SPECIFYING ADMISSIBILITY OF DYNAMIC DATABASE BEHAVIOUR USING TEMPORAL LOGIC

U.W.Lipeck, H.-D.Ehrich, M.Gogolla

Institut fur Informatik
Technische Universität Braunschweig
Postfach 3329, D-3300 Braunschweig
Fed. Rep. of Germany

This work uses temporal logic as a calculus for expressing integrity constraints that specify admissibility of dynamic database behaviour. Formulas are interpreted in state sequences representing dynamic behaviour. Our approach incorporates temporal quantification by "always", "sometime", and quantifiers bounded by intervals in state sequences. Moreover, dynamically changing domains of database values are considered. We then use special kinds of formulas as a language for dynamic constraints and give some hints how to specify in typical situations. For such formulas, a frame for monitoring constraints during runtime of a database is discussed which allows to characterize admissibility operationally.

1. INTRODUCTION

Many approaches to database design are suggesting a "conceptual"(or "logical") design step where a global information structure,the conceptual schema, should be defined independently of any implementation of a data model [5,21]. This step corresponds to the specification phase of general software design. Like that it has to bridge the gap between requirement analysis and implementation in a specific database management system. Therefore, a specification language is needed to describe the contents and the behaviour of a database in an abstract and exact way.

Specifying a conceptual schema essentially means defining the properties of database objects in the course of time. A database develops dynamically according to user transactions manipulating its contents, i.e. taking the database from one state into another. Thus the dynamic behaviour of a database can be represented by sequences of states. If a sequence obeys certain integrity constraints on database objects, it becomes "admissible". Such rules may refer to each single state, but generally they refer to several states at once or to entire intervals in the state sequence. Together with the syntactic description of database objects, the constraints make up the schema specification.

Since Hammer/McLeod [11] and Stonebraker [20] (both 1975) there have been numerous papers dealing with integrity constraints for databases. Most of them, however, are only concerned with "static" constraints, i.e. criteria for admissible states, disregarding dynamic contexts. The problem of "dynamic" constraints specifying admissible state sequences has rarely been addressed. Some preliminary ideas are contained in the ISO-report [12]; [15,16] consider pairs of before/after states.

Except for designing and implementing a database, a schema specification serves as a basis for monitoring dynamic behaviour, i.e. for checking admissibility of a state sequence during runtime. The problem of enforcing (static) integrity constraints [20,22] has recently found growing interest [6,14,23].

A new approach to dynamic constraints using temporal logic [13,17] has been initiated by Sernadas [18], Casanova et al.[3,4] and Golshani et al.[10]. We take up these ideas here, because temporal logic is, in our opinion, a convenient tool for modelling time-dependent database aspects (cf.[1]). By concentrating on the problem of specifying dynamic constraints without considering the problem of specifying and verifying transactions ([4],[24]), we restrict ourselves to a simpler form of temporal logic incorporating only the temporal formulas "always" and "sometime" as well as their bounded versions "always... until" and "sometime...before". We,however,generalize the work mentioned above since we are able to handle dynamically changing domains; we distinguish between "possible" and "actual" values by providing corresponding kinds of $\forall,\exists$ -quantifiers.

Such a logic calculus for formulating constraints must be based on specific concepts for modelling objects. Like Golshani et al.[10] we adopt a functional approach in the spirit of Buneman/Frankel [2] and Shipman [19]. Accordingly, a schema consists of sorts of entities, functions taking arguments and delivering results of specified sorts, and constraints expressed in temporal logic. It should be noted, however, that temporal logic is not bound to this approach to object modelling; it can be used with other approaches as well, e.g. relational ones as in [4].

In contrast to previous functional approaches, we introduce two syntactic levels, the data level and the object level. The data level comprises specifications of basic data types like BOOL,INT,etc. The data level has a fixed interpretation that does not vary in time and is often the same for large classes of database schemata. The object level, on the other side, contains sorts like Person,Project,etc. and functions on them whose interpretation varies in time, dependent on the database state. The object level will most probably be different for different database schemata.

In the next section, we present our version of temporal logic, based on our approach to object modelling. We define the syntax and semantics of temporal formulas including temporal quantification. The semantics is explained by interpreting formulas in infinite state sequences.

In section 3, conceptual schema specifications are restricted to special kinds of formulas which probably are sufficient to express dynamic database constraints in most applications. We use a suggestive syntax for constraints refering to intervals in a state sequence and we give some hints how to build constraints depending on the existence of objects. As an example, we specify parts of a database for an automobile market (cf. [12]).

Section 4 discusses principles for monitoring dynamic constraints during runtime of a database. For the constraint language chosen, we obtain an operational characterization of admissibility. This makes clear on which conditions a current state sequence will become admissible in the future, if it has already been accepted up to a present state.

Some preliminary ideas and results along these lines have been given in our previous paper [8].

2. TEMPORAL LOGIC

Temporal logic can be based on different approaches to object modelling, e.g. relational [4] or functional [10] ones. Here, we adopt the functional approach as described in [2,19], since it conforms in spirit with the well developed methodology and theory of data type specifications, especially the algebraic approach [7,9], and also has strong relationship with Chen's ER-model.

A conceptual schema specification is given in two levels, the data level and the object level.

On the data level, a <u>data specification</u> D-SPEC = $< S_D , \Omega_D , E_D >$ consists of data sorts S_D, data functions Ω_D between data sorts, and axioms E_D (most often equations) defining the behaviour of the data functions. The "syntactic" part of D-SPEC is called the <u>data signature</u> $\Sigma_D = < S_D , \Omega_D >$. For the purpose of this paper, we assume that the data specification is fixed, specifying standard data types like BOOL, INT, CHAR, TEXT, DATE etc.

The object level extends the data specification by giving an <u>object specification</u> O-SPEC = $< <S_O, \leq> , \Phi_O , C_O >$, consisting of a partially ordered set of object sorts $<S_O, \leq>$, object functions Φ_O between object and data sorts, and constraints C_O determining admissible states and state sequences. The partial ordering $\leq$ on S_O represents a generalization relationship between object sorts. The "syntactic" part of O-SPEC is called the <u>object signature</u> $\Sigma_O = < <S_O, \leq>, \Phi_O >$.

A <u>conceptual schema specification</u> CS-SPEC = D-SPEC + O-SPEC consists of a data specification D-SPEC and an object specification O-SPEC based upon D-SPEC. The syntactic part $\Sigma_{CS} = \Sigma_D + \Sigma_O$ is called a <u>conceptual schema signature</u>. In what follows, we will concentrate on O-SPEC assuming D-SPEC to be fixed. In order to illustrate the syntactic part of an object specification, we give an example. We make use of a <u>set</u> constructor as a parameterized object type: for each sort $s \in S_O$, "<u>set</u> s" denotes a new sort (of sets of objects of sort s), on which the predicate ϵ : s x <u>set</u> s --> BOOL for set membership is defined.

<u>Example:</u> We give part of the object signature of a simple automobile market database.

S_O : Manufacturer, Garage, Person, Car_Owner, Car, Model

$\leq$:
```
Manufacturer  ≤  Car_Owner
Garage        ≤  Car_Owner
Person        ≤  Car_Owner
```

Φ_O :
```
name      : Car_Owner --> TEXT
property  : Car_Owner --> set Car

models    : Manufacturer --> set Model

serialno  : Car --> INT
model     : Car --> Model
manuf     : Car --> Manufacturer
owner     : Car --> set Car_Owner
```

***

In order to complete this object signature up to an object specification, constraints C_O have to be added. Following [3,4,10,18] we propose temporal logic as an appropriate language for specifying static and dynamic database constraints. To introduce our version of temporal logic we first discuss the semantics, i.e. the interpretation, of object signatures.

Let $\Sigma = \Sigma_D + \Sigma_O$ be a conceptual schema signature. A <u>Σ-universe</u> is a mapping π

associating a set $\pi(s)$ of possible values to each object sort $s \in S_0$. A Σ-instance or Σ-state σ wrt π is a mapping associating a set $\sigma(s)$ of actual values to each object sort $s \in S_0$ and associating an actual function $\sigma(f)$: $\sigma(s_1) \times \ldots \times \sigma(s_n) \dashrightarrow \sigma(s_0)$ to each object function $f: s_1 \ldots s_n \dashrightarrow s_0$. These mappings are subject to the following conditions:

(i) $\forall s \in S_0 : \sigma(s) \subseteq \pi(s)$

(ii) $\forall s,t \in S_0 : s \leq t \Longrightarrow \pi(s) \subseteq \pi(t)$

(iii) $\forall s,t \in S_0 : s \leq t \Longrightarrow \sigma(s) \subseteq \sigma(t)$

For data sorts $s \in S_D$, we assume a fixed interpretation where all possible values are actual, i.e. $\sigma(s)=\pi(s)$. Moreover, for the set constructor, we have

(iv) $\forall s \in S_0 : \pi(\underline{set}\ s) = 2^{\pi(s)}$

(v) $\forall s \in S_0 : \sigma(\underline{set}\ s) \subseteq 2^{\sigma(s)}$

Let $X=\{x_1,x_2,\ldots\}$ be a set of variables and let each variable $x \in X$ have a sort $sort(x) \in S=S_D+S_0$. We assume that there is a countably infinite number of variables of each sort. We define the language constructs of temporal logic along with their semantics. Terms and atomic formulas are interpreted in a given state σ and a given local substitution α for a set $Y \subseteq X$ of variables, i.e. a mapping $\alpha: Y \dashrightarrow \bigcup_{s \in S} \sigma(s)$ such that, for each $x \in Y$, $\alpha(x) \in \sigma(sort(x))$. Formulas are interpreted in a given state sequence $\underline{\sigma} = (\sigma_0\ \sigma_1 \ldots)$ and a given global substitution $\underline{\alpha}$ defined like a local substitution with σ replaced by π and where all states are based on a fixed given universe π. We use the following notations: $[\sigma,\alpha](t)$ denotes the value of a term t in a state σ with a local substitution α; $[\underline{\sigma},\underline{\alpha}] \models \phi$ denotes the validity of a temporal formula ϕ in a state sequence $\underline{\sigma}$ with a global substitution $\underline{\alpha}$.

(1) Terms

(1.1) Syntax: Each variable $x \in X$ is a term of sort $sort(x)$.
Semantics: $[\sigma,\alpha](x) = \alpha(x)$

(1.2) Syntax: If $f: s_1 \ldots s_n \dashrightarrow s_0 \in \Phi_0$, and t_i is a term of sort s_i for $i=1,\ldots,n$, then $f(t_1,\ldots,t_n)$ is a term of sort s_0.
Semantics: $[\sigma,\alpha](f(t_1,\ldots,t_n)) = \sigma(f)([\sigma,\alpha](t_1),\ldots,[\sigma,\alpha](t_n))$

Remark: As a special case, constants c of sort s can be viewed as nullary functions $c: \dashrightarrow s$, and their semantics is an element $\sigma(c) \in \sigma(s)$.

(2) Atomic Formulas

(2.1) Syntax: Each boolean term t (i.e. sort(t)=BOOL) is an atomic formula.
Semantics: $[\sigma,\alpha] \models t$ iff $[\sigma,\alpha](t)=true$

(2.2) Syntax: If t,u are terms of the same sort, $t=u$ is an atomic formula.
Semantics: $[\sigma,\alpha] \models t=u$ iff $[\sigma,\alpha](t)=[\sigma,\alpha](u)$

(2.3) Syntax: If t is a term and $s \in S_0$ such that $s \leq \mathrm{sort}(t)$, then t is s is an atomic formula.

Semantics: $[\sigma,\alpha] \models t$ is s iff $[\sigma,\alpha](t) \in \sigma(s)$

(3) Formulas

(3.1) Atomic formulas

Syntax: Each atomic formula ϕ is a formula. Each occurrence of a variable in ϕ is free.

Semantics: $[\underline{\sigma},\underline{\alpha}] \models \phi$ iff $\underline{\alpha}$ is local in σ_0 and $[\sigma_0,\underline{\alpha}] \models \phi$.

Here, $\underline{\alpha}$ is local in σ_0 iff $\underline{\alpha}(x) \in \sigma_0(\mathrm{sort}(x))$ for each x occurring in ϕ, i.e. all objects substituted by $\underline{\alpha}$ for occurring variables exist in the state σ_0.

(3.2) Boolean connectives

Syntax: If ϕ, ψ are formulas, then so are $\neg\phi$, $\phi \wedge \psi$, etc. All free occurrences of variables in ϕ or ψ remain free in these formulas.

Semantics: $[\underline{\sigma},\underline{\alpha}] \models \neg\phi$ iff not $[\underline{\sigma},\underline{\alpha}] \models \phi$,

$[\underline{\sigma},\underline{\alpha}] \models \phi \wedge \psi$ iff $[\underline{\sigma},\underline{\alpha}] \models \phi$ and $[\underline{\sigma},\underline{\alpha}] \models \psi$, etc.

(3.3) Quantification over actual values

Syntax: If ϕ is a formula and $x \in X$ is a variable, then $(\forall x)\,\phi$ and $(\exists x)\,\phi$ are formulas. Each free occurrence of x in ϕ is no longer free in these formulas, whereas free occurrences of other variables remain free.

Semantics: $[\underline{\sigma},\underline{\alpha}] \models (\forall x)\,\phi$ iff for all $v \in \sigma_0(\mathrm{sort}(x))$, we have $[\underline{\sigma}, \underline{\alpha}\langle x \leftarrow v\rangle] \models \phi$. Here, $\underline{\alpha}\langle x \leftarrow v\rangle$ denotes $\underline{\alpha}$ changed for the argument x to yield the value v. Similarly, $[\underline{\sigma},\underline{\alpha}] \models (\exists x)\,\phi$ iff there is a $v \in \sigma_0(\mathrm{sort}(x))$ such that $[\underline{\sigma}, \underline{\alpha}\langle x \leftarrow v\rangle] \models \phi$.

(3.4) Quantification over possible values

Syntax: Like (3.3), with $\underline{\forall}$ and $\underline{\exists}$ instead of $\forall$ and $\exists$.

Semantics: Like (3.3), with π instead of σ_0 .

(3.5) Temporal quantification

Syntax: If ϕ is a formula, then always ϕ and sometime ϕ are formulas. Free occurrences of variables in ϕ remain free in these formulas.

Semantics: $[\underline{\sigma},\underline{\alpha}] \models$ always ϕ iff for all $i \geq 0$, $[\underline{\sigma}^i,\underline{\alpha}] \models \phi$.

Here, $\underline{\sigma}^i = (\sigma_i \; \sigma_{i+1} \; \ldots)$ denotes the tail of $\underline{\sigma}$ from the position i.

Similarly, $[\underline{\sigma},\underline{\alpha}] \models$ sometime ϕ iff there is an $i \geq 0$ such that $[\underline{\sigma}^i,\underline{\alpha}] \models \phi$.

Remark: We will also talk about interpretation in finite state sequences $(\sigma_0 \ldots \sigma_n)$. By this we mean the interpretation in the infinite state sequence $(\sigma_0 \ldots \sigma_n \; \sigma_n \; \sigma_n \ldots)$ remaining constant from position n on. Accordingly, by an interpretation of a temporal formula in a single state σ we mean the interpretation in the state sequence $(\sigma \; \sigma \; \sigma \ldots)$.

The well known duality of universal and existential quantification under negation, $\neg(\forall x)\,\phi \iff (\exists x)\neg\phi$, holds for both forms of quantification introduced here, over actual and possible values. Moreover, a corresponding duality principle holds for temporal quantification, too:

$\neg$ <u>always</u> ϕ <==> <u>sometime</u> $\neg\phi$ or, equivalently,

$\neg$ <u>sometime</u> ϕ <==> <u>always</u> $\neg\phi$.

The language of temporal logic gives an appropriate means to specifying database constraints; for an example see section 3. In many applications, however, situations have to be expressed where a temporal quantifier is not required to hold indefinitely, but is bounded in time by some other condition. Typically such conditions can be represented by the first state in a sequence where a given formula τ becomes true. In order to express such situations conveniently, two additional <u>bounded temporal quantifiers</u> are introduced :

<u>always</u> ϕ <u>until</u> τ

<u>sometime</u> ϕ <u>before</u> τ

Here, ϕ and τ are arbitrary temporal formulas. The semantics of bounded temporal quantification in a state sequence $\underline{\sigma}$ and a global substitution $\underline{\alpha}$ is as follows. Let $m = \min(\{\, j \geqslant 0 \mid [\underline{\sigma}^j, \underline{\alpha}] \models \phi \,\} + \{\infty\})$.

$[\underline{\sigma}, \underline{\alpha}] \models$ <u>always</u> ϕ <u>until</u> τ iff for all $i, 0 \leqslant i < m$, we have $[\underline{\sigma}^i, \underline{\alpha}] \models \phi$.

$[\underline{\sigma}, \underline{\alpha}] \models$ <u>sometime</u> ϕ <u>before</u> τ iff there exists an $i, 0 \leqslant i < m$, such that $[\underline{\sigma}^i, \underline{\alpha}] \models \phi$.

These definitions do not imply that τ ever becomes true. If wanted, this must be specified additionally by <u>sometime</u> τ . The bounded quantifiers again behave dually :

$\neg$ <u>always</u> ϕ <u>until</u> τ <==> <u>sometime</u> $\neg\phi$ <u>before</u> τ

$\neg$ <u>sometime</u> ϕ <u>before</u> τ <==> <u>always</u> $\neg\phi$ <u>until</u> τ

3. CONCEPTUAL SCHEMA SPECIFICATION

For most database applications, we feel it is sufficient to restrict the general form of temporal formulas given in the previous section to a simpler one. Additionally, we are interested to have constraint formulas that can be monitored during database operation; this aspect will be discussed in section 4.

Therefore, from now on we only consider formulas of the form

$\underline{\forall x_1} \ldots \underline{\forall x_n}$ <u>always</u> (ϕ ==> <u>always</u> ψ <u>until</u> τ) and

$\underline{\forall x_1} \ldots \underline{\forall x_n}$ <u>always</u> (ϕ ==> <u>sometime</u> ψ <u>before</u> τ)

where $x_1, \ldots, x_n$ are the free variables of the bracketed parts and ϕ, ψ and τ are nontemporal(!) formulas. Furthermore we will drop the first <u>always</u> and all $\underline{\forall}$-quantifications and use a more suggestive syntax:

<u>from</u> ϕ <u>always</u> ψ <u>until</u> τ

<u>after</u> ϕ <u>sometime</u> ψ <u>before</u> τ

Formulas involving "<u>always</u>" are called <u>universal constraints</u>, whereas for formulas involving "<u>sometime</u>" we use the notion <u>existential constraint</u>. A state sequence $\underline{\sigma}$ (finite or infinite) is called <u>admissible</u> wrt a set C of constraints, iff each $\gamma \in C$ (in its original form) is valid in $\underline{\sigma}$ with an arbitrary substitution. In this case we will also say that the state sequence $\underline{\sigma}$ satisfies the constraints C.

The from/after- and until/before-bounds may be left out liberately and then the formulas

always ϕ until τ
from ϕ always ψ
always ϕ

are understood to be semantically equivalent to:

from true always ψ until τ
from ϕ always ψ until false
from true always ψ until false

Of course the same applies to existential constraints by analogy.

Considering the time-variant nature of our constraints, it should be noted that the from/after-bounds establish a "less-equal" relation in time, whereas the until/before-bounds express a "less" relation in time.

In our approach it is possible to generally specify the existence of objects (of sort s) as a boolean function

exists: s --> BOOL

by giving the (defining) constraint

$\forall$ x : exists(x)

with x variable of sort s. Please note that we have to use the quantification $\forall$ over actual values (and not $\underline{\forall}$ over possible values). Thus exists(x) will be true exactly for those objects which are existent in the present state of the database. For the rest of the paper we assume "exists" predicates for all object sorts.

Example: We give a few constraints for our automobile market database whose signature has been specified in section 2 and is now enlarged by the following boolean object functions:

registered , deregistered : Car --> BOOL

The variables used here have the following sorts:

c,c′: Car ; co,co′,co": Car_Owner ; m: Model ;
mf: Manufacturer ; ms: set Model

(A) after exists(c) sometime registered(c) before deregistered(c)

(B) from registered(c) always registered(c) until deregistered(c)

(C) from deregistered(c) always deregistered(c) until ¬ exists(c)

(D) after c ∈ property(co) ∧ co is Manufacturer
sometime ∃ co′: c ∈ property(co′) ∧ co′ is Garage
before ∃ co": c ∈ property(co") ∧ co" is Person

(E) from c ∈ property(co) ∧ ¬ (co is Manufacturer)
always $\forall$ co′: c ∈ property(co′) ==> ¬ (co′ is Manufacturer)

(F) sometime ¬ exists(m)

(G) after ms=models(mf)
sometime exists(m) ∧ m ∈ models(mf) ∧ ¬ m ∈ ms

(H) $c \in \text{property}(co) \iff co \in \text{owner}(c)$

(I) $m=\text{model}(c) \implies m \in \text{models}(\text{manuf}(c))$

(J) $\text{serialno}(c)=\text{serialno}(c') \implies c=c'$

Constraints (A)-(C) require that, if a car exists in the database, it sometime must be registered and then will always be registered until it is deregistered. Once it is deregistered, it cannot be registered again. (D) and (E) express that cars are sold from manufacturers to garages and after this cannot be owned by manufacturers again. (F) and (G) demand that "old" models will disappear from the market and "new" models have to be produced. Constraints (H)-(J) refer to each single state of a given state sequence since they do not contain temporal quantifiers. Such constraints are called <u>static</u>. (H) for example expresses that the functions "property" and "owner" contain the information. The first constraints (A)-(G), in contrast, are called <u>dynamic</u>, since they only make sense when considering state sequences.

***

Concerning the expressive power of our constraint language, we can specify any nontemporal condition on database objects to hold in all or some states of a certain interval in a state sequence. Such an interval is bounded by two other nontemporal conditions on objects. If one wants to express conditions involving calendar dates, durations, deadlines etc. (i.e. real world time), appropriate objects have to be modelled by including additional functions into the signature. E.g., by using a function today: --> DATE and a variable d of sort DATE, the constraint

<u>after</u> $d=\text{today} \wedge \text{exists}(c)$ <u>sometime</u> registered(c) <u>before</u> today>d+90

says that a car should be registered within the first 90 days of its lifetime.

The exists predicates may be used to specify certain dependencies between the validity of constraints and the existence of objects involved therein. Let us consider some typical situations :

On the one hand, some formulas, e.g. a disjunction like "registered(c) $\vee$ registered(c')", may become true, even if an object substituted for one variable does not exist. The existence of an object can be required explicitly by a constraint of the form

<u>from</u> ϕ <u>always</u> ($\psi \wedge \text{exists}(x)$) <u>until</u> τ

where x is a variable which (probably) appears in ϕ and ψ.

On the other hand, some formulas, e.g. a conjunction of atomic formulas like "registered(c) $\wedge$ registered(c')", can only become true, if all objects involved do exist. But in general, objects may disappear from the database and perhaps sometime later re-appear. So there may be different life cycles or incarnations for one object. To restrict a constraint to the first incarnation of an object, the until (before) clause has to be modified:

<u>from</u> ϕ <u>always</u> ψ <u>until</u> ($\tau \vee \neg \text{exists}(x)$)

Another possibility is to demand the validity of a constraint exactly for all periods when an object is present in the database :

<u>from</u> ϕ <u>always</u> ($\text{exists}(x) \implies \psi$) <u>until</u> τ

4. OPERATIONAL CHARACTERIZATION OF ADMISSIBILITY

After we have introduced constructs for formulating a conceptual schema specification, we will now study principles of monitoring the specified constraints during runtime of a database. Although implementation aspects are not discussed, a framework for enforcing dynamic constraints will emerge.

Since dynamic constraints refer to sequences of database states or at least to intervals in sequences, the problem arises how to check relationships between past, present and future states in a current sequence. At each moment the present state has been reached from an initial state by performing some user transactions which induce corresponding state transitions. Whereas those past states are known, the future states cannot be predicted, if users may choose their transactions arbitrarily. Thus, any general monitoring can only deal with the database history up to the present state and check that sequence prefix for constraint violations.

But to avoid storing the entire history, the knowledge about it should be limited to a part necessary for checking any continuation. If, e.g., the start condition ϕ of a universal constraint "<u>from</u> ϕ <u>always</u> ψ <u>until</u> τ" became true in some past state for some (global) substitution $\underline{\alpha}$, then ψ has to be valid in that and all subsequent states until the termination condition τ occurs; so the information $\langle\psi,\tau,\underline{\alpha}\rangle$ meaning "(<u>always</u> ψ <u>until</u> τ) with substitution $\underline{\alpha}$" should be remembered for exactly that period. The key idea of monitoring is to maintain two dynamically changing sets of such notes, one set induced by universal constraints and the other one by existential constraints. These sets will be denoted as C_u and C_e, respectively.

In the following, we present and analyse a program called DB-MONITOR that

(i) executes user transactions (which are left unspecified here)
(ii) and checks the induced state sequence for constraint violations in the actual prefix.

Then, admissible state sequences can be characterized operationally, i.e. in terms of that monitor, what can be shown by verifying some program invariants.

Let a conceptual schema specification CS-SPEC be given with a set C of constraints. The frame of the program DB-MONITOR is given below. As global variables it uses σ for the actual database state, $\mathscr{a}$ for the actual user transaction, and two sets C_u, C_e consisting of triples $\langle\psi,\tau,\underline{\alpha}\rangle$ s.t. ψ,τ are nontemporal formulas and $\underline{\alpha}$ is a substitution. The procedure CHECK will be defined afterwards.

```
program DB-MONITOR :
   procedure EXECUTE;
      ( executes 𝒶 and changes σ accordingly );
   procedure CHECK;
      ...
   begin  ( initialize σ );
          CHECK;
          while true do
          begin ( input 𝒶 );
                EXECUTE; CHECK
          end
   end.
```

The following procedure CHECK manipulates the set variables C_u and C_e; let INSERT and DELETE denote the corresponding set operations. Moreover, a boolean function VALID(ϕ,$\underline{\alpha}$) computing ($[\sigma,\underline{\alpha}] \models \phi$) and a procedure ERROR that handles constraint violations, e.g. by undoing the last transaction, are used.

```
procedure CHECK :
   var φ,ψ,τ: nontemporal formula;  α: substitution;
   begin
(1)      for each "from φ always ψ until τ" ∈ C  do
            for each α do
               if VALID(φ,α) then INSERT(C_u, <ψ,τ,α> );
(2)      for each <ψ,τ,α> ∈ C_u  do
            if VALID(τ,α) then DELETE(C_u, <ψ,τ,α> )
            else if not VALID(ψ,α) then ERROR;
(3)      for each "after φ sometime ψ before τ" ∈ C  do
            for each α do
               if VALID(φ,α) then INSERT(C_e, <ψ,τ,α> );
(4)      for each <ψ,τ,α> ∈ C_e  do
            if VALID(τ,α) then ERROR
            else if VALID(ψ,α) then DELETE(C_e, <ψ,τ,α> );
   end CHECK.
```

It should be noted that the sequence of actions is essential within the two loop pairs (1);(2) and (3);(4), whereas these sections may be executed in parallel. The reader should verify that especially the critical situations, when start and termination conditions of a constraint coincide, are handled in accordance with former definitions.

The following theorem confirms that maintaining the sets C_u and C_e as above suffices to guarantee admissibility of a state sequence, if the present prefix has been accepted and if the (future) rest will satisfy the constraints in C, C_u and C_e.

We only consider state sequences ($\sigma_0\ \sigma_1$ ...) s.t. there are transactions inducing state transitions $\sigma_i \rightarrow \sigma_{i+1}$ (via the procedure EXECUTE). We say that, for $n \geq 0$, DB-MONITOR <u>accepts</u> (σ_0 ... σ_n) iff $\sigma_0,\ldots,\sigma_n$ are the states generated during the first n runs of DB-MONITOR taking appropriate transactions as input and not calling ERROR. Let C_u^n, C_e^n denote the contents of the variables C_u, C_e after the n-th execution of the while-loop in DB-MONITOR.

<u>Theorem:</u> Let $n \geq 0$ be given. The state sequence $\underline{\sigma}$ = ($\sigma_0\ \sigma_1$ ...) is admissible iff the conditions (i)-(iv) hold:

(i) DB-MONITOR accepts ($\sigma_0 \cdots \sigma_n$)

(ii) $\underline{\sigma}^{n+1}$ is admissible (i.e. $\underline{\sigma}^{n+1}$ satisfies C)

(iii) $[\underline{\sigma}^{n+1},\underline{\alpha}] \models$ (<u>always</u> ψ <u>until</u> τ) for each $\langle\psi,\tau,\underline{\alpha}\rangle \in C_u^n$

(iv) $[\underline{\sigma}^{n+1},\underline{\alpha}] \models$ (<u>sometime</u> ψ <u>before</u> τ) for each $\langle\psi,\tau,\underline{\alpha}\rangle \in C_e^n$

Thus, in each present state σ_n successfully reached by DB-MONITOR we know that the current state sequence is admissible, if we optimistically assume every constraint to be satisfiable in the future ($\underline{\sigma}^{n+1}$). In reality, there may be combinations of C, C_u and C_e excluding any continuation to an admissible state sequence, so that the monitor should be enhanced by some pre-computations. We will not follow this possibility here, but we have already considered a typical situation in [8].

The above theorem can be concluded from some "invariants" of the program DB-MONITOR.

<u>Lemma:</u> For all n$\geqslant$0, the following properties hold after the n-th execution of the while-loop:

(i) $\forall$ "<u>from</u> ϕ <u>always</u> ψ <u>until</u> τ" $\in C$ $\forall\underline{\alpha}$ $\forall j, j \leqslant n$:
$[\sigma_j,\underline{\alpha}] \models \phi \implies U_1(n,j,\psi,\tau,\underline{\alpha}) \vee U_2(n,j,\psi,\tau,\underline{\alpha})$
where $U_1(n,j,\psi,\tau,\underline{\alpha}) = (\exists\, l, j\leqslant l\leqslant n\colon [\sigma_l,\underline{\alpha}] \models \tau \wedge \forall\, k, j\leqslant k< l\colon [\sigma_k,\underline{\alpha}] \models \psi)$
and $U_2(n,j,\psi,\tau,\underline{\alpha}) = (\nexists\, l, j\leqslant l\leqslant n\colon [\sigma_l,\underline{\alpha}] \models \tau \wedge \forall\, k, j\leqslant k\leqslant n\colon [\sigma_k,\underline{\alpha}] \models \psi)$

(ii) $\forall\psi,\tau$ $\forall\underline{\alpha}$:
$\langle\psi,\tau,\alpha\rangle \in C_u^n \iff \exists\phi\ \exists j, j\leqslant n$: "<u>from</u> ϕ <u>always</u> ψ <u>until</u> τ" $\in C$
$\wedge\ [\sigma_j,\underline{\alpha}] \models \phi \wedge U_2(n,j,\psi,\tau,\underline{\alpha})$

(iii) $\forall$ "<u>after</u> ϕ <u>sometime</u> ψ <u>before</u> τ" $\in C$ $\forall\underline{\alpha}$ $\forall j, j\leqslant n$:
$[\sigma_j,\underline{\alpha}] \models \phi \implies E_1(n,j,\psi,\tau,\underline{\alpha}) \vee E_2(n,j,\psi,\tau,\underline{\alpha})$
where $E_1(n,j,\psi,\tau,\underline{\alpha}) = (\exists\, k, j\leqslant k\leqslant n\colon [\sigma_k,\underline{\alpha}] \models \psi \wedge \nexists\, l, j\leqslant l\leqslant k\colon [\sigma_l,\underline{\alpha}] \models \tau)$
and $E_2(n,j,\psi,\tau,\underline{\alpha}) = (\nexists\, k, j\leqslant k\leqslant n\colon [\sigma_k,\underline{\alpha}] \models \psi \wedge \nexists\, l, j\leqslant l\leqslant n\colon [\sigma_l,\underline{\alpha}] \models \tau)$

(iv) $\forall\psi,\tau$ $\forall\underline{\alpha}$:
$\langle\psi,\tau,\underline{\alpha}\rangle \in C_e^n \iff \exists\phi\ \exists j, j\leqslant n$: "<u>after</u> ϕ <u>sometime</u> ψ <u>before</u> τ" $\in C$
$\wedge\ [\sigma_j,\underline{\alpha}] \models \phi \wedge E_2(n,j,\psi,\tau,\underline{\alpha})$

Properties (i) and (iii) describe to what extent constraints have been satisfied up to the actual σ; (ii) and (iv) characterize C_u and C_e to consist of constraints that have been only "partially" satisfied (predicates U_2,E_2). The lemma can be proved using induction on n. Additionally, (i)$\wedge$(iii) is a sufficient condition that ERROR has not been called.

As a corollary to the theorem, we obtain an operational admissibility criterion for finite state sequences.

Corollary: The state sequence $\underline{\sigma} = (\sigma_0 \dots \sigma_n)$ is admissible iff it is accepted by DB-MONITOR s.t. $C_e^n = \emptyset$.

Proof: Since the continuation $\underline{\sigma}^{n+1}$ is defined to be the constant sequence $(\sigma_n\ \sigma_n \dots)$, all constraints in C affecting $\underline{\sigma}^{n+1}$ must have induced corresponding notes in C_u or C_e already during σ_n ; i.e., condition (ii) of the theorem becomes irrelevant. If the prefix is accepted, constraints in C_u remain satisfied and constraints in C_e remain unsatisfied, so that conditions (iii) and (iv) reduce to $C_e^n = \emptyset$. ***

This result does not only apply to cases when the database operation really gets stopped, but it also shows the special role of the set C_e in estimating the "consistency" of a database. C_e contains exactly every condition (ψ) for which a future state must exist s.t. it becomes valid. Only if C_e is empty, the database is in a fully consistent state; otherwise some transactions disposing of those conditions still need to be executed.

5. CONCLUSIONS

We have introduced a logic calculus for specifying dynamic database behaviour not only with a modeltheoretic semantics, but also with an operational semantics. Of course, this paper could only provide principal items of a specification language and method; some further adjustment to practical needs is still missing. There also remains a gap between the monitor frame given in the last section and any monitor implementation. It can partly be filled by existing work on the enforcement of static constraints. Our emphasis is on handling the non-static aspects of constraints which seem to have been neglected before.

On the theoretical side, it would be interesting to look for a monitoring scheme discovering any future constraint violation at the earliest possible moment. Therefore criteria are needed to decide whether a continuation to an admissible state sequence does exist or not; probably some pre-computations or logical deductions checking dynamic constraints for possible contradictions have to be done by a monitor. Especially, the available transactions should be taken into account.

REFERENCES

[1] Bolour,A./Anderson,L./Dekeyser,L./Wong,H.: The Role of Time in Information Processing. SIGMOD Record 12,3 (1982), 27-50

[2] Buneman,P./Frankel,R.E.: FQL - A Functional Query Language. Proc. ACM SIGMOD Int. Conf. on Management of Data 1979, 52-58

[3] Casanova,M.A./Furtado,A.L.: A Family of Temporal Languages for the Description of Transition Constraints. In: Proc. Workshop on Logical Bases for Data Bases, Toulouse 1982

[4] Castilho,J.M.V./Casanova,M.A./Furtado,A.L.: A Temporal Framework for Database Specifications. Proc. 8th Int. Conf. on Very Large Data Bases, Mexico 1982, 280-291

[5] Ceri,S.(ed.): Methodology and Tools for Data Base Design. North-Holland, Amsterdam 1983

[6] Cremers,A.B./Domann,G.: AIM, An Integrity Monitor for the Database System INGRES. Proc. 9th Int. Conf. on Very Large Data Bases (M.Schkolnick/ C.Thanos,eds.), Florence 1983, 167-170

[7] Ehrich,H.-D.: On the Theory of Specification, Implementation, and Parameterization of Abstract Data Types. Journ. ACM 29 (1982), 206-227

[8] Ehrich,H.-D./Lipeck,U.W./Gogolla,M.: Specification, Semantics, and Enforcement of Dynamic Database Constraints. Proc. 10th Int. Conf. on Very Large Data Bases (U.Dayal et al.,eds.), Singapore 1984, 301-308

[9] Goguen,J.A./Thatcher,J.W./Wagner,E.G.: An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In: Current Trends in Programming Methodology, Vol. IV (R.Yeh,ed.), Prentice-Hall, Englewood Cliffs 1978, 80-149

[10] Golshani,F./Maibaum,T.S.E./Sadler,M.R.: A Modal System of Algebras for Database Specification and Query/Update Language Support. Proc. 9th Int. Conf. on Very Large Data Bases, Florence 1983, 331-339

[11] Hammer,M.M./McLeod,D.J.: Semantic Integrity in a Relational Database System. Proc. Int. Conf. on Very Large Data Bases, 1975, 25-47

[12] ISO/TC97/SC5/WG3: Concepts and Terminology for the Conceptual Schema and the Information Base (J.J.van Griethuysen,ed.),ISO/TC97/SC5/WG3-N695,1982

[13] Manna,Z./Pnueli,A.: Verification of Concurrent Programs: The Temporal Framework. In: The Correctness Problem in Computer Science (R.S.Boyer/ J.S.Moore,eds.) Academic Press, London 1981, 215-273

[14] Nicolas,J.M.: Logic for Improving Integrity Checking in Relational Databases. Acta Informatica 18 (1982), 227-253

[15] Nicolas,J.M./Yazdanian,K.: Integrity Checking in Deductive Databases. In: Logic and Databases (H.Gallaire/J.M.Nicolas,eds.), Plenum Press, New York 1978, 325-344

[16] Richter,G.: Utilization of Data Access and Manipulation in Conceptual Schema Definitions. Inform.Syst. 6 (1981), 53-71

[17] Rescher,N./Urquhart,A.: Temporal Logic. Springer, Berlin 1971

[18] Sernadas,A.: Temporal Aspects of Logical Procedure Definition. Inform.Syst. 5 (1980), 167-187

[19] Shipman,D.W.: The Functional Data Model and the Data Language DAPLEX. ACM Trans. on Database Systems 6 (1981), 140-173

[20] Stonebraker,M.: Implementation of Integrity Constraints and Views by Query Modification. Proc. ACM SIGMOD Int. Conf. on Management of Data, San Jose 1975, 65-78

[21] Teorey,T.J./Fry,J.P.: Design of Database Structures. Prentice-Hall, Englewood Cliffs 1982

[22] Todd,S.: Automatic Constraint Maintenance and Updating Defined Relations. Proc. IFIP Congress 77 (B.Gilchrist,ed.), North-Holland, Amsterdam 1977, 145-148

[23] Weber,W./Stucky,W./Karszt,J.: Integrity Checking in Database Systems. Inform.Syst. 8 (1983), 125-136

[24] Kung,C.H.: A Temporal Framework for Database Specification and Verification. Proc. 10th Int.Conf. on Very Large Data Bases, Singapore 1984, 91-99