

Integrity Checking for Uncertain Data

Hendrik Decker^{*}
Instituto Tecnológico de Informática
E-46071 Valencia, Spain
hendrik@iti.upv.es

Davide Martinenghi[†]
Free University of Bozen/Bolzano
I-39100 Bolzano, Italy
martinenghi@inf.unibz.it

ABSTRACT

The uncertainty associated to stored information can be put in direct correspondence to the extent to which these data violate conditions expressed as semantic integrity constraints. Thus, imposing and checking such constraints provides a better control over uncertain data. We present and discuss a condition which ensures the violation tolerance of methods for integrity checking. Usually, such methods are supposed to work correctly only if all constraints are satisfied before each update. Applied to express and check conditions about uncertain data, violation tolerance means that stored data the uncertainty of which violates integrity can be tolerated while updates can be safely checked for introducing violations of constraints about uncertainty. We also discuss the soundness and completeness of violation-tolerant integrity checking and assert it for several methods.

1. INTRODUCTION

In general, the expressive power of arbitrary first-order predicate logic sentences, called “integrity constraints”, is needed in databases to express “semantic” information that goes beyond their otherwise comparatively simplistic positive factual content. In particular, conditions for characterising semantically imperfect data can be expressed by such constraints. The idea is that the database is considered to be free of semantically imperfect data if and only if its given state satisfies the imposed constraints. Otherwise, if any of the constraints is violated, i.e., not satisfied, the database is taken to contain imperfect data.

In this paper, we propose to gain a better control over uncertain data by expressing semantic properties about them

^{*}supported by the Spanish grant TIC2003-09420-C02i

[†]supported by the TONES IST project financed by the European Union 6th Framework Programme under contract number FP6-7603

(e.g., conditions that qualify stored data as uncertain) by integrity constraints. If each such property is satisfied, there are no uncertain data that would violate the constraints. Conversely, violation means that the data that are responsible for violation may qualify as uncertain. By capturing uncertainty properties of data by integrity constraints, it can be hoped to achieve a double benefit: firstly, to use the expressive power of the syntax of semantic integrity constraints also for describing arbitrarily general uncertainty properties of data; secondly, to make use of established integrity checking methods in order to efficiently check data also for uncertainty. Thus, monitoring and controlling stored and incoming new data and updates with regard to their potential uncertainty can be enhanced.

In section 2, we first try to gain a better understanding of the similarities and differences, respectively, between integrity and the lack of uncertainty, on one hand, and between violated integrity and uncertainty, on the other. Then, we claim that, in spite of seemingly severe differences, it is indeed possible to represent conditions for capturing uncertainty in the form of integrity constraints, and to check them by making use of well-known methods for integrity checking. In the remainder of the paper, we substantiate this claim. In section 3, we propose a general definition of the violation tolerance of any given integrity checking method. In section 4, we present and discuss a condition which ensures violation tolerance. Also, the soundness of violation-tolerant integrity checking is asserted for several methods. In section 5, we define and assert properties of completeness of violation-tolerant integrity checking. In section 6, we observe a preponderance of attention paid to satisfaction, and counter-balance it by highlighting violation of integrity. In section 7, we address related work and conclude.

2. INTEGRITY AND UNCERTAINTY

In subsection 2.1, we look at similarities, in 2.2 at differences between integrity and uncertainty. It will turn out that, regardless of any differences, the representation of conditions about uncertainty by integrity constraint syntax is possible, and also their efficient evaluation by well-known integrity checking methods.

2.1 Integrity and Uncertainty: Similarities

Traditionally, integrity constraints are a means to express correctness conditions with which all stored data must com-

ply. Upon each attempt to update data that may cause a violation of integrity, the constraints imposed on the database are checked, and the update is committed only if it does not cause integrity violation. For example, in a civil registry database containing information about citizens including their marital status, entering $\text{married}(\text{john}, \text{mary})$ will violate $\forall x \forall y \forall z (\text{married}(x, y) \wedge \text{married}(x, z) \rightarrow y = z)$, i.e., a constraint forbidding bigamy, if $\text{married}(\text{john}, \text{susan})$ is already stored. Also, in the presence of this tuple, the constraint $\forall x \forall y \text{married}(x, y) \rightarrow \text{person}(x, m) \wedge \text{person}(y, m)$ for requiring an entry for each spouse of each married couple, with marital status attribute set to $m(\text{arried})$ in the *person* table of the database will signal violation upon an attempt to delete the tuple $\text{person}(\text{susan}, m)$.

Also conditions for characterising database entries as uncertain can be expressed in the syntax of integrity constraints. Uncertain data can be understood to be either imprecise or incomplete in some sense, or indefinite, or precarious (i.e., unsafe, e.g., a potential security risk), or dubious (suspicious), or potentially malign, or have less than 100% (though not 0%) probability of truth or confidence. For example, $\text{uncertain} \leftarrow \text{person}(x, \text{null})$ qualifies each entry in the *person* table as uncertain by a namesake 0-ary predicate if the marital status of that entry is unknown, as represented by a *null* value. Or, $\forall x \text{person}(x, \text{null}) \rightarrow \text{person}(x, s) \vee \text{person}(x, m) \vee \text{person}(x, d) \vee \text{person}(x, w)$ says that each person with unknown marital status is either single or married or divorced or widowed. Similarly, entries of persons with birth date before the 20th century can be characterized as dubious. By amalgamating higher-order predicates into first-order terms [14], also sentences such as $\text{uncertain} \leftarrow \text{confidence}(\text{table-entry}(x, y), z) \wedge z < \text{threshold}$ may serve as constraints for capturing uncertainty about entries x in database tables y such that the confidence factor z of x is below a certain threshold (where *threshold* is some possibly parameterisable constant).

Now, it may perhaps be perceived as a bit of a stretch to also subsume data which violate conventional integrity constraints as uncertain, since they are usually considered to be certainly bad, while uncertain data are not certainly bad. But at least it seems fair to consider data which violate integrity as border cases of uncertain data. Be that as it may, we have seen that it is possible to qualify uncertain data in the syntax of integrity constraints. Hence, it should also be possible to check incoming data for violations of constraints about uncertainty (and, symmetrically, check deletions for having the effect of making existing data uncertain, such as deletions of person records which may render the marital status information of their spouse inconsistent or unknown).

Thus, the satisfaction of each constraint can be seen as a necessary condition for avoiding uncertainty about the stored information. Conversely, data that violate integrity undoubtedly convey some degree of uncertainty. Upon each attempt to update data that are potentially uncertain since they may violate integrity, the integrity constraints of the database should be checked. The update can certainly be committed if it does not cause integrity violation. Otherwise, a warning of uncertainty can be issued, and further action may be taken before or after rejecting or committing (a possibly modified version of) the update.

The plain evaluation of constraints upon each update may be unfeasibly expensive, due to their arbitrarily high query complexity. However, satisfaction or violation of constraints can be checked by well-known methods (e.g., [21, 5, 17, 22, 10, 4]) that are supposed to work in a significantly more efficient manner than plain evaluation. Yet, the price to be paid for this gain of efficiency is possibly as high as to make such methods useless in practice: in order to ensure the correctness of their output, each of them assumes that integrity be satisfied before the update, i.e., that there is no uncertainty whatsoever about the semantic correctness of stored data. In practice, this is clearly asking too much, for two reasons.

Firstly, it is unlikely in general that all semantic properties of interest are expressed as integrity constraints, so that integrity satisfaction would be equivalent with an absolute certainty about the data's correctness. This "semantic completeness" of the integrity constraints, however, is not the topic of this paper, because it is application-dependent, hard to quantify and qualify precisely, and thus cannot be expected in general. As we have already noted, satisfaction of constraints is a necessary condition for avoiding uncertainty (or whatever else is captured by the integrity conditions), but, as we have just seen, it is not a sufficient one.

The second reason why it is unlikely in practice that all integrity constraints are satisfied in each state is known to everybody with practical experience in working with large, real-life databases. For example, integrity checking is usually turned off for uploading bulk data, and is not always checked completely afterwards. This likely causes inconsistencies, constraint violations and uncertainty. Another example is the cleansing process of data warehousing, which rarely is in the position to avoid *all* integrity violations of data extracted from often inhomogeneous sources. More examples can be found in update-intensive databases. For instance, when triggers are used, their firing caused by some update may in turn cause the firing of badly controlled sequences of follow-up triggers that might not care or wait for successful integrity checks before taking action. Also, replication mechanisms in distributed databases that enter streaming data into an information store, or trade off consistency against availability, cannot be expected to always have sufficient time for exhaustive integrity checks (cf. [1]).

Now, the good news is that, as opposed to to common belief, many well-known approaches to integrity checking can indeed abandon the assumption that integrity be satisfied before each update, without sacrificing the certainty of their results and any of their efficiency. For uncertain data that violate integrity constraint, this means that integrity checking can tolerate them and leave the task of improving their integrity status to separate, possibly off-line processes. Such processes need not run, let alone be completed, before updates checked for integrity satisfaction are committed, as traditional approaches to integrity checking would require.

Given the categorical stance with which the assumption of constraint satisfaction before updates is postulated in virtually all approaches to database integrity, the preceding paragraph may read like a paradox. More such oddities are addressed in the following subsection, where we are going

to take a closer look at some differences between violated integrity and uncertainty.

2.2 Integrity and Uncertainty: Differences

In the previous section, we have proposed to see data that lack integrity as a border case of uncertain data. However, there is a significant difference. Data that lack integrity are not just uncertain, but definitely unwanted, while uncertain data may or may not have integrity. For example, the integrity constraint

$$violated \leftarrow emp(x), age(x, y), y < 14$$

expresses that integrity is violated by underage employment (because there is a law by which this constraint is enforced), while the formula

$$uncertain \leftarrow emp(x), age(x, y), y > retirement_age$$

expresses that overage employment data qualify as uncertain, in the sense of dubious (since, though not forbidden, it may contradict an employer’s general policy that a person beyond retirement age would remain employed). Another example is the integrity constraint

$$violated \leftarrow email(x), sent(x, y), received(x, z), y > z$$

which declares that integrity is violated if the sent-date of an email item is after its received-date (assuming that both x and y are normalised wrt the same time zone), and the formula

$$uncertain \leftarrow email(x, from(y)), suspect(y)$$

which classifies an email item x received from y as uncertain if the latter qualifies as suspect, although the message content may unexpectedly be valid and unproblematic. (It does not matter here how the *suspect* predicate is defined.)

Due to this semantic difference between data that violate integrity and data that qualify as uncertain, which has also been observed in [20], and in spite of the fact that the same syntax can be used to represent conditions for integrity and uncertainty, the use of known integrity checking methods for checking the uncertainty of data may be deemed problematic, if not unfeasible, for the following reason.

Virtually all methods for improving the efficiency of integrity checking assume that integrity be satisfied before a given update is checked for integrity preservation or violation. That way, the evaluation of the integrity status can focus on the relevant part of the data that are actually involved in, or affected by the update, while ignoring the rest, since it is known to satisfy integrity. This assumption, however, cannot be expected to hold for uncertain data, since they may very well be part of the stored data, i.e., not each stored data item can be assumed to lack uncertainty whenever some update needs to be checked for uncertainty. Hence, we are facing again essentially the same problem as addressed already in the previous section. In the following section, we are going to see that, surprisingly, the assumption can be abandoned without further ado.

3. VIOLATION TOLERANCE OF INTEGRITY CHECKING

Throughout we assume the usual terminological and notational conventions for relational and deductive databases, as known from the standard literature. With regard to approaches to database integrity, we refer to [21, 5, 17, 22, 4] and others as surveyed in [19]. However, the following definitions are independent of any concrete approach.

Different methods employ different notions of integrity satisfaction and violation, and use different criteria to determine these properties. In fact, each method, say, \mathcal{M} can be identified with its criteria, which in turn can be formalised as a function that takes as input a database (i.e., a set of database facts and rules), an integrity theory (i.e., a finite set of integrity constraints), and an update (i.e., a bipartite finite set of database clauses to be inserted and deleted, resp.), and outputs upon termination one of the values in $\{satisfied, violated\}$. For a database D and an update U , let D^U denote the updated database. Thus, the soundness of a method \mathcal{M} can be stated as follows.

Definition 1. (Soundness of integrity checking)

An integrity checking method \mathcal{M} is *sound* if the following holds, for each database D , each integrity theory IC that is satisfied in D , and each update U .

- (•) IC is satisfied in D^U if $\mathcal{M}(D, IC, U) = satisfied$.

For example, the approach in [21] generates a conjunction $\Gamma(U, IC)$ of simplifications of certain instances of those constraints in IC that are possibly affected by an update U , and asserts that, under the assumption that integrity is satisfied in the old state, integrity remains satisfied in the new state (i.e., $\mathcal{M}(D, IC, U) = satisfied$, in terms of definition 1) if and only if $\Gamma(U, IC)$ evaluates to *true* in the updated state D^U ; otherwise, integrity is violated. Under the same assumption, the approach in [22] runs an SLDNF-based resolution proof procedure, extended by some forward reasoning steps by which it is possible to delimit the search space to be traversed to those parts of the union of database and integrity constraints that are actually affected by a given update. The procedure asserts that integrity remains satisfied in D^U if the resulting search space is finitely failed; integrity is violated if the search space contains a refutation indicating inconsistency. In terms of the definition above, the $\mathcal{M}(D, IC, U)$ of [22] is the result of the traversed search space with given input from D, IC, U . In a similar manner, definition 1 can be instantiated for virtually any method of integrity checking in the literature.

Next, we formally define the notion of violation tolerance. As indicated above, the intuition of violation tolerance of an approach \mathcal{M} to integrity checking is that the existence of cases of violated constraints (which may indicate the violation of a condition on the uncertainty of related data, or, more generally, the violation of any semantic property as expressed by some constraint) needs to be tolerated while \mathcal{M} monitors and controls new cases of integrity violation as introduced by updates. Moreover, the cases of integrity that had been satisfied before the update are supposed to

remain satisfied afterwards. So, if the constraints express properties for preventing uncertainty, then those data that have not been uncertain should not be turned into uncertain ones by updates of other data, nor should uncertain data be introduced by any update. In order to capture this idea more formally, we first need to make precise what we mean by “cases”.

Definition 2. (Global variable, Case)

Let W be an integrity constraint.

- a) Each variable x in W that is \forall -quantified but not dominated by any \exists quantifier (i.e., \exists does not occur left of the quantifier of x in W) in the prenex normal form of W is called a *global variable of W* . Let $global(W)$ denote the set of global variables in W .
- b) $W\sigma$ is called a *case of W* if σ is a substitution such that $Range(\sigma) \subseteq global(W)$ and $Image(\sigma) \cap global(W) = \emptyset$.

Clearly, each variable in a constraint W represented in the standardised datalog denial form [22] is a global variable of W . Note that not any substitution whatsoever, but only substitutions of global variables of a constraint yield valid cases. Most integrity checking methods focus on certain cases of integrity constraints. These are simplified forms of original constraints that are sufficient for determining their satisfaction or violation upon given updates. In general, substitutions of non-global variables do not lead to valid simplifications.

For example, consider the integrity constraint $\exists x \forall y emp(y) \rightarrow sup(x, y)$ requiring the existence of an individual x who is superior of all employees y , in the database of some enterprise. Suppose that this constraint is satisfied when the tuple $emp(e)$ is inserted to the database, recording the hiring of a new employee e . Then, checking the instance $\exists x emp(e) \rightarrow sup(x, e)$ may very well evaluate to *true* while the original constraint may have become *false*, even if some new *sup* tuples are inserted along with $emp(e)$.

Further note that cases of an integrity constraint need not be ground, and that also each constraint W itself as well as each variant of W is a case of W .

With this, soundness of violation tolerance of an approach \mathcal{M} to integrity checking can be defined as follows.

Definition 3. (Violation tolerance)

An integrity checking method \mathcal{M} is *violation-tolerant* if the following holds, for each database D , each integrity theory IC , each finite set IC' of cases of constraints in IC that is satisfied in D , and each update U .

- ($\bullet\bullet$) IC' is satisfied in D^U if $\mathcal{M}(D, IC, U) = satisfied$.

Note that, even though there may well be an infinity of cases of constraints in IC , the finiteness requirement for IC' entails no loss of generality, as long as satisfaction of a set of constraints is supposed to be defined by requiring that each constraint be satisfied, as usual. Moreover, ($\bullet\bullet$) guarantees

satisfaction of any number of cases if \mathcal{M} returns *satisfied* for IC as its second argument, which is always finite.

Clearly, for violation-tolerant integrity checking, ($\bullet\bullet$) suggests to use the very same method \mathcal{M} as in the traditional case, where satisfaction of all of IC in D is required. Hence, with the relaxation of definition 3 that only some cases IC' but not necessarily all constraints are satisfied before the update, no loss at all of efficiency is associated, whereas the gains are immense: with a violation-tolerant method, it will be possible to continue database operations even in the presence of (obvious or hidden, known or unknown) cases of integrity violation (which for better or worse is rather the rule than the exception in practice), while maintaining the integrity of all cases which have complied with the constraints. Whenever \mathcal{M} is employed, no new cases of integrity violation will be introduced, while existing “bad” cases may disappear (by intention or even accidentally) by executing updates which have passed the integrity test of \mathcal{M} . So far, with the strict requirement of integrity satisfaction in the old state, not the least bit of integrity violation was tolerable. Hence, the known correctness results of virtually all approaches to database integrity would remain useless for the majority of all practical cases, and in particular for constraints about uncertainty, unless they can be shown to be violation-tolerant.

Of course, the preceding observations, as nice as they may be, would be void if no violation-tolerant method existed. Fortunately, however, most known approaches to database integrity that we have looked at so far are indeed violation-tolerant. A notable exception is the method in [10]. The following section introduces a sufficient condition by which it is fairly easy to check and assert violation tolerance.

4. A SUFFICIENT CONDITION FOR VIOLATION TOLERANCE

For a database D , an integrity theory IC , an update U and a finite set IC' of cases of constraints in IC that is satisfied in D , a straightforward special case of (\bullet) obviously is

- ($\bullet\bullet\bullet$) IC' is satisfied in D^U if $\mathcal{M}(D, IC', U) = satisfied$

For a given method \mathcal{M} , it is easy to see that its violation tolerance as expressed by ($\bullet\bullet$) directly follows from ($\bullet\bullet\bullet$) if the following condition is satisfied for each database D , each integrity theory IC , each finite set IC' of cases of constraints in IC that is satisfied in D , and each update U .

- ($\#$) If $\mathcal{M}(D, IC, U) = satisfied$ then $\mathcal{M}(D, IC', U) = satisfied$

Hence, with regard to definition 3, we immediately have the following result, for each sound approach \mathcal{M} to integrity checking.

THEOREM 1. \mathcal{M} is violation-tolerant if ($\#$) holds.

Proofs to verify ($\#$) for the approaches in [21, 5, 17, 22] are fairly easy because each of them generates simplified forms of constraints, such that, roughly speaking, the truth value

of the simplified form of any case of a constraint W in IC is implied by the truth value of the simplified form of W itself, from which (#) follows. The subsequent counter-example of a method \mathcal{M} which is sound according to definition 3 but not violation-tolerant shows that violation tolerance is not a matter of course. Methods such as [10, 4] also are not violation-tolerant, essentially because their optimisations of simplified constraints involves the elimination of redundancies that are independent of the database. However, when constraints are given in denial form (which is one of two common standard forms for representing integrity constraints), and when the optimisation phase of the method in [4] is applied separately to each individual constraint in IC instead of to IC as a whole, that method also is violation-tolerant.

Now, we construct a method \mathcal{M} which is not violation-tolerant. Let $\mathcal{M}(D, IC, U)$ be

1. *satisfied* (resp., *violated*) if $\exists x(p(x) \wedge x \neq a)$ is satisfied (resp., violated) in D^U , whenever IC contains $\leftarrow p(x)$ and U precisely consists of inserting $p(a)$.
2. *satisfied* (resp., *violated*) if IC is satisfied (resp., violated) in D^U otherwise.

The implementation of \mathcal{M} can recur on any standard method except for a special treatment of the constraint $\leftarrow p(x)$ and attempted insertions of $p(a)$. Clearly, \mathcal{M} is sound in the sense of definition 1, that is, if IC is satisfied in D then $\mathcal{M}(D, IC, U) = \textit{satisfied}$ entails the satisfaction of IC in D^U . (We remark that, here, also the converse holds, i.e., that satisfaction of IC entails $\mathcal{M}(D, IC, U) = \textit{satisfied}$.) Indeed, whenever IC holds in D and point 1 applies, $\mathcal{M}(D, IC, U) = \textit{violated}$, which correctly indicates that the update violates integrity; when point 2 applies, the evaluations of IC in D^U and $\mathcal{M}(D, IC, U)$ coincide by definition, so soundness is granted by definition.

Now, let us consider a database D which contains the sole fact $p(b)$, and $IC = \{\leftarrow p(x)\}$, i.e., integrity is not satisfied in D . Further, let U be as in point 1 above, and let $W' = \leftarrow p(a)$ be a case of the constraint in IC which obviously is satisfied in D . Although $\mathcal{M}(D, IC, U) = \textit{satisfied}$, W' is satisfied in D but not in D^U , i.e., the satisfied case W' is not preserved after the update, even though the corresponding checking condition given by \mathcal{M} is satisfied for IC . Thus, \mathcal{M} is sound according to def. 3 but not violation-tolerant.

To conclude this example, we remark that it can also be understood as an indication that approaches to integrity that implement a special case treatment for certain cases (e.g., cases with lower or higher uncertainty) tend to be less violation-tolerant (and, in general, more error-prone) than standard methods.

5. COMPLETENESS OF VIOLATION TOLERANCE

We note that def. 1 is only a statement about the soundness of \mathcal{M} . Completeness can be defined by the following version of the *only-if* half of (●).

Definition 4. (Completeness of integrity checking)
An integrity checking method \mathcal{M} is *complete* if the following holds, for each database D , each integrity theory IC that is satisfied in D , and each update U .

- (○) If IC is satisfied in D^U then $\mathcal{M}(D, IC, U) = \textit{satisfied}$.

For range-restricted integrity constraints and several significant classes of logic databases including relational ones, completeness has been shown to hold for the methods in [21, 17, 22, 4] and others in the respective original papers.

In analogy to the complementarity of (●) and (○), the question about the validity of the *only-if* half of (●●) in def. 3 arises. Not surprisingly, the *only-if* half of (#) is a sufficient condition for the *only-if* half of (●●), in full analogy to theorem 1. More precisely, it is easy to see that, for each each database D , each integrity theory IC , each finite set IC' of cases of constraints in IC that is satisfied in D , each update U and each integrity checking method \mathcal{M} that is complete for integrity checking (i.e., that satisfies (○)), the condition

(##) If $\mathcal{M}(D, IC', U) = \textit{satisfied}$ then $\mathcal{M}(D, IC, U) = \textit{satisfied}$

is sufficient for proving the completeness of violation tolerance of \mathcal{M} , i.e., to prove

(○○) If IC' is satisfied in D^U , then $\mathcal{M}(D, IC, U) = \textit{satisfied}$.

However, these formal results, though correct, are of limited value, since neither (##) nor (○○) hold for several of the most well-known methods, as the following simple example shows.

Let D consist of the fact $q(a)$ and the rule $p(x) \leftarrow q(x)$, $IC = \{\leftarrow p(x)\}$ and U be the insertion of $p(a)$. Then, the set of correct answers derivable from D would not change after committing the update, and hence, also all cases of IC that are satisfied in D remain satisfied in D^U . And indeed, approaches that notice the redundancy of the update, such as the method in [5], will simply stop right away, signalling integrity preservation for this example. However, the methods in [22] and [17], as well as several other ones will signal that U violates integrity (just the same as under the here not valid assumption that integrity was satisfied in D). Hence, neither (##) nor (○○) holds in general for several methods. We remark that the cited methods do not check the redundancy of the update since such additional checks require additional fact base accesses and may incur a considerable overhead in general.

Yet, we have the following completeness result.

THEOREM 2. (Completeness of violation tolerance)

Let \mathcal{M} be a sound and complete method for integrity checking, D a database, IC an integrity theory, U an update and W^* a case of a constraint W in IC such that W^* is satisfied in D . If W^* is not satisfied in D^U , i.e., if W^* is violated by U , then the following holds.

- (a) $\mathcal{M}(D, \{W\}, U) = \textit{violated}$,
- (b) $\mathcal{M}(D, IC, U) = \textit{violated}$.

Part (a) of theorem 2 holds because the violation of W^* in D^U entails that, *a fortiori*, W (and also IC) is violated in D^U , and hence completeness of \mathcal{M} entails that $\mathcal{M}(D, W, U)$ outputs *violated*. Part (b) is a direct consequence of part (a) and the soundness of \mathcal{M} .

To conclude this section, we conjecture that $(\circ\circ)$ holds for complete methods \mathcal{M} in databases where integrity violation cannot be corroborated by any update, i.e., where no case W^* of any constraint that is already violated in D can be violated anew by any update. As an example, consider relational databases that do not allow duplicate entries of tuples in any table. Further, only conjunctive queries (definite datalog denials) be allowed as integrity constraints. For such databases and integrity theories, the constraints are all of form $\leftarrow A_1, \dots, A_n$ ($n > 0$), where each A_i is an atom. Such denials may only be violated by the insertion of some ground fact A such that A unifies with one of the A_i . Conversely, if any update U of ground facts leaves all cases of constraints that are satisfied in D unviolated in D^U , then no denial can be violated at all (since each violation caused by U would introduce a violated case that has been satisfied in D). This intuitive argument remains to be verified by a more formal proof. In case it turns out to be correct, a generalization of the conjecture for denials with negation lends itself for verification, but we leave that to future research.

6. CHECKING VIOLATION

Soundness and completeness of integrity checking (definitions 1 and 4, respectively), as well as the definition of violation tolerance (definition 3) and the result about soundness of violation-tolerant integrity checking (theorem 1) are conceived with regard to checking the satisfaction of constraints, but nothing is explicitly stated wrt violation. This is due to the prevalent interest of integrity checking in the preservation of integrity satisfaction, i.e., in the “good” cases, while the “bad” cases of integrity violation tend to receive less attention. However, as unwanted as they may be, they deserve a comparable amount of attention. To reflect this is the objective of this section.

For instance, the soundness and completeness of integrity checking methods \mathcal{M} could in principle be defined in terms of violation in analogy to (\bullet) and (\circ) by the *if*- (soundness) and the *only-if* half (completeness) of condition (\diamond) below, for databases D , integrity theories IC that are satisfied in D and updates U , as follows.

(\diamond) IC is violated in D^U iff $\mathcal{M}(D, IC, U) = \textit{violated}$.

For two-valued definitions of integrity which do not admit values other than *satisfied* and *violated* (e.g., ‘unknown’, ‘undefined’, ‘overdefined’ or values for graded integrity), *not satisfied* is equivalent to *violated* and vice-versa. Hence, it follows by the definitions in sections 3–5 that conditions (\bullet) and the *if* half of (\diamond) , as well as (\circ) and the *only-if* half of (\diamond) , are equivalent.

Similarly, for each set IC' of cases of constraints in IC that is satisfied in D , the violation tolerance of an integrity checking method \mathcal{M} could also be defined via the following condition, in analogy to $\bullet\bullet$, as follows.

$(\diamond\diamond)$ If IC' is violated in D^U then $\mathcal{M}(D, IC, U) = \textit{violated}$.

For two-valued definitions of integrity and sound and complete methods for integrity checking that tolerate violation, it can be easily shown that $(\bullet\bullet)$ and $(\diamond\diamond)$ are equivalent, since the latter is simply the contraposition of the former.

So far, the results in this section (symmetries and relationships between satisfaction and violation on one hand, and between soundness and completeness, on the other) may appear as mere formal exercises without groundbreaking insights. However, for several methods, satisfaction and violation are not symmetrical issues. For instance, for many approaches, the termination of checking either the satisfaction or the violation of integrity cannot be guaranteed, while checking violation or, respectively, satisfaction will always terminate for large, significant classes of databases, constraints and updates. This is due to the semi-decidability of satisfiability in general. But there is more to the lack of symmetry between satisfaction and violation, to be looked at in the remainder of this section, by which the preceding formal investigations can be justified.

As a side remark, we firstly point out that soundness and completeness for checking violation of integrity is interesting because most of the approaches cited in this paper, and many others, do not only signal violation in case any constraint is no longer satisfied. Rather, they precisely identify the cases of constraints that are violated by the update. Such detail of information about uncertain data is very useful for repairing violations introduced by updates, e.g., in abductive procedures such as [6].

In general, completeness of integrity checking does not come for free. For example, several approaches to database integrity generate simplified conditions that are only sufficient but not necessary. That is, the satisfaction of such conditions ensures the preservation of integrity by given updates, but if they are not satisfied, then integrity is not necessarily violated. Thus, whenever these conditions are not satisfied, such methods need to perform further checks in order to determine the integrity status of the updated database. Examples of such approaches that lack completeness of checking integrity satisfaction (which, due to the analogies between (\bullet) , (\circ) and (\diamond) , can also be understood as a lack of soundness of checking integrity violation) are [10, 16, 11]. Despite their deficiencies, such methods may provide an advantage. In many applications, updates that violate integrity are rare. Therefore, a performance advantage for checking satisfaction which is obtained by a trade-off to the disadvantage of completeness (i.e., checking violation) may be convenient for such applications. (By the way, the lack of violation tolerance of the method in [10] is independent of its lack of completeness.)

Our last argument for defending the attention paid to checking constraints for violation is directly related to uncertainty. In databases with a propensity of containing uncertain information, the usual prevalence of satisfaction over violation may easily invert to the necessity of being more attentive to updates that possibly violate integrity rather than preserve it. Then, methods such as the aforementioned ones which only provide sufficient conditions for satisfaction but not

for violation have a definite disadvantage. As an example, imagine a database in which email coming in via a central email server node is stored. Further, imagine that the integrity constraints are such that they block spam. Because of high volumes and boost frequencies of incoming mail, not each spam message can be expected to be trapped since exhaustive checks might be unaffordable. Hence, suspicious email (e.g., with harmful attachments) may be stored and integrity cannot be expected to be 100% satisfied, since there may be more spam than correct email which does not violate integrity. In such settings, a violation-tolerant approach to integrity is in demand, and particularly one that is sound for violation. Whenever such a method indicates violation, the incoming message surely qualifies as spam and thus can safely be dumped. This behaviour is preferable to that of a method which would make every effort trying to guarantee satisfaction, i.e., making sure that a message which has passed the test really is not spam.

7. CONCLUSION

We have shown that it is possible to use integrity constraints and simplification methods for their evaluation as a means of expressing and checking conditions that qualify stored data as uncertain. This is due to two reasons. Firstly, any kind of semantic information can be expressed by arbitrary first-order syntax, be it integrity, uncertainty, or whatever properties that are wanted or unwanted. In general, any property of interest that needs to be checked and monitored can be represented in that form. This advantage of the expressive power of the syntax of integrity constraints is well-known and only needed to be recalled.

The disadvantage of such arbitrary syntax is that the evaluation of formulas expressed in it may be unfeasibly expensive. Since there are known methods with which the evaluation of constraints can be considerably simplified, integrity constraints in databases (i.e., properties required to hold in each state) are usually expressed in that syntax, in spite of its complexity. In the literature, these simplifications have been believed to be sound only in case the unsimplified forms are all satisfied before an updated state is checked for preserving satisfaction. Thus, the idea to use the simplification technology for something like checking uncertainty could not arise, because it would be unreasonable to assume a complete absence of uncertainty before an update is checked for satisfying or violating constraints about uncertainty.

However, as conveyed in this paper, many well-established methods for integrity checking can be used straightforwardly for also checking other semantically complex properties, such as constraints about uncertainty, even if the database contains uncertain information that violates these constraints. So, our findings about violation tolerance embody the second reason mentioned above, why it is possible to use known technology for integrity checking also for checking conditions about uncertainty. The result is significant, because, hitherto, all approaches to database integrity have required that no constraint be violated before any update could be checked efficiently for preserving or violating integrity.

In detail, we have qualified as violation-tolerant those approaches to database integrity that have the ability to relax

the assumption of full satisfaction of integrity before updates. With such approaches, the presence of uncertain information is admissible while the invariance of satisfied cases of imposed constraints about uncertainty can be checked efficiently. We also have defined and discussed sufficient conditions for the soundness and completeness of violation-tolerant methods. Most known methods (with some noteworthy exceptions) have turned out to be violation-tolerant, while completeness seems to be achievable only under particular circumstances.

In general, we believe that the possibility of abandoning the assumption of complete satisfaction of all constraints increases the applicability of integrity checking technology significantly.

To the best of our knowledge, the allegedly fundamental assumption that integrity needs to be satisfied before an update in order to perform correct and efficient integrity checking has never been challenged by other authors. Motro's work on "integrity and uncertainty" [20] uses a similar terminology and the related terms "validity" (soundness) and "completeness". However, his work is distinctly different from ours because Motro does not deal with methods for improving the efficiency of integrity checking, which, for the work presented in this paper, is key.

Nevertheless, it cannot be overlooked that "inconsistency tolerance" has recently become an important issue. Still, most work in this area is concerned with query answering in inconsistent databases; in this sense, the notions of repair (cf., e.g., [2]) and consistent query answering are crucial and yet very different from what this paper suggests.

Several paraconsistent logic approaches have received some attention in the recent past (cf., e.g., [9]), most of which, however, abandon classical first-order logic, possibly by resorting to multivalued logic, which we have not considered in this paper. Last, we note that standard resolution-based query answering (as in [22] and [15]) is in some sense also paraconsistent, in that it does not consider irrelevant database clauses for evaluating integrity constraints (considered as queries) upon an update.

In future work, besides checking whether other, more recent methods also tolerate integrity violation, we intend to broaden the notion of tolerance such that not only approaches for integrity checking, but also methods for repairing violations and for view updating can be applied in the presence of inconsistency and/or uncertainty.

8. REFERENCES

- [1] J.E. Armendáriz, H.Decker, F.D. Muñoz: Trying to Cater for Replication Consistency and Integrity of Highly Available Data. To appear in *Proc. DEXA Workshop LAAIC'06*.
- [2] L. Bertossi, J. Chomicki: Query Answering in Inconsistent Databases. In J. Chomicki, R. van der Meyden, G. Saake (eds), *Logics for Emerging Applications of Databases*, 43–83. Springer, 2004.
- [3] C.-L. Chang and R. Lee: Symbolic Logic and Mechanical Theorem Proving. *Computer Science Classics*. Academic Press, 1973.

- [4] H. Christiansen, D. Martinenghi: On Simplification of Database Integrity Constraints. *Fundamenta Informaticae* 71(4):371–417, A. Pettorossi, M. Proietti (eds.), IOS Press, 2006. See also [18].
- [5] H. Decker: Integrity Enforcement on Deductive Databases. In L. Kerschberg (ed), *Expert Database Systems*, EDS'86, 381–395. Benjamin/Cummings, 1987.
- [6] H. Decker: An Extension of SLD by Abduction and Integrity Maintenance for View Updating in Deductive Databases. *Proc. JICSLP'96*, 157–169, MIT Press, 1996.
- [7] H. Decker: Historical and Computational Aspects of Paraconsistency in View of the Logic Foundation of Databases. In L. Bertossi, G. Katona, K.-D. Schewe, B. Thalheim (eds), *Semantics in Databases*, 63–81. LNCS 2582, Springer, 2003.
- [8] H. Decker: Total Unbiased Multivalued Paraconsistent Semantics of Database Integrity. *DEXA Workshop LAAIC'05*, 813–817. IEEE Computer Society, 2005.
- [9] H. Decker, J. Villadsen, T. Waragai (eds): *Paraconsistent Computational Logic*. Proc. ICLP Workshop at FLoC'02. Dat. Skrifter vol. 95, Roskilde Univ., 2002.
- [10] A. Gupta, Y. Sagiv, J. D. Ullman, and J. Widom. Constraint checking with partial information. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 24-26, 1994, Minneapolis, Minnesota*, pages 45–55. ACM Press, 1994.
- [11] L. Henschen, W. McCune, and S. Naqvi. Compiling constraint-checking programs from first-order formulas. In H. Gallaire, J. Minker, and J.-M. Nicolas, editors, *Advances in Database Theory, February 1-5, 1988, Los Angeles, California, USA*, volume 2, pages 145–169. Plenum Press, New York, 1984.
- [12] A. Kakas, R. A. Kowalski, F. Toni: Abductive Logic Programming. *J. Logic and Computation* 2(6):719-770, 1992.
- [13] R. A. Kowalski: *Logic for Problem Solving*. Elsevier, 1979.
- [14] K. Bowen, R. A. Kowalski: Amalgamating Language and Metalanguage. In K. Clark, S.-A. Tärnlund (eds), *Logic Programming*, 153-172. Academic Press, 1982.
- [15] R. A. Kowalski, F. Sadri, P. Soper: Integrity Checking in Deductive Databases. In *Proc. 13th VLDB*, 61-69. Morgan Kaufmann, 1987.
- [16] S. Y. Lee and T. W. Ling. Further improvements on integrity constraint checking for stratifiable deductive databases. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 495–505. Morgan Kaufmann, 1996.
- [17] J. W. Lloyd, L. Sonenberg, R. W. Topor: Integrity constraint checking in stratified databases. *Journal of Logic Programming* 4(4):331–343, 1987.
- [18] D. Martinenghi: *Advanced Techniques for Efficient Data Integrity Checking*. PhD thesis, Roskilde University, Denmark, in *Datalogiske Skrifter* vol. 105, <http://www.ruc.dk/dat/forskning/skrifter/DS105.pdf>, 2005.
- [19] D. Martinenghi, H. Christiansen, H. Decker: Integrity Checking and Maintenance in Relational and Deductive Databases, and beyond. In Z. Ma (ed), *Intelligent Databases: Technologies and Applications*, to appear. Idea Group Publishing, 2006.
- [20] A. Motro. Integrity = validity + completeness. *ACM Transactions on Database Systems (TODS)* 14(4):480–502, 1989.
- [21] J.-M. Nicolas: Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 18:227–253, 1982.
- [22] F. Sadri, R. A. Kowalski: A Theorem-Proving Approach to Database Integrity. In J. Minker (ed), *Foundations of Deductive Databases and Logic Programming*, 313–362. Morgan Kaufmann, 1988.
- [23] J. Widom, S. Ceri: *Active Database Systems*. Morgan Kaufmann, 1996.