ELSEVIER

Decision Support Systems 16 (1996) 21–38

Decision Support Systems

# A decision support system for bureaucratic policy administration: An abductive logic programming approach

KayLiang Ong [a,*], Ronald M. Lee [b]

[a] Microelectronics and Computer Technology Corporation (MCC) 3500, West Balcones Center Drive, Austin, TX 78759-5398, USA

[b] Erasmus University Research Institute for Decision And Information Systems (EURIDIS), Burg. Oudlaan 50, 3062 PA Rotterdam, The Netherlands

**Abstract**

A bureaucracy can be viewed as a set of policies that governs the activities of its people. The purpose of these policies is to improve operational effectiveness and efficiency. However, manual administration of these policies is a tedious and often overwhelming task because it is too cognitively demanding to keep track of the complex relationships between the policies. As a result, these policies often consist of many inconsistencies (conflicts) as they evolve because there is no automated means to aid the administrators in detecting inconsistencies. In this paper, we present an approach that uses abductive logic programming for building a decision support system for the administration of bureaucratic policies. The system will help administrators decide the consistency of a policy with respect to the current set of policies and hence, prevent the introduction of inconsistent policies.

*Keywords:* Decision support systems; Bureaucracy; Policies; Logic programming; Abduction; Inconsistency; Deontic logic

## 1. Introduction

In a bureaucratic system, explicit policies are used to provide straight-forward guidelines of how the organization should operate to achieve greater efficiency and effectiveness. In order to cater for changing requirements and to balance simplicity, efficiency and effectiveness, policies are modified gradually by introducing new ones or removing old ones. Unfortunately, contrary to its intention, this process often introduces even more inconsistencies and thus, further degrades rather than improves its efficiency and effectiveness. Example of inconsistencies in policies are situations of dilemmas where an employee is both obligated and forbidden to perform a certain duty.

As we will elaborate further in the next section, the reason is because the process of identifying inconsistency manually, given even a small set of policies, is too cognitively demanding. In fact, as Lee [18] observed, bureaucracies are "sticky upward". That is, they tend to grow more easily than they shrink. Hence, the set of policies as well as the number of inconsistencies tend to grow.

In this paper, we will discuss an approach that facilitates the building of a decision support system for administrating bureaucratic policies using

* Corresponding author. E-mail: kayliang@mcc.com

0167-9236/96/$15.00 © 1996 Elsevier Science B.V. All rights reserved
SSDI 0167-9236(94)00054-9

logic programming technology. The purpose of such a decision support system is to help administrators decide, in an automated fashion, whether a particular new policy or old policy should be included into or excluded from the policy set respectively. In general, whenever an inconsistency is detected, the administrator will decide how to resolve the inconsistency. Explanation will be provided as a form of feedback to assist the administrator in the resolution process.

We will also address the more difficult of problem of detecting potential inconsistencies. Since logic programming system can only reason based on the given set of facts, the detection of inconsistencies is limited to the current scenario (A *scenario* is defined as the set of facts known or assumed at that moment in time). In this paper, we will discuss how logic programming can be extended with abductive reasoning and show how potential inconsistencies can be detected in some simulated future scenarios.

### 1.1. Information technology for policy administration

Meyer [20] discovers in his extensive empirical studies on the *Limits of Bureaucratic Growth* that bureaucracy tends to grow because of the "inability ... either to replace existing organizations or to reorganize and simplify them in bureaucratic systems." Similarly, policy makers are often overwhelmed by the complexity of the interrelationships between policies. They can not even detect the inconsistencies in the policies, let alone the even more difficult task of replacing and simplifying them. As new policy is added or an old policy is deleted, it is extremely difficult for one to foresee the potential impact and side effects. Therefore, any attempt to 'fix' the bureaucracy may actually do more damage by creating more inconsistencies in the policies. The solution, as Meyer [20] puts it, is to find "concepts of administration that permit ... replacement or fundamental reorganization of existing units in complex systems that otherwise tend toward persistence and growth." However, he warns that the idea of adding routine administration executed by humans will only worsen the situation.

The availability of information technology presents a new alternative to implement Meyer's solution. Certain well-structured but complex tasks such as detection of inconsistencies in the policies can indeed be automated. Imagine a policy administrator who intends to amend the existing policies by adding a new policy in view of new requirements. Three questions arise:

(a) Will this new policy conflict or be inconsistent with any of the existing policies?
(b) If so, what are the existing policies that are in conflict with this new policy?
(c) Why and how are they in conflict?

Given even a small set of policies as we will show in our example later, answering these questions is often non-trivial. Our goal is to develop a decision support system to aid the policy administrator in answering the above three questions in an acceptable time-frame. Such goal can be achieved by using logic programming technology to automate some of these mechanisms.

### 1.2. Relevant works and our approach

Sergot, Sadri, Kowalski, Kriwaczek, Hammond and Cory [33] demonstrate that logic formalism can indeed be expressive enough to model legislation (the British Nationality Act). Furthermore, they point out that "representation in logical form helps to identify and eliminate unintended ambiguity and imprecision." In addition, they suggest that "the rules and regulations that govern the management of institutions and organizations have exactly the same character as legal provisions". Even though they feel that a logic representation "can also help to derive logical consequences of the rules and therefore test them before they are put into force," they do not discuss how it can be done.

Rozenshtein and Minsky [30] develop a Prolog-based approach that provides a scheme for managing how a database system is accessed such that it is capable of "self-control". However, they do not address the problem of inconsistency among rules. Thus, their proposed scheme does not prevent the introduction of inconsistent rules.

Lee [15-19] is among the first to propose the use of formal logic and artificial intelligence in

modeling bureaucracies. Since bureaucracy serves the purpose of "officiating in the sense of issuing directives, granting permissions, enforcing prohibitions, waiving obligations and so forth," Lee [18] thus views bureaucracy as a deontic system for social and organizational control. Rules and procedures can be viewed as the "software" of the bureaucratic system used to control its operations. The use of deontic concepts, such as obligations, permissions and so forth, provides the additional expressiveness to model bureaucratic policies. This paper represents a further extension to address the problem of consistency checking of policies as they evolve.

Ryu [31] proposes the use of defeasible reasoning to resolve inconsistent conclusions derived from bureaucratic policies. This "repair" approach compromises on the fact that the set of policies are potentially inconsistent. Thus, as these policies are being applied in a certain situation, inconsistent conclusions may be derived. To resolve the inconsistency, various resolution strategies are introduced to decide the priority of some policies over another. Hence, conclusions based on those rules with higher priority are selected. On the other hand, our research focuses on a "preventive" approach such that potential inconsistencies in policies are removed before being used. Whenever a new policy is added to the bureaucracy, it is tested against the existing set of policies to determine if it will cause any inconsistencies.

Our approach is based on *horn clause logic* and *deontic logic*. Once the policies are formalized as logical rules (i.e. horn clauses) and integrity constraints (i.e. denials or negated conjunctions), it becomes feasible to perform automated reasoning using the existing logic programming systems. As a result, inconsistency of policies can be deduced using theorem proving techniques such as *resolution* [13,29]. Explanations are also provided as a form of feedback. Unfortunately, traditional logical inference procedure will only deduce inconsistencies with respect to the current set of facts, i.e. the current scenario. This is inadequate since the policies should also be consistent when applied to some likely future scenarios. To facilitate the exploration of potential inconsistencies, we develop a procedure based on abduction that extends the inference mechanism to simulate future scenarios.

In Section 2, we introduce the basic framework for the decision support system for policy administration. In Section 3, we define the logic model and in Section 4, we will demonstrate how inconsistency in policies can be detected and explained. Section 5 extends the logic model to include abductive reasoning in order to detect potential inconsistencies. Lastly, we conclude and discuss some future research directions in Section 6.

## 2. A decision support system for policy administration

In this section, we will describe what we mean by a decision support system for policy administration. We will first give an user view of the system and the set of functionalities available. Then, we describe the process how it is used and finally, the architecture of the system in terms of its components.

When an administrator interfaces with the decision support system, the following five functionalities are provided:

(a) **insert** Insert a new policy into the existing policies.
(b) **delete** Delete an existing policy.
(c) **verify** Verify the existing policies by checking if there is any inconsistency based on the current known facts.
(d) **explain** Explain why an inconsistency occurs. This will show all the relevant policies and how their relationships lead to the inconsistency.
(e) **explore** Explore for potential inconsistency by generating simulated scenarios in a systematic manner. The explain facility in (d) can be used to provide justification and all the assumed facts will be provided as part of the explanation.

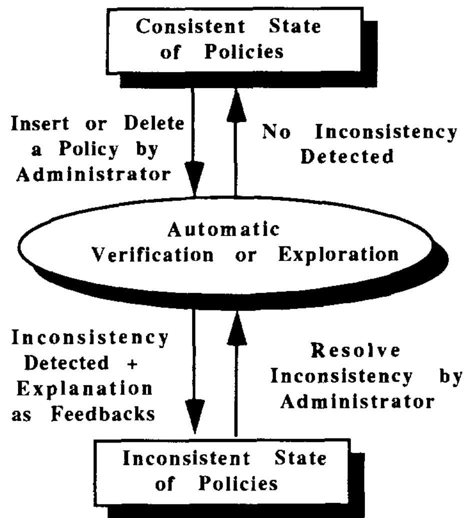



Fig. 1. The update process

Given these operations, the policy administrator has the ability to find out if any inconsistency has been introduced whenever a policy is inserted or deleted. This update process is iterative in nature such that if a violation is detected, the control is returned to the administrator to resolve the inconsistency. It is important to note that the decision support system we propose does not provide automatic resolution of inconsistency. The task, which is highly context-dependent and will require domain-specific knowledge, will be left to the administrator. The goal of the system is simply to aid the administrator in finding some of the potential problems that could be introduced when the existing policies are modified. The update process is shown in Fig. 1. The process will continue until a consistent state of the existing polices is reached.

The overall architecture of the decision support system is shown in Fig. 2. The update facility will determine the list of necessary checks required given the updates. The verification facility checks the consistency of the existing policies while the explanation facility traces the detection process and extract the relevant policies as explanation. The exploration facility extends the verification facility with the abductive procedure. Briefly speaking, the abductive procedure allows assumption of facts in the verification process. New facts can only be assumed on a need basis and they must be consistent with the existing policies, facts and other assumed facts.

The fact interface manages the retrieval of facts necessary for the verification process. It is important to note that the state of the existing policies may become inconsistent when facts are updated. In such case, we assume that the existing policies always take precedence over the facts and it is the responsibility of the data management system to flag for violation as the update of facts violates the existing policies.

## 3. Logic modeling of bureaucracy

In modeling bureaucratic policies, formal logic was chosen over other representational schemes because it has a sound and rigorous theoretical framework. Furthermore, the derivation of new facts from old facts can be mechanized by theorem-proving techniques. The most important work was done by Robinson [29] for the discovery of the *resolution* inference rule that lays the basic ground work for automated inference. This also leads to the creation of the first logic programming language, *Prolog* [6] and the proposal of using logic as a programming language [14]. For further details on logic programming and theorem-proving, the reader is referred to [1,4]. We will assume the basic knowledge of predicate logic and logic programming in general for the rest of the discussion.

In this section, we will describe the logic model for bureaucracy and how they are represented in horn clause logic.

### 3.1. The logic model

We define the logical framework for modeling bureaucracy as follows:

#### 3.1.1. Definition 3.1

A logic model for bureaucracy is a triple:

$< \mathbf{BF}, \mathbf{BR}, \mathbf{BIC} >$,

where

(a) **BF** is a set of ground formulae representing bureaucratic facts.

(b) **BR** is a set of deductive rules of the form:

$H \leftarrow B1 \; \& \; \ldots \; \& \; Bn$ where $n \geq 0$,

where H, B1, ..., Bn are atomic formulae and H does not have any predicate symbol in **BF**.

(c) **BIC** is a set of integrity constraints of the form:

$false \leftarrow B1 \; \& \; \ldots \; \& \; Bn$ where $n \geq 0$,

where B1, ..., Bn are atomic formulae.

(d) The logic model is **consistent** iff all integrity constraints are satisfied as follows:

$\forall c \in \mathbf{BIC}, \mathbf{BF} \cup \mathbf{BR} \models c$,

where $A \models B$ means A entails B.

Thus, condition (d) of the logic model specifies that all integrity constraints in **BIC** must be satisfied in the evolution of the logic model. Whenever there is an update, we have to ensure that condition (d) is satisfied. Otherwise, the logic model is inconsistent. Each of these components will be further elaborated in the following sections.

### *3.2. Bureaucracy as a deontic system*

*Deontic Logic* provides a logical system for analyzing moral and practical reasoning (the word *deontic* originated from a Greek word that means 'duly' or 'as it should be'). It is also referred to as *logic of obligation* and *logic of norms* [11]. Deontic meanings are highly relevant in the context of bureaucratic modeling because bureaucratic policies serve the purpose of imposing deontic constraints on actions to be performed by agents within the organization. Thus, these policies often contain deontic meanings using words such as "may", "must", "have to", "could", "required", etc. in their descriptions. In general, policies of a bureaucracy can be viewed as the instrument for determining the *deontic status* of all the governed individuals with respect to the kind of actions they are to perform.

Much of the work on deontic logic was stimulated directly or indirectly by von Wright [38]. For a complete illustration for the standard system for deontic logic, the reader is referred to [11]. Latest extension of deontic logic can be found in [36]. Lee [18] was among the first to apply deontic logic for computational modeling of bureaucratic policies.

Our approach of formalizing bureaucratic concepts in policies is based on the first axiomatization of deontic logic by von Wright [37]. Deontic logic is considered as a variant of modal logic and thus, only those axiomatizations suitable for the first-order horn clause logic paradigm are incorporated. The four deontic predicates that specify the deontic status impose on an action to be carried out by an agent are defined as follows:

**obligated Agent: Action**

**permitted Agent: Action**

**forbidden Agent: Action**

**waived Agent: Action**

Generally speaking, obligations and prohibitions (i.e. forbidden actions) are a form of duty while permissions and waivers are a form of discretion. An action that is not covered by one of these deontic predicates means there are no explicit rules to govern it and will require human intervention. The four predicates are logically related with the following axioms:

**permitted Agent: Action**

$\leftrightarrow$ **~ forbidden Agent: Action**

**obligated Agent: Action**

$\leftrightarrow$ **~ waived Agent: Action**

which can be read as: an agent is permitted to perform an action if and only if he is not forbidden to do so and an agent is obligated to perform an action if and only if he is not waived from doing so. We only model atomic actions here because it is not clear how complex actions should be handled in a logic framework and whether it can be implemented efficiently. We will deal with this issue in future research.

Since one deontic predicate is a logical negation of the other, we follow a simple and elegant approach proposed by Gelfond and Lifschitz [12] that supports both explicit negation as well as negation-by-failure. Here, we treat the negation

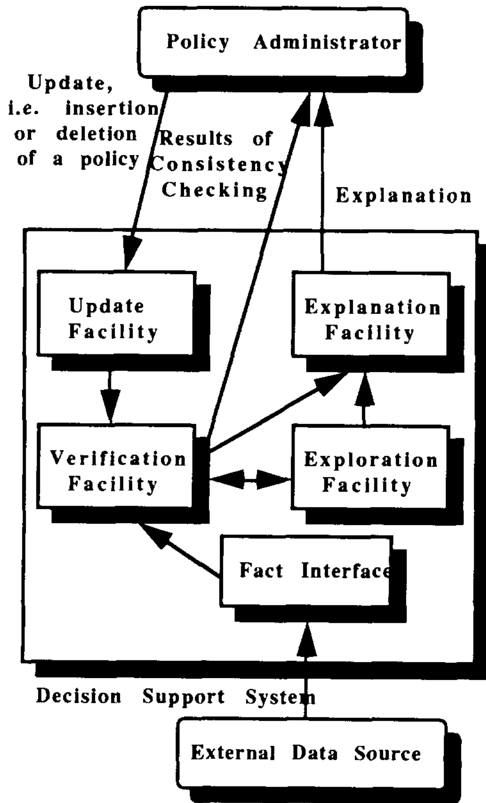



Fig. 2. Overall architecture of the decision support system

of the deontic predicates specially as explicit negation (Negation on other predicates is still treated as negation-by-failure). The predicates forbidden/1 and waived/1 are simply explicit negation of permitted/1 and obligated/1 respectively and vice versa. All negated form of these deontic predicates are rewritten into its corresponding form, i.e. ~waived(Agent, Action) are always rewritten into obligated(Agent, Action). In other words, the goal ~waived(Agent, Action) can only be satisfied if obligated(Agent, Action) is explicitly derived.

By allowing both the positive and negative form of deontic predicates, we have allowed the specification of logically inconsistent rules such as the following:

waived(Agent,Action) $\leftarrow$ ...

obligated(Agent,Action) $\leftarrow$ ...

Hence, additional integrity constraints are required to enforce the consistency and they will be discussed later.

### 3.3. Modeling bureaucratic facts

Bureaucratic facts are simply statements saying what is currently true in the bureaucratic system. These facts could be stored in a relational database system as records. Facts can also be information on documents or forms. In general, we assume these facts are somehow stored electronically and are accessible by the information system. As shown in Fig. 2, we assume there is a fact interface that will retrieve facts from all relevant external data sources and converted them into a relational or predicate format.

In general, bureaucratic facts are directly represented as logical facts, i.e. as a set of ground formulae. The fact that John is a teaching assistant is represented logically as follows:

teaching_assistant('John').

Later, we will introduce a framework that allow facts to be assumed in the inference process. This plays a vital role in the explore capability.

### 3.4. Modeling bureaucratic policies

Bureaucratic policies are regulations that determine certain outcomes based on some preconditions. In general, we can represent policies as logical implications or deductive rules shown in a general form as follows:

**conclusion $\leftarrow$ pre − conditions.**

which can be read as: if **pre-conditions** are true then we can accept that **conclusion** is true. A common class of bureaucratic policies are those that imposes deontic constraints on the action to be carried out by an agent. Such policies can be modeled as logical rules shown earlier with deontic status as its conclusion shown below:

**obligated(Agent : Action) $\leftarrow$ pre-conditions1.**
**permitted(Agent : Action) $\leftarrow$ pre-conditions2.**
**forbidden(Agent : Action) $\leftarrow$ pre-conditions3.**
**waived(Agent : Action) $\leftarrow$ pre-conditions4.**

which can be read as: if **pre-conditions** are true then we can conclude that an **Agent** is imposed with a new deontic status of **obligated**, **permitted**, **forbidden** or **waived** with respect to a particular **Action**. An example would be the following policy:

$$\text{obligated}(\text{university}:\text{insured}(X)) \leftarrow \text{employee}(X)$$

that imposes an obligation on the university to insure all its employees. An example of a policy that does not impose deontic constraint is as follow:

$$\text{student}(X) \leftarrow \text{teaching_assistant}(X).$$

that specifies that all teaching assistants must be students.

### 3.5. Modeling integrity constraints

Integrity constraints are meta-level logical statement about the possible allowable states [26]. Following the approach in [32], an integrity constraint statement is represented as a *denial*, i.e. a negated conjunction that specifies the conditions of what is not allowed and has the following general form:

$$\textbf{false} \leftarrow \textbf{pre}-\textbf{conditions}.$$

An example of an integrity constraint is a deontic dilemma such that an agent is both obligated to carry out an action and yet waived from doing so. It is specified as follows:

$$\text{false} \leftarrow \text{obligated}(\text{Agent}:\text{Action}) \ \&\ \text{waived}(\text{Agent}:\text{Action}).$$

Thus, if we can prove the existence of such situation defined as the conjunctions on the right-hand side of the implications with respect to the set of policies, then the policies are indeed inconsistent. In general, integrity constraints can be classified into different categories and they are listed below with a corresponding example:

(a) **logical**. This is the basic consistency requirement of the logical system. They should not be violated. e.g. ∀ predicate symbol p with arity n used,
false ← p(A1, ..., An) and ~ p(A1, ..., An).

(b) **natural**. These are natural laws and should not be violated.
e.g. false ← parent(X, X).

(c) **deontic**. These are rules devised by humans and can be violated.
e.g. false ← permitted(X: raise_salary(X)).

(d) **empirical**. These are generalized rules that try to capture most cases in the real world. They can be violated in exceptional cases.
e.g. false ← age(Person, Age) and Age > 150.

(f) **Implementational**. These are rules due to implementational requirements as in the case of database modeling where a certain field has to be of a certain type. Violations normally lead to the rejection by the system.
e.g. false ← age(Person, Age) and ~ integer_type(Age).

An important class of domain independent integrity constraints that are of particular interest to us are those integrity constraints that prevents *deontic dilemma*. These integrity constraints deal with situations of normative inconsistencies that are very relevant and universally applicable in all bureaucracies. They are as follows:

(a) false ← obligated(Agent : Action) and forbidden(Agent : Action)
It states that one should not be both obligated and yet forbidden to perform an action at the same time. Doing either or neither will not satisfy the normative constraints imposed on the agent. As violation is eminent, sanctions are unavoidable.

(b) false ← permitted(Agent : Action) and forbidden(Agent : Action).
It states that one should not be both permitted and yet forbidden to perform an action at the same time. Only no action can avoid sanctions. This also serves as the integrity constraint to prevent logical inconsistency because permitted/1 is an explicit negation of forbidden/1.

(c) false ← obligated(Agent : Action) and waived(Agent : Action).
It states that one should not be both obligated to perform and yet waived from performing an action at the same time. Only by performing the action can one avoid sanctions. This also serves as the integrity con-

straint to prevent logical inconsistency because obligated/1 is an explicit negation of waived/1.

(d) false ← obligated(Agent : Action1) and obligated(Agent : Action2) and exclusive(Action1,Action2).

It states that one should not be obligated to perform two actions that are exclusive. This means one is obligated to perform the impossible. The predicate exclusive/2 is special and specifies the exclusive relationship between two actions. For example, walking and sitting are exclusive.

### 3.6. Example: A logic model

We present an example for illustrating a logic model based on a sample set of policies. Suppose that we have a set of policies that deals with insurance coverage by the university. The policies are stated as follows:

The university is obligated to pay insurance for all its employees.
A teaching assistant is a full-time student.
A teaching assistant is an employee.
The University is waived from paying insurance for its students.

Consider the following domain-independent integrity constraint:

One should not be obligated and waived from performing the same action.

And the following fact is true:

John is a teaching assistant.

A complete logic model for these policies, facts and integrity constraints is shown as follows:

**R1**: obligated(university : insured(X)) ← employee(X).
**R2**: student(X) ← teaching_assistant(X).
**R3**: employee(X) ← teaching_assistant(X).
**R4**: waived(university : insured(X)) ← student(X).
**IC1**: false ← obligated(Agent : Action) and waived(Agent : Action).
**F1**: teaching_assistant('John').

where {**R1**, **R2**, **R3**} ∈ **BR**, {**IC1**} ∈ **BIC** and {**F1**} ∈ **BF**.

## 4. Verify, update and explain

In this section, we shall focus on the three basic facilities provided by the decision support system, i.e. the verification facility, the update facility and the explanation facility.

### 4.1. The verification facility

The verification facility performs consistency checking of the policies. Based on the definition of part (d) of the logic model defined in Definition 3.1, which we reproduce as follows:

(d) The logic model is **consistent** iff all integrity constraints are satisfied as follows:

$\forall c \in \mathbf{BIC}, \mathbf{BF} \cup \mathbf{BR} \models c,$

where $A \models B$ means A entails B.

In other words, we check for consistency if we can prove that all the integrity constraints are true with respect to the set of rules **BR** and facts **BF**, i.e. the current scenario. The set of current facts be retrieved from the external data source through the fact interface. As described earlier, all integrity constraints are represented as denials in the following form:

false ← B1 & ... & Bn, for $n > 0$.

To ensure the consistency of policies, we need to prove that all the integrity constraint statements are true. Conversely, to detect for any inconsistency, we need to prove that some of these integrity constraints are false. In other words, given an integrity constraint of the form shown above, if B1 and ... and Bn is proven to be true, we have detected an inconsistency. This can be achieved by applying *SLDNF-resolution* used in traditional logic programming system by treating B1 and ... and Bn as a goal clause. The inference procedure is well known and the reader is referred to [1] for further details.

Given a set of policies, once they are formalized into their corresponding logical representa-

tion as rules, facts and integrity constraints, the basic inference mechanism for detection is already made available by the logic programming system. This is precisely one of the main motivations why *horn clause logic* is selected for modeling the policies because the inference mechanism has already been developed.

#### 4.1.1. Example: consistency checking

To illustrate the consistency checking process, consider the university insurance example shown earlier and suppose that a new policy shown below is added:

**R4**: *waived(university : insured(X))*
$\leftarrow$ *student(X)*,

which says that: The university is waived from paying insurance for its students. The administrator will have a demanding task to figure out whether his update with the new policy can lead to any inconsistent side-effects of violating any integrity constraint. In this case, inconsistent conclusions can indeed be derived because based on these policies, we may have difficulty deciding whether the university is obligated to or waived from insuring John. Such inconsistency can be detected automatically using horn clause resolution with the integrity constraint **IC1** as a goal. Certainly, we can not reject the fact that John is a teaching assistant because it is the truth. In this case, the actual problem lies in the policies. Thus, either the new policy has to be rejected or some existing policies should be modified.

In practice, one will have to apply the implicit reasoning that the new policy is an exception to the first policy. Thus, the right conclusion is that the university is obligated to insure John. In other words, the intended meaning by the administrator is shown as follows:

**R1**: obligated(university : insured(X))
$\leftarrow$ employee(X).
**R2**: student(X) $\leftarrow$ teaching_assistant(X).
**R3**: employee(X) $\leftarrow$ teaching_assistant(X).
**R4′**: waived(university : insured(X))
$\leftarrow$ student(X) and ~ employee(X).
**IC1**: false $\leftarrow$ obligated(Agent : Action) and waived (Agent : Action).
**F1**: teaching_assistant('John').

Note that the new policy **R4** has been modified as the policy **R4′**. It has been rewritten with an additional goal such that the university is waived only if the student is not an employee.

### 4.2. The update facility

As defined in the logic model in Section 3.1, we have facts, deductive rules and integrity constraints. Any modification of any one of them may lead to new inconsistency. Hence, given any update, we have to verify that all the integrity constraints are satisfied. Such process can be very expensive computationally. Fortunately, we can reduce the number of integrity constraints evaluated based on the updated entity.

When the logic model is first established, we have to evaluate all the integrity constraints. After that, the consistency checking process is carried out in a restricted fashion as the logic model is updated *incrementally* such that only those integrity constraints affected by the update re-evaluation. Which integrity constraints are re-evaluated depends on the what is being updated and we refer to those integrity constraints that are affected by the update as the *relevant integrity constraints*.

Before we deal with the different cases of update, we need to define the notion of *dependency* of predicate symbols. Given a deductive rule of the form:

$$H \leftarrow B1 \& \ldots \& Bn \quad \text{where } n \geq 0.$$

The predicate symbol of H is *dependent* on the predicate symbol of B1, B2, ... and Bn. Furthermore, the dependency relationship is transitive, i.e. if the predicate symbol P1 is dependent on the predicate symbol P2 and the predicate symbol P2 is dependent on the predicate symbol P3, then P1 is dependent on P3.

Let us denote the set of predicate symbols used in an integrity constraints as **PS**. Based on the notion of dependency, we need to define the set of relevant integrity constraints whenever a deductive rule is inserted or deleted. An integrity constraint is relevant if any of the predicate symbols in its **PS** is dependent on the predicate

symbol of H, the head of the deductive rule to be inserted or deleted.

When an integrity constraint is inserted, it is the only relevant integrity constraint. When an integrity constraint is deleted, the set of relevant integrity constraints is an empty set.

When a fact F is inserted or deleted, an integrity constraint is relevant if any of the predicate symbol in its **PS** is dependent on the predicate symbol of F. However, as we have emphasized earlier, it is not the responsibility of the decision support system to ensure consistency when facts are being updated because it will be tremendously inefficient. We assume that the data management system will incorporate the set of policies and integrity constraints and use them to enforce the integrity of data. In other words, we assume the policies will have a higher precedence in terms of correctness whenever there is a conflict. As the result, the fact is always rejected when it violates the policies. A future extension of this work is to use the current deductive database technology for this purpose. In addition, several techniques for efficient evaluation of integrity constraints have been proposed [8,32] but it is not within the scope of this paper to discuss them.

### *4.3. The explanation facility*

The detection mechanism helps the administrator overcome the bottleneck in figuring out the correctness of this update with respect to the other policies. However, unless some feedback are provided to the administrator, it will not be useful to him if he does not understand why his update is rejected. In general, the following are the kinds of feedback the decision support system should provide:

(a) The set of integrity constraints that are violated.
(b) The scenario, i.e. set of facts and the policies that violate the integrity constraints.

The identification of (a) is always possible because when we have detected an inconsistency in the policies, we must have started our checking process using the violated integrity constraint as the top clause in our proof process. (b) is essentially an explanation as to why and how the application of some policies based on the current known facts leads to the violations of the integrity constraints. This explanation is normally provided as the logical *proof tree* that prove the integrity constraint to be false. Thus, in the example for the university insurance policies, the graphical presentation of the proof tree is shown in Fig. 3.

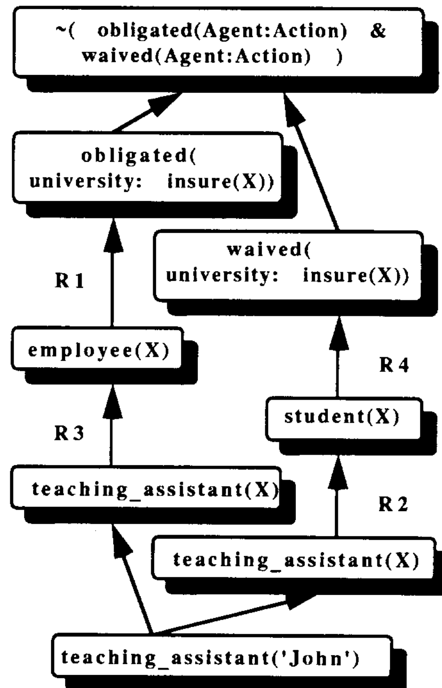


Fig. 3. Proof tree as explanation

By putting this proof tree in a more descriptive or graphical form, the administrator can be provided with some useful feedback as to why such inconsistency occurs and realize which and how the existing policies are affected by the updated policy. Through this feedback, the administrator will be able to narrow down the problems that lead to the inconsistency and proceed to fix the problem by rejecting or revising some of the policies.

## 5. The exploration facility: An abductive approach

So far, we have discussed the basic facilities for the detection of inconsistencies based on the

current set of known facts, which we also refer to as the *current scenario*. If no inconsistency is found, it only means that these policies are consistent at the moment with respect to the current scenario. Furthermore, not all the needed facts are available when the policies are being tested for consistency. Consider the same university insurance example such that we have no information about John being a teaching assistant. The logic model is as shown as follows:

**R1**: obligated(university : insured(X)) ← employee(X).
**R2**: student(X) ← teaching_assistant(X).
**R3**: employee(X) ← teaching_assistant(X).
**R4**: waived(university : insured(X)) ← student(X).
**IC1**: false ← obligated(Agent : Action) and waived(Agent : Action).

Without the fact that someone is a teaching assistant, these policies are indeed logically consistent if we apply horn clause resolution discussed in Section 4. However, based on human judgement, they would be considered potentially inconsistent because if this set of policies is applied to any situation where there exists someone is a teaching assistant (it does not have to be John), there will be a deontic dilemma as the integrity constraint **IC1** will be violated.

The detection of such potential inconsistencies is only feasible if we have the capability to explore future scenarios. In other words, we need to extend our logical reasoning process to systematically infer new assumptions. Thus, for the above example, we need to be able to assume that there exists someone who is a teaching assistant shown below:

$\exists_X$ teaching_assistant(X).

By making this assumption, we have generated a future scenario and the same detection mechanisms discussed in Section 4 can be applied to conclude that these policies are potentially inconsistent under such future scenario. Such simulation can indeed be achieved through *abductive reasoning*.

In the next few sections, we will discuss several extensions to logic programming in order to incorporate abductive reasoning. More importantly, we present a *greedy* approach that attempts to generate or abduct the required assumptions to detect for inconsistencies. By a "greedy" approach, we mean the desire to make as many assumptions as possible as long as the assumptions made are:
(a) relevant, i.e. they are required and useful in the inference process to prove the goal.
(b) consistent, i.e. they do not conflict with the other assumptions made and the existing rules and facts.

We will cover the abductive extensions in the following sections and show how they are motivated and implemented.

### 5.1. An abductive logic model

Charniak and McDermott [5] defines *abduction* as a form of plausible reasoning and was first introduced by the philosopher Charles Peirce [25] to mean a kind of hypothetical reasoning. For example, given that both A and A ← B are true, we can infer B as a possible explanation for A. In other words, we can assume that B is true in order to prove that A is true. Abduction has been applied in many Artificial Intelligence problems such as expert system reasoning [5], diagnosis [7], planning [9], plan recognition and diagnosis [22] and etc.

Abduction is in general NP-hard [3,22,27,28]. However, it does not mean that the solution we propose has no practical use. Currently, there is no feasible means for the administrator to manage bureaucratic rules in the current manual system and that the approach we propose will at least be able to provide some form of feedbacks. In principle, as soon as the system can automatically find one deontic dilemma, we will make the task of managing the rules easier. In addition, the frequency of modifications to the rules is normally low and that the detection can be performed in batch mode. We will deal with these optimizations in details in future work.

In order to incorporate abduction into the inference mechanism, we need to extend the definition of the logic model defined in Definition 3.1 as follows:

*5.1.1. Definition 5.1*

An abductive logic model for bureaucracy consists of five parts:

< **BF**, **BR**, **BA**, **BAP**, **BIC** > ,

where

(a) **BF** is a set of ground formulae representing bureaucratic facts where some or all predicate symbol are in **BAP**.
(b) **BR** is a set of deductive rules of the form: H ← B1 and ... and Bn where n > = 0 where H, B1, ..., Bn are atomic formulae and H does not have predicate symbols in both **BF** and **BAP**.
(c) **BA** is the set of abducted predicates where all predicate symbols in **BA** are in **BAP**.
(d) **BAP** is a set of predicate symbols that is abducible.
(e) **BIC** is a set of integrity constraints of the form:
false ← B1 and ... and Bn where n > = 0
where B1, ..., Bn are atomic formulae.
(f) The logic model is **consistent** iff all integrity constraints are satisfied as follows:
∀c ∈ **BIC**, let **BIC**′ = **BIC** − {c},
**BF** ∪ **BR** ∪ **BA**| = c
and **BF** ∪ **BR** ∪ **BA**satisfies **BIC**′
where A| = B means A entails B. Since all integrity constraints are in the form of denial, then **BF** ∪ **BR** ∪ **BA** satisfies **BIC**′ iff **BF** ∪ **BR** ∪ **BA** ∪ **BIC**′ is consistent.

The abductive framework extends the logic model discussed earlier by having a set of predicates that are abducible. We limit abduction to a set of predicates specified by the users and only those predicates not defined by rules, i.e. only factual predicates can be made abductive. This is a practical restriction in order to reduce the computational complexity required by abduction.

Part (f) of the definition treats each integrity constraint in **BIC** as a goal in the traditional formalization of abductive framework. The only difference here is that when an abductive solution is discovered, the solution represents a potential scenario that can derive an inconsistency.

Abductive predicates should not be abducted on an arbitrary basis because there might be an unlimited number of assumptions one could make. Furthermore, the abduction we make may introduce inconsistency in the reasoning process. A trivial case of inconsistent abduction is when one abducts both p(a) and ~ p(a). Thus, we have to impose more constraints into the abductive framework by including explicitly additional integrity constraints defined below:

*5.1.2. Definition 5.2*

Given an abductive logic model:

< **BF**, **BR**, **BA**, **BAP**, **BIC** > .

For each abducible predicate A with predicate symbol P and arity n in **BAP**, there is an integrity constraint of the form:

false ← P(F1,...,Fn) & ~ P(F1,...,Fn).

in **BIC** where F1, ..., Fn are free variables.

This will ensure that no logically inconsistent abduction is made in the inference process. When an integrity constraint is being evaluated, the rest of the integrity constraints **BIC**′ will be used to restrict the abduction. In general, whenever a predicate is abducted, we have to satisfy the remaining set of integrity constraints. It is specified in the following definition:

*5.1.3. Definition 5.3*

Given an abductive logic model:

< **BF**, **BR**, **BA**, **BAP**, **BIC** > ,

and an integrity constraint c ϵ **BIC** that is currently being evaluated, if we try to abduct the predicate A such that the predicate symbol of A is in **BAP**, then we can abduct A iff

{A} ∪ **BF** ∪ **BR** ∪ **BA** ∪ **BIC**′,

is consistent where **BIC**′ = **BIC** − {c}.

## 5.2. Abductive skolemization

Given an abductive goal, say p(A1, ..., An), if it is completely ground, i.e. all arguments A1, ..., An are bound to some constant values, we can abduct such a predicate in a straightforward manner by assuming p(A1, ..., An) is true. However, what if the abducible predicate is not ground, i.e. some arguments may currently contain free variables at the moment of abduction?

Here, we adopt an approach by [9] using skolemization.

Skolemization is a method used for eliminating the existential quantifier so that predicate calculus logical expressions can be reduced into a clausal normal form suitable for automated reasoning [4]. The result of the skolemization is to replace all existentially quantified variables with skolem constants, i.e. a form of existential instantiation. Instead of abducting the existentially quantified objects with certain property denoted by the predicate, we artificially create such objects and say that they have that property. Consider an example with the following rule:

$h(A) \leftarrow p(1,A) \ \& \ A = 2.$

Suppose we have the goal h(A) for evaluating against this rule and let p be an abducible predicate symbol. At the moment of abduction, we have an instantiation of p ( 1,A ) and that

$p(1,A)$ is true iff $\exists_X p(1,X)$.

Thus, we can abduct p(1,A) if we assume $\exists_X p(1,X)$ is true. Since we cannot abduct the existentially quantified predicate, we need to apply the process of skolemization. Thus, we create a skolem constant denoted as SK1 [1] and abduct the predicate p(1,SK1). As a result, the variable A is bound to the skolem constant SK1. The complete algorithm for abducting a predicate is shown below:

*5.2.1. Definition 5.4*

Given an abductive logic model:

**< BF, BR, BA, BAP, BIC >**

Given the predicate **P** such that the predicate symbol of **P** is in **BAP**, we can abduct **P** in the following order of sequence:

(a) If ∃ a previously abducted predicate **Q** such that **P** can unify with **Q** with the most common unifier $\theta$, then the abducted predicate is **P**$\theta$. An unifier $\theta$ for **P** and **Q** is defined as a set of substitutions such that $\mathbf{P}\theta = \mathbf{Q}\theta$. A substitution is a mapping from a variable to a term. We say a substitution set $\theta$ is more general than another substitution set v if ∃ a substitution set $\upsilon$ such that $\theta = \mathrm{v}\upsilon$ [1,4].

(b) If step (a) fails, let **B1**, ..., **Bn** be the bound arguments and **F1**, ..., **Fm** be the free arguments at the moment of abduction. Then for each free argument **Fi** for $1 \le i \le m$, create a new skolem constant **SKi**. Thus, the abducted predicate is **P** with all **Fi** substituted with **SKi** for $1 \le i \le m$.

[1] From this point onwards, we shall denote all skolem constants as SKi for i > 0.

*5.3. Late binding and abduction of equality*

Even with the skolemization procedure in Definition 5.4, we still have another problem for the example above. What we really want to bind the variable A with is the value 2, not any arbitrary skolem constant **SK1**. Otherwise we cannot prove h(A) even though there exists such a proof if we abduct p(1,2) rather than p(1,**SK1**). The problem arises because we have committed the value of the variable A to a skolem constant too early. The solution is to *make the equality relation, = abducible*, whether it is used both directly as a predicate and indirectly in the unification process. In other words, we would like to treat comparison of skolem constants differently such that it behaves like a logical variable that can be bound once and can be unbound upon backtracking. Thus, in the above example, after we have abducted p(1,**SK1**), A is bound to **SK1** and we are now ready to solve the goal **SK1** = 2. Since = is abducible and that one of the argument is a skolem constant, we can abduct the predicate **SK1** = 2. Thus, as a consequence, we have proven h(2) with the set of abducted predicates {p(1,**SK1**), **SK1** = 2}.

We refer to this process as the late binding of skolem constants such that we do not need to commit the actual value of the skolem constants until it is needed. Since the equality relation is reflexive, symmetric and transitive, we have to introduce the following equality axioms, which we refer to as **E1**, into the inference process:

$A = A,$

$A = A = B \leftarrow B = A,$

$A = C \leftarrow A = B \quad \& \quad B = C.$

Based on these equality axioms, we can partition all the skolem constants we have created so far into equivalent sets. Each equivalent set is associated with a bound value if it is bound or a null value denoted as **null** if it is unbound. Thus, each equivalent set and its bound value forms a pair and the set of all such pairs is denoted as *ES* as follows:

$ES = \{ (S1, V1), \ldots, (Sn, Vn) \}$

where $n$ is the number of equivalent sets, $S1, \ldots, Sn$ are sets of equivalent skolem constants and $V1, \ldots, Vn$ are their corresponding bound values.

### 5.4. Abductive negation

So far, we have discussed abduction only in the context of positive literal. Abduction of negative literal is necessary and can be easily incorporated into the abductive logic model using the same framework that has been discussed in Section 5.2. Consider the following rule:

h(A,B,) ← ~ p(A,B).

A greedy approach to prove h(A,B) is to abduct ~ p(SK1,SK2) where SK1 and SK2 are skolem constants. Here, we follow again the approach by Gelfond and Lifschitz [12] and Esghi and Kowalski [10] by treating the negation of abducible predicates as explicit negation.

Informally, given a negated abducible predicate ~ p(Al, ...,An) for $n \geq 0$, we can create new corresponding predicate symbol $p^*$ and rewrite the predicate as $p^*$(B1, ..., Bn, F1, ..., Fm) and abduct it in the similar fashion as specified in Definition 5.4. Definition 5.2 ensures that we do not introduce logical inconsistency in abducting the negated predicate. Therefore, based on the new rewriting scheme for negated predicate, Definition 5.2 can be revised as follows:

#### 5.4.1. Definition 5.5

Given an abductive logic model:

< **BF**, **BR**, **BA**, **BAP**, **BIC** >

For each abducible predicate A with predicate symbol P and arity n in **BAP**, we replace each occurrence of the negation of the predicate A with a new predicate with a new predicate symbol $P^*$. For each of these abductive predicates, there is an integrity constraint of the form:

false ← P(F1,...,Fn) & $P^*$(F1,...,Fn),

in **BIC** where F1, ..., Fn are free variables.

### 5.5. Abduction of inequality

Extending the abductive logic model to handle inequality is not as simple. Previous approaches such as [9] did not provide a complete treatment of inequality except enforcing the inequality of non-skolem constants that cannot be unified. In fact, it is often necessary to provide abduction of inequality as in the trivial case with the following rule:

h(A,B) ← p(A,B) & A ≠ B,

such that p is an abductive predicate. A greedy approach would be able to prove h(SK1, SK2) by abducting p(SK1, SK2) and SK1 ≠ SK2. To ensure consistent abduction of equality and inequality, we need to incorporate the following integrity constraint:

#### 5.5.1. Definition 5.6

Given an abductive logic model:

< **BF**, **BR**, **BA**, **BAP**, **BIC** > .

Whenever an equality of inequality with at least one skolem constant is abducted, there is an integrity constraint of the form:

false ← A = B & A ≠ B,

in **BIC** that should be evaluated. A and B are free variables such that at least one of them is bound to a skolem constant.

Unfortunately, to allow abduction of inequality, additional inequality axiom **E2** shown below:

A ≠ B ← A = C, B = D, C ≠ D,

is required to ensure the correct evaluation of equality and inequality in general. For example, consider the following rule:

h(A,B) ← p(A,B) & A = a & B = b & A = B.

such that p is an abductive predicate. We have the following sequence of abductive steps during inference process:

(a) Abduction of p(SK,SK2) where SK1 and SK2 are skolem constants.

(b) Abduction of the equality SK1 = a.
(c) Abduction of the equality SK2 = b.
(d) Abduction of the equality SK1 = SK2.

Following the abduction of SK1 = SK2 in step (d), we should evaluate the following integrity constraint to ensure consistency:

$$\text{false} \leftarrow A = B \;\&\; A \neq B.$$

Without the inequality axiom **E2**, we can not derive SK1 ≠ SK2 even though we know that SK1 = a, SK2 = b and a ≠ b. As a consequence, the integrity constraint is incorrectly satisfied and the abduction of SK1 = SK2 is incorrectly accepted. If the inequality axiom **E2** is incorporated into the inference process, we have the following instantiation of the axiom:

$$\text{SK1} \neq \text{SK2} \leftarrow \text{SK1} = \text{a}, \text{SK2} = \text{b}, \text{a} \neq \text{b}.$$

As a result, step (d) will not succeed because we can conclude that SK1 ≠ SK2 given that SK1 = a, SK2 = b and a ≠ b are true. Hence, the following integrity constraint:

$$\text{false} \leftarrow \text{SK1} = \text{SK2} \;\&\; \text{SK1} \neq \text{SK2}.$$

is not satisfied and abduction of the inequality SK1 = SK2 will be rejected. Without the inequality axiom the wrong conclusion h(SK1, SK2) is reached with the assumptionS {SK1 = a, SK2 = b, SK1 = SK2}.

### 5.6. Abductive unification

In this section, we present an extension to the traditional unification algorithm used in logic programming [1,4] that allows the abduction of equality for the skolem constants. We refer to this extended unification as the *abductive unification* algorithm. The algorithm is specified as the Pascal-like pseudo-code in the Appendix.

The abductive unification procedure only provides the capability of abducting equality but not inequality. Another procedure which allows abduction of inequality called *abductive disunification* is also developed. Due to the space constraint, we are not able to describe all aspects of the abductive extensions and a comprehensive treatment of the abductive procedure can be found in [23].

## 6. Conclusions and future directions

This research was primarily motivated by the need to provide an automated tool for policy administrator in the management of bureaucratic policies. Currently, inconsistencies in policies remain because there exists no feasible mechanisms for the administrator to detect and remove them since the inter-relationships between policies are often so complex that it is beyond human cognitive ability to understand, detect and remove them. As a result, we develop a decision support system based on logic programming paradigm using deontic concepts to model policies. This will assist the administrator to decide whether a policy should be rejected or not. Furthermore, we extend the inference process with abductive reasoning to explore inconsistencies of policies in future scenarios.

However, it should be noted that we are limited to the domain of formalizable policies and that it is not always possible to verify all future scenarios in checking for inconsistencies in these polices. Nevertheless, this research represents a step forward in providing an automated tool to help policy administrators. It is recognized that some *feedback for policy administrator in making a decision about changing the existing policies are certainly better than none at all*. In addition, the availability of such automated tool can serve as a useful instrument to help policy makers in developing, designing and testing new policies by revising them iteratively until they are well-designed.

A prototype called the ALP which stands for Abductive Logic Programming has been implemented in *Prolog* [23]. Applications have been developed to validate the approach. One is based on a subset of the actual library lending code used by the General Library at the University of Texas at Austin.

Several related future research problems have been identified. The first is to extend the scope of exploration into future scenarios by exploiting the deontic meanings in the rules. In particular, future scenarios that are likely to happen include scenarios that are obedient to current deontic conditions as well as scenarios that violates the current deontic conditions. Secondly, we would

like to extend the logic model towards a *closed* system capable of *self-control* such that all update actions that can change the policies of the model are regulated by the policies internal to the model. Hence, no additional mechanisms or policies external to the model is needed for controlling the consistency of the system.

Lastly, we also intend to further exploit the benefits of formalized policies. Once these policies are verified, they can be used to verify and enforce the correctness of the facts. In the real situation, facts residing in databases are large and we need to take into consideration the performance aspect of inference process. In order to support a more efficient execution of these policies, we intend to explore into the possibility of using deductive database system. Deductive database systems integrate logic programming paradigm with relational database technology [21,35,39]. They are suitable for both data intensive and knowledge-based applications. An implementation of such a system called *LDL* + + is currently available [2,34]. This will enable the control of the evolution of data using the policies as the constraints.

**Acknowledgements**

We would like to thank the editor, the referees who reviewed this paper as well as those who reviewed the original version of this paper [24]. Furthermore, we would like to acknowledge those Ph.D students at the University of Texas at Austin, namely Jim Baty, Kuo-Tay Chen, Sandy Dewitz, Steve Hargis, Hoguen Lee and Young Ryu, who have contributed with their comments and criticisms during this research effort.

## Appendix A. Abductive unification algorithm

Given A and B are two terms to be unified, the abductive unification algorithm is defined by the routines ABD_UNIFY/3 and ABD_UNIFY_SKOLEM/3 such that it will return TRUE if they are unifiable; otherwise it will return FALSE. In addition, there will be two side effects:

(a) a substitution list $\theta$ is produced and
(b) abduction of equality involving skolem constants

### A.1. Algorithm 1

```
Boolean ABD_UNIFY(X1, X2, θ)
{
  CASE (constant(X1) and constant(X2)):
    return (X1 = X2)        constant to constant
  CASE (functor(X1) and functor(X2)): s.t. X1 =
    f(A1, ..., Am) and X2 = f(B1, ..., Bm)
    functor to functor-break down into sub-arguments
    return (ABD_UNIFY(A1, B1, θ) and ...
    and ABD_UNIFY (Am, Bm, θ))
  CASE (variable(X1)):     assign X2 to X1
    IF (X1 occurs in X2)
    THEN   return (FALSE)
    ELSE   Add X1/X2 to the substitution
           list θ
           Apply the substitution to A and B
           return (TRUE)
  CASE (variable(X2)):
    return (ABD_UNIFY(X2, X1, θ))
  CASE (skolem(X1)):      unification of skolem
    constants
    IF (X1 occurs in X2)
    THEN   return (FALSE)
    ELSE   return
           (ABD_UNIFY_SKOLEM(X1,
           X2, θ))
  CASE (skolem(X2)):
    return (ABD_UNIFY(X2, X1, θ))
  DEFAULT:
    return (FALSE)
}
```

### A.2. Algorithm 2

```
Boolean ABD_UNIFY_SKOLEM(X1, X2, θ)
{  skolem constant must in the equivalent sets
   Let ∃i s.t. (Si, Vi) ∈ ES and X1 ∈ Si
   IF (skolem(X2))
   THEN
     Let ∃ j s.t. (Si, Vj) ∈ ES and X2 ∈ Sj
```

```
CASE (i = j):        identical skolem constant
                     no abduction is necessary
   return (TRUE)
CASE (Vi = null):    abduct the equality X1 =
                     X2 by collapsing Si and
                     Sj
   ES = (ES − {(Si, Vi), (Sj, Vj)} ∪ {(Si ∪
                     Sj, Vi)}
   return (TRUE)
CASE (Vj = null):    abduct the equality X1 =
                     X2 by collapsing Si and
                     Sj
   ES = (ES − {(Si, Vi), (Sj, Vj)}) ∪ {(Si
                     ∪ Sj, Vi)}
   return (TRUE)
DEFAULT:             abduct the equality X1 =
                     X2 by collapsing Si and
                     Sj if unifiable
   IF (ABD_UNIFY(Vi, Vj, θ))
   THEN  ES = (ES − {(Si, Vi), (Sj, Vj)})
                     ∪ {(Si ∪ Sj, Vi)}
         return (TRUE)
   ELSE  return (FALSE)
ELSE
 IF (Vi = null)      abduct the equality X1 =
                     X2 by assigning to a
                     skolem constant
 THEN   ES = (ES − {(Si, Vi)}) ∪ {(Si, X2)}
        return (TRUE)
 ELSE   return (ABD_UNIFY(Vi, X2, θ))
}
```

Note that the occur checks for both variables and skolem constants are normally omitted in logic programming systems such as Prolog for efficiency reasons. The core of the extension is in the ABD_UNIFY_SKOLEM/2 routine where we maintain additional information about the skolem constants in the equivalent sets **ES** during the unification process.

## References


[1] Apt, Krzysztof R., Logic Programming, Chap. 10, Handbook of Theoretical Computer Science, Ed. J. van Leeuwen, pp. 493–574, 1990.

[2] Arni, Natraj, Ong, KayLiang, Tsur, Shalom and Zaniolo, Carlo, LDL++: A Second Generation Deductive Database System, Submitted for publication, 1994.

[3] Bylander, Tom, Allemang, Dean, Tanner, Michael C., and Josephson, John R., Some results concerning the computational complexity of abduction, In Proc. of the 1st Int. Conf. on Principles of Knowledge Representation and Reasoning, pp. 44–45, Toronto, Ontario, Canada, 1989.

[4] Chang, Chin-Liang and Lee, Richard Char-Tung, Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973.

[5] Charniak, E. and McDermott, D., Introduction to Artificial Intelligence, Addison-Wesley Pub. Co., 1985.

[6] Colmerauer, A., Kanoui, H., Roussel, P. and Pasero, R., Un Systeme de Communication Homme-Machine en Francais, Groupe de Recherche en Intelligence Artificielle, Universite d'Aix-Marseille, 1973.

[7] Cox, P.T. and Pietrzykowski, Y., General Diagnosis by Abductive Inference, Proc. IEEE Symp. on Logic Programming, 1986, pp. 183–189.

[8] Das, Subrata, K. and Williams, Howard, M., A Path Finding Method for Constraint Checking in Deductive Databases, Data and Knowledge Engineering, 4, 1989, pp. 223–244, North-Holland.

[9] Eshghi, K., Abductive Planning with Event Calculus, Proc. 5th International Conference on Logic Programming, R. Kowalski and K. Bowen, (eds), 1988, pp. 562–579.

[10] Eshghi, K. and Kowalski, R.A., Abduction Compared With Negation By Failure, Proc. 6th International Conference on Logic Programming, Lisbon, Portugal, June 1989, MIT Press.

[11] Follesdal, Dagfinn and Hilpinen, Risto, Deontic Logic: An Introduction, in Deontic Logic: Introductory and Systematic Readings, Edited by Risto Hilpinen, D. Reidel Publishing Company / Dordrecht-Holland, 1971, pp. 1–35.

[12] Gelfond, Michael and Lifschitz, Vladimir, Logic Programs with Classical Negations Proceedings of the Seventh International Conference on Logic Programming, David D.H. Warren and Peter Szeredi (eds.), 1990, pp. 579–597.

[13] Genesereth, Michael R. and Nilsson Nils. J., Logical Foundations of Artificail Intelligence, Morgan Kaufmann, 1987.

[14] Kowalski, R.A. Predicate Logic as a Programming Language, in Proc. IFIP 1974, Stockholm, North-Holland, 1974, 569–574.

[15] Lee, Ronald M., Bureaucracies, bureaucrats and information technology, European Journal of Operational Research 19, (1984) 293–303.

[16] Lee, Ronald M. Automating red tape: The performative vs. informative roles of bureaucratic documents, Offices: Technology and People 1,2, (1984), 187–204.

[17] Lee, Ronald M., Bureaucracy as Artificial Intelligence, in Knowledge Representation for Decision Support, Proc. of IFIP WG 8.3 Working Conf. (Durham, England, July), North-Holland, Amsterdam, 1984.

[18] Lee, Ronald M., Bureaucracies as Deonatic Systems, ACM Transaction on Office Information Systems, Vol 6, No. 2, April 1988, pp 87–108.

[19] Lee, Ronald M. and Ryu, Young, DX A Deontic Expert System, Working Paper, (submitted to Data and Knowledge Engineering), 1992.

[20] Meyer, Marshall W., Stenvenson, William and Webster, Stephen, Limits to Bureaucratic Growth, Walter de Gruyter, 1985.

[21] Naqvi, Shamim and Tsur, Shalom, A Logical Language for Data and Knowledge Bases, Computer Science Press, New York, 1989.

[22] Ng, Hwee-Tou, A General Abductive System With Application to Plan Recognition and Diagnosis, Doctorate Dissertation, University of Texas at Austin, 1992.

[23] Ong, KayLiang, A. Formal Model for Maintaining Consistency of Evolving Bureaucratic Policies: A Logical and Abductive Approach, Doctorate Dissertation, University of Texas at Austin, 1992.

[24] Ong, KayLiang and Ronald, M. Lee, A Logic Model for Maintaining Consistency of Bureaucratic Policies, Proceedings of the 26th Hawaii International Conference on System Sciences, Maui, January 5-8, 1993.

[25] Peirce, C.S., Collected papers of Charles Sanders Pierce, Vol. 2, 1931-1958, (C. Hartshorn et al, eds.) Harvard University Press, 1931.

[26] Reiter, Raymond, On Integrity Constraints, Proceedings of 2nd Conference on Theoretical Aspects of Reasoning about Knowledge, Asilomar, CA 1988, p. 97.

[27] Reggia, James A., Nau, Dana S., Wang, Pearl Y., Diagnostic expert systems based on a set covering model, International Journal of Man-Machine Studies, 19:437-460, 1983.

[28] Reggia, James A., Nau, Dana S., Wang, Pearl Y., A formal model of diagnostic inference, I. problem formulation and decomposition, Information Sciences, 37:227-256, 1985.

[29] Robinson, J.A. A Machine-Oriented Logic Based on the Resolution Principle, J. ACM, V12, N1, 1965, 23-41.

[30] Rozenshtein, David and Minsky, Naftaly, Controlling the Use and Evolution of Database Systems: A Prolog-Based Approach, Journal of MIS, Summer 1986, VIII, N1.

[31] Ryu, Young, Defeasible Deontic Reasoning - A Formal Approach, Doctorate Dissertation, University of Texas at Austin, 1991.

[32] Sadri, Fariba and Kowalski, Robert, A Theorem-Proving Approach to Database Integrity, Ch. 9, Foundations of Deductive Databases and Logic Programming, Ed. by J. Minker, Morgan Kaufmann, Los Altos, 1987.

[33] Sergot, M.J., Sadri F., Kowalski R.A., Kriwaczek F., Hammond P. and Cory H.T., The British Nationality Act As A Logic Program, Comm. of ACM, May 1986, Vol. 29, Number 5.

[34] Tsur, Shalom, Arni, Natraj and Ong, KayLiang, The LDL+ + User Guide, MCC Technical Report, 1993.

[35] Ullman, Jeffrey, Principles of Database and Knowledge Base Systems, Volume I and II, Computer Science Press, 1990.

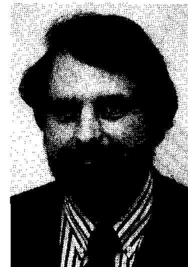
[36] Von Wright, Georg Henrik, A New System of Deontic Logic, in Deontic Logic: Introductory and Systematic Readings, Risto Hilpinen (eds), D. Reidel Pub. Co./Dordrecht-Holland, 1971, pp. 105-120.

[37] Von Wright, G.H., An Easy in Deontic Logic and the General Theory of Action. in Acta Philosophica, Vol. 12, North-Holland, Amsterdam, 1968.

[38] Von Wright, G.H. Deontic Logic, Mind 60 (1951) 1-15. Reprinted in Logical Studies (by G.H. von Wright), Routledge and Kegan Paul, London 1957, pp. 58-74.

[39] Zaniolo, Carlo, The Design and Implementation of a Logic-Based Language for Data Intensive Applications, in Proc. of the Int. Conf. on Logic Programming, 1988.

**KayLiang Ong** is currently a Member of Technical Staff at Microelectronics and Computer Technology Corporation (MCC), a consortium for industrial collaborative research. He received his Ph.D (1992) in Management Information Systems and his M.A (1988) and B.A (1986) in Computer Science from the University of Texas at Austin. He was also a visiting scientist to the Erasmus University Research Institute for Decision and Information Systems (EURIDIS). Since 1986, he has been involved in the research and development of deductive/logic databases and heterogeneous and distributed database integration. He is one of core developers of the LDL+ + deductive database system and has successfully transferred and deployed the new technology for commercial applications at various industrial sites. His research interests includes deductive databases, database integration, knowledge discovery/mining, logic programming and logic modeling especially on formalization of bureaucratic rules and procedures.



**Ronald M. Lee** is currently Director of the Erasmus University Research Institute for Decision and Information Systems (EURIDIS) of Erasmus University. Formerly, he was Associate Professor of Management Science and Information Systems Department at the University of Texas at Austin. He has a Ph.D in Decision Sciences (Wharton, 1980), and has previously served as a research scholar at the International Institute for Applied Systems Analysis in Vienna, Austria and as Visiting Professor of Management at the Universidade Nova de Lisbon, Portugal. His research focuses on the use of formal logic representation for management science applications. Current projects involve the use of logic modeling to represent and manage formal business communications systems focusing on bureaucratic systems (formalized communications within institutions) and electronic contracting systems (formalized communications between enterprises).