# INCONSISTENCY-TOLERANT KNOWLEDGE ASSIMILATION

Hendrik Decker
*Instituto Tecnológico de Informática, UPV, Campus de Vera 8G, Valencia, Spain*

Keywords: Inconsistency tolerance, knowledge assimilation, integrity maintenance, view updating, repair, databases.

Abstract: A recently introduced notion of inconsistency tolerance for integrity checking is revisited. Two conditions that enable an easy verification or falsification of inconsistency tolerance are discussed. Based on a method-independent definition of inconsistency-tolerant updates, this notion is then extended to a family of knowledge assimilation tasks. These include integrity maintenance, view updating and repair of integrity violation. Many knowledge assimilation approaches turn out to be inconsistency-tolerant without needing any specific knowledge about the given status of integrity of the underlying database.

## 1 INTRODUCTION

Knowledge assimilation (abbr.: *KA*) is the process of integrating new data into a body of information such that the latter's integrity remains satisfied (Kowalski, 1979; Miyachi et al, 1982; Decker, 1998; Kakas et al, 1998). For instance, KA takes place in data warehousing, decision support, diagnosis, quality assurance, content management, machine learning, robotics, vision, natural language understanding etc.

Also in more commonplace systems, KA is a very important issue. In databases, for example, a common instance of KA is *integrity maintenance*, i.e., when updates of relational tables are rejected or modified in order to preserve integrity. For example, the deletion of a row $r$ in table $T_1$ may not be possible without further ado if the primary key of $T_1$ is referenced by a foreign key constraint of table $T_2$. For maintaining the postulated referential integrity, the delete request for $r$ either necessitates the deletion of each row in $T_2$ that references $r$ or the insertion of a new or modified row $r'$ into $T_1$ with the same primary key values as in the referenced row $r$.

A somewhat more involved task of KA which subsumes integrity maintenance is *view updating*, i.e., the translation of a request for updating a virtual table of rows derivable from the view's defining query, to changes in the queried base tables. The goal of KA for view updating is to compute integrity-preserving translations for realizing update requests. Translations that would violate integrity are filtered out by KA. However, if, for some reason, integrity is violated, KA is called for to *repair* the violated constraints. Repairs that would violate other constraints are not valid. For instance, the deletion of referenced rows and the insertion of $r'$ in the preceding example are possible repairs for maintaining integrity.

All approaches to consistency-preserving KA employ some integrity checking mechanism, for making sure that the assimilation of a new piece of knowledge will not violate any constraint, i.e., that integrity satisfaction is an invariant of database state transitions. Usually, KA methods require that each constraint be satisfied by the underlying database before assimilating new knowledge, i.e., that integrity satisfaction is total. As shown in (Decker and Martinenghi, 2006b), many methods ensure that all consistent parts of the database remain consistent even when the strict requirement of total integrity satisfaction is waived. So, it comes as no surprise that this requirement can be abandoned for KA in general.

In section 2, we revisit definitions and results for inconsistency-tolerant integrity checking. In section 3, we discuss two conditions that ensure inconsistency tolerance. In section 4, we generalize the definitions and results of sections 2 and 3 to integrity maintenance, view updating and inconsistency repair. In the concluding section 5, we also address related work and look out to future research. Throughout, we use terms and notations of standard database logic.



Decker H. (2007).
INCONSISTENCY-TOLERANT KNOWLEDGE ASSIMILATION.
In *Proceedings of the Second International Conference on Software and Data Technologies - PL/DPS/KE/WsMUSE*, pages 198-205
DOI: 10.5220/0001331701980205

## 2 INCONSISTENCY-TOLERANT INTEGRITY CHECKING

Recall that integrity constraints are well-formed sentences of first-order predicate calculus. W.l.o.g., we assume they are represented in prenex form (i.e., roughly, all quantifiers explicitly or implicitly appear outermost), which subsumes prenex normal form (i.e., prenex form with all negations innermost) and denial form (i.e., clauses with empty conclusion). In each database state, they are required to be *satisfied*, i.e. true in, or at least consistent with that state. Otherwise, they are said to be *violated* in $D$.

An *integrity theory* is a finite set of integrity constraints. It is *satisfied* if each of its members is satisfied, and *violated* otherwise. Let $IC$ optionally stand for an integrity constraint or an integrity theory, and $D$ be a database. With $D(IC) = sat$, we express that $IC$ is satisfied in $D$, and $D(IC) = vio$ that it is violated. Moreover, for an update $U$, let $D^U$ denote the database obtained from executing $U$ on $D$; $D$ and $D^U$ then are referred to as *old* and *new* state, respectively.

Different integrity checking methods use different notions to define and determine integrity satisfaction and violation. Abstracting away from such differences, each integrity checking method $\mathcal{M}$ can be formalized as a function that takes as input a database, an integrity theory and an update (i.e., a bipartite finite set of database clauses to be deleted and inserted, resp.). It outputs the value *sat* if it has concluded that integrity will remain satisfied in the new state, and outputs *vio* if it has concluded that integrity will be violated in the new state. Thus, the soundness and completeness of $\mathcal{M}$ can be stated as follows.

**Definition 1** An integrity checking method $\mathcal{M}$ is *sound* if, for each database $D$, each integrity theory $I$ such that $D(I) = sat$ and each update $U$, the following holds.

$$\text{If } \mathcal{M}(D,I,U) = sat \text{ then } D^U(I) = sat. \quad (1)$$

Completeness of $\mathcal{M}$ can be defined dually to def. 1, by the only-if half of (1). Note that both definitions are impartial to the question whether *sat* and *vio* are the only values that could be output by $\mathcal{M}(D,I,U)$ (another value could be, e.g., *unknown*). However, for simplicity, we do not consider any semantics of integrity that would have values other than "satisfied" and "violated" for integrity.

Def. 1 and its dual apply to virtually any integrity checking method in the literature. Of course, each of them is defined for certain classes of databases, constraints and updates (e.g., relational or stratified databases, range-restricted constraints and transactions consisting of insertions and deletions of base facts). So, whenever we say "each" (database, integrity theory, update), we mean all those for which the respective methods are defined at all. From now on, each method $\mathcal{M}$ is assumed to be sound.

For significant classes of databases and integrity theories, soundness and completeness has been shown for methods in (Nicolas, 1982; Decker, 1986; Lloyd et al, 1987; Sadri and Kowalski, 1988; Christiansen and Martinenghi, 2006) and others; also their termination, as defined below, can be shown. Other methods, e.g., (Gupta et al, 1994; Lee and Ling, 1996), are only sound, i.e., they provide sufficient conditions that guarantee the integrity of the updated database. If these conditions do not hold, further checks may be necessary. For later (theorem 4), also the termination of methods is of interest.

**Definition 2** An integrity checking method $\mathcal{M}$ is said to be *terminating* if, for each database $D$, each integrity theory $I$ and each update $U$, the computation of $\mathcal{M}(D,I,U)$ halts and outputs either *sat* or *vio*.

Note that by def. 2, the computation of $\mathcal{M}(D,I,U)$ terminates, no matter whether $D(I) = sat$ or not. Thus, such an $\mathcal{M}$ is complete, although it is clear that complete methods are not necessarily terminating.

Definitions 1 and 2 are independent of the diversity of criteria by which methods often are distinguished, e.g., to which classes of databases, constraints and updates they apply, how efficient they are, which parts of the data in $(D,U,I)$ are actually accessed, whether they are complete or not, whether constraints are "soft" or "hard", whether integrity is checked in the old or the new state, or whether simplification steps are pre-compiled at schema specification time or taken at update time. Such distinctions are studied, e.g., in (Martinenghi et al, 2006b; Decker and Martinenghi, 2007) but do not matter much in this paper, except when explicitly mentioned.

Common to all methods is that they require *total integrity satisfaction*, i.e., before each update, each constraint must be completely satisfied. In (Decker and Martinenghi, 2006b), we have shown how this requirement can be relaxed. Informally speaking, it is in fact possible to tolerate (i.e., live with) individual inconsistencies in the database, i.e., hopefully minoritarian cases of violated constraints, while trying to make sure that updates do not cause any new cases of integrity violation, i.e., that the cases of constraints that were satisfied in the old state remain satisfied in the new state. The following definitions revisit previous ones in (Decker and Martinenghi, 2006b) for formalizing what we mean by "case" and "inconsistency tolerance" of integrity checking.

**Definition 3** Let $C$ be an integrity constraint. The variables in $C$ that are $\forall$-quantified but not dominated by any $\exists$ quantifier (i.e., $\exists$ does not occur left of $\forall$) are called *global variables of C*. For a substitution $\sigma$ of the global variables of $C$, $C\sigma$ is called a *case* of $C$.

For convenience, a case of some constraint in an integrity theory $I$ is shortly called a *case of I*.

Note that cases have themselves the form of integrity constraints, and need not be ground. In particular, each constraint is a case of itself.

We remark that the following definition is somewhat more succinct than its equivalent in (Decker and Martinenghi, 2006b).

**Definition 4** An integrity checking method $\mathcal{M}$ is *inconsistency-tolerant* if, for each database $D$, each integrity theory $I$, each case $C$ of $I$ such that $D(C) = sat$, and each update $U$, the following holds.

$$\text{If } \mathcal{M}(D,I,U) = sat \text{ then } D^U(C) = sat. \qquad (2)$$

In general, inconsistent cases may be unknown or not efficiently recognizable. However, by def. 3, inconsistency-tolerant methods are able to blindly cope with any degree of inconsistency. They guarantee that all cases of constraints that were satisfied in the old state will remain satisfied in the new state. Running such a method $\mathcal{M}$ means to compute the very same function as if total satisfaction were required. Since $\mathcal{M}$ does not need to be aware of any particular case of violation, no efficiency is lost, whereas the gains are immense: transactions can continue to run even in the presence of (obvious or hidden, known or unknown) violations of integrity (which is rather the rule than the exception in practice), while maintaining the integrity of all satisfied cases. Running $\mathcal{M}$ means that no new cases of integrity violation will be introduced, while existing "bad" cases may disappear (intentionally or even accidentally) by committing updates that have successfully passed the integrity test.

As shown in (Decker and Martinenghi, 2006b), inconsistency tolerance is available off the shelve, since most, though not all known approaches to database integrity are inconsistency-tolerant. The following examples illustrate this.

**Example 1** Let $C = \leftarrow b(x,y) \wedge b(x,z) \wedge y \neq z$ be the constraint that no two entries with the same ISBN $x$ in the relation $b$ about books must have different titles $y$ and $z$. Suppose $U$ is to insert $b(11111, logic)$. The simplification $C' = \leftarrow b(11111, y) \wedge y \neq logic$ is generated and evaluated by most methods, with output *sat* if the query $C'$ returns the empty answer, and *vio* otherwise. With the traditional prerequisite of total integrity satisfaction, this output says that $D^U$ satisfies or, resp., violates integrity. Now, suppose that $b(88888, t_1)$ and $b(88888, t_2)$ are in $D$, possibly together with many other facts in $b$. Clearly, the case $\leftarrow b(88888, t_1) \wedge b(88888, t_2) \wedge t_1 \neq t_2$ of $C$ is violated in $D$, i.e., integrity is not totally satisfied. However, the insertion of $b(11111, logic)$ is guaranteed not to cause any additional violation as long as the evaluation of $C'$ yields the output *sat*, i.e., as long as there is no other entry in $b$ with ISBN 11111.

Before, or instead of, evaluating simplified instances of relevant constraints, some methods, e.g. (Gupta et al, 1994), may reason on the integrity theory alone for detecting the possible invariance of integrity satisfaction by given updates. That, however, may fail to be inconsistency-tolerant, as illustrated below.

**Example 2** Consider $I = \{\leftarrow q(x), \leftarrow q(a), r(b)\}$, $D = \{q(a)\}$ and $U = insert\ r(b)$. Clearly, the case $\leftarrow q(a)$ of $\leftarrow q(x)$ is violated while all other cases of $I$ are satisfied. A typical simplification of $\leftarrow q(a), r(b)$ (which, unlike $\leftarrow q(x)$, is relevant for $U$) is $\leftarrow q(a)$; the conjunct $r(b)$ is dropped because $U$ makes it true. Methods that reason with possible subsumptions of simplifications by the integrity theory then easily detect that the simplification above is subsumed by the constraint $\leftarrow q(x)$. Using the intolerant assumption of total integrity satisfaction in the old state then leads to the faulty output *sat*, by the following argument: The constraint $\leftarrow q(x)$, which is assumed to be satisfied in $D$, is not relevant wrt $U$. Thus, it can be assumed to remain satisfied in $D^U$. So, since this constraint subsumes the simplification $\leftarrow q(a)$, integrity will remain satisfied. This argument, which is correct if integrity is totally satisfied in the old state, fails to be inconsistency-tolerant since it fails to identify the violated case of $\leftarrow q(a), r(b)$ caused by $U$.

# 3 VERIFYING AND FALSIFYING INCONSISTENCY TOLERANCE

To verify or falsify condition (2) of def. 4 for a given method can be laborious. However, there are various sufficient conditions by which inconsistency tolerance can be verified much more easily. Two of them are presented in theorems 1 and 4, below. The first has been used in (Decker and Martinenghi, 2006a,b) to verify inconsistency tolerance of the methods in (Nicolas, 1982; Decker, 1986; Lloyd et al, 1987; Sadri and Kowalski, 1988). The second is new. It also is a necessary condition, i.e., it also serves to falsify inconsistency tolerance. It arguably is even more apt to show or disprove the inconsistency tolerance of

the already mentioned and other methods. Theorem 1 states that inconsistency tolerance is entailed by the first condition, labeled (3) below.

**Theorem 1** A method $\mathcal{M}$ for integrity checking is inconsistency-tolerant if, for each database $D$, each integrity theory $I$, each case $C$ in $I$ such that $D(C) = sat$, and each update $U$, the following holds.

$$\text{If } \mathcal{M}(D,I,U) = sat \text{ then } \mathcal{M}(D,\{C\},U) = sat \quad (3)$$

**Proof** Clearly, (2) follows from the transitivity of (3) and (4):

$$\text{If } \mathcal{M}(D,\{C\},U) = sat \text{ then } D^U(C) = sat \quad (4)$$

where (4) obviously is a special case of (1). □

The second condition for verifying inconsistency tolerance is based on def. 5 below. Part *a)* is interesting also in itself because it provides a method-independent notion of inconsistency tolerance. Note that def. 5 does not require any constraint to be satisfied in $D$.

**Definition 5**
***a)*** For a database $D$ and an integrity theory $I$, an update $U$ *causes violation* if there is a case $C$ of $I$ such that $D(C) = sat$ and $D^U(C) = vio$. If, for each case $C$ of $I$, $D(C) = sat$ entails $D^U(C) = sat$, then $U$ is called *inconsistency-tolerant*.
***b)*** For an integrity checking method $\mathcal{M}$, we say that $\mathcal{M}$ *recognizes violation* if, for each database $D$, each integrity theory $I$ and each update $U$ that causes violation, $\mathcal{M}(D,I,U) = vio$.

Theorem 2 below relates the two parts of def. 5*a* and follows by definition. Theorem 3 is a corollary of 5*a* and def. 4. It states that updates can be checked for inconsistency tolerance by inconsistency-tolerant integrity checking methods. Theorem 4 relates def. 5*b* to def. 4.

**Theorem 2** For a given database and a given integrity theory, an update $U$ is inconsistency-tolerant if and only if it does not cause violation. □

We remark that theorem 2 would not hold if the semantics of integrity were not two-valued.

**Theorem 3** For a database $D$, an integrity theory $I$ and an inconsistency-tolerant integrity checking method $\mathcal{M}$, an update $U$ is inconsistency-tolerant if $\mathcal{M}(D,I,U) = sat$. □

In general, the only-if half of theorem 3 does not hold. For example, consider a view $p$ defined by $p(x,y) \leftarrow s(x,y,z)$ and $p(x,y) \leftarrow q(x), r(y)$ in a database $D$ in which $q(a)$ and $r(a)$ are the only tuples that contribute to the natural join of relations $q$ and $r$. Further, let $I$ consist of the constraint $\leftarrow p(x,x)$, and $U$ be the insertion of the tuple $s(a,a,b)$. Clearly, $U$ does not cause violation, since the case $C = \leftarrow p(a,a)$ is already violated in $D$. Hence, by theorem 2, $U$ is inconsistency-tolerant. However, the inconsistency-tolerant methods in (Lloyd et al, 1987; Sadri and Kowalski, 1988) and others compute and evaluate the simplification $\leftarrow p(a,a)$ of $\leftarrow p(x,x)$ and thus output *vio*. On the other hand, note that inconsistency-tolerant methods which check for idle updates (e.g., the one in (Decker, 1986)) identify $p(a,a)$ as idle (i.e., a consequence of the update that is already true in the old state) and hence output *sat*.

**Theorem 4** Let $\mathcal{M}$ be a terminating integrity checking method. Then, $\mathcal{M}$ is inconsistency-tolerant if and only if it recognizes violation.

**Proof**
*If:* Let $\mathcal{M}$ be a method that recognizes violation, $U$ an update such that $\mathcal{M}(D,I,U) = sat$, and $C$ a case of a constraint in $I$ such that $D(C) = sat$. We have to show that $D^U(C) = sat$. Since $\mathcal{M}(D,I,U) = sat$, it follows from theorem 2 that $U$ does not cause violation, i.e., there is no case of any constraint in $I$ that is satisfied in $D$ and violated in $D^U$. Thus, $D(C) = sat$ implies $D^U(C) = sat$. □
*Only if:* Let $\mathcal{M}$ be inconsistency-tolerant and suppose that $U$ causes violation. So, we have to show that $\mathcal{M}(D,I,U) = vio$. Since $U$ causes violation, there is a case $C$ such that $D(C) = sat$ and $D^U(C) = vio$. Hence, inconsistency tolerance of $\mathcal{M}$ entails by def. 3 that $\mathcal{M}(D,I,U) \neq sat$. Since $\mathcal{M}$ is terminating, it follows that $\mathcal{M}(D,I,U) = vio$. □

We remark that termination of $\mathcal{M}$ is used only in the proof of the *only-if* half. However, the last steps in each half of the proof rely on the assumption that the semantics of integrity is two-valued.

# 4 GENERALIZATIONS FOR KA

As already indicated, our focus is on the KA tasks of integrity maintenance across updates, satisfaction of view update requests, and reparation of violated integrity constraints. Common to each of them and also other KA tasks is that they generate updates as candidate solutions where the integrity of the state obtained by executing such an update is one of possibly several filter criteria for distinguishing valid can-

didates. Other criteria typically ask for minimality of (the effect of) updates, or use some additional preference ordering, to select among valid candidates. For instance, integrity maintenance may sanction a given update after having checked it successfully for integrity preservation, or otherwise either reject or modify it so that integrity remains invariant.

Since integrity checking is an integral part of KA, the requirement of total satisfaction of all constraints has traditionally been postulated also by all methods for tackling the mentioned tasks. However, this requirement appears as unrealistic for KA in general as for mere integrity checking. In fact, it can be abandoned just as well, as shown in theorem 5 below. The latter relies on the following definition, which in turn recurs of def. 5*a*.

**Definition 6**
A KA method $\mathcal{K}$ is *inconsistency-tolerant* if each update generated for tackling the task of $\mathcal{K}$ is inconsistency-tolerant.

Similar to definitions 1 and 4, def. 6 is as abstract as to apply to virtually all KA methods in the literature. We repeat that such methods originally have not been meant to be applied in case the current database state is inconsistent with its constraints. Strictly speaking, they are not even defined for such situations. However, the clue of inconsistency-tolerant methods is that, by definition, they produce reliable results even when they are run in situations for which they originally have not been thought for. And the justification for the definition of inconsistency tolerance is that many methods turn out to comply with it. Thus, def. 6 provides a basis for KA to be applicable also if the underlying database is not fully consistent with its integrity constraints. In particular, the generated updates still are going to achieve what they are supposed to achieve. More precisely, view updating methods compute updates that make update requests true, and repair methods turn violated cases of constraints into satisfied cases, while the overall state of integrity is not exacerbated by the respective updates.

We remark that theorem 3 does not readily provide a means to test a given KA method $\mathcal{K}$ for inconsistency tolerance, because def. 6 asks that *each* update that ever might be generated by $\mathcal{K}$ have that property. It might be said that def. 6 could be relaxed to the extent that not all, but just one of the generated updates would have to be inconsistency-tolerant. Then, a further test by an inconsistency-tolerant integrity checking method could act as a filter for eliminating updates that would cause violation. However, that would in fact amount to the definition of a modified KA method, extended by an inconsistency-tolerant integrity checking method. The following theorem reflects the usefulness of such integrity checking methods for KA.

**Theorem 5** Each KA method that uses an inconsistency-tolerant method to check updates for not causing violation is inconsistency-tolerant.

**Proof** This result follows straightforwardly from def. 6 and theorems 2 and 3. □

## 4.1 Inconsistency-tolerant View Updates

Theorem 5 serves to recognize several known view update methods as inconsistency-tolerant, due to their use of suitable integrity checking methods. Among them are the view updating methods in (Decker, 1990; Guessoum and Lloyd, 1990a,b), as stated in theorem 6 below. For convenience, let us name them $\mathcal{D}ec$ and $\mathcal{GL}$, respectively.

**Theorem 6**
***a)*** The view update method $\mathcal{D}ec$ is inconsistency-tolerant.
***b)*** The view update method $\mathcal{GL}$ is inconsistency-tolerant.

**Proof**
*a)* $\mathcal{D}ec$ uses the inconsistency-tolerant integrity checking method in (Decker, 1986) for filtering out generated update candidates that would cause violation. □
*b)* $\mathcal{GL}$ uses the inconsistency-tolerant integrity checking method in (Lloyd et al, 1987) for filtering out generated update candidates that would cause violation. □

A related method is described in (Kakas and Mancarella, 1990a,b). For convenience, let us name it $\mathcal{KM}$. It does not use any integrity checking method as a separate module, hence theorem 5 is not applicable. However, the inconsistency tolerance of $\mathcal{KM}$ can be tracked down as outlined in the remainder of this subsection.

For satisfying a given view update request, $\mathcal{KM}$ explores a possibly nested search space of "abductive" derivations and "consistency" derivations. Roughly, the goal of abductive derivations is to find successful deductions of a requested update, by which base table updates that satisfy the request are obtained; consistency derivations check these updates for integrity. Each update obtained that way consists of a set of positive and a set of negative literals that are all ground. Positive literals correspond to insertions, negative ones to deletions of rows in base relations.

For more details, which are not included here for lack of space, we refer the reader to the original papers as cited above. It may suffice here to mention that, for $\mathcal{KM}$, all constraints are assumed to be represented by denial clauses, so that they can be used as candidate input in consistency derivations.

It is easy to verify that, for an update request $R$, each update $U$ computed by $\mathcal{KM}$ satisfies $R$, i.e., $R$ is true in $D^U$ even if some constraint is violated in $D$. What is at stake is the preservation of integrity in $D^U$, for each case that is satisfied in $D$, while unknown or irrelevant cases that are violated in $D$ may remain to be violated in $D^U$. The following theorem states that satisfied cases are preserved by $\mathcal{KM}$.

**Theorem 7** The view update method $\mathcal{KM}$ is inconsistency-tolerant.

**Proof** By theorem 2, it suffices to show that each update computed by $\mathcal{KM}$ does not cause violation. To initiate a *reductio ad absurdum* argument, suppose that, for some update request in some database with some integrity theory $I$, $\mathcal{KM}$ computes an update $U$ that causes violation. Then, by def. 2 and def. 5*a*, there is a case $D'$ of some constraint $C$ in $I$ such that such that $D(C') = sat$ and $D^U(C') = vio$. Thus, *a fortiori*, $D(C) = sat$ and $D^U(C) = vio$. Hence, by the definition of $\mathcal{KM}$, there is a consistency derivation $\delta$ rooted at one of the base literals in $U$, that uses $C$ as input clause in its first step and terminates by deducing the empty clause. However, termination of any consistency derivation with the empty clause signals inconsistency, i.e., constraint violation. Hence, by definition, $\mathcal{KM}$ rejects $U$, because $\delta$ indicates that its root causes violation of $C$. Thus, $\mathcal{KM}$ never computes updates that would cause violation. □

## 4.2 Inconsistency-tolerant Repairs

Repairing a database that is inconsistent with its integrity constraints can be difficult, for several reasons. For instance, there may be (too) many alternatives of possible repairs, even if a lot of options are filtered out by minimality or other selection criteria. To choose suitable filtering criteria can be a significant problem on its own already. Also, repairs can be prohibitively costly, due to the complexity of constraints and intransparent interactions between them and the stored data; cf., e.g., (Lopatenko and Bertossi, 2007). And, worse, the existence of unknown inconsistencies (which is common in practice) may completely foreclose the repair of known constraint violations, under the traditional inconsistency-intolerant semantics of clasical first-order logic.

To see this, suppose that, for a database $D$, $C_0$ is a case of some constraint $C$, the violation of which is unknown, i.e., both $D \cup \{C_0\}$ and $D \cup \{C\}$ are inconsistent. Further, $C_1$ be a known violated case of the same or some other constraint, which is to be repaired. In general, all integrity constraints need to be taken into account for repairing violations, due to possible interdependencies between them. However, classical logic does not sanction any result of reasoning in an inconsistent theory, since anything (and thus nothing reliable at all) may follow from inconsistency. Thus, no repair of any known inconsistency can be trusted, unless it can be ensured that there is no unknown inconsistency. So, since it is hard to know about the unknown, repair may seem to be a hopeless task, in general.

Fortunately, inconsistency tolerance comes to the rescue. In the preceding example, an update $U_1$ such that $D^{U_1}(C_1) = sat$ can be obtained by running any inconsistency-tolerant view update method on the request to make $C_1$ true. Each terminating method will produce such an update $U_1$, independent of the integrity status of $C_0$, while all other cases of constraints that are satisfied in $D$ remain satisfied in $D^{U_1}$.

For a database $D$, inconsistency-tolerant view updating can in general be used either for repairing all violated constraints in one go, or, if that task is too big, for repairing violated (cases of) constraints incrementally, as follows. W.l.o.g., suppose that all constraints $C_1,...,C_n$ ($n > 1$) are represented as denial-like clauses of form $violated \leftarrow B_i$ $(1 \leq i \leq n)$, where *violated* be a distinguished view predicate that is not used for any relation in $D$, and $B_i$ is an existentially closed formula with predicates defined in $D$. A constraint of that form is satisfied if and only if $B_i$ is not true in the given database state. So, to repair all violated constraints in one go, the view update request $\sim violated$ can be issued in $D \cup \{C_1,...,C_n\}$, asking that *violated* be not true (cf. (Decker at al, 1996). It is easy to see that any terminating inconsistency-tolerant view update method will return the required repair.

Otherwise, the following incremental approach may be tried. For each $i$ at a time, the update request $\sim B_i$ be issued and satisfied, if possible, by an inconsistency-tolerant view update method. Clearly, the end result will in general depend on the sequence of the $C_i$. Here, as with any policy for choosing among alternative updates for satisfying a request, application-specific considerations may help.

For instance, suppose the management of some enterprise has decided to dissolve their research department. In the database of that enterprise, let a foreign key constraint of the *works-in*(*EMP*,*DEPT*) relation ask for the occurrence of the second attribute's

value of each tuple of *works-in* in the primary key's value of some tuple in the *dept* relation. To repair the cases of this constraint that have become violated by the deletion of the tuple *dept*(*research*), the following updates can be performed.

First, a downsized new research-oriented department is established by inserting the fact *dept*(*investigation*). No violation of any key constraint is caused by that. Then, for each employee *e* of the defunct research department, the tuple *works-in*(*e*, *research*) either is dropped (i.e., *e* is fired) or replaced by *works-in*(*e*, *investigation*), or replaced by *works-in*(*e*, *development*), for some already existing department *development*.

As an aside, we remark that the last two of the three alternaive repairs of this example, which is quite typical for reorganizing enterprise departments, may also serve to criticize the adequacy of the usual minimality criteria in the literature, since they comply with none of them.

More importantly, note that each such repair is not acceptable by any inconsistency-intolerant method that would insist on total integrity satisfaction, because some violated cases of constraints are likely to survive across updates. However, each repair that does not cause violation of any of the mentioned constraints is sanctioned by inconsistency-tolerant methods that check the preservation of all satisfied cases.

## 5 CONCLUSION

The semantic consistency of data is a major concern of knowledge engineering. Consistency requirements usually are expressed by integrity constraints. Knowledge assimilation methods are employed for preserving constraint satisfaction across changes. To go for total satisfaction, as most known approaches do, is unrealistic. To relax that, we have revisited and extended a notion of inconsistency tolerance. We have shown that it is possible to use existing KA methods for checking and preserving integrity upon updates, for satisfying view update requests and for repairing violated constraints, even if the knowledge suffers from inconsistencies.

Arguably, our concept of inconsistency tolerance is less complicated and more effective than the one associated to the field of consistent query answering (CQA) (Bertossi and Chomicki, 1999) and others, as documented in (Bertossi et al, 2005). The latter of course have several other merits of their own that are not questioned by inconsistency tolerance as discussed in this paper. In fact, we expect that our work, and in particular our notion of inconsistency-tolerant repair, can be beneficial for the further development of CQA. We intend to look into this in future research. We also intend to investigate the capacity of inconsistency tolerance of advanced procedures such as in (Dung et al, 2006).

## ACKNOWLEDGEMENTS

The author wishes to thank Davide Martinenghi for utterly useful discussions.

This work has been partially supported by FEDER and the Spanish MEC grant TIN2006-14738-C02-01.

## REFERENCES

Arenas, M., Bertossi, and Chomicki, J. (1999). Consistent Query Answers in Inconsistent Databases. *Proc. 18th PODS*, 68-79. ACM Press.

Bertossi, L., Hunter, A. and Schaub, T. (2005). *Inconsistency Tolerance*. Springer LNCS vol. 3300.

Christiansen, H. and Martinenghi, D. (2006). On Simplification of Database Integrity Constraints. *Fundam. Inform.* 71(4):371-417.

Decker, H. (1987). Integrity enforcement on deductive databases. *Proc. EDS'86*, 381-395. Benjamin/Cummings.

Decker, H. (1990). Drawing Updates From Derivations. *Proc. 3rd ICDT*, 437-451. Springer LNCS vol. 470.

Decker, H. (1998). Some Notes on Knowledge Assimilation in Deductive Databases. *Transactions and Change in Logic Databases*, 249-286. Springer LNCS vol. 1472.

Decker, H. and Martinenghi, D. (2006a). Checking Violation Tolerance of Approaches to Database Integrity. *Proc. 4th ADVIS*, 139-148. Springer LNCS vol. 4243.

Decker, H. and Martinenghi, D. (2006b). A Relaxed Approach to Integrity and Inconsistency in Databases. *Proc. 13th LPAR*, 287-301. Springer LNCS vol. 4246.

Decker, H. and Martinenghi, D. (2007). Getting Rid of Straitjackets for Flexible Integrity Checking. *To appear in Proc. DEXA'07 Workshop FlexDBIST-07*.

Decker, H., Teniente, E. and Urpí, T. (1996). How to Tackle Schema Validation by View Updating. *Proc. 5th EDBT*, 535-549. Springer LNCS vol. 1057.

Dung, P. M., Kowalski, R. A. and Toni, F. (2006). Dialectic proof procedures for assumption-based, admissible argumentation. *Artif. Intell.* 170(2):114-159.

Guessoum, A. and Lloyd, J. (1990a) Updating Knowledge Bases. *New Generation Comput.* 8(1):71-89.

Guessoum, A. and Lloyd, J. (1990b) Updating Knowledge Bases II. *New Generation Comput.* 10(1):73-100.

Gupta, A., Sagiv, Y., Ullman, J. and Widom, J (1994). Constraint checking with partial information. *Proc. 13th PODS*, 45-55. ACM Press.

Kakas, A. and Mancarella, P. (1990a). Database Updates through Abduction. *Proc. 16th VLDB*, 650-661, Morgan Kaufmann.

Kakas, A. and Mancarella, P. (1990b). Knowledge Assimilation and Abduction. *Truth Maintenance Systems*, 54-70. Springer LNCS vol. 515.

Kakas, A., Kowalski, R. A. and Toni, F. (1998). The Role of Abduction in Logic Programming. *Handbook of Logic in Artificial Intelligence and Logic Programming*, 235-324. Oxford University Press.

Kowalski, R. A. (1979). *Logic for Problem Solving*. North-Holland, 1979.

Lee, S. Y. and Ling, T. W. (1996) Further improvements on integrity constraint checking for stratifiable deductive databases. *Proc. 22nd VLDB*, 495-505. Morgan Kaufmann.

Lloyd, J., Sonenberg, L. and Topor, R. (1987). Integrity constraint checking in stratified databases. *J. Logic Progr.* 4(4):331-343.

Lopatenko, A. and Bertossi, L. (2007). Complexity of Consistent Query Answering in Databases under Cardinality-Based and Incremental Repair Semantics. *Proc. 11th ICDT*, 179-193. Springer LNCS vol. 4353.

Martinenghi, D., Christiansen, H. and Decker, H. (2006). Integrity Checking and Maintenance in Relational and Deductive Databases and Beyond. In Zongmin Ma (ed): *Intelligent Databases: Technologies and Applications*, 238-285. Idea Group, 2006.

Miyachi, T., Kunifuji, S., Kitakami, H., Furukawa, K., Takeuchi, A. and Yokota, H. (1984). A Knowledge Assimilation Method for Logic Databases. *New Generation Comput.* 2(4):385-404.

Nicolas, J. M. (1982). Logic for improving integrity checking in relational data bases. *Acta Informatica* 18:227-253.

Sadri, F. and Kowalski, R. A. (1988) A theorem-proving approach to database integrity. *Foundations of Deductive Databases and Logic Programming*, 313–362. Morgan Kaufmann.