INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

# Abduction for Knowledge Assimilation in Deductive Databases

**Hendrik Decker**

appeared in R. Baeza-Yates (ed), *Proc. SCCC'97*, Valparaiso (Chile), IEEE Press, 1997
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1997-13, September 1997

# Abduction for knowledge assimilation in deductive databases

Hendrik Decker
Institut für Informatik, Universität München
hdecker@informatik.uni-muenchen.de

## Abstract

*We present a simplified version of SLDAI, an SLD-based proof procedure extended by Abduction and Integrity maintenance, for knowledge assimilation in deductive databases. For an update request in a database which satisfies its integrity requisite, SLDAI computes updates (hypothetical insertions or deletions of facts about base predicates) which satisfy the request while maintaining integrity. That is illustrated by an example.*

## 1. Introduction

Deductive databases (DDBs) are logic theory systems that change over time. State changes are triggered by update requests, which ask for the assimilation of new or revised knowledge into the database. Knowledge assimilation (KA) supports the incorporation of actualized knowledge. View updating is a special case. We focus on the use of abductive logic programming (ALP) for KA, particularly the integrity-preserving satisfaction of update requests. ALP homogenizes queries and updates: Queries can be interpreted as update requests; inferred hypotheses which justify answers can be interpreted as updates which satisfy the requests.

Essentially, KA means to satisfy an update request by modifying the data from which those that are requested to change are derived. In simple cases, that can be done by tracing attempts to derive the request, and drawing updates from the reasons of failure of the attempts, as e.g., in [2] [9] [10]. For example, the request of inserting p can be satisfied by inserting q if p is defined by the clause p← q. However, integrity constraints may invalidate updates drawn from failed derivations. In the example, the presence of q might be forbidden under certain conditions by some constraint. Thus, the goal of KA is to make a request become true in the database without violating any integrity constraint.

For integrity checking, it is useful to reason forward from updates, as in [1] [12] [14] [6]. Reasoning forward exploits that integrity violation can only be caused by updates or hypotheses assumed to satisfy an update request. However, difficulties arise if updates or hypotheses have to be propagated forward through literals of opposed polarity. For example, the insertion of q may or may not engender the deletion of p if p is defined by p← ~q and other clauses.

The KA procedure SLDAI [3] uses a simple kind of forward reasoning through literals of opposed polarity: For a hypothesis $L_h$ and a clause L← B with literal L' in B such that the atoms of $L_h$ and L' unify by some mgu θ but $L_h$ and L' are of opposed polarity, SLDAI infers ~Lθ← ~Lθ as a consequence. Intuitively, such a clause expresses that the negation of Lθ (in the head) can be taken to hold if it is not possible to prove Lθ. In fact, this seems to be the easiest way to capture possible negative consequences of hypotheses. The import of such inference steps is discussed in more detail in [6] [3]. In this paper, we present a simplified version of SLDAI which is not complicated by technical refinements and optimizations as incorporated in [3]. Also, a technical bug is the original version is fixed.

## 2. Preliminary definitions

An *SLDAI clause* (shortly, *clause*) is of form L← B, where L is the *head* of the clause, its *body* B is a (possibly empty) conjunction of literals. L is either a positive or a negative literal. The head may also be empty, in which case the clause is a *denial*. A *database clause* is a clause with a positive literal as its head. A *database* is a finite set of database clauses. A *fact* is a clause with an empty body and either a positive or a negative literal as its head. For a fact L←, the "←" is sometimes omitted. As usual, *queries* and integrity constraints are expressed as denials. An *integrity requisite* is a finite set of integrity constraints.

We denote conjunction by &, negation by ~, set union by $\cup$, set difference by –. For a literal L, |L| denotes the atom of L. The *complement* $\overline{L}$ of L is ~L if L is positive, and |L| if L is negative. Let $\varepsilon$ be the identity substitution, { } the empty set, [ ] the empty clause. Let $\forall$(W) be the universal closure of a formula W.

As usual, we distinguish a set of extensional *base predicates* and *base facts*. For a database D, the set F of all ground base facts in D is the *fact base* of D. As usual, we require that no base predicate occurs in the head of any clause in D – F. Database facts in D – F are permitted, but if there is a fact in D – F, then it must not be base.

An *update* is a finite set U of ground base facts such that, for each L in U, $\overline{L}$ is not in U. For an update U and a database D (its "current state"), the "updated state" U(D) is the database obtained from D by adding each positive fact in U to D and by deleting each fact L in D for which ~L is a negative fact in U. An *update request* is a query. For an update request $\leftarrow$ B in a database D and an integrity requisite I, the *KA problem* is to find an update U of base facts such that $\exists$(B) is true in U(D) and I is satisfied in U(D). An integrity requisite I is satisfied in a database D if there is a partial stable model [15] of D (which is unique if D is stratified) in which each constraint in I is true (cf. [4]). Often, an additional criterion of minimality (according to some measure) of updates is imposed, for filtering among multiple solutions for update requests. In this paper, we do not explicitly impose any such criterion. However, denials in I can be perceived as filters, and by definition, solutions are restricted already by requiring that updates consist of a distinguished kind of facts which, in ALP, are called "abducible":

A literal is *abducible* if it is either a positive base literal or a negative literal. A literal is *updatable* if it is abducible and ground. A *hypothesis* is an updatable fact.

For a set T of clauses, a literal L in a clause C is *processable in* T if it does not flounder in T. L *flounders* in T if L is abducible, L is not ground, L occurs in the body of C, and there is no clause in T the head of which unifies with L. Note that the head of a clause can unify with a negative literal only if it is itself negative.

Let $C_1 = L \leftarrow B_1$, $C_2 = L_2 \leftarrow B_2$ be two clauses with non-empty head, and $L_1$ a literal in $B_1$ such that $|L_1|$ and $|L_2|$ unify by some mgu $\theta$. An *SLDAI resolvent of* $C_1$ *on* $L_1$ *and* $C_2$ *on* $L_2$ (and also an *SLDAI resolvent of* $C_2$ *on* $L_2$ *and* $C_1$ *on* $L_1$) *using* $\theta$ is defined as follows. Let B be obtained by dropping $L_1$ from $B_1$.

*a*) If $L_1$ and $L_2$ are of the same polarity, then $L\theta \leftarrow (B \,\&\, B_2)\theta$ is a (normal) SLDAI resolvent of $C_1$ and $C_2$.

*b*) If $L_1$ and $L_2$ are of opposed polarity and L is positive, then $\sim L\theta \leftarrow \sim L\theta$ is an SLDAI resolvent of $C_1$ and $C_2$.

## 3 The basic structure of SLDAI computations

As the procedures of [8] [10], SLDAI alternates between *refutation* and *consistency* phases of reasoning. In the refutation phase, SLDAI attempts to reduce a given update request to the empty clause, with the union of the database and a set of hypotheses as input set. Whenever a selected updatable literal L (say) is not resolvable with any clause from the input set, SLDAI establishes $L \leftarrow$ as a new hypothesis and attempts to prove its consistency in a subsidiary computation rooted at $L \leftarrow$. As in [8] [10], the selection of an updatable literal L' in the consistency phase may recursively invoke the refutation phase with the update request $\leftarrow \overline{L'}$.

While the refutation phase attempts to derive [ ], the consistency phase attempts to terminate each derivation with a non-empty clause (since deriving [ ] would mean inconsistency of the root). The refutation phase reasons only backward, but the consistency phase explores the consequences of hypotheses also by reasoning forward.

## 4 The simplified definition of SLDAI

<u>Definition</u> (*SLDAI refutation*)

Let D be a database with fact base F and integrity requisite I, C an update request, and H a finite set of hypotheses.

An SLDAI refutation of C in (D, F, I) assuming H with answer $\theta$ and justification J is a sequence $(C_0, H_0)$, ...., $(C_n, H_n)$ $(n>0)$ and a sequence of substitutions $\theta_1, ..., \theta_n$ such that $C_0 = C$, $H_0 = H$, $C_n = [\,]$, $H_n = J$ and $\theta = \theta_1 ... \theta_n$. Moreover, for each i, $0 \le i < n$, a literal $L_i$ in $C_i$ which is processable in $D \cup H_i$ is selected, and one of (1) (2) holds.

(1) There is a clause C' in $D \cup H_i$ the head of which unifies with $L_i$ by mgu $\theta_{i+1}$, such that $C_{i+1}$ is a normal SLDAI resolvent of $C_i$ and C' on $L_i$. If $C' \in F$, then $H_{i+1} = H_i \cup \{C'\}$ else $H_{i+1} = H_i$.

(2) There is no clause as in (1), $L_i$ is updatable, $C_{i+1}$ is a normal SLDAI resolvent of $C_i$ and $L_i \leftarrow$, and $\theta_{i+1} = \varepsilon$. Moreover, there is a subsidiary SLDAI consistency proof of $L_i \leftarrow$ in (D, F, I) assuming $H_i \cup \{L_i\}$ with justification $H_{i+1}$.

An SLDAI refutation of C in (D, F, I) with *computed update* U is an SLDAI refutation of C in (D, F, I) with justification J assuming { } and $U = J - \{h \in F$, or h is negative and $\overline{h} \notin F\}$.

Step (2) above requires that the consistency of the selected updatable literal $L_i$ be shown in case $L_i$ is neither in F nor in $H_i$. The incremented set of hypotheses $H_{i+1}$ in (1) and (2) needs to include input clauses that are in F because otherwise, they may later be inadvertently overridden by their complement in a consistency proof. That can be seen, e.g., by running $\leftarrow q \,\&\, \sim q$ in $D = F = \{ q \leftarrow \}$

If, for a selected literal $L_i$, neither (1) nor (2) applies, and if no subsidiary consistency proof flounders, then the attempt to construct an SLDAI refutation is said to *fail*. If each literal in $C_i$ flounders, or if a subsidiary consistency proof flounders, then the attempt of an SLDAI refutation is said to *flounder*. Since non-ground abducible literals are never processed in [10], SLDAI flounders less often than [10]. For instance, SLDAI computes that the update of inserting $q(a, a)$ satisfies the insert request $\leftarrow p(a, a)$ in $D = \{ p(x, y) \leftarrow q(x, y) \,\&\, q(y, z) \}$, while [10] flounders.

At the very end of a refutation, the actual update $U$ for satisfying the request is obtained by removing from the justification $J$ each positive fact that already is in $F$, as well as each negative fact the complement of which is not in $F$, which notably includes each non-base fact in $J$.

Definition (*SLDAI consistency proof*)

Let $D$, $F$, $I$ be as in the previous definition. Further, let $L\leftarrow$ be a hypothesis, $H_0$ a finite set of hypotheses, and $S = (D - F) \cup I$ (i.e., $S$ is the database schema).

An SLDAI consistency proof of $L\leftarrow$ in $(D, F, I)$ assuming $H_0$ with justification $J$ is a finite tree $\tau$. Each node $C_i$ ($1 \leq i \leq n$, $n > 0$) of $\tau$ is a non-empty clause, to which a set $H_i$ of hypotheses is associated. The root of $\tau$ is $C_1 = L\leftarrow$, $C_n$ is a leaf of $\tau$ and $H_n = J$. Moreover, a literal $L_i$ in $C_i$ which is processable in $S$ is selected, and (0) (1) (2) hold.

(0) If $C_1$ is not base, then $\leftarrow \overline{L}$ is a child node of $C_1$. Other child nodes of the root, if any, are given by (1).

(1) For each non-leaf node $C_i$, $H_i = H_{i-1}$, and one of (1.1) (1.2) holds.

 (1.1) $L_i$ is either selected in the head, or $L_i$ is selected in the body and is not updatable. Each SLDAI resolvent of $C_i$ on $L_i$ and any clause in $S \cup H_{i-1}$ (modulo variants) is a child node of $C_i$.

 (1.2) $L_i$ is selected in the body and is updatable. If $L_i \in F \cup H_{i-1}$ or each SLDAI refutation attempt of $\leftarrow \overline{L}_i$ in $(D, F, I)$ assuming $H_{i-1}$ fails, then the normal SLDAI resolvent of $C_i$ and $L_i \leftarrow$ is a child node of $C_i$.

(2) For each leaf node $C_i$, none of (1.1) (1.2) is applicable, and one of (2.1) (2.2) holds.

 (2.1) $L_i$ is selected in the head or $L_i$ is not updatable. Moreover, $H_i = H_{i-1}$.

 (2.2) $L_i$ is selected in the body and $L_i$ is updatable, and there is an SLDAI refutation of $\leftarrow \overline{L}_i$ in $(D, F, I)$ assuming $H_{i-1}$ with justification $H_i$.

The index $i$ above enumerates the (non-deterministic) sequence by which nodes are processed. This sequence may proceed depth-first or breadth-first or may even jump from branch to branch, but it is supposed to start at the root. In fact, it suffices to associate a set of hypotheses only to each leaf if the search proceeds depth-first. Note that $H_0$ is not associated to any node. For a leaf $C_i$, $H_i$ is determined by the reason of failure to infer any more clause from $C_i$. Thus, $H_i$ can be seen as a justification of the failure to derive the empty clause, i.e., of proving consistency.

Similar to the procedures of [8] and [10], the implementation of simultaneous search of several branches in consistency proofs of SLDAI needs careful synchronization, since the termination of one branch has an effect on the hypotheses of the others.

Similar to [8] [10], step (0) implicitly resolves the hypothesis $L\leftarrow$ at the root with an input clause of form $\leftarrow L \,\&\, \overline{L}$, in step (0).

Steps in (1) describe normal resolution steps where current hypotheses are taken into account as candidate input, but the fact base is not. That is because the complement of facts in $F$ may have to be assumed in order to successfully terminate a consistency proof. Such facts will then be among those to be removed from the database for satisfying the update request.

In (1.2), the derivation needs to continue by resolving $L_i$ only if $\overline{L}_i$ fails to be abductively provable. As opposed to [8] [10], failure to refute $\leftarrow \overline{L}_i$ in (1.2) is required to be finite, and $L_i$ is not added to the hypotheses. The latter conforms to the modification of [8] in [7]. If, on the other hand, $\leftarrow \overline{L}_i$ can be successfully refuted, as in (2.2), then the derivation can be terminated, since the consistent assumption of $\overline{L}_i$ prevents the derivation to continue by resolving $L_i$. Also all other reasons to terminate the derivation are captured in (2).

If one of the branches in $\tau$ terminates in the empty goal [], then the attempt of proving consistency of the root is said to *fail*. If there is no such branch and, in one branch, there is a node in which each literal flounders, then the attempt of proving consistency of the root is said to *flounder*.

More accurate definitions for SLDAI would have to incorporate ranks as for SLDNF in [11], such that circularity of definitions is avoided.

The correspondence of SLDAI consistency proofs and finitely failed SLDNF trees is not as close as the correspondence of refutations by SLDAI and SLDNF. Indeed, a finitely failed search for SLDAI refutations is different from an SLDAI consistency proof. Each leaf node of each branch in an SLDAI consistency proof may contribute to the computed justification (while non-leaf nodes never do), and each node inherits the hypotheses of previously reached leaves. SLDAI would possibly compute unsound justifications if hypotheses could not be taken into account across branches. For example, an attempt to refute $\leftarrow \sim o$ in the database $\{ o \leftarrow p, o \leftarrow q, p \leftarrow \sim q, q \leftarrow \sim p \}$ could incorrectly succeed if the two branches of the consistency proof of $\sim o \leftarrow$ had no knowledge of each other's hypotheses).

It can be shown that SLDAI is sound, in the following sense: For an SLDAI refutation of a query $\leftarrow B$ in $(D, F, I)$ with computed answer $\theta$, computed justification $J$ and computed update $U$, $\forall(B\theta)$ as well as each constraint in $I$ is true in all Herbrand models of $U(D) \cup J$, and there is a partial stable model of $U(D)$ in which $\forall(B\theta)$ is true and $I$ is satisfied. Moreover, it can be shown that each answer, justification and update computed by the procedure in [10] is also computed by SLDAI.

## 5 KA with SLDAI – an example

Each fact in relation room consists of the room number and the department which owns the room. Facts in the relation course consist of the subject of the course and the responsible department. Facts in lecture contain course subject, room number, weekday and the hour of the course. Facts in room_equipment record pieces of equipment in rooms identified by their number, and course_equipment specifies equipment needed for courses. The clauses *22 – 24* determine under which conditions rooms are either inadequate or acceptable for certain courses.

Note that, in the "worst case", a room may have to be taken as acceptable for a course (by clause *24*) even if it is inadequate. The condition is that the same department must be responsible for both the course and the room.

*1* room(27, math)
*2* room(36, math)
*3* room(48, cs)
*4* room(69, cs)
⋮

*5* course(logic, math)
*6* course(java, cs)
*7* course(prolog, cs)
*8* course(db, cs)
⋮

*9* lecture(logic, 27, mo, 10)
*10* lecture(java, 48, tue, 11)
*11* lecture(prolog, 69, thu, 11)
*12* lecture(db, 36, fri, 10)
⋮

*13* room_equipment(27, board)
*14* room_equipment(27, flip_chart)
*15* room_equipment(36, super_8)
*16* room_equipment(36, overhead)
*17* room_equipment(48, video)
⋮

*18* course_equipment(logic, board)
*19* course_equipment(java, video)
*20* course_equipment(prolog, overhead)
*21* course_equipment(db, overhead)
⋮

*22* inadequate(R, C) <— course_equipment(C, E) & ~ room_equipment(R, E)

*23* acceptable(R, C) <— ~ inadequate(R, C)

*24* acceptable(R, C) <— room(R, D) & course(C, D)

*25* <— lecture(C, R, W, H) & ~ acceptable(R, C)

<u>*update request:*</u> *delete* room_equipment(36, overhead)

The rationale behind that may be that, within one department, it is easier to get along with (or do something against) inadequacies, than in rooms which do not belong to the department responsible for the course. In general, however, the database schema permits that departments may share their rooms among each other. The integrity constraint states that there must not be any lecture held in a room which is not acceptable. The update request asks to remove the overhead projector from room number 36 (perhaps because it needs repair or it is needed somewhere else).

Variable symbols in the example are denoted by capital letters, constants are strings of digits or lower case characters. For convenience, clauses are numbered. The rank of each derivation is denoted by an underlined number at the root.

If the update request was simply executed, then integrity would be violated, as can be easily seen by querying the integrity constraint (clause *25*): The lecture of the database course held on Friday mornings in room 36 could not do without an overhead, since the latter is needed for the db course according to fact *21*. Instead of giving up after detecting inconsistency, SLDAI attempts to avoid inconsistency, as shown in the derivations below.

Selection of literals proceeds from left to right. The top level derivation of SLDAI starts with the goal to refute the query ← ~room_equipment(36, overhead), which represents the update request. It terminates after one step with the answer that the request can be satisfied by actually deleting the fact room_equipment(36, overhead) and inserting course(db, math). Though this is not a satisfying answer, it is instructive to look at its computation.

Selection of the root literal ~room_equipment(36, overhead) yields the subsidiary attempt of rank 1 to show that the hypothesis ~room_equipment(36, overhead) is consistent. (The successful termination of derivations in the consistency phase is denoted by == at the tip of the derivations.) For that purpose, the hypothesis is propagated through the occurrence of a matching literal in clause *22*. This kind of forward reasoning step is known from [14].

The next step, with input clause *23*, is a forward propagation step which generates a negative consequence, as described in the preliminary definitions. The selected literal inadequate(36, C) is of opposed polarity to the occurrence of a matching atom in *23*. Hence, the derived resolvent is the clause ~acceptable(36, C) ← ~acceptable(36, C).

The literal in the head is propagated in the next step into the denial *25*. The leftmost literal in the resolvent is then resolved with each fact about lecture with 36 in its second argument. In the considered excerpt of the database, there is just one such fact, which yields the resolvent ← ~acceptable(36, db). That spawns a subsidiary refutation attempt of rank 2, rooted at the complementary goal ← acceptable(36, db). Let us consider the failed branch on the left hand side first. It fails to refute the root goal because the subsidiary attempt to prove that the hypothesis ~inadequate(36, db) ← is consistent fails, as can be seen in the derivation of rank 3, below the failed refutation.

Besides the alternation between attempts of refutation and consistency proofs, that derivation illustrates two more ALP features. First, the root ~inadequate(36, db) ← is rewritten in the first step to the goal ← inadequate(36, db). In general, it is sufficient to do that only for negative hypotheses which are not about base predicates. Such rewritings can be seen as resolution steps with implicitly assumed denials representing the law of contradiction (in our example: ← inadequate(36, db) & ~inadequate(36, db)). Second, the last step of the attempt to show consistency of the hypothesis ~inadequate(36, db) ←, which leads to [ ] and means contradiction, features that hypotheses assumed in the course of the derivation are taken into account as candidate input clauses. In our example, the hypothesis ~room_equipment(36, db) ←, assumed at the single first step of the top-level derivation of rank 0, is used as input in the derivation of rank 3, which shows that the subsidiary hypothesis ~inadequate(36, db) ← is inconsistent.

<u>0</u> <–– ~ room_equipment(36, overhead)

| *input hypothesis* *~room_equipment(36, overhead)*

*output hypotheses* [ ] *~room_equipment(36, overhead), course(db, math)*

**1**

```
~room_equipment(36, overhead)  <—
                |
                | 22
                |
inadequate(36, C)  <—  course_equipment(C, overhead)
                |
                | 23
                |
~acceptable(36, C)  <—  ~acceptable(36, C)
                |
                | 25
                |
<—  lecture(C, 36, W, H)  &  ~acceptable(36, C)
                |
                | 12
                |
<—  ~acceptable(36, db)
                |
                |
                |
```

*output hypotheses* == *~room_equipment(36, overhead), course(db, math)*

**2**

```
            <—  acceptable(36, db)
              /                    \
             / 23                   \ 24
            /                        \
<—  ~inadequate(36, db)          <—  room(36, D)  &  course(db, D)
            |                             |
            |                             | 2
            |                             |
          fail                   <—  course(db, math)
                                          |
                                          | input hypothesis course(db, math)
                                          |
                                         [ ]  output hypothesis course(db, math)
```

**3**

```
~inadequate(36, db) <—
            |
            |
            |
<—  inadequate(36, db)
            |
            | 22
            |
<—  course_equipment(db, E)  &  ~room_equipment(36, E)
            |
            | 21
            |
<—  ~room_equipment(36, overhead)
            |
            | input hypothesis  ~room_equipment(36, overhead)
            |
           [ ]
```

Now, let us turn to the successful branch of the tree of rank 2 on the right-hand side, rooted at ← acceptable(36, db). Its last non-empty goal is ← course(db, math). Mere integrity checking, as in [6], would only test if integrity is satisfied in the updated state, and hence would fail to resolve the goal ← course(db, math). That, together with the failure of the left-hand side branch, would show that the hypothesis on rank 1 is inconsistent, and hence would show that deleting room_equipment(36, overhead) without further ado would lead to integrity violation.

As opposed to that, SLDAI attempts to modify the extensional part of the database (i.e., the fact base) such that the request can be satisfied without integrity violation. Hence, SLDAI continues by hypothesizing the otherwise failing literal course(db, math). The subsidiary derivation of rank 3 which proves consistency of that hypothesis is not shown; it terminates successfully after one forward step through *24*. Thus, using the consistent hypothesis course(36, math) as input shows that acceptable(36, db) is provable. That in turn shows that, under the hypothesis of course(db, math), also ~room_equipment(36, overhead) is consistent. That finally shows that the original update request can be consistently satisfied by actually updating the assumed hypotheses, i.e., by deleting room_equipment(36, overhead) and inserting course(db, math).

Except the one forward step for proving consistency of course(db, math), the derivations above show the complete search space of SLDAI for the given update request. However, if in the right-hand side derivation of rank 2, the rightmost literal of ← room(36, D) & course(db, D) would have been selected instead of the leftmost, then we would have obtained another solution to satisfy the request, viz. delete room_equipment(36, overhead) and insert room(36, cs). The dependence of ALP procedures on the employed selection function, and also a way of achieving independence, is addressed in more detail in [3].

Clearly, both possible outcomes of SLDAI for satisfying the given update request are not satisfactory: It is not reasonable to simply belie the database that the db course would also be done by the math department, or that room 36 would also belong to computer science. But SLDAI cannot really be blamed for coming up with such solutions, since, after all, the database does not know any better. In fact, it is now the knowledge engineer's turn to tell the database more about the meaning of relations room and course, and to provide more flexibility to the database clauses, such that scheduling and room allocation for courses can be supported more satisfactorily.

Let us consider the following modification of the database: Two referential constraints (*31* and *32*) be added which express the uniqueness of the ownership of rooms and courses. Further, clause *22* for defining inadequateness be modified by the extra condition that a room which lacks some piece of equipment E (say) for a certain course is inadequate only if there is no substitute for E. Also, clauses *26 − 30* which define admissible substitutes be added to the database, as follows. (An extension of SLDAI for processing such kind of clausal updates is described in [5].)

```
22'  inadequate(R, C) <—   course_equipment(C, E) &
                           ~ room_equipment(R, E) & ~ subs(E, R, C)

26   subs(E, R, C) <—  substitute(E, R, C, E') & room_equipment(R, E')

27   substitute(super_8, 36, C, video)

28   substitute(video, R, java, slide_projector)

29   substitute(E, R, C, E') <—  query_the_user(substitute(E, R, C, E'))

30   <—  substitute(video, R, C, overhead)

31   <—  course(C, D) & course(C, D') & D ≠ D'

32   <—  room(C, D) & room(C, D') & D ≠ D'
```

For brevity, the definition of query_the_user (cf. [13]) is not elaborated. Its idea is to enter a dialogue with the user, to ask her for possible substitutes of equipment if none is found in the database.

With the modified database, the user obtains the solution that the overhead can be removed from room 36 if it is replaced by a flip chart. Ranks 0 and 1 which compute this solution are the same as before, except that the output hypothesis is different. Branches of rank 2 – 5 which prevent the previous stupid answer and lead to the more intelligent new answer are depicted below.

We first comment on the right-hand side branch of attempting to refute the goal ← acceptable(36, db), which previously succeeded with the stupid suggestion to insert course(db, math). That hypothesis is now resolved with the new referential constraint that departments responsible for courses are unique. There are two input clauses for the resolvent, viz. the hypothesis itself and the fact that db is a computer science course. The resolvents are disequalities which are evaluated as built-in predicates. The inconsistency of the hypothesis is shown by the right-hand side branch. This prevents the output of the hypothesis as an answer to the initial update request. Instead, the hypothesis is withdrawn on backtracking.

Next, let us have a look at what happens to the hypothesis ~ inadequate(36, db), which previously was inconsistent. Now that there is the possibility to replace missing equipment by some appropriate substitute, it turns out to be consistent, due to the user's input and another hypothesis, which is assumed at the end of the derivation of rank 4. That hypothesis is shown to be consistent in the derivation of rank 5, below. We assume the user inputs that a flip chart is a possible substitute for the overhead in room 36 for the database course. That input leads to the goal ← room_equipment(36, flip_chart). Since the relation room_equipment is an extensional base relation, it can be updated in order to satisfy that goal, provided the update does not violate integrity.

```
2                  <—— acceptable(36, db)
                       /                 \
                      / 23                \ 24
                     /                     \
<—— ~inadequate(36, db)                  <—— room(36, D) & course(db, D)
                    |                        |
 input hypothesis   | ~inadequate(36, db)    | 2
                    |                        |
                   [ ]                   <—— course(db, math)
 output hypothesis    ~inadequate(36, db)    |
                                             |
                                             |
                                            fail


3            course(db, math) <——
                         |
                         | 31
                         |
             <—— course(db, D') & math ≠ D'
                          /   \
 input hypothesis course(db, math) /   \ 8
                        /       \
      <—— math ≠ math       <—— math ≠ cs
                |                 |
 built-in evaluation |            | built-in evaluation
                |                 |
               ==                [ ]
```

```
3      ~inadequate(36, db) <——
                |
                |
  <——  inadequate(36, db)
                |
                | 22'
                |
  <——  course_equipment(db, E)  &  ~room_equipment(36, E)  &  ~subs(E, 36, db)
                |
                | 21
                |
  <——  ~room_equipment(36, overhead)  &  ~subs(overhead, 36, db)
                |
                | input hypothesis  ~room_equipment(36, overhead)
                |
  <——  ~subs(overhead, 36)
                |
                | (see subsidiary refutation of rank 4)
                |
               == output hypotheses ~inadequate(36, db), room_equipment(36, flip_chart)


4      <——  subs(overhead, 36, db)
                |
                | 26
                |
  <——  substitute(overhead, 36, db, E')  &  room_equipment(36, E')
                |
                | 29
                |
  <——  query_the_user(substitute(overhead, 36, db, E'))  &  room_equipment(36, E')
                |
                : user input
                : query_the_user(substitute(overhead, 36, db, flip_chart))
                |
  <——  room_equipment(36, flip_chart)
                |
                | input hypothesis  room_equipment(36, flip_chart)
                |
               [ ]  output hypothesis  room_equipment(36, flip_chart)


5      room_equipment(36, flip_chart)  <——
                |
                :
                :
                |
               ==  output hypothesis  room_equipment(36, flip_chart)
```

Thus, SLDAI hypothesizes room_equipment(36, flip_chart) ← and proves its consistency, on rank 5. That succeeds after two steps (which are not shown), through *22* and *23*. (The optimized version of SLDAI in [3] does not even need to take any step at all for proving consistency of this hypothesis). The successful consistency proof sanctions the use of the hypothesis as input clause in the last step of the refutation of ← subs(overhead, 36), on rank 4. The success of that refutation in turn sanctions the consistency of the hypothesis ~inadequate(36, db) ←, on rank 3. Thus, that hypothesis can be used as input in the last step of the successful refutation of ← acceptable(36, db), on rank 2.

That brings us back to the consistency proof of ~room_equipment(36, overhead), on rank 1, which now can be terminated successfully. That consistency proof is not depicted since its structure is the same as before. However, the hypotheses which lead to its success are now different: Instead of the (meanwhile inconsistent) course(db, math), we now get room_equipment(36, flip_chart). Together with ~room_equipment(36, overhead) and ~inadequate(36, db), it is output as hypothesis for justifying the answer to the original update request to remove the overhead from room 36, on rank 0. The justification consisting of the three hypotheses is finally translated into an extensional update which consists of deleting room_equipment(36, overhead) and inserting room_equipment(36, flip_chart). Nothing needs to be done wrt ~inadequate(36, db), since that holds by default in the updated database.

Now, the user could go on by querying the database in which room there is a flip chart, then request the removal of the flip chart from room 27 such that it is available for room 36, etc., but we do not go through that here. However, let us finally consider a slight modification of the example, just to show that user input is not necessarily needed for satisfying update requests which would otherwise violate integrity. Suppose the database additionally contains the clause

substitute(overhead, _, _, board) ←

which says that, no matter which course and which room, a board is always a possible substitute for an overhead. Then, the computation of rank 4 above will branch into a subtree by using the new clause besides clause *29* as input. That subtree then leads to the hypothesis room_equipment(36, board). That turns out to be consistent, and thus yields the update of deleting room_equipment(36, overhead) and inserting room_equipment(36, board).

The update computed by SLDAI must finally be committed or rejected by the user, and realized by separate extensional updates.

## References


[1] Decker: Integrity enforcement on deductive databases, in Kerschberg (ed): Expert Database Systems, Morgan Kaufmann, 1987.

[2] Decker: Drawing updates from derivations, ECRC Technical Report KB-65, 1989; short version in Proc. 3rd ICDT, Springer, 1990.

[3] Decker: An extension of SLD by abduction and integrity maintenance for view updating in deductive databases, Proc. JICSLP'96, MIT Press, 1996.

[4] Decker: A model-theoretic semantics of integrity constraints in deductive databases, to appear in Proc. Workshop Logic Programming & Knowledge Representation (LPKR'97),16 October 1997.

[5] Decker: One abductive logic programming procedure for two kinds of updates, to appear, 1997.

[6] Decker, Celma: A slick procedure for integrity checking in deductive databases, Proc. 11th ICLP, 1994.

[7] Dung: An argumentation-theoretic foundation for logic programming, J. Logic Programming 22, 1995.

[8] Eshghi, Kowalski: Abduction compared with negation by failure, Proc. 7th ICLP, MIT Press, 1989.

[9] Guessoum, Lloyd: Updating knowledge bases I / II, New Generation Computing 8 / 10, 1990 / 1991.

[10] Kakas, Mancarella: Database updates through abduction, Proc. 16th VLDB, 1990;

[11] Lloyd: Foundations of Logic Programming, Springer, 1987.

[12] Lloyd, Sonenberg, Topor: Integrity constraint checking in stratified databases, J. Logic Programming 4, 1987.

[13] Sergot: A query-the-user facility for logic programming, in Degano, Sandevall (eds): Integrated Interactive Computer Systems, North-Holland, 1983.

[14] Sadri, Kowalski: A theorem-proving approach to database integrity, in Minker (ed): Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, 1988.

[15] Saccà, Zaniolo: Stable models and non-determinism in logic programs with negation, Proc. PoDS'90, ACM Press, 1990.



**Acknowledgement**

François Bry hosted me as a visitor at the computer science department of the University of München, where most of this paper was written. I also appreciate the detailed feedback of an anonymous colleague who reviewed an early version of the paper.