Expression and Enforcement of Dynamic Integrity Constraints

Iliano Cervesato
Dipartimento di Informatica
Universita' degli Studi di Torino
Corso Svizzera 185,
10149 Torino, Italy
iliano@di.unito.it

Christoph F. Eick
Department of Computer Science
University of Houston
4800 Calhoun
Houston, TX, 77204-3475, USA
ceick@cs.uh.edu

Abstract

The interest in consistency enforcement in the field of database and in expert systems is nowadays widespread. Special attention has been given in the literature to the subtopic of *static integrity constraints*. This paper centers instead on the automatic enforcement of *dynamic consistency constraints*, i.e. those integrity constraints that cannot be checked by solely inspecting the most recent state of a data- or knowledge-base. A logical formalism that extends first-order logic with a temporal dimension is introduced for their specification. An algorithm aimed at identifying the portion of the dynamic integrity constraints of a temporal database relevant for a given update is presented. This algorithm is then specialized to conventional databases. Prolog prototypes exist for both versions.

1 Introduction

Consistency constraints play an important role in almost any software design process. Violating consistency constraints results in erroneous system behavior and therefore has to be avoided, especially if knowledge is shared by multiple users. Consequently, application programs that update knowledge have to contain code that enforces the consistency constraints that underlie the particular application area. Usually, the code that enforces consistency constraints is significantly longer than the code needed to perform a particular update operation.

Consequently, it seems attractive to relieve application programmers from the task of having to enforce consistency constraints by moving the responsibility to perform this task from application programs to the system software such as database management systems, knowledge base management systems, expert system shells, or CASE-tools. In order to enforce constraints automatically, it is necessary to centralize the specification of constraints. Consequently, the consistency constraints that hold in a particular application area have to be specified in the conceptual schema that describes the semantics of the application area. System software will then enforce these constraints automatically relieving the application programmer from the task of consistency enforcement - application programmers need only to code update operations, but no longer code consistency checks.

However the topic of consistency enforcement has been kept into consideration only recently and the availability and use of this essential tool are not widespread. In fact, the majority of commercial DBMSs do not provide tools aimed at enforcing but basic forms of consistency (uniqueness of primary keys, etc.). The situation is a little better in the field of knowledge bases systems for several shells, like KEE, provide enforcement tools. The main reason why commercial products do not usually provide general consistency enforcement facilities is their computational cost. Checking the consistency of a database after an update is a high overhead, akin to that of several queries. The situation worsens considerably in a distributed environment.

In this paper, we will focus on a special subproblem of consistency enforcement: the automatic enforcement of *dynamic consistency constraints*. A dynamic constraint is a constraint that cannot be checked by inspecting the most recent state of a database (knowledge base). For example, the constraint "salaries never decrease" refers to objects (here salaries) of former states of the database. On the other hand, the uniqueness of social security numbers can be checked by inspecting the most recent state. The uniqueness constraint is therefore a static constraint. The importance of dynamic constraints has been recognized by research on temporal and historical databases (see for example [2, 3, 13, 17, 19]).

Although the automatic enforcement of consistency constraints is rarely supported in commercial software systems, a number of useful techniques have been proposed in the literature for this purpose, mainly for the enforcement of static consistency constraints. Most of these papers assume that first order predicate calculus (or some of its variations) is used to specify integrity constraints, and provide techniques that decide which constraints (out of a set of constraints) have to be checked for a given update. Moreover, the proposed algorithms simplify the constraints to be checked, if possible. This approach was originally proposed by Nicolas in [14], and later extended and modified by many researchers (most notably by [9, 10, 16]).

Recently, rule-based programming paradigms gained some popularity for databases. These efforts found their expression in two directions: in deductive databases [1, 15] which augment the query-processing capabilities of database management systems by supporting deductive, Prolog-style production rules, and in active databases that support data-driven production rules [5, 6]. Research in these two areas demonstrated that production rules provide a very suitable framework for specifying and enforcing consistency constraints. In deductive databases constraints can be expressed and enforced by Prolog-style predicates relying on classical resolution techniques [10, 15].

On the other hand, active databases, such as POSTGRES [18] and HIPAC [12], support data-driven production rules that facilitate the coding of exceptions handlers for constraint violations. The STARBUST DBMS [20] supports set-oriented production rules integrated with DBMS transaction concepts. In their framework, a transaction computes a set of update-operations that it is going to perform, triggering active rules that react to these changes, e.g. reject violations of consistency constraints by cancelling the transaction. Moreover, [5] introduces activation pattern controlled rules for the purpose of consistency enforcement. They are rules augmented with an additional left-hand side - the activation pattern - which matches calls of particular commands, sensitizing the rules for the execution of these commands. Finally, in the ODE object-oriented database system [7] constraints are associated with objects. The associated constraints are checked each time a member function (or the constructor) is called for the particular object.

Section 2 recalls some background material. Section 3 gives a detailed account of dynamic integrity constraints by presenting a logical formalism that allows to express them in their most general form. Section 4 describes an algorithm for enforcing the constraints expressed by means of the formalism of section 3. Section 5 specializes this algorithm to conventional databases. Finally, section 6 concludes this paper with an evaluation.

2 Background

Although the following techniques can be applied also in other frameworks, we will present our enforcement algorithm for dynamic consistency constraints with respect to a database, moreover a relational database [4].

A relational database is logically partitioned into a time-vaying extension and an intension that does not change over long period of time. The *extension* comprises the actual set of data (tuples) contained in the database. These data are grouped into relations. The *intension* of the database corresponds to the set of application-specific rules those data should obey, for instance the format of the relations. It imposes restrictions concerning the structure of the relations of the database and constraints with respect to their content.

From the point of view of logic, a database can be regarded as a set of ground atomic formulas describing the tuples in its extension together with more complex formulas representing the structural and consistency constraints in its intension. In this way, the database is conceptually partitioned into a set of logical formulas that change frequently (its extension) and sets of formulas relatively time-invariant (its intension). Each time the database is updated, its extension changes, either by inserting some new tuples or by deleting old ones, or even both in response to a *modify* command. Each update will be seen as a transaction, i.e a sequence of database commands that have to be carried out without interruptions. The synchronization aspects of a transaction will be ignored.

The extension of the database will be called a *state* when the temporal dimension is to be stressed. In particular, a database state will be seen as the content of the database between two consecutive updates. The succession of states a database passed through since it was created will be called its *history*. Given a database D, its history up to the current time t_{curr} can be described by the following state-time succession:

$$H(D,t_{curr}) = (d_0,t_0), (d_1,t_1), ..., (d_{curr},t_{curr}).$$

where $d_0, d_1, ..., d_{curr}$ is the complete, exhaustive and ordered sequence of all the database states D passed through while $t_0, t_1, ..., t_{curr}$ form a linear sequence of time points. d_0 is the initial state (usually the empty state) and d_{curr} is the current state of D, at time t_{curr}.

Even though historico-temporal databases [3, 13, 17] are not the exclusive topic of this paper, the main stream of the discussion will suppose that the complete database history is available. Therefore the language and algorithms that will be devised in the following sections are tailored for historical and temporal databases. That formalism will be weakened in section 5 to deal with the more common *current-state-only* databases.

3 Expression of dynamic constraints

A *dynamic integrity constraint* is a consistency constraint that cannot be expressed without referring to the states of the database. A dynamic constraint is not concerned with the content of the database but with what happens to it over time. Therefore, any formalism aimed at expressing such constraints must incorporate an explicit temporal dimension.

One way to specify and enforce constraints is to restrict the possible states of a database. This can be done by augment first order predicate calculus with the capability of specifying assertions whose truth-value is computed with respect to a particular point of time. Consider the dynamic integrity constraint: "salaries never decrease". Assuming that salaries are stored in a relation *salary(person, amount)*, it can be described as follows:

$$\forall t_1\ \forall t_2\ \forall p\ \forall s_1\ \forall s_2\ (\text{salary}(p, s_1)_{t_1} \wedge \text{salary}(p, s_2)_{t_2} \wedge t_1 < t_2 \rightarrow s_1 \leq s_2)$$

the subscripts t_1 and t_2 refer to the history of the database: f_t inquires if the formula f is true at time t - by convention, t_{curr} is kept implicit. For example, salary(Fred, 55555) evaluates to true if Fred's salary is 55555.

Alternatively instead of restricting states, we could restrict the operations that potentially violate constraints, thanks to the linearity of time. This approach would specify the above constraint as:

$$\forall p\ \forall s_1\ \forall s_2\ (\text{salary}(p, s_1) \wedge \text{MODIFY}(\text{salary}, p, s_2) \rightarrow s_1 \leq s_2)$$

Where *MODIFY(salary, person, <new-value>)* changes the value of the attribute *amount* of the salary of *person* (assumed as a key) to <*new-value*>. Note that the above constraint is much easier to enforce, since it only refers to the current state and update operations that potentially violate the constraint, but not to the history of the database. It requires to impose syntactical restrictions on operations performed on a database.

Like any consistency constraint, a dynamic consistency constraint is triggered by a data change. It is highly inefficient to check all the constraints for a particular update operation: in fact very few of them are relevant for the update. A convenient way to alleviate this problem consists in matching the update against the representation of the integrity constraint [14]. Particular expressions in the text of an integrity constraint that can be matched against the expression of an update operation in order to select only the possibly relevant integrity constraints is called an *activation pattern* in [5, 6].

In this paper, we assume that only the following three operations that can be used to manipulate databases are the insertion of a tuple, its deletion and querying. *assert/2*, *retract/2*, and *query/2* (where *P/n* is the informal notation commonly used in logic programming to express that the predicate *P* has arity *n*) are the corresponding activation patterns; they are first order predicates in our constraint language[1]. The first argument of these predicates is a *generic tuple* (i.e. an expression that looks like a tuple except the possible presence of variables), whereas its second argument is a time tag. As in the static case [14], pre-interpreted symbols (as >, =, +, ...) are needed to allow an easy formulation of these constraints.

In the following, we will illustrate our constraint specification language by discussing seven example, six of which involve dynamic constraints. In the next section, we will use these examples to illustrate and explain our constraint enforcement algorithm.

The predicate *assert/2* makes possible to express constraints concerning the assertion of individual tuples in the database. Its general pattern is

assert(<tuple>,<time>).

It evaluates to true if the tuple <tuple> is/was inserted into the database state that corresponds to the time tag <time>.

A few examples are in order. The integrity constraints they express are shown in a sequence of increasing complexity.

1. "The tuple *a(b,c,d)* can be asserted in the database only at time *44580784*"

   $\forall t(\mathrm{assert}(a(b,c,d),t) \rightarrow t=44580784)$

2. "A tuple having a pattern such as *a(b,X,d)*, where any value can be inserted for the variable *X*, can be asserted only in a state having an even time tag"

   $\forall t \forall X(\mathrm{assert}(a(b,X,d),t) \rightarrow (t \bmod 2)=0)$

3. "A tuple of the form *a(b,X,Y)*, where *X* and *Y* are variables, can be asserted in the database only if a tuple of the form *e(f,Y,Z)*, where *Y* and *Z* are variables, and in particular *Y* must be the same as in the new tuple, has been asserted in a previous state of the database"

   $\forall t \forall Y(\forall X\ \mathrm{assert}(a(b,X,Y),t) \rightarrow \exists t_1(t_1 \leq t \wedge \exists Z\ \mathrm{assert}(e(f,Y,Z),t_1)))$

The syntax and behavior of *retract/2* are similar to those of *assert/2*. The only difference is that it deals with the retraction of a tuple. Its pattern is

retract(<tuple>,<time>)

and it evaluates to true, when <tuple> was/is deleted from the database state that carries the time tag <time>. Consider the following examples:.

[1] Actually, our language provides a fourth operation, namely *transac/2*, for dealing with transactions. I will not be discussed due to space limitations.

4. "Once asserted, the tuple *a(b,c,d)* can never be removed"

$\forall t\ \neg retract(a(b,c,d),t)$

5. "A tuple can be removed from the database only if it has previously been inserted in it"

$\forall X \forall t(retract(X,t) \rightarrow \exists t_1(assert(X,t_1) \wedge t_1<t))$

Note that (4) only requests that the tuple a(b,c,d) is not deleted from the database using a retract command. However, it does not disallow any other changes; e.g. it could modified or manipulated by other operations. (5) could be read as the minimum condition to remove a tuple from the database: its presence in it. It is however weaker, for the same reasons just pointed out for (4). Note that (5) is useless since a tuple must be searched and found in order to be deleted.

The predicate *query/2* models a simplified (tuple-at-a-time) form of the query operation of relational and deductive databases. It has the following pattern:

query(<tuple>,<time>).

It succeeds simply if the database contains(ed) the tuple <tuple> when it is (was) in the state identified by <time>.

query/2 is used to assess the presence of a tuple in a particular state of the database's history. Therefore this predicate deals with static aspects of the database: its extension at a particular instant. *query/2* is the basic tool for expressing static integrity constraints. Every static constraint can in fact be formulated using only this predicate and a very limited quantifiers pattern similar to what will be presented in the section 5. Note that a formula containing only *query/2* predicates can express also dynamic constraints, e.g. our constraint "salaries never decrease" could be written as follows:

$\forall t_1\ \forall t_2\ \forall p\ \forall s_1\ \forall s_2\ (query(salary(p,s_1),t_1) \wedge query(salary(p,s_2),t_2) \wedge t_1<t_2 \rightarrow s_1 \leq s_2)$

It should be noted that *assert/2* (*retract/2*) is only true when the particular tuple was inserted (resp. deleted) at the particular point of time. Furthermore, we assume that if an update is performed at time t, it becomes visible at time t+1. That is, if we assume that a tuple p has not been in the database before, and was asserted at time t, and was not retracted at time t+1, *assert(p,t)*, *query(p,t+1)*, *query(p,t+2)* are true, and *assert(p,t+1)*, *query(p,t)* are false.

The following example (6) expresses a static constraint in this way. Example (7) illustrates how *query/2* and the previous predicates can cooperate to define new dynamic integrity constraints.

6. "At any time, there must be in the database a unique tuple of the form *a(b,c,X)*, where any value can be substituted to the variable *X*"

$\forall t(\exists X\ query(a(b,c,X),t) \wedge \forall X \forall Y(query(a(b,c,X),t) \wedge query(a(b,c,Y),t) \rightarrow X=Y))$

7. "A tuple of the form *a(b,X,d)* can be asserted in the database only if it is not already there and some tuple of the form *e(Y,X)* is contained in it"

$\forall t \forall X(assert(a(b,X,d),t) \rightarrow \neg query(a(b,X,d),t) \wedge \exists Y\ query(e(Y,X),t))$

What has been done in this section is to define informally a language for expressing dynamic constraints. As mentioned earlier, the language formalism that underlies our constraint language is first order logic. The connectives and quantifiers are the usual ones. All the predicate symbols in our language are interpreted. They have their meaning hard-wired into the language. These predicate symbols comprise the activation patterns (*assert/2*, *retract/2* and *query/2*), as well comparison operators, such as > and =.

Our constraint language allows for interpreted (such as +, *mod*, etc.) as well as for uninterpreted function symbols in constraints. The former are not needed (nor allowed) inside the activation patterns. This restriction is made in order to avoid underspecified expressions involving interpreted operators when simplifying the constraints. This is, in general, a difficult task that only recently received some attention, most notably in the context of logic and constraint programming; for example, in the CLP language described in [8].

The kind of logic to be used merits some notes. So far, a single-sorted (i.e. pure) first order logic has been implicitly assumed. However, for the kind of problems that have to be coped with, a typed (or multi-sorted) logic (see [11]) is more adequate. Anyhow, the single-sorted case will continue to be adopted for the sake of simplicity.

4 The enforcement algorithm

In the previous section, first order predicate calculus has been extended to permit a direct expression of dynamic integrity constraints (namely, allowing activation patterns and temporal variables and constants). What will be faced in this section is how to enforce them.

Our constraint enforcement algorithm computes the set of relevant constraints for a given update operation and check them. The following steps are performed:

1. select the dynamic integrity constraints that are potentially affected by the update;
2. simplify them as much as possible;
3. check the simplified constraints if they are not elementary (i.e. of the form *true* of *false*);

It should be mentioned that due to space limitation we will describe the algorithm informally in this paper. These phases will be explained in detail while carrying out an example. Suppose the set of integrity constraints associated to the database contains all those exemplified in the previous section. Therefore, the database is consistent if and only if the formulas (1) through (7) are valid. The constraints stated in the last section are here summarized.

1. $\forall t(\text{assert}(a(b,c,d),t) \rightarrow t=44580784)$
2. $\forall t \forall X(\text{assert}(a(b,X,d),t) \rightarrow (t \bmod 2)=0)$
3. $\forall X\ \text{assert}(a(b,X,Y),t) \rightarrow \exists t_1(t_1 \leq t \wedge \exists Z\ \text{assert}(e(f,Y,Z),t_1)))$
4. $\forall t\ \neg\text{retract}(a(b,c,d),t)$

5. $\forall X \forall t(\mathrm{retract}(X,t) \rightarrow \exists t_1(\mathrm{assert}(X,t_1) \wedge t_1<t))$

6. $\exists X\ \mathrm{query}(a(b,c,X),t)\ \wedge \forall X \forall Y(\mathrm{query}(a(b,c,X),t) \wedge \mathrm{query}(a(b,c,Y),t) \rightarrow X=Y))$

7. $\forall t \forall X(\mathrm{assert}(a(b,X,d),t) \rightarrow \neg \mathrm{query}(a(b,X,d),t) \wedge \exists Y\ \mathrm{query}(e(Y,X),t))$

Suppose that the database contains only the two tuples *a(b,c,e)* and *e(c,c)*, and that its time tag is *14905*. The database is consistent since the only static constraint is (6) and it is satisfied. The database is now updated by asserting the tuple *a(b,c,d)*. The previously stated steps are performed in order.

The first step prescribes to select the relevant dynamic integrity constraints. This phase can be performed by matching the update against the atoms that constitute the integrity constraints and select the constraints for which a successful match is found - if the two do not match it is impossible to violate a constraint, because it does not apply in the context of the particular update operation.. Because of the syntactic restriction imposed on the integrity constraints, the classical unification algorithm is enough for this task. Once found, a match can conveniently be represented as a substitution for the variables that occur in the integrity constraint (since the update is always ground). Note that if there are two occurrences of an atom in the same formula that match the update, then that formula is selected twice: once for each successful matched occurrence.

In the example, the update can be represented as

assert(a(b,c,d),14905).[2]

All the constraints are matched except (6), that is static - it will be treated by some static constraints enforcing algorithm for instance [9, 10, 14]. The selected constraints are the following: the matches are represented by underlining the matching atom and showing the substitution

1. $\forall t(\underline{\mathrm{assert}(a(b,c,d),t)} \rightarrow t=44580784)$ $\quad\sigma_1 = \{t/14905\}$
2. $\forall t \forall X(\underline{\mathrm{assert}(a(b,X,d),t)} \rightarrow (t \bmod 2)=0)$ $\quad\sigma_2 = \{X/c,t/14905\}$
3. $\forall X\ \underline{\mathrm{assert}(a(b,X,Y),t)} \rightarrow \exists t_1(t_1 \leq t \wedge \exists Z \mathrm{assert}(e(f,Y,Z),t_1)))$ $\quad\sigma_3 = \{X/c,Y/d,t/14905\}$
5. $\forall X \forall t(\mathrm{retract}(X,t) \rightarrow \exists t_1(\underline{\mathrm{assert}(X,t_1)} \wedge t_1<t))$ $\quad\sigma_5 = \{X/a(b,c,d),t_1/14905\}$
7. $\forall t \forall X(\underline{\mathrm{assert}(a(b,X,d),t)} \rightarrow \neg \mathrm{query}(a(b,X,d),t) \wedge \exists Y\ \mathrm{query}(e(Y,X),t))$
   $\sigma_7 = \{X/c,t/14905\}$

The second step of the enforcement procedure consists of simplifying the constraints produced by step 1. This is done by instantiating each selected constraint by applying the substitution found in the previous step. The obtained formulas are then simplified as much as possible by means of the elementary rules of logic (as in [2, 14]) and arithmetic, and by using some basic properties of the updates, as for example that no two updates can occur to the same database state. The simplification step is now

[2] Note that the same syntax (here, *assert/2*) is used to express different concepts: the effective update here and an activation pattern in a constraint.

illustrated in the context of the example. As in [14], **T** and **F** represent the truth values *true* and *false* respectively.

1. assert(a(b,c,d),14905) $\rightarrow$ 14905=44580784)
   T $\rightarrow$ **F**
   F

2. assert(a(b,c,d),14905) $\rightarrow$ (14905 mod 2)=0
   T $\rightarrow$ **F**
   F

3. assert(a(b,c,d),14905) $\rightarrow \exists$t1($t_1 \leq 14905 \wedge \exists$Z assert(e(f,Y,Z),$t_1$))
   T $\rightarrow \exists t_1(t_1 \leq 14905 \wedge \exists$Z assert(e(f,Y,Z),$t_1$))
   $\exists t_1(t_1 \leq 14905 \wedge \exists$Z assert(e(f,d,Z),$t_1$))

5. $\forall$t(retract(a(b,c,d),t) $\rightarrow$ assert(a(b,c,d),14905) $\wedge$ 14905<t)
   $\forall$t(retract(a(b,c,d),t) $\rightarrow$ **T** $\wedge$ 14905<t)
   $\forall$t(retract(a(b,c,d),t) $\rightarrow$ 14905<t)

7. assert(a(b,c,d),14905) $\rightarrow \neg$query(a(b,c,d),14905) $\wedge \exists$Y query(e(Y,c),14905)
   T $\rightarrow \neg$query(a(b,c,d),14905) $\wedge \exists$Y query(e(Y,c),14905)
   T $\wedge$ **T**
   T

The *query/2* statements have been simplified by querying the database in the current state (remember that the current state of our example database contains only the tuples *a(b,c,e)* and *e(c,c)*, but not *a(b,c,d)* which will be included in the next state of the database corresponding to the time tag *14906*).

The simplification procedure can lead to three results. If at least one constraint has been falsified (i.e. **F** was derived), then an inconsistency has been found. Therefore, no further step is needed. If all the constraints are reduced to **T**, then the update does not violate the database. Step 3 can then be skipped. In all the other cases, the procedure must go on with step 3, which decides which constraints have to be checked for the particular update.

In the example, the update is invalid since it falsifies the constraints (1) and (2). In order to be able to proceed with step 3 without changing example, we assume that the constraints (1) and (2) have suddenly been dropped by the system administrator. Taking this into consideration, constraints derived from the three constraints (3), (5), and (7), are still in contention after phase 2.

Summarizing the situation, constraint (3) was simplified to become

3'. $\exists t_1(t_1 \leq 14905 \wedge \exists$Z assert(e(f,d,Z),$t_1$))

it refers to the past of the database, and has to be checked by querying the history of the database: if a tuple matching *e(f,d,?)* has not been inserted at some time in the past, the constraint is violated, and the insertion of *a(b,c,d)* has to be rejected.

Constraint (5) has been simplified to:

5'. $\forall t(retract(a(b,c,d),t) \rightarrow 14905<t)$.

It expresses that the retraction of the tuple *a(b,c,d)* is allowed in any state having a time label greater than *14905*. This label cannot be checked since it refers to states in the future. Since it does not involve the current update, it can be assumed to be true (**T**). However, that simplified constraint must be recorded for future use in the database: it will become a new integrity constraint that must be satisfied by all the successive updates.

Finally, constraint (7) evaluates to **T**, which means that this constraint need not be checked for the particular update.

In summary, for the particular update only a single dynamic constraint (3') has to be checked.

So far our algorithm looks quite similar to the one proposed by Nicolas in [14] for static constraints. The remainder of this section will focus on complications that arise in our enforcement algorithm due to the special nature of temporal constraints. Due to the lack of space we will discuss these complications informally.

The first singularity arises from interactions between *query/2* and the predicates *assert/2* and *retract/2*, which make it necessary to modify the matching algorithm used in step 1 to decide if a constraint can potentially be violated. In fact, an insertion or a deletion becomes visible (i.e. can be queried) in the successive state of the database. For example, if we have the constraint

8. $\forall t\ (query(p,t) \rightarrow t < 450)$

and p is asserted at time 550, our generalized unification algorithm would unify *assert(p,550)* with *query(p,t)* obtaining the simplified constraint *551 < 550* which evaluates to **F** (since *query(p,551)* is trivially true); consequently, the update would be rejected. Similarly, for the constraint 8' given below

8'. $\forall t\ (\neg query(p,t) \rightarrow t < 450)$

the retraction of p at time 450 (or later) would be rejected by our enforcement algorithm - as stated in (8') p should not become false after time 449.

A second complication arises from the fact that dynamic constraints can be violated implicitly as time passes by. Consider again constraint 8', and let us assume that has never been asserted or retracted during the history of the database. The algorithm presented so far, will consider 8' irrelevant for any update operation that was performed on our example database and will not check the constraint. When time 450 is reached, the above constraint becomes violated.

To avoid these problems, it becomes necessary to consider any constraints containing *query/2* to be relevant for any update, and to simplify the above constraint by substituting the current time for the universally quantified temporal variable. Note that the constraint 8' can now be evaluated for time 450. Consequently, constraints containing *query/2* patterns universally quantified on temporal variables have to be evaluated for the current point of time, possibly detecting violations of constraints that originally referred to the future.

Therefore, if *r* (which is different from *p*) is asserted at time 450, our enforcement algorithms derives the following simplified constraint 8" from 8', which has to be checked in the database:

8". query(p,450)

In the case that a constraint involves multiple such universally quantified temporal variables, all possible substitutions of the current time for each of these variables have to be considered by the enforcement algorithm.

A third singular aspect of the previously described enforcement algorithm is that it can derive undecidable temporal formulas. Suppose for instance the following constraint reaches step 3 of the algorithm:

9. ∃t assert(p,t)

If p has previously been inserted into the database, it evaluates to **T**. Otherwise, no truth value can be assigned to it since it refers to a course of actions to be taken in the future. A conservative approach would consider this constraint false (for it can make the database inconsistent in the future). It seems anyway more natural to ignore it, and therefore behave as if it were true.

The previously discussed algorithm can be practically implemented in several forms. One way is to use a system that directly applies the algorithm and enforces constraints at run-time. Another approach would be to use a precompiler that augments the application programs with code that enforces the dynamic consistency constraints. If our application program contains code that asserts *a(b,c,d)*, then the precompiler would add code that enforces the simplified constraint 3' to it, but no code for any other dynamic constraints. This is a necessary condition for the efficiency of any automatic constraint enforcement algorithm: only those constraints that can potentially be violated by a particular update should be checked. A Prolog prototype of the algorithm presented in this section has been implemented.

5 Restricted Dynamic Constraints

This section is primarily aimed at defining a form of the previously presented formalism weak enough to be conveniently used in conventional databases. In order to achieve that goal, the possibility of getting rid of step 3 by restricting the range of the expressible constraints is investigated. In fact, a (simplified) integrity constraint goes through step 3 if and only if it refers to a database state different from the current one.

All the formulas that refer only to the current state (once instantiated) have the following pattern:

∀t φ

where

- φ is a formula that does not contain any quantifier applied to a variable standing for a time label, and
- the only occurrences of *t* can be found as the second argument of *assert/2*, *retract/2* and *query/2*.

This syntactic restriction seems to be a huge lost in expressive power. However, it is our belief that, apart from some specifically temporal application, the integrity constraints commonly used do not need to refer to what happened in the past history of the database. The only thing of interest is the current content of the database and the updates that are currently made on it. The second part of the restriction forbids a direct manipulation of the time labels as when requiring that a certain property holds only after a certain instant. This constraint would not be sound in a temporal or historical database.

As a consequence of this restriction the constraints 2, 3 and 5 of the previous example are ruled out. The integrity constraints that satisfy this syntactic restriction have been called *restricted dynamic constraints*, or *RDC*s. The only RDCs of section 3 were shown in the examples 4, 6 and 7. Note that all the traditional static integrity constraints are RDCs in this framework.

Before saying more about RDCs, notice that the use of time tags is so strictly ruled in them that they can be dropped or, better, made implicit. In this way, the predicates defined in section 3 do not need their second argument anymore. They become therefore *assert/1*, *retract/1* and *query/1*.

With this new syntax, the examples 4, 6 and 7 are written as follows:

4 $\neg$retract(a(b,c,d))

6 $\exists$X query(a(b,c,X)) $\wedge$ $\forall$X$\forall$Y(query(a(b,c,X)) $\wedge$ query(a(b,c,Y)) $\rightarrow$ X=Y)

7 $\forall$X(assert(a(b,X,d)) $\rightarrow$ query(a(b,X,d)) $\wedge\exists$Y query(e(Y,X)))

RDCs were devised with the purpose of eliminating phase 3 of the enforcement procedure in a conventional database. Thus, in order to enforce constraints expressed as RDCs, it is enough to perform the following steps:

1. select the dynamic integrity constraints that are involved by this update;
2. simplify them as much as possible;

Note that in many cases this procedure allows to enforce integrity constraints by looking only at the constraints and at the update. In fact, unless a constraint contains the *query/1* predicate, there is no need to access the (extension of the) database. Since there are relatively few constraints in general, the check can be done in main memory. If a *query/1* is encountered, a very well defined, very focused search in the database has to be carried on. This is nearly the only cost of enforcing (dynamic) constraints for this is the only case in which an access to the database (in secondary storage) is needed.

6 Conclusions

The topic of consistency enforcement in database and in knowledge based systems is nowadays very popular, even though only few such systems provide tools for expressing and automatically checking integrity constraints. However, the subproblem of static integrity constraints is now relatively well understood and several algorithm have been devised for their enforcement. On the other hand, dynamic integrity constraints have received only little attention, and it is difficult to find this topic treated in the literature but from a very general point of view.

The paper focused on the automatic enforcement of dynamic consistency constraints. A logical formalism for the specification of dynamic constraints has been presented that extends first order predicate logic by a temporal dimension and by the availability to refer to operations that perform changes. Nicolas' classical enforcement algorithm [14] for the enforcement of static consistency constraints has been extended to cope with dynamic constraints. It should also be mentioned that we implemented our algorithm in a PROLOG environment.

Our current research focuses on the validation of the presented algorithm, and on its integration into a knowledge base management that supports temporal queries. Due to the novelty of this research there are many other questions that deserve further exploration. How does one cope with constraints that refer to events in the future and what should be their role in a database management system - we gave an example of this problem in the paper (constraint 5')? A subclasses of dynamic constraints that can be enforced by conventional database management systems that only store the current state (RDCs) has been identified. Our algorithm has been adapted to treat this important subclass.

References

1. Ceri S., Gottlob G., Tanca L.: "What You Always Wanted to Know About Datalog (And Never Dared to Ask)", *IEEE Transactions on Knowledge and Data Engineering*, 1989, pp 146-166.

2. Cervesato I., Eick C.F.: "Specification and Enforcement of Dynamic Consistency Constraints", in *Proceedings of the International Conference on Information and Knowledge Management* (*CIKM-92*), Baltimore, MD, Nov. 1992, pp 193-200.

3. Clifford J., Tansel A.U.: "On an Algebra for Historical Relational Databases: Two Views", in *Proceedings of the ACM SIGMOD conference*, Austin TX, 1987, pp 247-265.

4. Elmasri R., Navathe S.B.: "*Fundamentals of Database Systems*", Addison-Wesley, 1989.

5. Eick C.F.: "Activation Pattern Controlled Rules: Towards the Integration of Data-Driven and Command-Driven Programming", *Journal of Applied Intelligence*, vol. 2, 1992, pp. 75-91.

6. Eick C.F., Werstein P.: "Rule-Based Consistency Enforcement for Knowledge-Based Systems" Accepted for *IEEE Transaction on Knowledge and Data Engineering*, 1990.

7. Gehani N., Jagadish H. V.: "ODE as an Active Database: Constraints and Triggers, in *Proc. Int. Conf. on Very Large Databases*, Barcelona, 1991, pp. 327-336.

8. Jaffar J., Lassez J.L.: "Constraint Logic Programming" in *Acts of the 14th POPL*, Munich, West-Germany, January 1987, pp 111-119.

9. Kobayashi I.: "Validating Database Updates", *Information Systems* 9 (1), 1984, pp 1-17.

10. Kowalski R.A., Sadri F., Soper P.: "Integrity Checking in Deductive Databases", in *Proceedings of the 13th VLDB*, Brighton 1987, pp 61-69.

11. Lloyd J.W.: "*Foundations of Logic Programming*", Springer Verlag, II ed., 1987.

12. McCarthy D.R., Dayal U.: "The Architecture of an Active Database System", in *Proc. ACM SIGMOD Conf. on Management of Data*, Portland, 1989, pp. 215-224.

13. Navathe S.B., Ahmed R.: "A Temporal Relational Model and a Query Language", *Information Sciences* 49, 1998, pp 147-175.

14. Nicolas J.-M.: "Logic for Improving Integrity Checking in Relational Databases", *Acta Informatica* 18, 1982, pp 227-253.

15. Olive A.: "Integrity Constraint Checking in Deductive Databases", in *Proc. VLDB-Conference*, Barcelona, 1991, pp. 513-524.

16. Qian X., Weiderhold G.: "Knowledge-Based Integrity Constraints Validation", in *Proc. Int. Conf. on Very Large Databases*, Kyoto, 1986, pp 3-12.

17. Snodgrass R., Ahn I.: "A Taxonomy of Time in Databases", in *Proceedings of the ACM SIGMOD conference*, Austin TX, 1987, pp 236-246.

18. Stonebraker M., Hansen H., Potomianos S.: "The POSTGRES Rule Manager", *IEEE Transactions on Software Engineering*, vol. 14, no. 7, 1988, pp. 897-907.

19. Su S., Chen H.: "A Temporal Knowledge Representation Model OSAM*/T and its Query Language QQL/T", in *Proc. VLDB-Conference,* Barcelona, 1991, pp. 431-442.

20. Widom J., Ceri S.: "Deriving Production Rules for Constraint Maintenance", in *Proc. VLDB-Conference,* Brisbane, 1990, pp. 566-577.