REPRESENTING ACTIONS IN LOGIC PROGRAMMING

AND

ITS APPLICATIONS IN DATABASE UPDATES

Phan Minh Dung
Division of Computer Science
Asian Institute of Technology
GPO Box 2754, Bangkok 10501, Thailand
E-mail:dung@cs.ait.ac.th

Abstract

We give a sound and complete method for representing actions in normal logic programming. The method allows both reasoning about the past as well as reasoning about the future where abduction can be "implemented" by deduction. We show that our framework provides a simple but powerful theoretical and practical framework for reasoning about updates. We give an abstract simplification method for reasoning about abstract actions generalizing Nicolas's well-known simplification method. Further, we also point out the interesting relations between partial evaluations, constructive negation, abductions, and reasoning about actions. We compare our method with a related method of Gelfond and Lifschitz where we show the incompleteness of the later.

1. Introduction

The ultimate goal of a theory of actions is to provide a method for verifying whether a dynamical system meets its specification. Such theories have been the subject of much interest in AI. Often it has been done by studying some toy examples. Gelfond and Lifschitz [GL2] recently have proposed a new methodology for studying actions which we believe is more likely to find applications in practice. Here, a particular methodology for representing actions can be viewed as a translation from a simple "source" language for describing actions into a "target language" -for instance into logic programming. Thus the soundness and completeness of each particular translation become precise mathematical questions. Unfortunately, Gelfond and Lifschitz's translation suffers from many severe problems. First of all, it is not complete. Further in many examples, it yields multiple answer sets where only one of them captures the intended semantics (see chapter 5). Since the approach is purely declarative, an appropriate proof procedure remains to be developed, thus making it unlikely to be applicable in practice.

We believe that the best way to mechanize the process of reasoning about actions is to find a way to represent actions in logic programming. Doing so will make it possible to use the powerful Prolog systems to perform reasoning. Many of the proposals for formalizations of action in logic programming have been made [AB,E]. Unfortunately, these proposals are adequate only for a simple kind of temporal reasoning called "temporal projection" where we are given a description of the initial state of the world, and use the properties of actions to determine what the world will look like after a series of action is performed. Moreover, these approaches can be used only if the description of the initial world is complete since the "closed world assumption" is applied to every predicate [GL2].

Our goal is to provide a simple but powerful logic programming based framework for reasoning about actions which combines the novelties of previous works and at the same time overcomes their limitations. Adopting Gelfond and Lifschitz's methodology, our method is a translation from a simple source language for describing actions into the target language of normal logic programming[1]. The difference to ealier approaches [AB,E] is that we don't apply the "closed world assumption" on each predicate. Instead, a modification of Clark's predicate completion is used which is "closed" at any future situation but open at the current situation. The result is a simple, sound and complete method which allows both reasoning about the future and the past where abduction can be "implemented" by deduction. Since the target language of our translation is the ordinary logic programming, we can apply the well-developed proof procedures of normal logic programming directly as proof procedures for reasoning about action. A main result of this paper is the abstract simplification method for reasoning about actions generalizing Nicolas's well-known simplification method [N].

Finding efficient methods for checking the integrity of database updates is of vital importance for the applications of databases. Since database updates are nothings but actions performed on database states, it is obvious that systems which are capable of reasoning about actions must be applicable to integrity checking of database updates. But surprisingly, until now, not much works have been done to study this relationship. In this paper, we will address this problem by showing that our framework provides a simple but powerful theoretical and practical framework for reasoning about updates. We will show how the well-known techniques of partial evaluation and contructive negations [AD,C,CW,LS], can be applied within our framework to provide a simple, intuitive and efficient methods for checking the integrity of database updates.

The paper is organized as follows: In chapter two, a simple relational language for describing actions is given. In chapter three, we introduce our method for

---

[1]We avoid the use of "classical" negation by considering ¬F as a new fluent if F is a fluent name. Thus we make a difference between Holds(¬F,s) and not Holds(F,s).

representing actions in normal logic programming and prove its soundness and completeness. Chapter 4 applies our theory to integrity checking of database updates. Chapter 5 discusses the relations between our method and that of Gelfond and Lifschitz [GL2] where we prove the incompleteness of the later. We summarize the pros and cons of our approach in the conclusion.

2. A Relational Language for Representing Actions

In this chapter, we introduce a relational language for representing actions generalizing the propositional language given in [GL2]. The alphabet of our language is a triple <FLU,ACT,CONST> where FLU is a finite set of predicate fluent names and ACT is a finite set of action names, and CONST is a set of individual constants. An *individual term* is either an individual constant or a variable. A *fluent atom* is an expression of the form $F(t_1,..,t_n)$ where F is an n-ary predicate fluent name and t_i's are individual terms. A *fluent literal* is a fluent atom L or the negation ¬L of a fluent atom L. The complement of a positive fluent literal L is ¬L. Similarly, the complement of a negative fluent literal ¬L is L. The *complement* of a fluent literal L is denoted by *L**. An *equality constraint* is either an equation t = t' or an inequation t ≠ t' between individual terms. The *complement* of an equality (resp. inequality) constraint t = t' (resp. t ≠ t') is t ≠ t' (resp. t = t'). The *complement* of an equality constraint C is denoted by C*. A *fluent expression* is a nonempty disjunction of fluent literals and equality constraints. An *action expression* is of the form $A(t_1,..,t_n)$ where A is an n-ary action name and t_i's are individual terms. We often use F and A to denote fluent expressions and action expressions, respectively.

An *e-formula* is an expression of the form

$$A \text{ causes } L_0 \text{ if } L_1,...,L_n$$

where A is an action expression, L_i's are fluent literals.

A *v-formula* is of the form

$$F \text{ after } A_1,...,A_n$$

where F is an fluent expression and A_i's are action expressions. If n=0 then a v-formula is called an init-formula and written as *initially F*.

A *relational domain description* is a finite set of e- and v-formulae.

Example 1 [GL2] The domain of the famous Yale Shooting Problem is characterized by the following propositions:

initially ¬Loaded

initially Alive
Load causes Loaded
Shoot causes ¬Alive if Loaded
Shoot causes ¬Loaded ■

Example 2 The following relational database schema together with the insert and delete update operations is a simple example of a relational domain description.

FLU = {Student,Receive}, and ACT = {$Insert_{recv}$,$Delete_{recv}$} where Student is an unary and Receive a binary fluent name, and $Insert_{recv}$ as well as $Delete_{recv}$ are binary action names. The e-formulae describing the effect of the updating operations are

$Insert_{recv}(x,y)$ causes Receive(x,y)
$Delete_{recv}(x,y)$ causes ¬Receive(x,y) ■

The semantics of a domain description is described by its models which are defined in the following.

A *pre-state* over an alphabet <FLU,ACT,CONST> is a triple <DOM,τ,τ'> where DOM is a set of individuals, called the individual domain, and τ is an one-to-one mapping from CONST into DOM, and τ' is an assignment which assigns the identity relation on DOM to =, and the complement of the identity relation to ≠. ■

An *assignment* of a set of variable $X = \{x_1,..,x_n\}$ wrt pre-state <DOM,τ,τ'> is a mapping Θ from X into DOM. For any action expression A(X) and any assignment Θ of X wrt a pre-state J, A(X)Θ is called an *J-instance* of A(X). The J-instances of fluent expressions, e-formulae, v-formulae are defined similarly. J-instances of expressions are also called J-ground expressions.

Remark We assume that J-ground fluent expressions are always simplified such that no equality constraints appear in it.

The set of all J-ground action expressions is denoted by $ACTION_J$. Similarly, the set of all J-ground fluent atoms is denoted by BF_J.

A *state* over a pre-state J is a subset of BF_J. The set of all states over a pre-state J is denoted by ST_J.

An positive J-ground fluent literal L holds in a state σ iff $L \in \sigma$. A negative J-ground fluent literal ¬L holds in σ iff $L \notin \sigma$. A J-ground fluent expression $L_1 \vee ... \vee L_n$ holds in σ if some L_i, $1 \leq i \leq n$, holds in σ.

A *transition function* Φ over a pre-state J is a mapping from $ACTION_J \times ST_J$ into ST_J. A *structure* over J is a pair (σ_0,Φ) where σ_0 is a state (the initial state) over J, and Φ is a transition function over J.

We say that a J-ground v-formula F after $A_1,...,A_n$ is true in a structure $M=(\sigma_0,\Phi)$, if F holds in the state $\Phi(A_n,\Phi(A_{n-1},...,\Phi(A_1,\sigma_0)..))$, and that it is false otherwise. A (possibly nonground) v-formula F after $A_1,...,A_n$ is true in a structure M if each J-instance of it is true in M.

A structure $M=(\sigma_0,\Phi)$ over a pre-state J is a *model* of a domain description D if every v-formula from D is true in M, and for each J-ground action expression A, each J-ground fluent literal L, and each state σ, the following conditions are satisfied:

(1) If there exists an J-instance of the form

$$A \text{ causes } L \text{ if } L_1,...,L_n$$

of some e-formula in D such that L_i's are true in σ then L holds in $\Phi(A,\sigma)$.

(2) Otherwise, L holds in $\Phi(A,\sigma)$ iff L holds in σ and L is not affected by any action whose preconditions are satisifed in σ.[2]

For example, the transition function of the Yale Shooting Problem is the following:

$$\Phi(\text{Load},\sigma) = \sigma \cup \{\text{Loaded}\}$$

$$\Phi(\text{Shoot},\sigma) = \begin{cases} \sigma\backslash\{\text{Loaded},\text{Alive}\}, & \text{if Loaded} \in \sigma \\ \sigma, & \text{otherwise} \end{cases}$$

$$\Phi(\text{Wait},\sigma) = \sigma$$

A relational domain description is *consistent* if it has a model. A v-formula E is *entailed* by a relational domain description D, written $D \models E$, if it is true in each model of D.

Now, we give a simple sufficient condition for the consistency of a domain description.

An action name A is said to be *selfcontradicted* in D if D contains two e-formulae whose variables are disjoint, of the following form:

$$A(X) \text{ causes } L(X) \text{ if } L_1(X),...,L_n(X)$$
$$A(Y) \text{ causes } \neg L(Y) \text{ if } K_1(Y),...,K_m(Y)$$

---

[2] L is affected by an action A in state σ if there exists an J-ground e-formula of the form A causes L if $L_1,...,L_n$ or of the form A causes L* if $L_1,...,L_n$ such that all L_i's hold in σ.

such that L(X) and L(Y) are unifiable with mgu θ and there exist no such pairs i,j such that $K_j(Y) = L_i(Y)^*$ and $L_i(X)\theta$ and $L_i(Y)\theta$ are unifiable.

Now we can prove the first result about the consistency of domain description.

Lemma 1 *Let D be a domain description containing no v-propositions. Then D is consistent if D contains no selfcontradicted actions.* ■

Remark From now on we consider only domain description containing no selfcontradicted actions.

3. Transforming Relational Domain Description into Logic Programs

Our method translates a domain description D into a normal logic program αD. αD uses variables of three sorts: situation variables s,s',..., fluent variables f,f',.., action variables a,a',...[3] The only situation constant is S0. The fluent terms are the fluent literals where for any fluent atom L, ¬L is another distinct term[4]. The action terms are the action expressions. *The only (recursive) function symbol* is a situation function symbol [s|a] denoting the situation resulted after the action a is performed in situation s.

Let D be a domain description. Then by D_E,D_V we denote the sets of all e- and v-formulae in D, respectively. The corresponding program $\alpha D = \alpha D_E \cup \alpha D_V$ is defined as follows:

αD_E consists of the translation of the individual formulae from D_E together with the standard rule:

$$Holds(f,[s|a]) \leftarrow Holds(f,s), \text{ not } Ab(f,a,s)$$

The translation of an e-formula A causes L if $L_1,...,L_n$ consists of 3 rules:

$$Holds(L,[s|A]) \leftarrow Holds(L_1,s),...,Holds(L_n,s)$$
$$Ab(L,A,s) \leftarrow Holds(L_1,s),...,Holds(L_n,s)$$
$$Ab(L^*,A,s) \leftarrow Holds(L_1,s),...,Holds(L_n,s)$$

A v-formula $L_1 \vee ... \vee L_m \vee C_1 \vee .. \vee C_k$ after $A_1...A_n$ where L_i's are fluent literals and C_i's are equality constraints, is translated into a denial

---

[3]Using a sorted language implies that all atoms in the rules of a program are formed according to the syntax of the sorted language. Further, we will always assume that all the substitutions we will consider are of appropriate sorts.

[4]Note that ¬¬L is not a fluent term

$\leftarrow$ Holds(L_1*,[S0|A_1|...|A_n]),...,Holds(L_m*,[S0|A_1|...|A_n]),C_1*,...,C_k*

αD_V consist of the translations of the v-formulae in it.

Now we want to define the semantics of αD. First, clauses of the form

Holds(F,[s|A]) $\leftarrow$ Holds(L_1,s),...,Holds(L_n,s)
Ab(F,A,s) $\leftarrow$ Holds(L_1,s),...,Holds(L_n,s)

in αD_E are transformed respectively into the following form

Holds(f,[s|a]) $\leftarrow$ f = F, a = A, Holds(L_1,s),...,Holds(L_n,s)
Ab(f,a,s) $\leftarrow$ f = F, a = A, Holds(L_1,s),...,Holds(L_n,s)

Let Holds(f,[s|a]) $\leftarrow$ E_1
Holds(f,[s|a]) $\leftarrow$ E_2
.....
Holds(f,[s|a]) $\leftarrow$ E_n

Ab(f,a,s) $\leftarrow$ E_1'
....
Ab(f,a,s) $\leftarrow$ E_m'

be all the transformed clauses.[5]

Then ***$comp_E(D)$*** is defined as the first order theory consisting of the following sentences together with the corresponding Clark's equality theory CET [L].

$\forall f \forall a \forall s($ Holds(f,[s|a]) $\leftrightarrow E_1 \vee ... \vee E_n$)

$\forall f \forall a \forall s($ Ab(f,a,s) $\leftrightarrow E_1' \vee ... \vee E_m'$)

Remark The completion semantics is defined only for Holds(f,[s|a]) and not for Holds(f,s).

Definition 1 *μD is defined as the following multi-sorted first order theory*

$$\mu D = comp_E(D) \cup \alpha D_V \cup ICS_D$$

where ICS_D = { $\forall x_1..\forall x_n($ *Holds*($\neg F(x_1,..,x_n)$,S0) $\leftrightarrow$ *not Holds*($F(x_1,..,x_n)$,S0)) | *F is an n-ary fluent name* }

■

---

[5]Note that the standard clause Holds(f,[s|a]) $\leftarrow$ Holds(f,s), not Ab(f,a,s) is among these clauses.

Example 3 Let D be the domain description of the Yale Shooting Problem (see example 1). αD consists of the following clauses:

αD_V: $\leftarrow$ Holds(Loaded,S0)
$\leftarrow$ Holds(¬Alive,S0)

αD_A: Holds(Loaded,[s|Load]) $\leftarrow$
Ab(Loaded,Load,s) $\leftarrow$
Ab(¬Loaded,Load,s) $\leftarrow$

Holds(¬Alive,[s|Shoot]) $\leftarrow$ Holds(Loaded,s)
Ab(Alive,Shoot,s) $\leftarrow$ Holds(Loaded,s)
Ab(¬Alive,Shoot,s) $\leftarrow$ Holds(Loaded,s)

Holds(¬Loaded,[s|Shoot]) $\leftarrow$
Ab(Loaded,Shoot,s) $\leftarrow$
Ab(¬Loaded,Shoot,s) $\leftarrow$

Holds(f,[s|a]) $\leftarrow$ Holds(f,s), not Ab(f,a,s)

Thus μD is $\alpha D_V \cup ICS_D \cup CET$ together with the following sentences:

$\forall f \forall a \forall s($ Holds(f,[s|a]) $\leftrightarrow$ (f=Loaded $\wedge$ a=Load $\vee$
f=¬Alive $\wedge$ a=Shoot $\wedge$ Holds(Loaded,s) $\vee$
f=¬Loaded $\wedge$ a=Shoot $\vee$
Holds(f,s) $\wedge$ not Ab(f,a,s)))

$\forall f \forall a \forall s($ Ab(f,a,s) $\leftrightarrow$ (f=Loaded $\wedge$ a=Load $\vee$
f=¬Loaded $\wedge$ a=Load $\vee$
f=Alive $\wedge$ a=Shoot $\wedge$ Holds(Loaded,s) $\vee$
f=¬Alive $\wedge$ a=Shoot $\wedge$ Holds(Loaded,s) $\vee$
f=Loaded $\wedge$ a=Shoot $\vee$
f=¬Loaded $\wedge$ a=Shoot))

It is clear that μD $\models$ Holds(Loaded,[S0|Load]). Hence μD $\models$ Holds(Loaded,[S0|Load|Wait]). Therefore μD $\models$ Holds(¬Alive, [S0|Load|Wait|Shoot]) ■

The equivalence of μD and D is showed in the following theorem.

Theorem 1 (Soundness and Completeness)

Let D be a relational domain description and E be a v-formula. Then

$$\mu D \models \alpha E \quad \textit{iff} \quad D \models E$$ ■

4. Application: Checking Integrity of Database Updates

In this chapter, we apply our theory to develop an powerful theoretical and practical framework for reasoning about database updates. The result is an abstract simplification method for reasoning about abstract actions generalizing Nicolas's well-known simplification method [N]. Further, we also point out the interesting relations between partial evaluations, constructive negation, abductions, and reasoning about actions.

A relational database schema is a finite set of predicate symbols RDS. A relational database state over a schema RDS is a finite set of ground atoms of predicates from RDS. The semantics of a relational database is defined using the closed world assumption. A ground positive (resp negative) literal L is true wrt a relational database state DB, written DB $\models_{CWA}$ L if L $\in$ DB (resp. L* $\notin$ DB).

Integrity constraints are conditions which the database has to satisfy when it changes through the time. Formally, we say that an integrity constraint IC is satisfied in a database state DB if DB $\models_{CWA}$ IC. For the sake of simplicity, in this paper, we consider only integrity constraints which are denials, i.e. expressions of the form $\leftarrow L_1,...,L_n,C_1,...,C_m$[6] where L_i's are literals and C_i's are equality constraints.

Databases are changed through updates. Updates can be either primitive updates or compound updates. Similar to Wallace [W], (compound) updates are defined according to the following syntax:

Update ::= Primitive Update
foreach X: Cond(X) do Primitive Update
Update1;Update2

Primitive Update ::= insert(B) | delete(B)

where Cond(X) is a conjunction of literals with free variables in X , and B is an atom.

The result of inserting a ground atom L into a database state DB is DB $\cup$ {L}. The result of deleting a ground atom L from a database state DB is DB \ {L}.

Up1;Up2 is a sequential composition of Up1 and Up2 which is defined by first performing Up1 then Up2.

A "parallel" update foreach X: Cond(X) do Up(X) represents a "simultaneously"

---

[6] All clauses are universially quantified at the head.

performing of all updates Up(X)δ[7] for which Cond(X)δ holds in the current state of the database. For example, the result of performing the update foreach x,y: p(a,x) do insert(q(x)) upon the database state DB = {p(a,b),p(b,c),p(a,d)} is the database state DB $\cup$ {q(b),q(d)}.

The database resulted from updating a relational database state DB by an update Up is denoted as result(DB,Up). An integrity constraint IC is satisfied by an update Up if result(DB,Up) $\models_{CWA}$ IC.

It is clear that an update language is in fact a language for describing actions on the database states. Given a (compound) update Up, a domain description D_{Up} is defined as follows:

1) For each n-ary predicate symbol p appearing in Up, D_{Up} contains the following e-formulae:

$insert_p(x_1,..,x_n)$ causes $p(x_1,..,x_n)$ and

$delete_p(x_1,..,x_n)$ causes $\neg p(x_1,..,x_n)$

2) For each "parallel" update foreach X:Cond do Pup(X) in Up, D_{Up} contains the following e-formula

PU causes L(X) if Cond(X)

where L(X) = p(X) if Pup(X) = $insert_p(X)$, and L = $\neg$p(X) if Pup = $delete_p(X)$, and PU is the string "foreach X:Cond do Pup(X)" interpreted as a propositional action name.

3) No other formulae are contained in D_{Up}

For each (compound) update Up, define

$$P_{Up} = \alpha D_{Up} \cup \{ \text{Holds}(L,S0) \leftarrow \mid L \in DB \} \cup \{ \text{Holds}(\neg L,S0) \leftarrow \text{notHolds}(L,S0) \mid L = F(x_1,..,x_n), \text{ F is an n-ary fluent name} \}$$

The following theorem shows how our theory can be applied in checking the integrity of database updates.

---

[7] δ is a ground substitution over X

Theorem 2 *The integrity constraint* $IC = \leftarrow L_1,..,L_n,C_1,..,C_m$ *is satisfied by the (compound) update Up on the database state DB if and only if*

$$comp(P_{Up})^8 \models \leftarrow Holds(L_1,[S0|Up]),...,Holds(L_n,[S0|Up]),C_1,...,C_m$$

■

It follows immediately

Theorem 3 *The integrity constraint* $IC = \leftarrow L_1,..,L_n,C_1,..,C_m$ *is satisfied by the (compound) update Up on the database state DB if and only if the SLD-CNF*[9] *tree for the goal* $\leftarrow Holds(L_1,[S0|Up]),...,Holds(L_n,[S0|Up]),C_1,..,C_m$ *with respect to the program* P_{Up} *finitely fails.*

■

From the theorem 3, it is clear that efficient methods for checking the integrity of database updates can be developed using the well-studied techniques of constructive negation and partial evaluation in logic programming [AD,C,CW,LS]. As an example, we will show in the following how the well-known simplification method [N] can be obtained from theorem 3 through partial evaluation. We need some new notations.

Definition 2 Given a domain description D and a v-formula E. A set S of init-formulae is said to be an explanation of E if following conditions are satisfied:

(1) $D \cup S$ is consistent
(2) $D \cup S \models E$

Let S,S' be explanations of E wrt D. We say that S' is less specific than S wrt D if $S \models S'$.

■

It is clear that if S,S' are explanations of E wrt D then $S \vee S'$ is also an explanation of E wrt D which is less specific than both S and S'. Hence, the least specific explanation of E wrt D, if exists, represents a collection of all possible explanations of E wrt D.

Definition 3 A domain description D is said to be *normal* if each v-formula E possesses at least one least specific explanation S such that $D \cup E \models S$. ■

---

[8] $comp(P_{Up})$ is the Clark's completion of the program P_{Up} as defined in [L].

[9] SLD-CNF stands for SLD resolution augemented with constructive negation as failure [C].

Now we can define the abstract simplification method for reasoning about actions:

The Abstract Simplification Method

Given: *A normal domain description D, a v-formula E and a structure* (σ,Φ).

Goal: *Checking whether* (σ,Φ) *satisfies* $D \cup \{E\}$.

Step 1: *Computing a least specific explanation S of E wrt D.*

Step 2: *Checking whether S is satisfied in* σ. *If yes then* (σ,Φ) *satisfies* $D \cup \{E\}$. *Otherwise* (σ,Φ) *does not satisfy* $D \cup \{E\}$.

It is not difficult to see that Nicolas's simplification method [N] is a special case of the above abstract simplification method where the e-formulae are either primitive updates or sequential composition of primitive updates. Further, the method employed to compute the least specific explanation by Nicolas can be obtained from the theorem 3 using partial evaluation with unfolding of negative literals, techniques which have been studied extensively in [AD,CW].

Example 4 (Continuation of example 2)

Let D be the domain description given in example 2. Assume that the database has to satisfy the integrity constraint saying that no student can receive both an SERC-grant and a British Council award which is represented by the fluent expression F = ¬Student(x) ∨ ¬Receive(x,SERC) ∨ ¬Receive(x,BC).

Now we want to know whether it is possible to award Mark with a SERC grant. In other words, we have to determine whether the database resulted from the update $\text{insert}_{\text{reiv}}$(Mark,SERC) satisfies the integrity constraint F.

It is not difficult to see that the set consisting of the following init-formulae

initially ¬Student(Mark) ∨ ¬Receive(Mark,BC)
initially ¬Student(x) ∨ ¬Receive(x,SERC) ∨ ¬Receive(x,BC) ∨ (x≠Mark)

is a least specific explanation for the v-formula F after $\text{Insert}_{\text{recv}}$(Mark,SERC). If the initial database state satisfies the integrity constraint F, then the updated database satisfies F iff the initial database state satisfies the first init-formula. This is exactly Nicolas's simplification method
■

Wallace in [W] has developed a meta-level rule-based procedure for checking the integrity of database updates. The difference between our and Wallace's approach

is twofolds. First, our method transforms updating procedures into "equivalent" simple logic programs which have a simple but precise semantics. From this transformation, the application of partial evaluation for checking the integrity of database updates follows immediately and naturally. This is in strong constrast to the meta-level rule-based procedure of Wallace for which no clear and precise semantics has been offered [W]. Further our method is based on a general but yet simple theory for representing actions in logic programming where database updates are only one of many possible applications.

A (compound) update is said to be legal if no integrity constraint is violated. One possible way to determine the legality of a (compound) update is to specify the conditions, called permissible conditions, under which a primitive update can be performed without violating the integrity constraints. It is not difficult to see that such permissible conditions are nothing but a least specific explanation of the integrity interpreted as a v-formula. This points out that the recently proposed approach to semantics of database updates in [R] can be obtained from our through partial evaluation.

Up to now, we have assumed that the fluents are in some ways indepentdent to each other. The value of a fluent can only affected by an action which has a direct effect on this fluent. This is a severe restriction to the expresibility of our language. For example, it is not possible to describe the semantics of view updates of general deductive databases in a relational language. The problem of describing the "indirect" effects of actions is called the ramification problem in the AI literature. In [K2], Kowalski proposed to address this problem by distinguishing between basic fluents and derived fluents where the value of a derived fluent can only be indirectly affected by a change of the basic fluents. This in fact is a simple adaptation of the idea of distinguishing between intentional and extensional predicates in view updates. A formal development of this idea is a topic of future work.

5. Relations to Gelfond and Lifschitz's Approach

To have a framework capable of more general kinds of temporal reasoning other than the temporal projection, Gelfond and Lifschitz [GL2] have proposed the extended logic programming [GL1] as the target language. The reason here is that extended logic programming offers a mechanism to represent "explicit" negation. As we have showed in previous chapters, there is no real need for the use of "explicit" negation to represent actions in logic programming. Further, we will show in this chapter that unfortunately the Gelfond and Lifschitz's translation is not complete and in a lot of cases, it does not capture the intuitive semantics. These shortcommings together with the lack of an appropriate proof procedure makes it difficult to apply this approach in practice.

Remark In this chapter, we will consider only propositional domain description.

Let D be a domain description without similar actions[10]. The corresponding program πD consists of the translation of the individual propositions from D and the four standard rules:

$$Holds(f,[s|a]) \leftarrow Holds(f,s), \text{ not } Ab(f,a,s)$$
$$\neg Holds(f,[s|a]) \leftarrow \neg Holds(f,s), \text{ not } Ab(f,a,s)$$
$$Holds(f,s) \leftarrow Holds(f,[s|a]), \text{ not } Ab(f,a,s)$$
$$\neg Holds(f,s) \leftarrow \neg Holds(f,[s|a]), \text{ not } Ab(f,a,s)$$

A v-proposition F after $A_1,...,A_n$ is translated into

$$Holds(F,[S0|A_1|...|A_n])$$

The translation of an e-proposition A causes F if $P_1,..P_n$ consists of 2n+2 rules:

$$Holds(F,[s|a]) \leftarrow Holds(P_1,s),...,Holds(P_n,s)$$
$$Ab(|F|,a,s) \leftarrow \text{not } \neg Holds(P_1,s),...,\text{not } \neg Holds(P_n,s)$$
$$Holds(P_i,s) \leftarrow \neg Holds(F,s), Holds(F,[s|a])$$
$$\neg Holds(P_i,s) \leftarrow \neg Holds(F,[s|a]), Hold(P_1,s),...,Hold(P_{i-1},s), Hold(P_{i+1},s),...,Hold(P_n,s).$$

where |F| = F and |¬F| = F for each fluent name F.

The following theorem shows the soundness of the translation π.

Theorem 4 [GL2] Let D be a domain description without similar e-propositions. For any v-proposition P, if πD entails πP, then D entails P. ■

Gelfond and Lifschitz have given no results about the completeness of their transformation. The following theorem shows that the Gelfond and Lifschitz's transformation is not complete.

Theorem 5 *The Gelfond and Lifschitz's transformation π is incomplete in the sense that there exists a domain description D and a v-proposition P such that D entails P but πD does not entail πP.*

Proof Let D = { Shoot causes ¬Alive if Alive }. It is clear that D entails P with P = ¬Alive after Shoot. Let Z = { Ab(Alive,Shoot,s) | s = [S0|w] with w ∈ {Shoot}* }. It is not difficult to see that Z is an answer set of πD. It is obvious that πP = ¬Hold(Alive, [S0|Shoot]) does not belong to Z. Thus, πD does not entail πP. q.e.d. ■

---

[10]Two different e-propositions are similar if they differ only by their preconditions.

Another interesting problem which has been left open by Gelfond and Lifschitz in [GL2] is the question of how many answer sets πD has. The authors of [GL2] gave a prediction that under general condition, πD has an unique answer set. The following example shows that in general, there are more than answer sets for πD where only one of them captures the intended semantics.

Theorem 6 *In general, πD can have more than one answer sets where only one of them captures the intended semantics.*

Proof Let D: Initially Alive
Parachute causes ¬Alive if Stormweather

The corresponding program has two answer sets

$Z_1 = \{ \text{Ab(Alive,s)} \mid s = [S0|w], w \in \{\text{Parachute}\}^* \} \cup \{\text{Holds(Alive,S0)}\}$
$Z_2 = \{ \text{Holds(Alive,s)} \mid s = [S0|w], w \in \{\text{Parachute}\}^* \}$
$\cup \{ \neg\text{Holds(Stormweather,s)} \mid s = [S0|w], w \in \{\text{Parachute}\}^* \}$

It is obvious that only Z_1 captures the intended semantics of D for if we don't know anything about the weather, it is impossible to say anything about the outcome of parachuting. Z_2 is completely counterintuitive.
■

Conclusion

We have proposed a simple, sound and complete method for representing actions in logic programming. We have showed that it is posssible in our framework to reason both about the future and about the past with both forms of reasoning: abduction and deduction. We have showed that applying our method to database updates provides an efficient procedure for checking the integrity of database updates without actually performing the updates. Then we compared our method with the recently proposed method of Gelfond and Lifschitz whereby we pointed out the incompleteness of the later. Our method can be viewed as a combination of methods proposed in [GL2,AB,E] where we try to inherit the best from each and at the same time avoid their shortcommings.

To extend our method for more applications, much work remains to be done. Here it is necessary to extend the power of the source language to include parallel as well as nondeterministic actions. Finding a way to represent parallel and nondetermincitic actions in logic programming will be the real challenge for a successful application of logic programming in verifying dynamical systems. Maybe, here works done by Kowalski and Sergot as well as many other researchers [K2,K3,S1,S2] could be of some help. A more concrete extension of our work is to study whether our method could be applied to checking the integrity of updates defined in other language as that of Manchanda and Warren [MS] as well as to dtermine whether other methods for checking the integrity of

database updates [SK] can be modelled within our framework.

Recently, Denecker and De Schreye [DS] has indepentdently proposed a similar but technically somewhat simpler transformation from a propositional source language into a normal logic programming. Denecker and De Schreye's work corresponds roughly to chapter 3 in our paper where the transformation is restricted to propositional domain.

Acknowledgement

This work has been partially supported by the Abduction Group at Imperial College under a grant from Fujitsu.

References

[AB] Apt K., Bezem M. 'Acyclic Programs' in Proc. of 7th ICLP'90

[AD] Aravindan C., Dung P.M. 'Partial deduction of logic programs wrt well-founded semantics' in Proc. 3th Int. Conference on Algebraic and Logic Programming, LNCS 632, 1992, Springer Verlag

[C] Chan D. 'Constructiv negation based on the completed database' in Proc. of 5th ICLP, 1988, MIT press

[CW] Chan D., Wallace M. 'A treatment of negation during partial evaluation' in Meta-programming in Logic Programming (eds.) Abramson H. amd Rogers M.H., 1989, MIT Press.

[CDT] Console L., Dupre D.T., Torasso P. 'On the Relationship between Abduction and Deduction' in J. Logic and Computation, 1991

[DS] Denecker M., De Schreye D. 'Representing incomplete knowledge in abductive logic programming', Draft, personal communication, March 1993.

[EK] Eshghi K., Kowalski R.A. 'Abduction compared with Negation as Failure' in Proc. of 6th ICLP'89

[E] Evans C. 'Negation as failure as an approach to the hanks and McDermott problem' in Proc. of Second Int Symp. on AI, 1989

[GL1] Gelfond M, Lifschitz V. 'Logic Programs with Classical Negations', in Proc. of the ICLP'90

[GL2] Gelfond M., Lifschitz V. 'Representing Actions in Extended Logic Programming', in Proc. of the JICSLP'92

[K1] Kowalski R.A. 'Logic for problem solving', Elsevier North Holland, New York, 1979

[K2] Kowalski R.A. 'Database Updates in Event Calculus', J. of Logic Programming, 1992

[K3] Kowalski R., Sergot M. 'A Logic-based Calculus of Events' in New generation Computing 1986, Vol 4, 67-95

[K4] Kowalski R.A. 'Logic programming in AI' in Proc. of IJCAI'91

[L] Lloyd J.W. 'Foundations of Logic Programming', Second edition, Springer verlag, 1987

[LS] Lloyd J.W., Shepherson J.C. 'Partial evaluation in logic programming' J. of logic programming, 1992

[MW] Manchanda S., Warren D.S. 'A logic-based language for database updates' in Foundations of Deductive databases and Logic Programming' J. Minker (ed.) 1988

[N] Nicolas J.M. 'Logic for improving integrity checking in relational databases', in Acta Informatica, 18:227-253,1982

[R] Reiter R. 'On formalizing database updates: Preliminary report', in Proc. of 3rd International Conf. on Extending Database Technology, 1992

[SK] Sadri F., Kowalski R.A. 'A theorem proving approach to database integrity' in Foundations of Deductive databases and Logic Programming' J. Minker (ed.) 1988

[S1] Shanahan M. 'Prediction is Deduction and Explanation is Abduction' in Proc. of IJCAI'89

[S2] Shanahan M. 'Explanations in the Situation Calculus', in Proc. of IJCAI'93

[W] Wallace M. 'Compiling Integrity Checking into Update Procedure' in Proc. of IJCAI'91