; we define support based on *cautiously* and *boldly* warranted policies.

Definition 7 (Cautiously Warranted Policy) *A policy γ is said to be* cautiously warranted *for user U and relation R, from the argument structure $\langle \mathcal{P}, \mathcal{C} \rangle$ for $Use(u, r, \gamma)$ iff $\exists\, T \in \mathcal{P}$ s.t. $Judge(T) = Warranted$, and either $\mathcal{C} = \emptyset$ or $\forall\, T' \in \mathcal{C}$ we have $Judge(T') = Unwarranted$.*

Definition 8 (Boldly Warranted Policy) *A policy γ is said to be* boldly warranted *for user U and relation R, from the* *argument structure* $\langle \mathcal{P}, \mathcal{C} \rangle$ *for* $Use(u, r, \gamma)$ *iff* $\exists T \in \mathcal{P}$ *s.t.* $Judge(T) = Warranted$.

An automated system based on these concepts can rank the recommendations for policy selection based on how they are warranted (cautiously first, then boldly); for boldly warranted policies, the ranking can be based on the number of argument trees in $\mathcal{P}$ that are marked as warranted, or some combination between that number and the number of marked warranted trees in $\mathcal{C}$. More complex rankings could make use not only of the structure of trees but also of more sophisticated ways of comparing arguments. If additional information is available, for instance a preference or reliability order among formulas in Δ, the system can make use of this information to rank the policies.

Finally, in the case that after this process no policy is warranted, the user will have to intervene in the end decision. The system can aid in this manual process by providing detailed explanations of the individual decisions, *e.g.*, showing the different argument trees built for a policy argument, the dialectical marking process step by step, etc.

5. Ongoing and Future Work

In this work we have started to explore the incorporation of classical logic argumentation as a solution for the problem of determining a suitable set of inconsistency management policies that may be applied to a particular relation. As part of ongoing work in this direction, we are currently studying properties that should be imposed on this framework in order for the resulting system to be well-behaved from the point of view of the user. The next logical step is then to implement the argument-based policy selection process as part of a system that is currently under development for the use of IMPs, using existing implementations of classical logic argumentation systems (Efstathiou and Hunter 2008; 2009). Furthermore, we will also investigate the viability of transforming our framework into a defeasible logic formalization, and thereby take advantage of defeasible logic implementations such as those for ABA (Gaertner and Toni 2007) and DeLP (García and Simari 2004).

The meta-policy level arguments deal, in the most part, with user preferences. As future steps we would also like to investigate how related work in priority and preference management in abstract argumentation (Amgoud and Cayrol 2002; Modgil 2006) might be applied in this setting.

Independently of how a set of policies has been selected for application, once this set is determined it is necessary to decide how to compute the set of tuples that compose the output of the application of the policies. This problem has to be considered for two different cases: the single functional dependency case and the multi-dependency case. For the multi-dependency case, (Martinez et al. 2008) proposes a sequential application of the different policies and two semantics for the output, based on the assumption of the existence of a partial order among the functional dependencies. The same strategy can be applied for the single functional dependency case if it is possible and sensible to specify an order among the selected policies. A different semantics of application of policies in both cases is to regard the set of selected policies as one composite complex policy (a *super policy*). Inevitably, conflicts will arise regarding what tuples are part of the result of applying the super policy; certain policies will require some tuples to be deleted or modified, while others might require to leave the same tuples unmodified. A mechanism is then necessary for deciding which tuples will be part of the output of a super policy. Once again, it seems that an argumentative approach may be useful in solving this problem. Arguments for and against deleting, modifying, and keeping tuples can be constructed automatically depending on the policies' specifications, and the composition of the output will then be determined by means of a dialectical process. Part of the ongoing work in this project is the formalization of such a mechanism.

6. Acknowledgments

The first author of this paper was funded in part by AFOSR grant FA95500610405, ARO grant W911NF0910206 and ONR grant N000140910685.

References

Amgoud, L., and Cayrol, C. 2002. Inferring from inconsistency in preference-based argumentation frameworks. *Journal of Automated Reasoning* 29(2):125–169.

Besnard, P., and Hunter, A. 2001. A logic-based theory of deductive arguments. *Artif. Intell.* 128(1-2):203–235.

Besnard, P., and Hunter, A. 2008. *Elements of Argumentation*. The MIT Press.

Efstathiou, V., and Hunter, A. 2008. Algorithms for effective argumentation in classical propositional logic: A connection graph approach. In *FoIKS*, 272–290.

Efstathiou, V., and Hunter, A. 2009. An algorithm for generating arguments in classical predicate logic. In *ECSQARU*, 119–130.

Gaertner, D., and Toni, F. 2007. Computing arguments and attacks in assumption-based argumentation. *IEEE Intelligent Systems* 22(6):24–33.

García, A. J., and Simari, G. R. 2004. Defeasible logic programming: an argumentative approach. *Theory and Practice Logic Programming* 4(2):95–138.

Grant, J., and Hunter, A. 2006. Measuring inconsistency in knowledgebases. *J. Intell. Inf. Syst.* 27(2):159–184.

Hunter, A., and Konieczny, S. 2005. Approaches to measuring inconsistent information. In *Inconsistency Tolerance*, 191–236.

Lozinskii, E. L. 1994. Resolving contradictions: A plausible semantics for inconsistent systems. *Journal of Automated Reasoning* 12(1):1–31.

Martinez, M. V.; Pugliese, A.; Simari, G. I.; Subrahmanian, V. S.; and Prade, H. 2007. How dirty is your relational database? An axiomatic approach. In *ECSQARU 2007*, 103–114.

Martinez, M. V.; Parisi, F.; Pugliese, A.; Simari, G. I.; and Subrahmanian, V. 2008. Inconsistency management policies. In *KR 2008*, 367–376.

Modgil, S. 2006. Hierarchical argumentation. In *JELIA 2006*, 319–332.