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

Ludwig
Maximilians–
Universität
München

# One Abductive Logic Programming Procedure for Two Kinds of Updates

**Hendrik Decker**

appeared in B. Freitag et al (eds), *Proc. DYNAMICS'97*, Port Jefferson (New York)
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1997-16, September 1997

# One Abductive Logic Programming Procedure for Two Kinds of Updates

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

**Abstract**

We distinguish schema updates and user updates. A crucial benchmark for the assimilation of new knowledge via updates is the enforcement of integrity. The procedures SLIC for integrity checking of schema updates and SLDAI for integrity maintenance of user updates are forged into the abductive procedure SLDAI*, for satisfying schema and user update requests. SLDAI* computes modifications of the extensional part of the database in order to satisfy the requests and to preserve integrity. Similar to other abductive procedures for knowledge assimilation, SLDAI* homogenizes queries and updates: Queries can be interpreted as user update requests; inferred hypotheses which justify computed answers can be interpreted as updates which satisfy the requests. Moreover, abductive consistency proofs are generalized in SLDAI* to implement hypothetical schema updates.

## 1 Two kinds of updates and procedures

Various kinds of updates can be encountered in contemporary database systems as well as in the database research literature. In this paper, we distinguish schema updates and user updates. *Schema updates* means that "intensional" database clauses are requested to be inserted, deleted or modified explicitly. Insertions and deletions of clauses defining database views or integrity constraints are typical cases of schema updates. *User updates* means that an update request is expressed by the specification of an arbitrary closed formula which is required to become true in the updated state. View updates realized by modifications of "extensional" database clauses are a typical case of user updates.

Also, various kinds of procedures for enforcing integrity constraints upon updates can be met in database systems and in the literature. For instance, in [2], a generalization of SLDNF called SLIC is discussed. SLIC checks if an update consisting of a finite set of addtions and deletions of database clauses would violate integrity. Its efficiency has compared favourably to other integrity checking procedures which preceded SLIC (cf. [1]). A shortcoming of SLIC is that its reaction to integrity violation is passive, in the sense that it simply rejects update requests that otherwise would violate integrity.

In response to that, the active database procedure SLDAI has been proposed in [3]. For a given user update request, SLDAI computes modifications of the extensional database such that the request becomes satisfied and integrity remains satisfied in the updated state. However, a shortcoming of SLDAI (and also of other, comparable abductive procedures in the literature) is that it cannot readily handle schema updates.

For example, SLDAI can compute integrity-preserving solutions for satisfying the user update request *update*($\forall x\,(p(x) \vee \sim q(x))$). It asks for an update of the extensional database such that the specified formula is satisfied in the new state. However, SLDAI is not even defined for taking care of the schema update request expressed by *insert*($p(x) \leftarrow q(x)$), let alone to take appropriate actions in case the insertion of that clause would violate integrity. On the other hand, SLIC is defined for, but simply rejects such schema update requests in case they would lead to integrity violation.

In general, schema updates differ from user updates. For example, the schema update *insert*($p(x) \leftarrow q(x)$) is different from the user update request *update*($\forall x\,(p(x) \vee \sim q(x))$). The latter can be satisfied by deleting all facts about $q$ (provided that thereby, integrity will not be violated). As opposed to that, the former will leave $q$ unchanged (unless integrity maintenance would necessitate changes also to the $q$ relation).

In an update request of the form *update*($B$), $B$ is an arbitrary closed first-order formula. In effect, *update*($B$) asks for a modification of the extensional base facts, such that $B$ becomes true in the updated state and the integrity constraints remain satisfied. If several user update requests with arguments $B_1, ..., B_k$ ($k>1$) are issued at a time, in a single transaction, then they can be conjoined into one request of form *update*($B_1 \,\&\, ... \,\&\, B_k$). The user may express delete requests by negated (sub)formulae. For instance, the request *update*($\sim p(a)$) asks that $\sim p(a)$ be true in the updated state, which means that (the derivability of) $p(a)$ is to be deleted.

In a schema update request of the form *insert*($A \leftarrow B$) (resp., *delete*($A \leftarrow B$)), $A \leftarrow B$ is a database clause. Either one but at most one of $A$, $B$ may be empty. An update of this form requests the physical insertion or deletion, respectively, of the clause $A \leftarrow B$. Additionally, integrity is required to be preserved. That can be achieved by additionally inserting and deleting appropriate base facts as necessary. If $A$ is absent, then *insert*($\leftarrow B$) (respectively, *delete*($\leftarrow B$)) requests the insertion (deletion) of the integrity constraint $\leftarrow B$. For *insert*($\leftarrow B$), an additional modification of the base facts such that the resulting state satisfies integrity, including the inserted constraint, may be due. For *delete*($\leftarrow B$), nothing but to remove the specified constraint needs to be done.

## 2 SLDAI*

SLDAI [3] attempts to satisfy user update request by modifying the base facts from which those that are requested to change are derived. For example, the request of inserting $p$ can be satisfied by inserting $q$ if $p$ is defined by the clause $p \leftarrow q$. 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, SLDAI may be forced to assume further changes in order to make the request become true without violating any integrity constraint. The same idea also applies if the insertion or deletion of a clause, requested by a schema update, leads to a violation of integrity. For enabling SLDAI to also satisfy schema updates, it needs to process not just hypothetical base facts, but also hypothetical insertions and deletions of arbitrary clauses. This very feature can be adapted from SLIC, which yields SLDAI*, as defined in section 2.2.

For checking whether an assumed change of a base fact leads to integrity violation, it is useful to reason forward from updates, as, e.g., in [12] [2] [3]. 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 $p \leftarrow \sim q$ and other clauses.

Similar to SLIC, SLDAI* uses a simple kind of forward reasoning through literals of opposed polarity: For a hypothesis $L_h$ and a clause $L \leftarrow B$ with literal $L'$ in $B$ such that the atoms of $L_h$ and $L'$ unify by some mgu $\theta$ but $L_h$ and $L'$ are of opposed polarity, SLDAI* infers $\sim L\theta \leftarrow \sim L\theta$ as a consequence. Intuitively, such a clause expresses that the negation of $L\theta$ (in the head) can be taken to hold if it is not possible to prove $L\theta$. This seems to be the easiest way to capture possible negative consequences of hypothetical updates. The import of such inference steps is discussed in more detail in [2]. The definition of SLDAI* in 2.2 is based on the simplified version of SLDAI described in [5], which is not complicated by technical refinements and optimizations as incorporated in [3].

## 2.1 Preliminary definitions

In this paper, a *clause* is always of form $L \leftarrow B$, where the *head* L is is either a positive or a negative literal, and the *body* B is a (possibly empty) conjunction of literals. 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 $L\leftarrow$, the "$\leftarrow$" is sometimes omitted. As usual, *queries* and integrity constraints are expressed as denials. An *integrity requisite* is a finite set of integrity constraints. We denorte conjunction by &, negation by ~, disjunction by $\vee$, set union by $\cup$, set difference by –. For a literal L, let $|L|$ denote the atom of L. The *complement* $\overline{L}$ of L is $\sim L$ 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. No base predicate must occur 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 (extensional) *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. For a schema (or, resp., user) update request UR in a database D with integrity requisite I, the problem is to compute an update U of base facts such that UR is satisfied in the new state and integrity remains satisfied. If UR is a user update then the new state is (U(D), I). If UR is a schema update, then the new state is (U(D'), I'), where D' and I' are obtained by inserting or deleting the clause specified in UR (i.e., I'=I if the argument of UR is a databse clause, and D'=D if the argument of UR is a denial). An integrity requisite I is *satisfied* in a database D if there is a partial stable model [11] of D in which each constraint in I is satisfied (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 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.

This definition is due to [2]: 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 $\theta$. A (*normal* or *SLIC*) *resolvent of* $C_1$ *on* $L_1$ *and* $C_2$ *on* $L_2$ (synonymously, a *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 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 a SLIC resolvent of $C_1$ and $C_2$.

## 2.2 The definition of SLDAI*

Similar to other abductive procedures based on [7] [8], 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 $L \leftarrow$. 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.

**Definition** (*SLDAI* refutation*)

Let D be a database with fact base F and integrity requisite I, C a query, 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, $0 \leq i < n$, 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 $D \cup H_i$ which unifies with $C_i$ on $L_i$ by mgu $\theta_{i+1}$, such that $C_{i+1}$ is a normal 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 selected in the body of $C_i$, $L_i$ is updatable, $C_{i+1}$ is a normal resolvent of $C_i$ and $L_i \leftarrow$, and $\theta_{i+1} = \varepsilon$. If $L_i \in F \cup H_i$, then $H_{i+1} = H_i \cup \{L_i\}$. Else, 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 [8], SLDAI* flounders less often than [8]. For instance, SLDAI* computes that the update of inserting $q(a, a)$ will satisfy the insert request $\leftarrow p(a, a)$ in the database $D = \{ p(x,y) \leftarrow q(x, y) \,\&\, q(y,z) \}$, while [8] 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, by the way, includes each non-base fact in $J$.

**Definition** (*SLDAI* consistency proof*)

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

An SLDAI* consistency proof of $C$ in $(D, F, I)$ assuming $H_0$ with justification $J$ is a finite tree $\tau$. Each node $C_i$ ($1 \le i \le n$, $n > 0$) of $\tau$ is a non-empty clause, to which a set $H_i$ of hypotheses is associated. The root of $\tau$ is $C_1 = C$, $C_n$ 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 of form ${\sim}A \leftarrow$ and $A$ is not base, then $\leftarrow A$ is a child node of $C_1$. Other child nodes of the root, if any, are obtained according to (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 body and is not updatable, or $L_i$ selected in the head. Each normal and each SLIC 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 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 $H_i$.

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

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.

Step (0) implicitly resolves the hypothesis $L\leftarrow$ at the root with an input clause of the form $\leftarrow L \,\&\, \overline{L}$. 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 deleted 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 [7] [8], 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 [7] in [6]. 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 $L_i$.

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 if, 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*.

The correspondence of SLDAI* consistency proofs and finitely failed SLDNF trees is not as close as the correspondence of refutations in 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). Thus, the implementation of a 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. More accurate definitions for SLDAI* would have to incorporate ranks as for SLDNF in [9], such that circularity of definitions is avoided.

### 2.3 Satisfying two kinds of update requests with SLDAI*

We are going to describe how to translate an update request UR in a given database state into a pair consisting of a database state and a start clause. SLDAI* is then run from the start clause in the associated state. The task is to compute a (possibly empty) set of hypotheses which represents an update of the extensional database such that the updated state satisfies UR and the integrity requisite. For a schema update which requests the deletion of $A \leftarrow B$ (say), the start clause on which SLDAI* is run is of form $\sim A \leftarrow \sim A$, which captures that consequences of $A \leftarrow B$ are deleted if and only if they cannot be proved.

Let D be a database which satisfies an integrity requisite I, F be the fact base of D and and UR be an update request. Below, we describe on which start clause and in which state SLDAI* is run in order to compute an integrity-preserving update for satisfying UR.

If UR = *update*(B), then an SLDAI* refutation proof of $\leftarrow B$ in (D, F, I) is attempted.

If UR = *insert*($A \leftarrow B$) and A is not empty, then an SLDAI* consistency proof of $A \leftarrow B$ in $(D \cup \{A \leftarrow B\}, F, I)$ is attempted.

If UR = *delete*($A \leftarrow B$) and A is not empty, then an SLDAI* consistency proof of $\sim A \leftarrow \sim A$ in $(D - \{A \leftarrow B\}, F, I)$ is attempted.

If UR = *insert*($\leftarrow B$), then an SLDAI* consistency proof of $\leftarrow B$ in $(D, F, I \cup \{\leftarrow B\})$ is attempted.

If UR = *delete*($\leftarrow B$), nothing but to physically delete $\leftarrow B$ from I needs to be done.

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 $\theta$, 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, for an SLDAI* consistency proof of $L \leftarrow B$ in (D', F, I') with computed update U, there is a partial stable model of U(D') in which I is satisfied, where D' and I' are the result of inserting or, resp., deleting the clause in the original schema update request.

## 3 An example

The origin of the following example is due to [10]. 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. Clauses *22* – *24* determine under which conditions rooms are either inadequate or acceptable for certain courses.

In the "worst case", a room may have to be taken as acceptable for a course (by clause *24*) even though it might be inadequate. The condition is that the same department must be responsible for both the course and the room. The underlying rationale 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 responsible department. In general, however, 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 urgently 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 indicated by an underlined number at the root.

| | | |
|---|---|---|
| *1* room(27, math) | *5* course(logic, math) | *9* lecture(logic, 27, mo, 10) |
| *2* room(36, math) | *6* course(java, cs) | *10* lecture(java, 48, tue, 11) |
| *3* room(48, cs) | *7* course(prolog, cs) | *11* lecture(prolog, 69, thu, 11) |
| *4* room(69, cs) | *8* course(db, cs) | *12* lecture(db, 36, fri, 10) |
| ⋮ | ⋮ | ⋮ |

| | |
|---|---|
| *13* room_equipment(27, board) | *18* course_equipment(logic, board) |
| *14* room_equipment(27, flip_chart) | *19* course_equipment(java, video) |
| *15* room_equipment(36, super_8) | *20* course_equipment(prolog, overhead) |
| *16* room_equipment(36, overhead) | *21* course_equipment(db, overhead) |
| *17* room_equipment(48, video) | ⋮ |
| ⋮ | |

*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> *update*(~room_equipment(36, overhead))

Execution of the given user update request without further ado would violate clause *25*, since the db course held on Friday mornings in room 36 could not do without an overhead, according to fact *21*. However, SLDAI* attempts to avoid inconsistency, as shown below. Selection of literals proceeds from left to right. SLDAI* starts on rank 0 with the goal to refute ← ~room_equipment(36, overhead), which represents the update request. It terminates after one step with the answer that the request can be satisfied by deleting room_equipment(36, overhead) and inserting course(db, math). Though this is not a satisfying answer, it is instructive to look at its computation by subsidiary derivations. Successful termination of derivations in consistency proofs is indicated by == at their tips.

```
0        <—  ~ 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)*

[ ]

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. For that purpose, the hypothesis is propagated through the occurrence of a matching literal in clause *22*. This kind of forward reasoning is known from [12]. The next step, with input clause *23*, is a forward step with a negative consequence (cf. section 2): The selected literal inadequate(36, C) is of opposed polarity to the matching atom in *23*. Hence, the derived SLIC resolvent is ~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 sample 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 show that the hypothesis ~inadequate(36, db) ← is consistent fails, as shown by the derivation of rank 3 below the failed refutation.

Besides the alternation between attempts of refutation and consistency proofs, that derivation illustrates two more SLDAI* features. First, the root ~inadequate(36, db) ← is rewritten in the first step to the goal ← inadequate(36, db). It can be shown that, in general, such steps need to be taken only for negative non-base hypotheses. 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 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.

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). Up to that point, mere integrity checking with SLIC essentially would have taken the same steps of rank 0, 1 and 2. Mere integrity checking only tests if integrity is satisfied in the updated state "as-is", 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 modifications would lead to integrity violation.

Thus, SLDAI* continues by hypothesizing the otherwise failing literal course(db, math). For brevity, the subsidiary derivation of rank 3 which proves consistency of that hypothesis is not shown; it terminates successfully after one forward step through clause *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 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).

If, in the right-hand side derivation of rank 2, the rightmost literal of the goal ← room(36, D) & course(db, D) would have been selected instead of the leftmost, then a different solution for satisfying the initial request would be computed, viz. the deletion of room_equipment(36, overhead) and the insertion of room(36, cs). The dependence of abductive logic programming 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 rules, such that scheduling and room allocation for courses can be supported more satisfactorily.

Let us consider the following schema 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* be modified by the extra condition that a room which lacks some piece of equipment E 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.

*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)

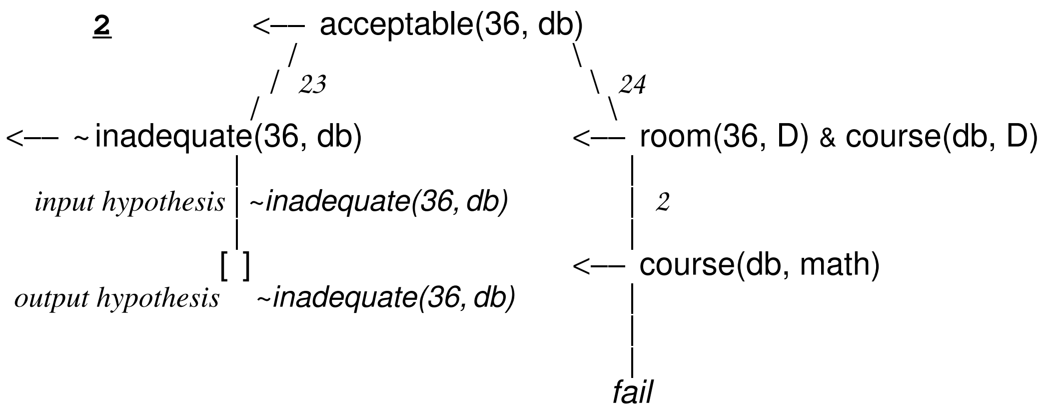
*31* <–– course(C, D) & course(C, D’) & D ≠ D’

*32* <–– room(C, D) & room(C, D’) & D ≠ D’

For brevity, the definition of the predicate query_the_user is not elaborated (cf. [13]). 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. Also for brevity, the SLDAI* consistency proofs for deleting clause *22* and inserting *22’* and *26* – *32*, which proceed according to the definition in section 2.3, are not elaborated. We leave this to the reader, and just mention that none of the consistency proofs for *22*, *22’* and *26* – *32* necessitates the assumption of any additional hypothesis. In other words, the proofs go through straightforwardly.

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 hypotheses are 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 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 now is resolved with the new constraint that the courses’ assignment to responsible departments is 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 inconsistency proof prevents the output of the hypothesis as a possible update for satisfying the initial update request. Instead, the inconsistent hypothesis is withdrawn on backtracking.



```
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
              |                |
              ==              [ ]
```

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, as shown by the proof on rank 3, below. The success of that consistency proof is due to the user's input during the subsidiary refutation of rank 4, and another hypothesis, which is assumed at the end of that refutation. Let us assume the user has input that a flip chart is a possible substitute for the overhead in room 36 for the database course. That input leads to the resolvent ← room_equipment(36, flip_chart). Since the relation room_equipment is extensional, it can be updated in order to satisfy that goal, provided the update does not violate integrity. The corresponding hypothesis room_equipment(36, flip_chart) is shown to be consistent in the derivation of rank 5, below. For brevity, the two SLIC steps needed to successfully terminate that consistency proof are not shown explicitly.

```
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) <—
                   |
                   :  (a step through 22' with a SLIC resolvent, and then another one through 23)
                   |
                  ==  output hypothesis  room_equipment(36, flip_chart)
```

Thus, SLDAI* hypothesizes room_equipment(36, flip_chart) ← and checks for its consistency, on rank 5. Since none of the denials depends on the hypothesis via a path going through an even number of negations, the hypothesis must be consistent. Hence, the consistency proof immediately terminates successfully, and 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 elaborated, since its structure is the same as in the previous database. 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 of removing the overhead from room 36, on rank 0. This justification of three hypotheses is finally translated into the extensional update of deleting the fact room_equipment(36, overhead) and inserting the fact room_equipment(36, flip_chart). Nothing needs to be done wrt the hypothesis ~inadequate(36, db), since it 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 now. 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 that

(*) substitute(overhead, _, _, board) <—

is requested to be inserted into the schema. It says that, no matter which course and which room, a board is always a possible substitute for an overhead. Again, processing this schema update with SLDAI* succeeds without any further changes. With (*) inserted, the computation of rank 4 above will branch into a subtree by using the inserted clause besides clause *29* as an additional input clause. 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).

Each update proposed by the outcome of SLDAI* must finally be committed or rejected by the user.

## References


[1] Celma, Mota, Garcia, Decker: Comparing and synthesizing procedures for integrity checking in deductive databases, Proc. 10th ICDE, IEEE Press, 1994.

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

[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: Abduction for knowledge assimilation in deductive databases, to appear in Proc. 17th Int'l Conf. of the Chilean Computer Society (SCCC'97), IEEE Press, 1997.

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

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

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

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

[10] Nüssel, Decker, Celma, Casamayor: A complete proof procedure for efficient integrity checking, Proc. 3rd Int'l Workshop on the Deductive Approach to Information Systems and Databases, LSI, UPC Barcelona, 1992.

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

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

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


### Acknowledgement


François Bry hosted me as a visitor at the computer science department of the University of München, where this paper was written. I also appreciate reviewers' feedback.