Specification and Enforcement of Dynamic Consistency Constraints

Iliano Cervesato and Christoph F. Eick
Department of Computer Science
University of Houston
Houston, TX 77204-3475
e-mail: (iliano,ceick)@cs.uh.edu

Abstract

This paper centers on the automatic enforcement of dynamic consistency constraints — dynamic constraints are constraints that cannot be checked by solely inspecting the most recent state of a data- or knowledge base. A logical formalism for the specification of dynamic constraints is presented that extends first order predicate logic by a temporal dimension and by the capability of restricting update operations that potentially violate constraints. In general, a large number of constraints exist in real world databases. It has been recognized that it is highly inefficient to check all these constraints for a particular update operation, because very few of those constraints are relevant for the update. To alleviate this problem, an algorithm is presented which determines if a dynamic constraint needs to be checked for a particular update.

1 Introduction

Consistency constraints (frequently called *semantic integrity constraints* in the literature) 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 underly the particular application area. Usually, the code that enforces consistency constraints is significantly longer than the code needed to perform a particular update operation. For example, when an application program inserts a new social security number it has to check at least two constraints: first, that the inserted social security number consists of 9 digits, and second, that it is unique.

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.

Although this idea looks commercially quite attractive, it has not become commercial reality yet. The existing commercial systems, e.g. relational database management systems or AI-programming environments, only partially support the specification and automatic enforcement of consistency constraints. In current practice, the enforcement of consistency constraints still rests on the shoulders of application programmers.

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, a constraint "salaries never decrease" cannot be checked by simply inspecting the contents of a database, because it 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 — due to the fact that the constraint has been specified in the context of a single state and not of multiple states. The uniqueness constraint is therefore a static constraint. The importance of dynamic constraints has been recognized by research on temporal and/or historical databases (see for example [13] and [9]).

Appeared in the *Proceedings of the International Conference on Information and Knowledge Management — CIKM'92*, (Y. Yesha editor), pp. 193–200, Baltimore, MD, 8–11 November 1992.

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 [10], and later extended and modified by many researchers (most notably by [5]).

Recently, rule-based programming paradigms gained some popularity for databases. These efforts found their expression in two directions: in *deductive databases* 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 in databases. 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 (see for example [6], and [11]).

On the other hand, active databases, such as POSTGRES [12] and HIPAC [8], support data-driven production rules that facilitate the coding of exceptions handlers for constraint violations. The STARBUST DBMS [14] supports set-oriented production rules that are integrated with DBMS transaction concepts. In their framework, a transaction computes a set of update-operations that it wants to perform for a particular database triggering active rules that react to these changes, e.g. reject violations of consistency constraints by cancelling the transaction. Moreover, [1] introduces activation pattern controlled rules — a generalization of classical data-driven production rules — for the purpose of consistency enforcement. Activation pattern controlled rules augment rules by an additional left-hand side – the activation pattern – which matches calls of particular commands and sensitizes production rules for the execution of particular commands. In the proposed generalization, rules are not triggered by data changes but rather by the intent to perform a data change, which is important for consistency enforcement: classical data-driven mechanisms can only react to the violation of constraints (e.g. by undoing the change that violated the constraint) but not prevent the violation itself. Finally, in the ODE object-oriented database system [3] constraints are associated with objects. The associated constraints are checked each time a member function (or the constructor) is called for the particular object.

Unfortunately, we are not aware of any papers that directly center on the automatic enforcement of dynamic constraints. The main idea of the paper is to generalize Nicolas' enforcement algorithm for dynamic constraints. Section 2 introduces temporal databases and presents our formalism for the specification of dynamic constraints. Section 3 describes an enforcement algorithm for dynamic constraints. Section 4 concludes the paper.

2 A Language for Dynamic Constraints

Although our techniques can also be applied in other frameworks, we assume that dynamic constraints are enforced with respect to a relational database. A relational database can be regarded as a set of relations whose contents changes over time. The current contents of a relation is called its *extension*, whereas the contents of all the relations that belong to a database is called the *extension of the database*. On the other hand, certain parts of a relational database do not change over long periods of time, because they express the law / rules / constraints of the application area. These parts, we call the *intension* of the relational database. It imposes restrictions concerning the structure of the relations of the database and constraints with respect to their content.

From a logical point of view, a database can be considered to be a set of logical (atomic) formulas describing the tuples of the extension of the database together with logical formulas describing the structural and consistency constraints of the intension of the database. In this way, the database is conceptually partitioned into a set of logical formulas that frequently change with time (its extension) and sets of logical formulas that do not or very rarely change over time (its intension). Each time the database is modified its extension changes. Each update can be seen as a sequence of commands that have to be carried out as an atomic unit. The extension of the database will be called a *state* in the following when its temporal dimension is to be stressed.

We will view the *history* of a database as a sequence of states linked by updates (simple or complex) that transformed a database from one state to the other. In particular, we see the history of a database D, denoted by H(D) as a complete, exhaustive, ordered sequence of state-time pairs that occurred during the evolution of the database D.

$H(D) := (s_0, t_0), (s_1, t_1), ..., (s_{curr}, t_{curr})$ with
$t_1 < t_2 < ... < t_{cur}$ are points of time
s_i for $i = 1, ..., curr$ are states of the database

In the above, s_0 denotes the initial state of the database — usually the empty state – and s_{curr} denotes the current state. The above definition associates every state with a time tag that uniquely identifies it with respect to other state. Note that two temporally different states of a database D can be actually the same.

In the following discussions we will assume the complete history of databases is available. Therefore, the language and algorithms that will be discussed in the next section are tailored for historical and temporal databases; namely, we assume that H(D) is accessible for our enforcement algorithm.

One way to express dynamic constraints is to augment first order predicate calculus with the capability of specifying assertions whose truth-value is computed with respect to a particular point of time. Assuming that salaries are stored in a relation $salary(person, amount)$, the constraint "salaries never decrease" could be described as follows:

(1a) $\forall t1 \forall t2 \forall p \forall s1 \forall s2 \quad (salary(p, s1)_{t1} \quad \wedge \quad salary(p, s2)_{t2} \ \wedge \ t1 < t2 \ \Rightarrow \ s1 \leq s2)$

In the above, the subscripts t1 and t2 refer to the history of the database, inquiring if a particular formula f is true at time t, denoted by f_t. If t is omitted, f is evaluated in the context of the current state. For example, $salary(Fred, 55555)$ evaluates to true if Fred's salary is \$55555. In the above, we specify that no state should be succeeded by another state in which the salary of the same person decreases.

One way to specify and enforce constraints is to restrict the possible states of a database, as in (1a). However, it has been frequently overlooked that this is not the only way to specify and enforce constraints. Alternatively instead of restricting states, we could restrict the operations that potentially violate constraints. We could reformulate the above constraint by referring to the changes that might potentially violate this constraint — disallowing salary decreasing modifications. This alternate approach would specify the above constraint as follows:

(1b) $\forall s1 \forall s2 \forall p(salary(p, s1) \ \wedge \ Modify(salary, p, s2) \ \Rightarrow \ s1 \leq s2)$

We assume that `Modify(salary,<person>,<new-value>)` is the operation to modify the salary of a person; e.g., `Modify(salary,Fred,66666)` modifies the salary of Fred to 66666. The above constraint should be read as follows: if there are modifications of salaries of a person p, and p's old salary is s1, then her/his new salary should be greater than or equal to the old salary. Note that the above constraint is much easier to enforce, because it only refers to the current state and update operations that potentially violate the constraint, but not to the history of the database, as (1a) does. On the other hand, the second approach requires to imposing syntactical restrictions on operations performed with respect to a database. In the second approach the constraint refers to calls of particular operations — calls of Modify-operations in the above example — which we will call activation patterns [2] in the following.

We believe that both temporal references and activation patterns are needed in order to cope with dynamic constraints. That is, in addition to references to tuples in the database (such as `salary(Fred,66666)`) our constraint language refers to calls of operations that access or manipulate the database, and uses temporal variables and constants to refer to events in the history of the database. Consequently, in our constraint language constraints are described by first order predicate calculus formulas that refer to the database contents and activation patterns, and which use temporal variables and constants.

Assuming this framework, our enforcement algorithm needs three inputs

- the history of the database
- the update operation for which constraint violations have to be prevented
- the set of dynamic consistency constraints that have to be enforced for the particular database

Our enforcement algorithm takes the above inputs and derives the set of constraints — that has to be checked for the particular update operation in order to guarantee the consistency of the database.

However, before we describe our enforcement algorithm, it is necessary to introduce our constraint specification language, which is the subject of the remainder of the section. Due to the lack of space we will only describe a simplified version of our constraint language.

In this paper, we assume that only the following three operations can be used to manipulate databases: assert/2, retract/2, and query/2, where P/n is the informal notation commonly used in the logic programming area to express that the predicate P has arity n.

In general, activation patterns are represented as predicates in our constraint language. The first argument of these predicates is a generic tuple (i.e. an expression that looks like a tuple except for the fact that it can contain variables), whereas its second argument is is a time tag. Moreover, pre-interpreted symbols (such as $\leq$ or $=$) are supported in our constraint language.

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

The predicate assert/2 makes it possible to express constraints concerning the insertion of individual tuples into the database. Its general pattern is `assert(<tuple>,<time>)`. The assert-predicate evaluates to true if the tuple `<tuple>` is/was inserted into the database state that corresponds to the time tag `<time>`.

Let us illustrate how the assert-predicate works using several examples:

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

$\forall t(assert(a(b,c,d),t) \Rightarrow t = 44580784)$

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

$\forall t \forall X(assert(a(b,X,d),t) \Rightarrow (t \ mod \ 2) = 0)$

3. "A tuple of the form a(b,X,Y), where X and Y are variables, can be only asserted in the database if a tuple of the form e(f,Y,Z) has been inserted earlier to the database."

$\forall t \forall Y \forall X(assert(a(b,X,Y),t) \Rightarrow \exists t_1(t_1 \leq t \wedge \exists Z assert(e(f,Y,Z),t_1)))$

The syntax and behavior of retract/2 are similar to assert/2. The only difference is that it deals with the deletion tuples. 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:

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

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

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

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

The application of assert/2 and retract/2 results in a new state in the history of the database. Furthermore, we assume for the rest of the paper that if an assert/2 or retract/2 operation is applied to a database at time t, their changes become visible in the newly generated state, which is tagged by $t+1$.

The predicate query/2 allows us to formulate queries with respect to the relational database. It has the form

`query(<tuple>,<time>)`

and evaluates to true, if the state of the database at time `<time>` contains `<tuple>`; otherwise, it evaluates to false. Query/2 allows us to refer to the history of a database, e.g. our constraint "salaries never decrease" could be written as follows:

$\forall t1 \forall t2 \forall p \forall s1 \forall s2((query(salary(p,s1),t1) \wedge (query(salary(p,s2),t2) \wedge t1 < t2) \Rightarrow s1 \leq s2)$

It should be noted that the assert-predicate is only true when the particular tuple was inserted 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, and 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 a unique tuple of the form a(b,c,X) in the database, where any value can be substituted for the variable X"

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

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

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

Note that the execution of a query/2 command does not change the state of a database. Therefore, query/2 does not affect the history of a database.

What has been done in this section is to define informally a language for expressing dynamic constraints.

However, before we represent our enforcement algorithm, it is necessary to explain the semantics and the processing of constraints in more detail. As mentioned earlier, the language formalism that underlies our constraint language is first order logic. The connectives and quantifiers are the usual ones. However, due to reasons given later, we assume that it is possible to recognize and distinguish references to temporal variables and constants syntactically from references to tuples, and activation patterns, which are assert, retract, and query. Furthermore, in our language we assume that all predicate symbols are interpreted. They have their meaning hard-wired into the language. Predicate symbols include activation patterns, such as assert, as well comparison operators, such as $<$.

Moreover, our constraint language allows for interpreted as well as for uninterpreted function symbols in constraints. However, interpreted function symbols (such as $+$, mod, etc) are not allowed inside activation patterns. This restriction derives from the operational aspects of the theorem proving techniques that underly our approach. This restriction is made in order to avoid underspecified expressions involving uninterpreted operators when simplifying the constraints. This is, in general, a difficult task that only recently received some attention by scientists, most notably in the context of logic and constraint programming; for example, in the CLP language described in [4]. On the other hand uninterpreted function symbols can be used freely in our language. For example, the arguments of assert/2, retract/2, and query/2 are not interpreted, as we have already mentioned above.

Finally, the kind of logic to be used for consistency enforcement deserves some discussion. So far, a single-sorted (i.e., pure) first order logic has been implicitly assumed. However, we feel that a typed (or multi-sorted) logic (see for example [7]) is more adequate for consistency enforcement — we believe that typed frameworks are more suitable for specifying and enforcing constraints that apply for a particular set of objects. However, for the sake of simplicity, we will continue our discussions assuming that a single-sorted logic is used for implementing our enforcement tool, which is the focus of the next section.

3 The Enforcement Algorithm

In the previous section, first order predicate calculus has been extended to permit a direct expression of dynamic integrity constraints (namely, we allow references to activation patterns and temporal variables and constants).

Our constraint enforcement algorithm computes the set of constraints that have to be checked for a given update operation out of a set of dynamic constraints. This computation is done in three steps:

1. Select the dynamic integrity constraints that potentially are affected by the update.
2. Simplify them as much as possible.
3. Remove those constraints from the set received in step (2) that are elementary or which refer to events in the future. The remaining constraints are the output of the algorithm — the set of constraints that need to be checked for the particular update operation.

It should be mentioned that due to space limitation we will describe the algorithm informally in this paper. More specifically, we will use the example (1) through (7), which have been redisplayed below, to explain the different phases of the algorithm.

1. $\forall t(assert(a(b,c,d),t) \Rightarrow t = 44580784)$

2. $\forall t \forall X(assert(a(b,X,d),t) \Rightarrow (t\ mod\ 2) = 0)$

3. $\forall t \forall Y \forall X(assert(a(b,X,Y),t) \Rightarrow \exists t_1(t_1 \leq t \wedge \exists Z assert(e(f,Y,Z),t_1)))$

4. $\forall t \sim retract(a(b,c,d),t)$

5. $\forall X \forall t(retract(X,t) \Rightarrow \exists t_1(assert(X,t_1) \wedge t_1 < t))$

6. $\forall t \exists V(query(a(b,c,V),t) \wedge \forall X \forall Y(query(a(b,c,X),t) \wedge query(a(b,c,Y),t) \Rightarrow X = Y))$

7. $\forall t \forall X(assert(a(b,X,d),t) \Rightarrow \sim query(a(b,X,d),t) \wedge \exists Y query(e(Y,X),t))$

We assume that the database is currently consistent, that it only contains two tuples a(b,c,e) and e(c,c), and that its current time tag is 14905. Now we assume that this database is updated by asserting the tuple a(b,c,d) which is represented in our temporal framework as follows:

```
assert(a(b,c,d),14905)
```

We will demonstrate in the following how our enforcement algorithm copes with this insertion. The first step, preselects those dynamic integrity constraints that are potentially violated by the update. This is done by matching the update operation against the constraints — if the two do not unify it is impossible to violate a constraint, because it does not apply in the context of the particular update operation. Because of the syntactical restrictions we impose on our constraint specification language, the classical unification algorithm is powerful enough to perform this task. If the update matches the constraint in at least one part the match is represented as a substitution for the variables that occur in the integrity constraint. This substitution is important, because it specifies the context in which the particular constraint can be violated by the update. Also note that if there are two occurrences of an atom in the same formula that match the update, then that formula is selected twice — two constraints potentially have to be checked for the update.

Consequently, all constraints are matched against

```
assert(a(b,c,d),14905)
```

with the exeception of (6) that is static and will not be treated by our algorithms, but rather by Nicolas' algorithm for static constraints.

Constraints (1), (2), (3), (5), and (7) match the above call successfully with the bindings indicated below. On the other hand, the match failed for the constraint (4); therefore, no substitution is given for it below.

1. $\forall t(assert(a(b,c,d),t) \Rightarrow t = 44580784)$ σ_1 = {t/14905}

2. $\forall t \forall X(assert(a(b,X,d),t) \Rightarrow (t\ mod\ 2) = 0)$ σ_2 = {X/c,t/14905}

3. $\forall t \forall Y \forall X(assert(a(b,X,Y),t) \Rightarrow \exists t_1(t_1 \leq t \wedge \exists Z assert(e(f,Y,Z),t_1)))$ σ_3 = {X/c,Y/d,t/14905}

4. $\forall t \sim retract(a(b,c,d),t)$

5. $\forall X \forall t(retract(X,t) \Rightarrow \exists t_1(assert(X,t_1) \wedge t_1 < t))$ σ_5 = {X/a(b,c,d),t1/14905}

7. $\forall t \forall X(assert(a(b,X,d),t) \Rightarrow \sim query(a(b,X,d),t) \wedge \exists Y query(e(Y,X),t))$ σ_7 = {X/c,t/14905}

The second step of the enforcement procedure consists of simplifying the constraints that have been received in step (1). This is done by instantiating each selected constraint by applying the substitution found in the previous step. The obtained formula is then simplified as much as possible by means of the elementary rules of logic and arithmetic, and by using rules that underly the semantics of our operators assert/2, query/2, etc. For our example, we obtain the results listed below, in which 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. $assert(a(b,c,d),14905) \Rightarrow \exists t_1(t_1 \leq 14905 \wedge \exists Z assert(e(f,Y,Z),t_1))$
$T \Rightarrow \exists t_1(t_1 \leq 14905 \wedge \exists Z assert(e(f,Y,Z),t_1))$
$\exists t_1(t_1 \leq 14905 \wedge \exists Z assert(e(f,d,Z),t_1))$

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. $assert(a(b,c,d),14905) \Rightarrow \sim query(a(b,c,d),14905) \wedge \exists Y query(e(Y,c),14905)$
$T \Rightarrow T \wedge T$
T

The query/2 statements have been simplified in the above by querying the database (remember that the current state of our example database only contains 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 described simplification algorithm can lead to three results. If at least one constraint has been falsified, then an inconsistency has been found. Therefore, no further step is needed, and the update is rejected. If all the constraints are reduced to T, then the update cannot violate any consistency constraints, because none of the constraints applies for the particular update. No constraints need to be checked for the particular update, and step (3) can 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 order to be able to proceed with step (3) without changing the example, we assume that constraints (1) and (2) have been dropped by the system administrator. Taking this into consideration, constraints derived from the three constraints (3), (5), and (7), are still in contention.

Constraint (3) was simplified to become:

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

The above constraint refers to the past of the database, and has to be checked by querying the history of the database: if a tuple that matches `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'. $\forall t(retract(a(b,c,d),t) \Rightarrow 14905 < t)$

Constraint (5') expresses that the retraction of the tuple a(b,c,d) is allowed in any state having a time label greater than 14905. This constraint cannot be violated at the current point of time, since it refers to states in the future. However, this does not exclude the constraint from becoming violated in the future.

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 constraint ((3')) has to be checked.

So far our algorithm looks quite similar to the one proposed by Nicolas [10] 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 and activation patterns. Due to the lack of space we will discuss these complications informally.

The first complication arises from interactions between the query-predicate and the assert- and retract-predicate, which make it necessary to modify our unification algorithm. For example, if `T` is a temporal variable `(assert(a(b,c,d),12))` unifies `(query(a(b,c,d),T)` by binding `T` to 13 (remember that a(b,c,d) is visible in the next state which is 13); similarly, `retract(a(b,c,d),12)` unifies `(~query(a(b,c,d),T)` with `T` being bound to 13. In general, the unification algorithm used in step (1) that decides if a constraint can potentially be violated has to take some special cases into consideration that do not occur in the classical unification algorithm — namely, retract and assert operations unify query-operations as outlined above. For example, if we have a 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 $551 < 450$ which evaluates to F; consequently, the update would be rejected. Similarly, for the constraint (8') given below

8'. $\forall t \; (\sim query(p,t) \;\Rightarrow\; t < 450)$

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.

The 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 p is initially false and 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 — note that p never has been retracted. When time 450 is reached, the above constraint becomes violated.

To avoid these problems, it becomes necessary to consider constraints that have form

$\forall t ...(\; ... \wedge query(p,t) \wedge \; ... \;\Rightarrow\; ...)$

$\forall t ...(\; ... \wedge \sim query(p,t) \wedge \; ... \;\Rightarrow\; ...)$

to be relevant for any update, and to simplify the above constraint by substituting the current time for the all-quantified temporal variable. Note that the constraint (8') can now be evaluated for time 450. Consequently, query-, assert-, and retract-predicates that contain all-quantified temporal variables have to be evaluated for the current point of time, possibly detecting violations of constraints that originally referred to the future.

For example, 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". $\sim query(p,450) \;\Rightarrow\; 450 < 450$

8". $query(p,450)$

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

$\forall t_1 \forall t_2 \forall t_3 \; P(t_1,t_2,t_3)$

in which t_1, t_2, t_3 are temporal variables and P is an arbitrary formula with free variables t_1, t_2, t_3 is given, and it is relevant at the current point of time 450; then the following constraints have to be processed in step (2) by the algorithm:

$\forall t_2 \forall t_3 P(450,t_2,t_3)$
$\forall t_1 \forall t_3 P(t_1,450,t_3)$
$\forall t_1 \forall t_2 P(t_1,t_2,450)$
$\forall t_1 P(t_1,450,450)$
$\forall t_2 P(450,t_2,450)$
$\forall t_3 P(450,450,t_3)$
$P(450,450,450)$

In general, things become quite complicated when multiple temporal variables are involved in a constraint.

There are various ways the previously discussed algorithm can be applied in practice. 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 application programs by code that enforces consistency constraints. If the application program contains code that asserts `a(b,c,d)`, then the precompiler would add code that enforces the simplified constraint (3') to the application program, but not any code that enforces any other dynamic constraints. This property is very important for the efficiency of any automatic constraint enforcement algorithm: only those constraints should be checked that potentially can be violated by a particular update; otherwise, the enforcement system would be highly inefficient. For example, it is a waste of time to check the uniqueness of social security numbers in the case that a social security number is deleted — it is impossible to violate this constraint by a deletion.

4 Conclusions

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 for the enforcement of static consistency constraints [10] 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')). Which subclasses of dynamic constraints can be enforced by conventional database management systems that only store the current state? For example, "salaries do not decrease" is a constraint that can be enforced without having to refer to the history of the database, as discussed in the paper. Is it possible to recognize this important subclass of dynamic constraints syntactically?

References

[1] Eick C.F., Werstein P.: "Rule-Based Consistency Enforcement for Knowledge-Based Systems", accepted for publication in IEEE Transactions on Knowledge and Data Engineering, to appear end of 1992.

[2] 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.

[3] 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.

[4] Jaffar J., Lassez J.L.: "Constraint Logic Programming" in Proc. 14th POPL-Conference, Munich, 1987, pp. 111-119.

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

[6] Kowalski R., Sadri F., Soper P.: "Integrity Checking in Deductive Databases", in Proc. VLDB Conf., Brighton, 1987, pp. 61-69.

[7] Lloyd J.W.: "Foundations of Logic Programming", Springer Verlag, Second Edition, 1987.

[8] 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.

[9] Navathe S.B., Ahmed R.: "A Temporal Relational Model and a Query Language", Information Sciences 49, 1989, pp. 147-175.

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

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

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

[13] 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.

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