

Checking Violation Tolerance of Approaches to Database Integrity

Hendrik Decker¹ and Davide Martinenghi²

¹ Instituto Tecnológico de Informática Ciudad Politécnica de la Innovación E-46071 Valencia, Spain <code>hendrik@iti.es</code> supported by Spanish grant TIC2003-09420-C02	² Free University of Bozen/Bolzano Faculty of Computer Science I-39100 Bolzano, Italy <code>martinenghi@inf.unibz.it</code> supported by EU project TONES IST FP6-7603
---	--

Abstract. A hitherto unquestioned assumption made by all methods for integrity checking has been that the database satisfies its constraints before each update. This consistency assumption has been exploited for improving the efficiency of determining whether integrity is satisfied or violated after the update. Based on a notion of violation tolerance, we present and discuss an abstract property which, for any given approach to integrity checking, is an easy, sufficient condition to check whether the consistency assumption can be abandoned without sacrificing usability and efficiency of the approach. We demonstrate the usefulness of our definitions by showing that the theorem-proving approach to database integrity by Sadri and Kowalski, as well as several other well-known methods, can indeed afford to abandon the consistency assumption without losing their efficiency, while their applicability is vastly increased.

1 Introduction

Virtually all known approaches to integrity checking assume that each constraint be satisfied in the “old state”, before the update. At least, this means to require that the union of the database and its integrity constraints be consistent, as, e.g., in [16]. Many other approaches, such as [15, 12] and others, even require that each constraint be a theorem, or a logical consequence of the database, which of course entails consistency. Only under this quite strong consistency assumption (which, by the way, is also quite unrealistic, in terms of large, “real-life” databases that typically contain some amount of inconsistent data) are the known approaches for integrity checking guaranteed to correctly indicate integrity satisfaction in the “new state”, after the update has been committed.

For the approaches described in [15, 4, 12, 16], this consistency assumption can be significantly relaxed without risking that their integrity invariance guarantees would go astray. Informally speaking, it can be shown that, even if there is any number of cases of integrity violation in the old state, the rest of the database will remain satisfied in the new state if the outcome of the respective approach indicates that the given update does not violate integrity, i.e., does not introduce new cases of violation.

After some preliminaries in section 2, where we also give an abstract definition of the correctness of integrity checking, we define and discuss the property of violation tolerance in section 3. This property is not obvious in general and has to be analysed individually for each given integrity checking approach. Independently of any approach, however, we develop in section 3 a general condition by which it is possible to check whether a given correct integrity checking approach is violation-tolerant. In subsection 3.2 we apply this condition to the theorem-proving approach to database integrity in [16] and prove that it is indeed violation-tolerant. Then we recapitulate results about the violation tolerance of approaches in [15] [12] and ascertain that also the integrity checking method in [4] is violation-tolerant. After addressing related work in section 4, we conclude in section 5 with an outlook on a broader notion of violation tolerance.

2 Preliminaries

Throughout we assume the usual terminological and notational conventions for relational and deductive databases, as used in the cited writings about database integrity [15, 4, 12, 16].

The following definitions are independent of any concrete method. We only point out that integrity constraints are usually conceived as closed well-formed formulae of first-order predicate calculus in the underlying language of the database. Two standard representations of integrity constraints are in use: either prenex normal form (where all quantifiers are outermost and all negation symbols are innermost) or denial form (datalog clauses without head). Unless explicitly mentioned, we are impartial about these representations, i.e., when speaking of an integrity constraint W , it may be given in any of these two forms.

Different methods to check database integrity employ different notions to define integrity satisfaction and violation, and use different criteria to determine these properties. Such criteria are always meant to be more efficient than to plainly evaluate the integrity status of all integrity constraints upon each update. 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), a finite set of integrity constraints, and an update (i.e., a bipartite finite set of database clauses to be deleted and inserted, resp.), and outputs upon termination one of the values $\{satisfied, violated\}$. (In general, a multivalued range is conceivable for also dealing with unknown, under- or over-determined integrity, or graded levels of satisfaction or violation, as, e.g., in [7]; for simplicity, we only deal with two-valued integrity here.)

For a database D and an update U , let, for convenience, D^U denote the updated database. Thus, the correctness of an approach \mathcal{M} can be stated in the following form.

Definition 1 (Correctness of integrity checking). *An integrity checking method \mathcal{M} is correct if, for each database D , each finite set IC of integrity constraints such that D satisfies IC , and each update U , the following holds.*

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

For instance, the approach in [15] 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., $\mathcal{M}(D, IC, U) = \text{satisfied}$, in terms of the definition above) 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 [16] runs an SLDNF-based resolution proof procedure, extended by some forward reasoning steps by which it is possible to delimit the search space 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 [16] is the result of the traversed search space with given input from D, IC, U .

Similarly, the $\mathcal{M}(D, IC, U)$ of the integrity checking method in [12] is determined by the outcome of running SLDNF resolution with essentially the same input as in the approach of [16]. For details of technical and conceptual differences between the latter two approaches, which are not directly relevant to the objectives of this paper, we refer the reader to [16].

3 Violation tolerance

Next, we formally define the notion of violation tolerance. Rather than using the perhaps more popular term “inconsistency tolerance”, we prefer to speak of violation tolerance because (in)consistency is a slightly more general concept than integrity satisfaction (violation, resp.); we also want to avoid terminological interferences with the discussion in [16] of differences between the consistency approach and the theoremhood approach of integrity checking.

As indicated in section 1, the intuition of violation tolerance of an approach \mathcal{M} to integrity checking is that we want to tolerate (or, rather, be able to live with) cases of violated constraints as long as we can ensure that no new cases of integrity violation are introduced, such that the cases of integrity that had been satisfied before the update will remain satisfied afterwards. Thus, 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 $\text{global}(W)$ denote the set of global variables in W .*

b) *The formula $W\sigma$ is called a case of W if σ is a substitution such that $\text{Range}(\sigma) \subseteq \text{global}(W)$ and $\text{Image}(\sigma) \cap \text{global}(W) = \emptyset$.*

Clearly, each variable in a constraint W represented by a normal datalog denial is a global variable of W . Note that cases of an integrity constraint need not be ground, and that each constraint W as well as each variant of W is a case of W .

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

Definition 3 (Violation tolerance). \mathcal{M} is violation-tolerant if, for each database D , each finite set IC integrity constraints, each finite set IC' of cases of constraints in IC such that D satisfies IC' , and each update U , the following holds.

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

Even though there may be an infinity of cases of constraints in IC , the finiteness requirement for IC' entails no loss of generality, since integrity satisfaction is defined compositionally (i.e., a finite set of constraints is satisfied if each of its elements is satisfied). Moreover, $\mathcal{M}(D, IC, U) = \text{satisfied}$ guarantees satisfaction of any number of cases that have been satisfied in D .

Clearly, for checking integrity with a violation-tolerant method \mathcal{M} , (**) suggests to compute the very same function as in the traditional case, where satisfaction of all of IC in D is required. Hence, with this relaxation, 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 that 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 that 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, 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, all known approaches to database integrity that we have checked so far for violation tolerance do enjoy this property. The following subsection introduces a sufficient condition by which it is fairly easy to check and assert violation tolerance.

3.1 A sufficient condition for violation tolerance

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

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

This is already pretty close to (**), which we have identified above as the desirable property of violation tolerance. It is easy to see that, for a given method \mathcal{M} , (**) directly follows from (***) if the following condition is satisfied for each database D , each finite set of integrity constraints IC , each finite set IC' of cases of constraints in IC such that IC' is satisfied in D , and each update U .

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

Hence, with regard to definition 3, we immediately have the following result.

Theorem 1. *Let \mathcal{M} be an approach to integrity checking by which the satisfaction and violation of a finite set of constraints can be determined. Then, \mathcal{M} is violation-tolerant if (#) holds.*

Proofs by which (#) is verified for the approaches in [15] and [12] are fairly easy because both generate 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. Violation tolerance of the method in [4] is also easily verified: essentially, it generates all ground facts (which are always finitely many in a range-restricted database without function symbols) that are either false in the old and true in the new state, or vice-versa, corresponding to all facts that are effectively inserted in or deleted from D , respectively. For each such fact, simplified forms of potentially affected constraints are obtained and evaluated that are essentially the same as those in [15], so that the verification of (#) for the approach in [4] can recur on the latter.

In section 3.2, we show that also the approach in [16] is violation-tolerant, and, again, the proof is fairly easy. However, it would be wrong to think that violation tolerance comes for free with any correct approach to integrity whatsoever. The following counter-example shows that this is indeed not the case.

Example 1. We construct here a method for integrity checking \mathcal{M} for which (#) does not hold, i.e., $\mathcal{M}(D, IC, U) = \text{satisfied}$ does not entail that $\mathcal{M}(D, IC', U) = \text{satisfied}$, where IC' is a set of cases of constraints in IC . 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 $IC = \{\leftarrow p(x)\}$ and U precisely consists of inserting $p(a)$.
2. *satisfied* (resp., *violated*) if IC is satisfied (resp., violated) in D^U otherwise.

Clearly, \mathcal{M} is correct in the sense of definition 1, i.e., if $\mathcal{M}(D, IC, U) = \text{satisfied}$ then IC is satisfied in D^U (here, actually, the converse holds too). Indeed, whenever IC holds in D and point 1 applies, $\mathcal{M}(D, IC, U) = \text{violated}$, 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 correctness is trivial.

Now, let U and IC be as in point 1 above. Further, let D be a database containing the sole fact $p(b)$, and $W' = \leftarrow p(a)$ be a case of the constraint in IC . Note that W' is satisfied in D but IC is not. Although $\mathcal{M}(D, IC, U) = \text{satisfied}$, 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 M is satisfied for IC . Therefore \mathcal{M} is correct but not violation-tolerant.

3.2 Violation tolerance of Sadri & Kowalski's approach

In this subsection, we are going to verify the condition (#) above for the approach to integrity checking in [16].

Before recalling the function $\mathcal{M}(D, IC, U)$ of this approach for verifying (#), it should be interesting to note that none of the proofs of the theorems and corollaries in [16] effectively makes use of the assumption that integrity is satisfied in the old state D_0 except the completeness results following from theorems numbered 4 and 5 in [16].

For inferring from those theorems the completeness of \mathcal{M} with regard to checking integrity violation, it is argued that, “since $\text{Comp}(D_0)$ is consistent with the constraints, any inconsistency after the transaction must involve at least one of the updates.” Clearly, this builds on the assumption that integrity is satisfied in the old state. However, rather than completeness of computing violation, what is of interest to us here is the generalisation of correctness results of approaches to check integrity satisfaction.

Related to the fact that proofs in [16] do not make use of the assumption that integrity is satisfied in the old state, a certain form of inconsistency tolerance of the approach \mathcal{M} in [16] with regard to integrity violation can be observed, in the following sense: whenever $\mathcal{M}(D, IC, U) = \text{violated}$, then the correctly indicated violation of integrity is independent of the integrity status before the update. However, rather than inconsistency tolerance with regard to integrity violation, what we are after in this paper is violation tolerance with regard to integrity satisfaction, as expressed in (*). The independence of detecting integrity violation by the approach in [16] from the integrity status before the update is fairly trivial, and has been addressed above only in order to be precise about what and what not we are dealing with.

As the main result of this subsection, we are going to prove the following.

Theorem 2. *The approach \mathcal{M} to integrity checking by Sadri & Kowalski is violation-tolerant.*

Proof. First, we recall the function $\mathcal{M}(D, IC, U)$ associated to \mathcal{M} , as described at the end of section 2. It determines integrity violation and satisfaction by the existence or, respectively, absence of a refutation in the search space of the theorem-prover defined in [16] with an element from U as top clause. Thus, to show violation tolerance of \mathcal{M} , we need to verify

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

i.e., that the search space, say, $\mathcal{T}(D, IC', U)$ of \mathcal{M} with top clause from U and input from $D \cup IC'$ is finitely failed if the search space $\mathcal{T}(D, IC, U)$ of \mathcal{M} is finitely failed (in the latter search space, input from IC is considered, instead of IC' , as in the former). To see that this holds, assume that $\mathcal{M}(D, IC, U) = \text{satisfied}$, i.e., $\mathcal{T}(D, IC, U)$ is finitely failed.

Now, we recall that, in each derivation δ of \mathcal{M} , at most one denial is taken as input clause, for resolving a literal selected in the head of a clause in δ . As assumed above, each derivation in $\mathcal{T}(D, IC, U)$ is finitely failed. It remains to be shown that each derivation δ in $\mathcal{T}(D, IC', U)$ from the same root which has IC' instead of IC in the set of candidate input clauses is also finitely failed.

For that, we distinguish the two cases that δ either does or does not use an input clause from IC' . If it does not, then, by definition of the proof procedure

in [16], δ necessarily is also a derivation in $\mathcal{T}(D, IC, U)$ (up to a possible permutation of the sequence of used input clauses) and hence is finitely failed. If it does, then, by definition of IC' , that input clause is a case I' of some constraint I in IC . Now, to initiate the conclusive *reductio-ad-absurdum* argument of this proof, suppose that δ is not finitely failed, i.e., it terminates in the empty clause. Since I is more general than I' , it follows from the definition of \mathcal{M} that a refutation δ^* in $\mathcal{T}(D, IC, U)$ can be constructed which is almost the same as δ except, instead of I' , uses I as an input clause and continues, possibly with less instantiated variables and up to a possible permutation of input clauses, in the same manner as δ until the empty clause is reached. This, however, contradicts the assumption that $\mathcal{T}(D, IC, U)$ is finitely failed. \square

In case negation may occur only in literals of denials in IC but not in D , a shorter proof is possible. To see then that that the search space with input from is finitely failed if taking input from IC finitely fails, simply assume that $\mathcal{T}(D, IC', U)$ would contain a refutation. Then, by the lifting lemma [2], there must be a refutation in $\mathcal{T}(D, IC, U)$, which contradicts the assumption.

Below we illustrate violation tolerance of the approach in [16] by an example adapted from [11].

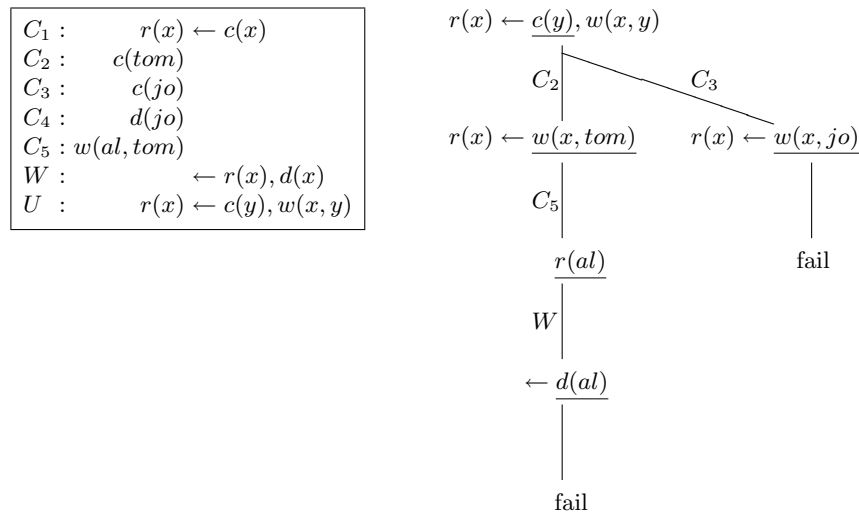


Fig. 1. Clauses and derivation tree of example 2.

Example 2. Consider a database D consisting of clauses C_1 – C_5 shown in figure 1 for unary relations r (regular residence), c (citizen), d (deported) and binary relation w (works for) and the integrity constraint W , expressing that it is impossible to both have a regular residence status and be registered as a deported person at the same time. The given update U inserts a new rule asserting that people working for a registered citizen also have regular residence status.

Clearly, W is not satisfied in D , since $r(jo)$ is derivable via C_1 and C_3 , and $d(jo)$ is a fact (C_4). However, $W' = \leftarrow r(tom), d(tom)$ is a case of W that is satisfied in D since $d(tom)$ does not hold in D .

With U as top clause and $D \cup \{W\}$ as input, the approach of [16] traverses the search space given by the tree shown in figure 1 (selected literals are underlined). Since this tree is finitely failed, we can conclude that U will not introduce new cases of inconsistency: all cases of integrity constraints that were satisfied in D remain satisfied in D^U . In particular, W' is also satisfied in D^U .

4 Related work

To the best of our knowledge, no other author has ever cared to render moot the putatively fundamental assumption that integrity would need to be satisfied before an update in order to correctly perform a simplified integrity check.

Notwithstanding this, interesting work has been going on in recent years under the banner of “inconsistency tolerance”. The majority of work in this area is concerned with query answering in inconsistent databases in connection with repairs of violated integrity (cf., e.g., [1]). Consistent query answering defines answers to be correct if they are logical consequences of each possible repaired state of the database, i.e., in each state which satisfies integrity and differs from the given violated state in some minimal way.

To query constraints or simplified forms thereof for evaluating their integrity status is usual in many approaches. However, using consistent query answering techniques for that is very different from what the results of this paper suggest. Rather than catering for repairs at query time (which often is impossible, e.g., due to untimeliness or because inconsistency is hidden and not even perceived) and determining truth values for all possible repaired states (which may be unfeasibly inefficient), the results of this paper justify that it is possible to simply “live with” inconsistencies as given by integrity violation. Instead of being pre-occupied with obtaining perfectly consistent database states in which integrity is a hundred percent satisfied, advantage can be taken of the fact that, in practice, the majority of stored information is consistent and can be queried without further ado. In other words, the use of a violation-tolerant approach to integrity checking allows a straightforward use of standard query evaluation procedures.

Repairing integrity violation by modifications of the database is not dealt with in this paper (cf., e.g., [9, 17] for abductive or active database techniques for repairing violation). However, with violation-tolerant integrity checking, the repair of violated cases of integrity can be delayed and dealt with at more convenient points of time (e.g., off business hours or when the system workload is lower). The price to be paid for this convenience, of course, is that querying data causing violated cases of integrity may yield answers that are not in accordance with the intended semantics, as expressed by the constraints.

Related to inconsistency tolerance, also a variety of paraconsistent logic approaches have received some attention (cf., e.g., [8]). Most of them, however, deviate significantly from classical first-order logic as the basis of database logic,

which we do not. However, standard resolution-based query answering (which is used for implementing the approach of [16] in [11]) can be characterised as a procedural form of paraconsistent reasoning (as done in [10, 5, 6]) that merits attention with regard to violation tolerance. It does not take irrelevant database clauses into account for evaluating constraints upon some update, even if such clauses would be involved in some case of integrity violation that has not been caused by the update but by some earlier event. In general, logic-programming-based reasoning in databases does not exhibit any explosive behaviour as predicted by classical logic in the presence of inconsistency, which would render each given answer worthless, but reasons soundly in consistent subsets of relevant clauses. This paraconsistency aspect qualifies logic programming as an ideal conceptual paradigm for violation-tolerant approaches to database integrity.

5 Conclusion

First, we have defined violation tolerance of approaches to database integrity as their capacity of abandoning the assumption that integrity is satisfied before each update, i.e., of admitting cases of integrity violation while preserving the invariance of satisfied cases, without forfeiting efficiency. Based on that, we then have defined a condition by which any method for determining integrity satisfaction or violation upon given updates can be checked for violation tolerance. We have successfully performed this check for some of the most well-known methods. We have traced this check in detail for the approach of Sadri & Kowalski, and have seen that it can indeed abandon the assumption of integrity satisfaction before each update without impairing its efficiency.

In general, by abandoning the assumption of integrity satisfaction, the applicability of approaches successfully checked for violation tolerance not only is not curtailed, but in fact formidably increased. To require that assumption would have the effect of making such approaches useless for many practical applications (e.g., replicated databases and data warehousing, to name just two of the most prominent application areas) where cases of intermittent or undetected integrity violation are rather the rule than the exception. The results of this paper, however, encourage the use of well-established approaches that hitherto have been known to function only in an “academically clean” context with a hundred percent absence of violated integrity constraints.

It could be asked why we have preferred to look at rather “old” methods for integrity checking, given that many more approaches have been presented and discussed in the literature more recently. To defend this preference is easy: most of the methods that have emerged later build, at least partially, on the results and achievements of those we have looked at, and improvements often have been just marginal, or interesting mainly for “advanced” scenarios that yet are hardly usual in practice, although some notable exceptions have been identified in [14]. Moreover, we expect that the task of checking more recent approaches for violation tolerance will become easier when violation tolerance checks of more fundamental methods can be referred to.

Apart from checking other, more recent approaches for violation tolerance, e.g., the one presented in [3], we also intend to investigate the viability of such approaches for integrity checking in distributed databases using lazy replication, where full satisfaction of integrity is particularly hard, if not impossible to achieve. Moreover, we have in mind to broaden the notion of violation tolerance such that also methods for actively repairing violated constraints, for integrity-preserving view updating and, more generally, for abductive belief revision can be checked to be applicable in scenarios with manifest violations of integrity constraints, without immolating effectiveness and efficiency of these methods.

References

1. L. Bertossi: Consistent Query Answering in Databases. *ACM SIGMOD Record* 35(2):68–77, 2006.
2. C.-L. Chang and R. Lee: Symbolic Logic and Mechanical Theorem Proving. *Computer Science Classics*. Academic Press, 1973.
3. 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 [13].
4. H. Decker: Integrity Enforcement on Deductive Databases. In L. Kerschberg (ed), *Expert Database Systems*, EDS’86, 381–395. Benjamin/Cummings, 1987.
5. 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.
6. H. Decker: A Case for Paraconsistent Logic as Foundation of Future Information Systems. *Proc. CAiSE’05 Workshops*, vol. 2, 451–461. FEUP edicoes, 2005.
7. H. Decker: Total Unbiased Multivalued Paraconsistent Semantics of Database Integrity. *DEXA Workshop LAAIC’05*, 813–817. IEEE Computer Society, 2005.
8. H. Decker, J. Villadsen, T. Waragai (eds): *Paraconsistent Computational Logic*. Proc. ICLP Workshop at FLoC’02. Dat. Skrifter vol. 95, Roskilde Univ., 2002.
9. A. Kakas, R. A. Kowalski, F. Toni: Abductive Logic Programming. *J. Logic and Computation* 2(6):719–770, 1992.
10. R. A. Kowalski: *Logic for Problem Solving*. Elsevier, 1979.
11. R. A. Kowalski, F. Sadri, P. Soper: Integrity Checking in Deductive Databases. In *Proc. 13th VLDB*, 61–69. Morgan Kaufmann, 1987.
12. J. W. Lloyd, L. Sonenberg, R. W. Topor: Integrity constraint checking in stratified databases. *Journal of Logic Programming*, 4(4):331–343, 1987.
13. 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.
14. 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.
15. J.-M. Nicolas: Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 18:227–253, 1982.
16. 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.
17. J. Widom, S. Ceri: *Active Database Systems*. Morgan Kaufmann, 1996.