Updates, Induction and Abduction in Deductive Databases

Ch. Vrain and **D. Laurent** [1]

1 Introduction

In this paper, we consider an application of Inductive Logic Programming to a recently introduced model of Datalog^{neg} databases ([12]) in which every insertion and every deletion can be performed in a *deterministic* way.

The problem of updating deductive databases has been widely studied in recent years and can be stated as follows: when inserting (resp. deleting) a fact f over an intensional predicate, the set of extensional facts stored in the database must be modified so that f can be deduced (resp. cannot be deduced) from the updated database. This way of performing updates, based on abduction, has been considered in many approaches (see [3], [4], [10]).

Unfortunately, in many cases, there are several ways to modify the set of stored facts, and this is known as the *non-determinism* of intensional updates (see [3], [12]). In these cases, abduction is used in order to compute all the possibilities and it is up to the user (or the system) to choose among them; when no choice is done the update is not performed. This shows that, in a sense, in these approaches, "old data" have priority over "new data."

For example, let D be a database containing the following rules:

$$\text{RULES}: \quad \begin{array}{ll} anc(x,y) & \leftarrow par(x,y) \\ anc(x,y) & \leftarrow par(x,z), anc(z,y) \end{array}$$

Inserting the intensional fact $anc(b,d)$ by modifying only the set of extensional facts can be performed either by inserting $par(b,d)$ or by choosing n constants $\alpha_1, \ldots, \alpha_n$ and inserting the facts $par(b,\alpha_1)$, $par(\alpha_1,\alpha_2), \ldots, par(\alpha_{n-1},\alpha_n)$, $par(\alpha_n,d)$.

Recent approaches to database updating ([11], [12], [13]) do not consider updating through abduction and consider a radically different method. Indeed, in [13], as well as in [11] and [12], instead of "pushing down" intensional updates to extensional updates, the authors propose to keep track of every insertion and every deletion. Since in these methods intensional updates are not pushed from the head to the body of the rules, this results in deterministic approaches to updating. We now give the main ideas of the approach of [12].

According to [12], the cases of non-determinism are overcome as follows: given an intensional or extensional fact f, the insertion (resp. the deletion) of f is "stored" in the database by means of the positive (resp. negative) ground literal f (resp. $\neg f$). As a consequence, the database consists of a set of rules and of a consistent set of positive or negative ground literals $\mathcal{L}$.

In order to insert a fact f while preserving the consistency of $\mathcal{L}$, we simply store f in $\mathcal{L}$ and we eventually remove $\neg f$ from $\mathcal{L}$. Similarly, in order to delete a fact f while preserving the consistency of $\mathcal{L}$, we simply store $\neg f$ in $\mathcal{L}$ and we eventually remove f from $\mathcal{L}$. Moreover, the literals of $\mathcal{L}$ can derive other facts, using the rules.

It is important to note in this respect that, in order to consider only *consistent* derivations, negative literals of $\mathcal{L}$ are seen as *exceptions* to the rules, as shown in the following example.

Example 1. Let $D = (\mathcal{L}, \text{RULES})$ be a database where:

$$\begin{array}{l} \mathcal{L}^+ = \{par(a,b), par(b,c), par(c,d), par(e,f), par(f,d)\}, \\ \mathcal{L}^- = \{anc(a,c),\ anc(e,b),\ anc(e,c)\} \end{array}$$

$$\text{RULES}: \quad \begin{array}{l} anc(x,y) \leftarrow par(x,y) \\ anc(x,y) \leftarrow par(x,z), anc(z,y) \end{array}$$

If we insert the fact $anc(a,c)$ in D then this fact moves from $\mathcal{L}^-$ to $\mathcal{L}^+$. Deleting $anc(a,d)$ from the database obtained after the insertion results in the following:

$$\begin{array}{l} \mathcal{L}^+ = \{par(a,b), par(b,c), par(c,d), par(e,f), par(f,d), \\ \qquad anc(a,c)\}, \\ \mathcal{L}^- = \{anc(e,b),\ anc(e,c),\ anc(a,d)\} \end{array}$$

We note that the deleted fact $anc(a,d)$ is an exception to the second rule, since the facts $par(a,b)$, $par(b,c)$ and $par(c,d)$ are true. □

It is important to note that, contrary to approaches to updating through abduction, in our approach, "new data" are given priority over "old data", i.e., updates are *always* performed in such a way that no choice is required. However, although this approach correctly solves the problem of deterministic updates, it has the following drawbacks:

1. In some cases, abduction may be required.
2. The storage of inserted and deleted facts implies an overhead.
3. Rules may have so many exceptions that they become irrelevant.

The first point above is addressed in this paper in Section 4. We show that extending the model just introduced by special rules, called *update rules,* allows for performing abduction in a deterministic way when updating databases. The third point has been ignored in the past whereas the second one has been addressed in [12] in terms of storage optimization. However, as shown in [12], in the context of Datalog^{neg} databases, there are few cases where the storage of the fact to be inserted or deleted can be avoided.

We propose to cope with points 2 and 3 above, using *induction,* as follows: updates are processed as in [12], and when the database contains too many facts over intensional predicates (inserted or deleted), we automatically "learn" new rules, based on the semantics of the database. After the learning phase, the resulting database is intended to:

- have a semantics containing the semantics of the original database,
- contain new rules reflecting "better" the "past experience,"
- contain *fewer* facts than the original database, and, in the best case,
- contain *only* inserted facts over extensional predicates, and no deleted facts.

[1] LIFO, Université d'Orléans, BP 6759, 45067 Orléans Cedex 02 (France) email: {cv,laurent}@lifo.univ-orleans.fr

In our example, we learn the following rules:

$$anc(x,y) \leftarrow \quad par(x,y)$$
$$anc(x,y) \leftarrow \quad par(x,z), par(z,y)$$

This means that the database has evolved in such a way that the facts about the predicate $ancestor$ correspond actually to the predicate $grand\_father$. As a result, with these new rules at hand, the storage of the facts in $\mathcal{L}^-$ is no longer required.

In previous works on updating databases and theory revision, interactive systems as the system Clint [18] were used. In the learning phase, the system MULT_ICN of [15] seems to be appropriate for the following two main reasons:

1. Interactive systems are not adapted to our approach, since for sake of efficiency, rules are not learned at each update. Indeed, rules are changed only when the number of exceptions is high enough to guess that the rules are no more relevant.
2. The underlying semantics of MULT_ICN is the well-founded semantics ([21]), which is precisely the semantics the approach of [12] is based on.

When applying empirical learning, the new rules may be substantially different from the rules as originally specified during the design of the database. In order to maintain the rules as close as possible to the original ones, we take into account the dependency graph during the learning phase.

Nevertheless, MULT_ICN is based on the notions of extensional coverage of positive examples and of extensional rejection of negative examples. When the specification of the examples and/or of the background knowledge is partial, rejecting the negative examples can be impossible. We address in Section 3 the problem of using abduction to complete the set of examples and/or the knowledge base.

2 Logic Programming

A Datalog^{neg} database consists in a set of clauses containing no function symbols other than constants. For the sake of simplicity, if p is a predicate symbol of arity n and if $t_1, t_2, \ldots, t_n$ are terms, then the formula $p(t_1, t_2, \ldots, t_n)$ is simply written as $p(\vec{t})$. A predicate is called *intensional* (w.r.t. $\mathcal{P}$) if it appears in the head of at least one rule, and every other predicate is called *extensional* (w.r.t. $\mathcal{P}$) ([20], [5]). Given a program $\mathcal{P}$, the set of intensional predicates is denoted by I_PRED, and the set of extensional predicates is denoted by E_PRED.

The Herbrand base associated to a program $\mathcal{P}$, denoted by HB, is the set of all facts that can be obtained from predicate symbols and constant symbols appearing in the program. The set of all ground instances of $\mathcal{P}$, denoted by $Inst(\mathcal{P})$, is the set of all ground clauses that can be obtained by replacing all variables in $\mathcal{P}$ with constants.

A set Int of ground literals can be decomposed into two sets of facts, $Int = Int^+ \cup \neg.Int^-$, where $Int^+ = \{f \in HB \mid f \in Int\}$ and $Int^- = \{f \in HB \mid \neg f \in Int\}$. A *partial interpretation* Int is a consistent set of ground literals, i.e., $Int = Int^+ \cup \neg.Int^-$ with $Int^+ \cap Int^- = \emptyset$.

Let f be a fact. If l is the positive literal f (resp. the negative literal $\neg f$) then $-l = \neg f$ (respectively $-l = f$).

The well-founded semantics of a program $\mathcal{P}$ ([21]) is defined by two operators, denoted by T and U, which compute true facts and false facts, respectively. These operators are defined as follows: let Int be a partial interpretation,

- $T(Int)$ is the set of the facts f for which there exists in $Inst(\mathcal{P})$ a rule $f \leftarrow l_1, \ldots, l_k$ such that $l_i \in Int$, for every $i = 1, \ldots, k$.
- U is the unfounded operator: $U(Int)$ is the union of all sets Unf which satisfy the following: for every fact f in Unf and for every rule $f \leftarrow l_1, \ldots, l_k$ in $Inst(\mathcal{P})$,
$$\exists i \in \{1, \ldots, k\} : -l_i \in Int \text{ or } l_i \in Unf.$$

As shown in [21], $T \cup \neg.U$ is a monotonic operator w.r.t. set inclusion over partial interpretations. Its least fixpoint is a partial interpretation called the *well-founded model* of $\mathcal{P}$. We denote this model by $\mathcal{M}(\mathcal{P}) = \mathcal{M}^+(\mathcal{P}) \cup \neg.\mathcal{M}^-(\mathcal{P})$, facts in $\mathcal{M}^+(\mathcal{P})$ are said to be *true* and facts in $\mathcal{M}^-(\mathcal{P})$ are said to be *false* (w.r.t. $\mathcal{P}$).

Moreover, let us consider a program $\mathcal{P}$ that can be decomposed into two programs $\mathcal{P}_1$ and $\mathcal{P}_2$, where $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) defines a set LP_1 (resp. LP_2) of predicates with $LP_1 \cap LP_2 = \emptyset$. When $\mathcal{P}_1$ is built only with predicates of LP_1, we have defined in [22] the semantics of $\mathcal{P}_2$ w.r.t. the semantics of $\mathcal{P}_1$, written $\mathcal{M}(\mathcal{P}_2, \mathcal{M}(\mathcal{P}_1))$, and we have shown that:

$$\mathcal{M}(\mathcal{P}_2, \mathcal{M}(\mathcal{P}_1)) \cup \mathcal{M}(\mathcal{P}_1) = \mathcal{M}(\mathcal{P}_2 \cup \mathcal{P}_1)$$

3 Abduction during the Learning Phase

The system MULT_ICN is a multiple predicate empirical learning system from positive and negative examples and background knowledge. Its aim is to learn a logic program defining a set TP of target predicates which are specified by a set E of ground literals: E can be decomposed into $E^+ \cup \neg.E^-$, where E^+ is the set of positive examples and E^- is the set of negative examples. Knowledge about a set BP of basic predicates, satisfying BP $\cap$ TP $= \emptyset$ is also provided to the system. It is specified either by a program, BK or by its model, $\mathcal{M}(\text{BK})$.

The system MULT_ICN learns general logic programs, i.e., sets of rules defining the target predicates p_k, $p_k(\vec{x}) \leftarrow l_1, \ldots, l_p$ where l_i is:

- either a literal over a predicate of TP$\cup$BP containing only variables, among which at least one belongs to the head of the rule or to some $l_j, j < i$,
- or a literal $y_u = y_v$ or $y_u \neq y_v$ where y_u and y_v are variables in the head of the rule or in a literal $l_j, j < i$.

Building Rules – The process to build rules relies on the notion of extensional coverage and of extensional rejection, defined in a three valued logic as follows:

Definition 1. *Let $\mathcal{P}$ be a set of rules defining the target predicates, let E be the set of examples, let* BK *be a set of rules and/or facts defining the basic predicates and let e be a fact.*

- $\mathcal{P}$ *extensionally covers* e *if there exists a rule* $l \leftarrow l_1, \ldots, l_n$ *in* $Inst(\mathcal{P})$ *such that* $e = l$ *and, for every* $i = 1, \ldots, n$, $l_i \in E \cup \mathcal{M}(\text{BK})$.
- $\mathcal{P}$ *extensionally rejects* e *if, for all rules* $l \leftarrow l_1, \ldots, l_n$ *in* $Inst(\mathcal{P})$ *such that* $e = l$*, there exists a literal* l_i $(i = 1, \ldots, n)$ *satisfying* $-l_i \in (E \cup \mathcal{M}(\text{BK}))$. □

The general algorithm of MULT_ICN can then be outlined as follows: *as long as some positive examples are not covered, build a rule that extensionally covers some positive examples and extensionally rejects all the negative ones.*

This notion is not sufficient to ensure that the learned program is complete and consistent. For instance, if q is the target predicate, the program composed of the clause $q(\vec{x}) \leftarrow q(\vec{x})$ extensionally covers the positive examples and rejects the negative ones, but all ground atoms are false w.r.t. the well-founded semantics. To deal with this problem, the notion of recursive dependencies has been introduced in [14] where it is shown that the programs $\mathcal{P}$ built by MULT_ICN are complete and consistent, i.e., we have

$$E^+ \cup \neg.E^- \subseteq \mathcal{M}(\mathcal{P} \cup \text{BK}).$$

Non extensional coverage versus extensional rejection – Most empirical systems ([16, 17]) search for a program that covers no negative examples whereas MULT_ICN searches for a program that rejects all negative examples. When the specifications of the background knowledge BK and/or of the examples are partial, this is no longer equivalent.

Example 2. Let us consider the domain $D = \{0, 1, 2, 3, 4\}$ and:

$$\begin{aligned} E =& \{even(0), \neg even(1), \neg even(3)\} \\ & \text{where } even(2) \text{ and } even(4) \text{ are undefined,} \\ \mathcal{M}^+(\text{BK}) =& \{succ(0,1), succ(1,2), succ(2,3), succ(3,4)\}, \\ \mathcal{M}^-(\text{BK}) =& \overline{\mathcal{M}^+(\text{BK})}. \end{aligned}$$

The program $\mathcal{P}_1$ composed of the clause:

$even(X) \leftarrow succ(X, Y), \neg even(Y)$

covers no negative examples but it is not correct, since its well-founded semantics is $\{even(1),\ even(3),\ \neg even(0),\ \neg even(2),\ \neg even(4)\}$. On the other hand, $\mathcal{P}_1$ does not reject $even(3)$ and therefore it is not acceptable for MULT_ICN. □

This shows that non extensional coverage and extensional rejection are not equivalent and that the constraint put in the system MULT_ICN, that is to say all negative examples must be extensionally rejected by the program, is stronger than in the other systems. The notion of extensional rejection linked with the notion of recursive dependencies provides us with an efficient way of validating the learned program. Nevertheless, the class of programs that can be learned is more restricted than in the classical approaches. When the specification is too partial, it could happen that there exists no programs that extensionally cover the positive examples and reject the negative ones. To deal with this problem, different solutions could be adopted:

- relaxing the constraint and accepting programs that cover no negative examples and that eventually do not reject all the negative ones,
- learning first non recursive clauses, as done in the system SKIL [9],
- using abduction to propose new facts during the construction of the program.

The second solution is not applicable, when learning normal programs, since the process of adding new clauses is no longer monotonic.

When using the first solution, the efficient way of checking the correction of the learned program is lost. Consequently, this can be acceptable only if after learning, we propose, by means of abduction, new facts that will enable to reject all negative examples.

We now address the problem of using abduction to complete the knowledge base during the construction of the learned program. This can be useful either to enable to cover positive examples or to enable to reject negative ones. A similar idea had been developed in [18] in the case of interactive learning of definite programs.

Example 3. Let us first consider only definite programs. In this case, the well-founded semantics of a program $\mathcal{P}$ corresponds to the classical semantics of definite logic programs, defined as the least fixpoint of the immediate consequence operator $T_{\mathcal{P}}$, under the closed world assumption.

Let us consider the domain $D = \{a, b, c\}$ and:

$$\begin{aligned} E =& \{p(a), p(c), \neg p(d)\} \\ & \text{where } p(b) \text{ is undefined,} \\ \mathcal{M}^+(\text{BK}) =& \{r(a,c), r(c,b), r(d,b), q(b)\}, \\ \mathcal{M}^-(\text{BK}) =& \overline{\mathcal{M}^+(\text{BK})} - \{q(a), r(c)\}, \\ & \text{where } q(a) \text{ and } r(c) \text{ are undefined.} \end{aligned}$$

The first problem here is that usually, when applying abduction, the logic program defining intensional predicates is fixed, whereas in our case, it is not yet built. Let us suppose that for instance, in order to solve this learning problem, we build the clause:

$C_1 : p(X) \leftarrow q(X)$

This clause rejects the negative example $p(d)$ but it does not cover the positive example, $p(a)$. In order to achieve this, we can make the assumption that $q(a)$ is true. Let us then suppose that we accept this clause with the assumption that $q(a)$ is true. This will imply that $p(b)$ is true (since $q(b)$ is true). A new clause must now be built in order to cover $p(c)$. Let us consider the clause:

$C_2 : p(X) \leftarrow r(X, Y), p(Y)$

It extensionally covers $p(c)$, but unfortunately it also covers $p(d)$, which is a negative example. Consequently, this program is not acceptable.

Let us now suppose that instead of building the first clause C_1, we build the clause:

$C_3 : p(X) \leftarrow r(X)$

This clause rejects the negative example $p(d)$ but it covers no positive examples. It can cover $p(c)$ if we make the assumption that $r(c)$ is true. Let us then suppose that we accept C_3 with the assumption that $r(c)$ is true. A new clause must now be built in order to cover $p(c)$. Let us consider again the clause C_2 above.

This clause extensionally covers $p(a)$, but it does not extensionally rejects $p(d)$. In order to achieve this, we can make the assumption that $p(b)$ is false. The program composed of the clauses C_2 and C_3 extensionally covers the positive examples and rejects the negative ones under the hypothesis that $r(c)$ is true and $p(b)$ is false.

The atom $p(b)$ is built over the target predicate p and therefore we have to check that the semantics of the learned program with respect to $\mathcal{M}(\text{BK}) \cup \{r(c)\}$ is correct for $p(b)$. □

This example shows that:

a- When abduction is interleaved with the learning process, the assumptions that are made after the construction of a clause can lead to a deadlock for learning a complete and consistent program. To cope with this problem, we should keep several interesting clauses with their corresponding abduced hypothesis.

b- The abduction can be processed over facts built with basic predicates or with target predicates. In this latter case, this gives us new examples and the semantics of the learned program must be correct with respect to these new examples.

Example 4. Let us now consider the domain $D = \{a, b, c, d, e\}$ and:

$$\begin{aligned} E =& \{p(a), p(c), p(d), \neg p(e)\} \\ & \text{where } p(b) \text{ is undefined,} \\ \mathcal{M}^+(\text{BK}) =& \{r(a,c), r(c,b), r(d,e)\}, \\ \mathcal{M}^-(\text{BK}) =& \overline{\mathcal{M}^+(\text{BK})}. \end{aligned}$$

Let us suppose that we build the first clause:

$p(X) \leftarrow r(X, Y), \neg p(Y)$

This clause covers the positive example $p(d)$ and extensionally rejects $p(e)$. It could enable to cover $p(c)$ if we make the assumption that

$p(b)$ is false. As performing abduction entails choices that could later lead to a deadlock, it seems preferable to postpone the abduction process and to try to build a new rule that covers $p(d)$. □

Example 5. Let us consider the domain $D = \{a, b, c, d\}$ and:

$$\begin{aligned} E =& \{p(a), \neg p(c), p(d)\} \\ & \text{where } p(b) \text{ is undefined,} \\ \mathcal{M}^{+}(\text{BK}) =& \{\underline{r(a,c), r(c,b)}, s(d,a), s(c,b), s(a,b)\}, \\ \mathcal{M}^{-}(\text{BK}) =& \overline{\mathcal{M}^{+}(\text{BK})}. \end{aligned}$$

Let us suppose that we build the first clause:
$p(X) \leftarrow r(X,Y), \neg p(Y)$

This clause covers the positive example $p(a)$ but it does not extensionally reject $p(c)$. In order to achieve this, we have to make the assumption that $p(b)$ is true.

Let us now consider the second rule:
$p(X) \leftarrow s(X,Y), p(Y)$

This rule covers $p(d)$. In order to extensionally reject $p(c)$ we have to make the assumption that $p(b)$ is false, which is not consistent with the previous assumption that $p(b)$ is true. □

This example shows an important difference between learning definite logic programs and learning normal logic programs. In the first case, making the assumption that a fact should be true (respectively false) enables to enlarge the positive (respectively negative) part of the semantics of the program, and this is therefore useful to cover (respectively reject) more positive (respectively negative) examples. On the other hand, when learning normal logic programs, this is no longer true: positive or negative facts can enable both to cover positive examples or to reject negative ones.

To sum up:

- When it is not possible to build a rule that extensionally covers some positive examples, abduction can be performed to suggest the truthvalue of facts, that were previously undefined and that will enable to cover more positive examples.
- When it is not possible to build a rule that rejects all the negative examples, abduction can be performed in order to reject the remaining negative examples.
- Using abduction during the learning process can lead to further deadlocks. Therefore, it seems advisable to keep several rules with their corresponding hypotheses.
- The number of rules that reject all the negative examples but cover no positive examples or that cover some positive examples but that do not reject all the negative ones can be large. Therefore, some criteria must be developed to keep only the "most interesting ones".
- When assumptions are made about undefined ground atoms built over target predicates, these atoms become new positive or negative examples. Positive examples must be covered by some clauses. On the other hand, previously built clauses must also reject the new negative examples.
- Finally, let us recall that the notions of extensional coverage of the positive examples and of extensional rejection of the negative one are not sufficient to ensure the correctness of the learned programs. It should be interesting to study how to perform abduction according to the well-founded semantics of the learned program (and no longer according to the notions of coverage and of rejection).

4 Abduction through Update Rules

Although updating without using abduction, as described in the introduction, allows to deterministically update deductive databases, there are many cases where it is appropriate to "push down" updates, i.e., to use abduction.

Consider for example a database containing the facts

$$penguin(tweete),\ whale(jumbo)$$

and the following three rules:

$$\begin{aligned} fly(x) &\leftarrow bird(x) \\ bird(x) &\leftarrow penguin(x) \\ mammal(x) &\leftarrow whale(x) \end{aligned}$$

Clearly, this database derives the two facts $fly(tweete)$ and $mammal(jumbo)$.

Consider now the deletion of the facts $fly(tweete)$ and $mammal(jumbo)$. According to our approach, the resulting database consists in the three rules above together with the set of instantiated literals

$$\mathcal{L} = \{penguin(tweete),\ whale(jumbo),\ \neg fly(tweete),\ \neg mammal(jumbo)\}.$$

Thus, we have that:

1. $tweete$ is a bird that does not fly, and
2. $jumbo$ is a whale that is not a mammal.

It is well-known that point one above is actually acceptable (i.e., there are exceptions to the first rule), whereas the second is not (i.e., there are *no* exceptions to the third rule).

Thus, it turns out that, in some cases, abduction is not suitable, whereas it is expected in some other cases. In order to cope with this difficulty, the approach to updates of [12] has been extended in [8] as follows: in addition to ground literals and to standard Datalog^{neg} rules called *query rules,* special rules, called *update rules,* are considered in the database.

The role of update rules is to compute the *side effects* of a given update. Syntactically, these rules are of the form $L_0 \leftarrow L_1$, where L_0 and L_1 are positive or *negative* literals. In the example above, let us consider the following update rule:

$$\neg whale(x) \leftarrow \neg mammal(x)$$

stating that, when deleting a fact such as $mammal(a)$, then $whale(a)$ must be false. Thus, update rules can be seen as a way to specify in which case and how abduction has to be performed when updating the database.

In the framework of update rules, the database Δ of our previous example would be defined as a triple $(\mathcal{L}, UR, QR)$ where $\mathcal{L} = \{penguin(tweete),\ whale(jumbo)\}$ and where:

$$\begin{aligned} UR: \quad \neg whale(x) &\leftarrow \neg mammal(x) \\ \\ QR: \quad fly(x) &\leftarrow bird(x) \\ bird(x) &\leftarrow penguin(x) \\ mammal(x) &\leftarrow whale(x) \end{aligned}$$

In this case, the deletion of the fact $fly(tweete)$ is performed as above, since no update rule is triggered. But, when deleting $mammal(jumbo)$, the update rule in UR is triggered, entailing that $whale(jumbo)$ must now be false. In this case, in order to be consistent with this conclusion, the fact $whale(jumbo)$ is removed from $\mathcal{L}$.

As a consequence, after these two deletions, the resulting database $\Delta' = (\mathcal{L}', UR, QR)$ is defined by

$$\mathcal{L}' = \{penguin(tweete),\ \neg fly(tweete), \neg mammal(jumbo)\}.$$

The semantics of such a database is computed as follows:

1. Apply recursively the update rules to the literals in $\mathcal{L}$, call $\xi_{\mathcal{L}}$ the result.
2. Apply the rules in QR to the literals of $\xi_{\mathcal{L}}$, as shown in the previous section.

In our example, we obtain:

1. $\xi_{\mathcal{L}} = \{penguin(tweete), \neg fly(tweete), \neg mammal(jumbo), \neg whale(jumbo)\}$.
2. Thus, the rules in QR allow for the derivation of the only fact $bird(tweete)$. Indeed, the third rule does no longer apply, due to the update rule which performs abduction, and $\neg fly(tweete)$ is an exception to the first rule, to which no abduction is applied.

As a consequence, the semantics of the updated database is:

$$\{penguin(tweete),\ bird(tweete),\ \neg fly(tweete),\ \neg mammal(jumbo),\ \neg whale(jumbo)\},$$

which is the expected result.

We now describe more formally how exceptions are computed and how updates are performed in the presence of update rules.

As mentioned earlier, a database Δ is composed by a consistent set of ground literals, $\mathcal{L}$, and by two sets of rules: a set of update rules, usually denoted by UR, and a set of query rules, usually denoted by QR. Query rules are standard Datalog^{neg} rules, whereas update rules are rules of the form $L_0 \leftarrow L_1$, where L_0 and L_1 are literals.

Given a set of update rules UR, we denote by $Inst(UR)$ the set of all instantiations of the rules of UR, using the constants of the underlying Herbrand universe, and we consider two operators, ξ and ϑ, defined as follows: for every partial interpretation Int,

$$\begin{aligned} \xi(Int) =& Int \cup \{head(r) \mid r \in Inst(UR) \wedge body(r) \in Int\} \\ \vartheta(Int) =& Int \cup \{body(r) \mid r \in Inst(UR) \wedge head(r) \in Int\}. \end{aligned}$$

Roughly, the operator ξ can be seen as the immediate consequence operator associated with the rules in UR, whereas ϑ works on update rules in the "opposite direction", that is, from the head to the body. In other words, ξ performs *deduction* whereas ϑ performs *abduction* on update rules.

As shown in [8], ξ and ϑ are monotonic, and thus, each of the sequences:

$$\begin{cases} \xi^0(Int) = & Int \\ \xi^k(Int) = & \xi(\xi^{k-1}(Int)), \ \text{ for } k > 0, \end{cases}$$

$$\begin{cases} \vartheta^0(Int) = & Int \\ \vartheta^k(Int) = & \vartheta(\vartheta^{k-1}(Int)), \ \text{ for } k > 0, \end{cases}$$

has a limit, denoted by $lfp(\xi, Int)$ or ξ_{Int} and by $lfp(\vartheta, Int)$ or ϑ_{Int}, respectively.

Definition 2 - Consistent Set of Update Rules. *A set UR of update rules is* consistent *if for every ground literal L, the set ξ_L is a partial interpretation.* □

Definition 3 - Database. *A* database *is a triple $\Delta = (\mathcal{L}, UR, QR)$ where:*
– $\mathcal{L}$ is a partial interpretation,
– UR is a consistent set of update rules,
– QR is a set of usual Datalog^{neg} rules. □

Definition 4 - Consistent Database. *A database $\Delta = (\mathcal{L}, UR, QR)$ is said to be consistent if $\xi_{\mathcal{L}}$ is a partial interpretation. The set $\xi_{\mathcal{L}}$ is called* the exception set. □

As explained earlier, the semantics of a consistent database $\Delta = (\mathcal{L}, UR, QR)$ is computed as follows:

1. compute first the exception set $\xi_{\mathcal{L}}$, then
2. apply the query rules to the result of step 1 above.

Step 2 is performed in our approach by using the well founded semantics. However, since this computation may lead to contradictions with respect to step 1, we define two operators T^* and U^* as follows ([8]): Let $\xi_{\mathcal{L}} = \xi^+_{\mathcal{L}} \cup \neg.\xi^-_{\mathcal{L}}$, and let T and U be the well founded operators associated to the program consisting of the facts in $\xi^+_{\mathcal{L}}$ and of the rules in QR. For every partial interpretation Int, we define:

$$T^*(Int) = T(Int) \setminus \xi^-_{\mathcal{L}} \qquad \text{and} \qquad U^*(Int) = U(Int) \cup \xi^-_{\mathcal{L}}.$$

As shown in [8], T^* and U^* as defined above are monotonic and thus, the sequence $[M_i]_{i \geq 0}$ defined by:

$$\begin{cases} M_0 = & \emptyset\ , \text{ and} \\ M_i = & T^*(M_{i-1}) \cup \neg.U^*(M_{i-1}), \ \text{for } i > 0 \end{cases}$$

has a limit. This limit is a partial interpretation, which is called the *model of* Δ, and is denoted by: $M(\Delta) = M^+(\Delta) \cup \neg.M^-(\Delta)$. Moreover, it is easily seen that $\xi_{\mathcal{L}} \subseteq M(\Delta)$.

As in the case where no update rules are present, we consider that updating a database means inserting or deleting a fact, while leaving the database consistent. More precisely, let f be a fact, call l the literal f or $\neg f$ following that f is inserted or deleted. Then, updates are performed as follows:

- *Step* 1 : Place the literal l in $\mathcal{L}$.
- *Step* 2 : Remove from $\mathcal{L}$ every literal whose consequences through update rules generate a contradiction to l or to any of its consequences through update rules.

In order to perform Step 2 above, abduction is performed as follows: starting with the literal l to be put in the database, the set ξ_l of all its consequences through update rules is computed, and then, we perform abduction (i.e., we apply the operator ϑ) starting with the interpretation $-\xi_l = \xi^-_l \cup \neg.\xi^+_l$ in order to compute all literals that are in contradiction to the introduction of l in the database. This is precisely what is stated in the definition below.

Definition 5 - Database Update. *Let $\Delta = (\mathcal{L}, UR, QR)$ be a database and let l be a literal. Define the* update *of l in Δ, denoted by $upd(l, \Delta)$, to be a database $(\mathcal{L}', UR, QR)$ defined by: $\mathcal{L}' = \big(\mathcal{L} \setminus lfp(\vartheta, -\xi_l)\big) \cup \{l\}$.* □

We note that, in our model, *two* kinds of negation are present: negation in update rules, which corresponds to classical negation, and negation in query rules, which corresponds to the negation of the well founded semantics. In this respect, we make the following assumption, in the spirit of [2, 7]: for every atom A if $\neg A$ is derived through the update rules, then $\neg A$ still holds with respect to the query rules. We refer to [8] for more details on this topic.

What is important to note with respect to negation is that update rules can be seen as way to perform abduction and that the negation associated to update rules is *not* negation as failure. So, it seems that, at least as far as updates are concerned, the semantics of negation associated to abduction is that of classical negation, instead of negation as failure, as usual in deduction.

Summarizing the relationship between abduction and update rules as defined in this paper, we argue that:

1. Update rules are an appropriate tool to specify abduction with standard Datalog^{neg} rules that we have called query rules.
2. When updating databases with update rules, abduction is also necessary with these rules, in order to maintain the database consistent.
3. The distinction between update rules and query rules reveals that two types of negation are used and that abduction seems to be related to classical negation, whereas, as seen previously, when performing induction, the well-founded negation is used.

5 Position with respect to the Workshop Focus

We have considered a deductive database model which allows to perform insertions and deletions of extensional or intensional facts in a deterministic manner, but at the cost of an overhead with respect to traditional approaches. We recall that:

1. Our method of updating is *not* systematically based on abduction, but rather, on the notion of exception.
2. Abduction is used, only when explicitely specified by means of special rules, called update rules.
3. Inductive techniques are used to generate new rules that enable to decrease the number of exceptions.
4. Our method of updating and of learning rules is based on a three valued semantics which is basically the well-founded semantics ([21]).
5. No interaction with the user is required during update processing as well as during learning new rules.

A problem with our approach is that the learning phase processed by MULT_ICN may fail, because, in the model of the database, too many atoms are unknown. In this case, we believe that, using abduction, it should be possible to "complete" the model of the database in such a way that learning becomes possible. We note that, by doing so, it may happen that interaction with the user be required in order to choose among the hypotheses generated through abduction.

Most learning systems are based either on two valued logics (true-false) [18, 17, 16] or on four valued logics (true-false-undefined-inconsistent) [23, 18]. In this work, we have considered a three valued logics (true-false-undefined), and so it should be interesting to study the impact of the semantics on abduction.

On the other hand, note that MULT_ICN learns general logic programs, i.e., programs with positive and negative literals in the bodies of the rules. In some works, negation is dealt with through the introduction of new predicates representing the negative parts of the predicates in the underlying alphabet. This enables to treat negation with definite programs in which abduction can be applied. In our work, however, we do not treat negation in this way, and therefore, we intend to study:

1. How to perform abduction in the context of three valued semantics such as the well-founded semantics or Fitting semantics [6]. One of the main points in this respect seems to be underlying structures of proof-trees.
2. Learning is usually performed by adding (resp. deleting) rules or facts in a given program, so as the semantics of the program increases (resp. decreases). In presence of negation, this is no longer true, and so, new criteria to control the impact of the introduction of new hypotheses should be developed.

REFERENCES

[1] Abiteboul S., 1988. Updates, a New Frontier, Intl. Conf. on Database Theory ICDT, *LNCS*, 326, Springer-Verlag.

[2] Alferes J.J., Pereira L.M., 1992. On Logic of program semantics with two kinds of negation. Joint Intl. Conf. and Symp. of Logic Programming.

[3] Atzeni P., Torlone R., 1992. Updating Intensional Predicates in Datalog, *Data & Knowledge Engineering*, 8.

[4] Bry F., 1990. Intensional Updates: Abduction via Deduction, Intl. Symposium on Logic Programming.

[5] Ceri S., Gottlob G., Tanca L., 1990. *Logic Programming and Databases,* Surveys in Computer Science, Springer-Verlag.

[6] Fitting M., 1985. A Kripke-Kleene semantics for logic programs. Journal of Logic Programming, 2(4).

[7] Gelfond M., Lifschitz, V., 1990. Logic Programming with Classical Negation. Seventh Intl. Conf. of Logic Programming.

[8] Halfeld Ferrari Alves M., Laurent D., Spyratos N., 1995. Update Rules in Datalog Programs, *Intl. Conf. on Logic Programming and Non-Monotonic Reasoning, LNAI,* 928, Springer-Verlag.

[9] Jorge A., Brazdil P., 1996. Architecture for Iterative Learning of Recursive Learning Definitions. Advances in Inductive Logic Programming. L. de Raedt (Ed.), IOS Press (1996) pp. 206-218.

[10] A.C. Kakas, P. Mancarella, 1990. Database Updates through Abduction, 16th Intl. Conf. on Very Large Databases VLDB, Brisbane (Australia).

[11] Laurent D., Phan Luong V., Spyratos N., 1992. Deleted Tuples are Useful when Updating through Universal Scheme Interfaces, 8th *IEEE Intl. Conf. ICDE.*

[12] Laurent D., Phan Luong V., Spyratos N., 1993. Updating Intensional Predicates in Deductive Databases, 9th *IEEE Intl. Conf. ICDE.*

[13] Leone N., Palopoli L., Romeo M., 1995. A Language for Updating Logic Programs and its Implementation. *Journal of Logic Programming,* vol. 23, number 1, pp. 1-61.

[14] Martin L., Vrain C.: A Three-Valued Framework for the Induction of General Logic Programs. Advances in Inductive Logic Programming. L. de Raedt (Ed.), IOS Press (1996) pp. 219-235.

[15] Martin L., Vrain C., 1995. MULT_ICN : an empirical multiple predicate learner. *Fifth Intl ILP Workshop*, L. De Raedt (Ed.), pp. 129-144.

[16] Muggleton S., Feng C., 1992. Efficient Induction of Logic Programs. Inductive Logic programming. The A.P.I.C. Series N^o 38, S. Muggleton (Ed.), Academic Press.

[17] Quinlan J.R., 1990. Learning Logical Definitions from Relations. *Machine Learning Journal*, Vol. 5, Kluwer Academic Publishers.

[18] de Raedt L., 1992. Interactive theory revision, an inductive logic programming approach. Academic Press Limited.

[19] de Raedt L., Bruynooghe M., 1992. Belief Updating from Integrity Constraints and Queries, Artificial Intelligence, 53:291-307.

[20] Ullman J.D., 1989. *Principles of Databases and Knowledge Base Systems,* Vol. I-II, Comp. Sc. Press.

[21] Van Gelder A., Ross K.A., Schlipf J.S., 1991. The well-founded Semantics for General Logic Program. Journal of the ACM, 38(3).

[22] Vrain C., Laurent D., 1995. Apprentissage de règles et Bases de données déductives (French). Technical Report 95-5, LIFO, Université d'Orléans.

[23] Wrobel S., 1994. *Concept Formation and Knowledge Revision,* Kluwer Academic Publishers.