Deriving Optimized Integrity Monitoring Triggers from Dynamic Integrity Constraints

M. Gertz [1] and U.W. Lipeck

Institut für Informatik, Universität Hannover, Lange Laube 22, D-30159 Hannover, Germany

Modern approaches to integrity monitoring in active databases suggest to generate triggers from constraints as part of database design and to utilize constraint simplification techniques for trigger optimization. Such proposals, however, have been restricted to static conditions only. In this paper, we show how to derive triggers from dynamic integrity constraints which describe properties of state sequences and which can be specified by formulas in temporal logic. Such constraints can equivalently be transformed into transition graphs which describe such life cycles of database objects that are admissible with respect to the constraints: Nodes correspond to situations in life cycles, and edges give the (changing) conditions under which a change into another situation is allowed. If object situations are stored, integrity monitoring triggers can be generated from transition graphs for all situations and all critical database operations. Additionally, new simplification techniques can be developed by identifying characteristic preconditions in the graphs and by utilizing invariants. Maintenance of object situations can be supported by triggers as well.

Key words: Active databases; Constraint simplification; Dynamic integrity constraints; Integrity enforcement; Object life cycles; Temporal formulas; Triggers

1 Introduction

Numerous papers have been written on specifying and monitoring integrity constraints in databases since the first large database conferences [5,14] (for an overview see [6,34]). But the idea to have a universal monitor which can check arbitrary constraints for arbitrary applications has only partially been successful up to now in spite of many proposals on strategies for constraint simplification, following [24,13] and others. They have aimed at restricting integrity enforcement to relevant updates

[1] partially supported by the Volkswagen-Stiftung

and data, i.e. to updates which can violate a constraint, and to data that can be affected by such updates. Yet monitors seem to slow down transaction processing so that no commercial system and only few research prototypes (e.g., [29]) have included *full* integrity subsystems. Nowadays, at least domain, primary key and foreign key constraints are supported: They improve expressiveness of the relational model only to the level of the ER (entity-relationship) model in the sense that equivalent structures can be expressed; foreign keys (referential integrity constraints) are needed in this context as a counterpart to the inherent constraints of relationships. There still remain many kinds of constraints to be expressed in addition to ER diagrams and hence as well in addition to relational database schemes.

Thus, the only way to really guarantee integrity seemed to be the design of integrity preserving transactions [10,30] at schema definition time which, however, would have excluded using ad-hoc transactions at database runtime.

Modern database systems, however, apart from offering further types of table constraints, make another kind of general integrity maintenance realizable: enforcement by triggers. What has been sketched by [4] can now be realized in systems like STARBURST or POSTGRES [12,27] that offer event-condition-action rules or triggers (database procedures activated on the occurrence of certain operations in transactions); commercial systems like DB2, INGRES, ORACLE, or SYBASE also have started to offer related mechanisms though yet in restricted forms. Of course, such instruments should be used carefully and with optimizations only, since in principle complete monitors could be imitated by them (every database operation triggers checks of all constraints and rollbacks in the case of constraint violation). The database designer who knows his/her application has to specialize the original (descriptive) constraints into (operational) rules as few and as local as possible; but (s)he can apply just those systematic simplification techniques mentioned earlier – hopefully supported by computer aided design tools [3,32]. And in addition to generating checks, the designer can utilize triggers even for active corrections of constraint violations, which hardly can be generated automatically (for first attempts in that direction see [23,32,1,28,8]). Thus transformation of integrity constraints into efficient and active triggers has become a design task for relational databases.

In this paper we want to study *dynamic* integrity constraints, i.e. constraints on state sequences instead of single states as in the static case. Based on our former work on monitoring schemes for such dynamic constraints [21,18,25] and on rules for transforming constraints into transaction specifications [18,19], we now want to explain how to use such constraint analysis techniques in the presence of ad-hoc transactions, i.e. arbitrary unforeseen sequences of insert–, update–, and delete–operations, provided that appropriate trigger mechanisms are available.

By using temporal logic with operators like **always**, **sometime**, **before**, etc. rather general dynamic constraints can be expressed, starting with conditions on state transitions and also including long-term relations between database states. We have shown

in [21,17,18] that it is possible to construct from temporal formulas such transition graphs that describe exactly those state sequences which are admissible with regard to the constraints. Thus checks on state sequences can be reduced to tests on state transitions, and monitoring a dynamic constraint is replaced by monitoring changing static conditions. The graphs represent life cycles of database objects with regard to the temporal formulas: Nodes correspond to situations, and edges give the conditions under which a change into another situation is allowed.

If life cycle rules are given in a descriptive manner, e.g., taken from policy rules of an enterprise, it is preferable first to express them as temporal constraints and then to transform them into such graphs which display a stepwise and more operational view. If, in particular, such a transformation is applied to a combination (conjunction) of constraints, the resulting graph displays the corresponding joint stepwise behavior of formerly separate descriptive rules, since the graph construction has already dealt with interdependencies between the constraints. If, however, the transitional behavior is already clear from initial application analysis, it is possible to start directly with graphic representations of life cycles. In contrast to simple (single-step) transitional constraints, transition graphs represent multi-step constraints which require history-dependent and implicit control like state machines do. I.e., maintenance of minimal necessary historical information and control of allowed transitions become intrinsic tasks of monitoring such dynamic integrity constraints. A pure restriction to single-step transitional constraints would prevent direct descriptive or operational specification of multi-step conditions, and it would rely on having the needed historic information already encoded in the database states. Our approach, however, shows how to systematically design a reliable monitoring for rather flexible behavioral constraints.

In any case, there appear new basic problems in designing triggers for monitoring purposes. Even the naive imitation of a monitor does not apply, since it does not make sense to check a dynamic constraint in a single state, i.e. the state after a transaction. Thus it will be necessary to consider the different situations according to the transition graphs together with the corresponding edge conditions, i.e. to adapt the monitoring mechanism to processing transition graphs. That requires to extend the database by tables of current object situations corresponding to the graph nodes together with further historical information needed for monitoring; their maintenance, however, can again be done by triggers without involving the user.

And constraint simplification has to be considered as well. Classic simplification relies on assuming that the constraint is valid before executing a transaction and on checking the same constraint after the transaction. Such techniques utilize that many constraint parts and instances remain invariant under database operations and thus need not be checked explicitly after the transaction. For dynamic constraints and transition graphs, it does not make sense to assume the whole constraint in the state before a transaction, but we will explain how preconditions corresponding to the current situations can easily be identified in transition graphs. For a large class of

constraints, namely those having so-called iteration-invariant transition graphs, invariance of these preconditions again leads to considerable simplifications. But even partial invariants can be utilized, even if the condition to be checked afterwards is different from the condition before.

Although it is well accepted that in particular modelling the behavior of a database system plays an important role in designing real world applications, especially in the field of object-oriented databases [25,33], only a few authors have considered the topic of specifying and enforcing dynamic integrity constraints (not only transitional constraints). Chomicki [2] presented a "history-less" method which is used to monitor dynamic integrity constraints formulated in a past-temporal logic (PTL). Therefore past-temporal operators like, e.g., **always/sometime in the past** and **previous** are used, stating that always/sometime in the past or, respectively, in the previous state a condition must hold. Similar to our approach every database state is augmented with auxiliary relations which are used to check a dynamic constraint in the poststate of a transaction, i.e. checks of dynamic constraints are reduced to checks on static conditions formulated using the auxiliary relations. In his approach these relations are derived from all subformulas of the specified constraints, whereas in our approach only the minimum information needed for monitoring is kept in the object situations corresponding to nodes of transition graphs. Moreover, we only use one auxiliary relation per constraint to represent the current situation of each object. Chomicki's approach has been implemented in the active DBMS Starburst [31] such that triggers are utilized to check these static conditions. Apart from using a different underlying monitoring approach, our trigger derivation considers in particular optimizations and simplifications of triggers.

Further work which use the more intuitive transition graphs (object life cycles) to monitor dynamic constraints has been presented in [26] and [38]. In [26] a method is proposed considering dynamic integrity constraints specified in a past-temporal logic. This approach utilizes reversed transition graphs derived from the underlying constraints and checks temporal permissions as enabling conditions for the occurrences of state changing operations, i.e. transactions. In [38] dynamic constraints are studied which may be formulated in a so-called mixed-temporal propositional logic (MTL) where post- and future-temporal quantifiers may be combined in formulas; monitoring algorithms can be based on so-called bitransition graphs which incorporate and generalize transitions graphs like in our approach.

In this paper we will now show how to systematically generate and even simplify triggers for monitoring dynamic integrity constraints efficiently if we proceed with transition graphs as constraint representations. The relevant notions of dynamic constraints and transition graphs are explained in Section 2, the underlying trigger system and execution assumptions in Section 3. The construction of transition graphs from temporal formulas is briefly sketched in the appendix. Section 4 is the main part where rules for deriving triggers from transition graphs are introduced. In Section 5 the integration of our approach into the database design is discussed. Also some fur-

ther simplifications are sketched to improve efficiency and expressiveness on integrity checking. In Section 6 we conclude with some remarks on the presented approach and sketch our future work on integrity enforcement in active databases.

2 Integrity Constraints and Transition Graphs

In this section we describe the syntax and semantics of dynamic integrity constraints built by applying temporal quantifiers. We then discuss transition graphs which can be constructed from such temporal formulas and how transition graphs are utilized to monitor dynamic integrity constraints at database runtime. For this, we present a general monitor schema.

2.1 Dynamic Integrity

Static integrity constraints are used to describe properties of database states. They restrict the set of possible states to those states which are admissible concerning the constraints. To express static constraints over relational databases we use formulas of first-order predicate logic, which may contain *variables* $x : D$ referring to tuples over a relation scheme D or to values of a datatype D.

In order to specify properties of state sequences, we make use of temporal logic which extends predicate logic by special operators relating states within sequences.

Instead of usual tuple variables, we use variables which refer to primary key values of tuples, i.e. to permanent tuple identities, since we assume that primary keys cannot be updated. Such a variable will be called *object variable* (since it represents an object in the sense of the entity-relationship model). It, however, may be used like a tuple variable in expressions "$x.A$" to denote the current value of attribute A in the tuple with key x.

Definition 1 Temporal formulas *are built from nontemporal formulas, i.e. formulas of classic predicate logic, by iteratively applying logical connectives and*

- *quantifiers over possible objects or data* ($\underline{\forall}, \underline{\exists}$) *and current objects* ($\forall, \exists$)*, i.e. the tuples in a relation at a certain state,*
- *a* **next** *operator referring to the next state,*
- *temporal quantifiers* **always** ..., **sometime** ... *referring to all or some states in the future,*
- *bounded temporal quantifiers* **always/sometime** ... **before/until** ... ,
- *and additional quantifiers* **from/always-from** ... **holds**.
  The precise construction can be seen within Definition 3.

Whereas the **next** operator specifies that a condition must hold only in the next state the temporal quantifiers **always** and **sometime** specify that a condition must hold in all subsequent future states or in some future states, respectively. By means of a **before** or **until** clause a temporal quantified condition can be specified to hold in future states before/until a certain end condition becomes true. The **from**/ **always-from** clauses require a condition to hold starting from the first state/from every state where a start condition becomes true.

Definition 2 *Temporal formulas are interpreted in infinite or finite state sequences*

$$\underline{\sigma} = \langle \sigma_0, \sigma_1, \ldots \rangle \text{ or } \underline{\sigma} = \langle \sigma_0, \ldots, \sigma_{n-1} \rangle, n \geq 0.$$

For $n = 0$, $\underline{\sigma}$ is the empty state sequence to be denoted as $\underline{\lambda}$. $I(\underline{\sigma})$ denotes the index range *of $\underline{\sigma}$, i.e. $I(\langle \sigma_0, \sigma_1, \ldots \rangle) = \mathbb{N}$, and $I(\langle \sigma_0, \ldots, \sigma_{n-1} \rangle) = \{0, \ldots, n-1\}$. $\mid \underline{\sigma} \mid$ defines the* length *of $\underline{\sigma}$, i.e. $\mid \langle \sigma_0, \sigma_1, \ldots \rangle \mid = \infty$, and $\mid \langle \sigma_0, \ldots, \sigma_{n-1} \rangle \mid = n$, for $\underline{\lambda}$ holds $I(\underline{\lambda}) = \emptyset$ and $\mid \underline{\lambda} \mid = 0$. $\underline{\sigma}_i$, $i \in \mathbb{N}$, denotes the* i-th ordered tail sequence *of $\underline{\sigma}$, i.e. $\underline{\sigma}_i = \langle \sigma_i, \sigma_{i+1}, \ldots \rangle$ if $I(\underline{\sigma}) = \mathbb{N}$, and $\underline{\sigma}_i = \langle \sigma_i, \sigma_{i+1}, \ldots, \sigma_j \rangle$ if $max(I(\underline{\sigma})) = j$.*

The semantics of temporal formulas are given in the following definition by induction on the formula structure.

Definition 3 *Let $\varphi, \varphi_1, \varphi_2, \tau, \alpha$ be temporal formulas and ρ an atomic nontemporal formula, i.e.* **true**, **false**, *or an expression $p(t_1, \ldots, t_n)$ with a predicate p and terms t_i. Let $\underline{\sigma}$ be a state sequence, and let θ be a substitution, which maps variables to possible objects or data. To denote that a temporal formula φ is* valid *in a state sequence $\underline{\sigma}$ for a substitution θ of its free variables, we write $[\underline{\sigma}, \theta] \models \varphi$. This holds under the following conditions:*

(0) Nontemporal atoms:

$[\underline{\sigma}, \theta] \models \rho$ — *iff for a nonempty state sequence $\underline{\sigma}$, ρ is valid in the first state σ_0 for the substitution θ (as known from predicate logic). For $\underline{\sigma} = \underline{\lambda}$, $[\underline{\lambda}, \theta] \models \rho$ holds for $\rho \equiv$* **true** *only.* [1]

(1) Propositional connectives:

$[\underline{\sigma}, \theta] \models \neg\varphi$ — *iff not* $[\underline{\sigma}, \theta] \models \varphi$.

$[\underline{\sigma}, \theta] \models \varphi_1 \wedge \varphi_2$ — *iff* $[\underline{\sigma}, \theta] \models \varphi_1$ *and* $[\underline{\sigma}, \theta] \models \varphi_2$.

$[\underline{\sigma}, \theta] \models \varphi_1 \vee \varphi_2$ — *iff* $[\underline{\sigma}, \theta] \models \varphi_1$ *or* $[\underline{\sigma}, \theta] \models \varphi_2$.

(2) Tail operator:

$[\underline{\sigma}, \theta] \models$ **next** φ — *iff* $[\underline{\sigma}_1, \theta] \models \varphi$. *Thus for* $\mid \underline{\sigma} \mid = 0$ *or* $\mid \underline{\sigma} \mid = 1$, $[\underline{\sigma}, \theta] \models$ **next** φ *holds iff* $[\underline{\lambda}, \theta] \models \varphi$.

(3) Unbounded temporal quantification:

(3a) $[\underline{\sigma}, \theta] \models$ **always** φ — *iff for all $i \in I(\underline{\sigma})$,* $[\underline{\sigma}_i, \theta] \models \varphi$. *Thus* **always** φ *is valid in $\underline{\sigma} = \underline{\lambda}$.*

(3b) $[\underline{\sigma}, \theta] \models$ **sometime** φ *iff there exists an* $i \in I(\underline{\sigma})$*, such that* $[\underline{\sigma}_i, \theta] \models \varphi$*. Thus* **sometime** φ *is not valid in* $\underline{\sigma} = \underline{\lambda}$*.*

(4) *Bounded temporal quantification:*
Let $\mu\tau$ *be the index of the first occurrence of* τ*,*
i.e. $\mu\tau = min(\{j \in I(\underline{\sigma}) \mid [\underline{\sigma}_j, \theta] \models \tau\} \cup \{\infty\})$.

(4a) $[\underline{\sigma}, \theta] \models$ **always** φ **before** τ *iff for all* $i \in I(\underline{\sigma})$*:* $i < \mu\tau \;\Rightarrow\; [\underline{\sigma}_i, \theta] \models \varphi$.

(4b) $[\underline{\sigma}, \theta] \models$ **sometime** φ **before** τ *iff there exists an* $i \in I(\underline{\sigma})$*:* $i < \mu\tau$ *and* $[\underline{\sigma}_i, \theta] \models \varphi$.

(4c) $[\underline{\sigma}, \theta] \models$ **always** φ **until** τ *iff for all* $i \in I(\underline{\sigma})$*:* $i \leq \mu\tau \;\Rightarrow\; [\underline{\sigma}_i, \theta] \models \varphi$.

(4d) $[\underline{\sigma}, \theta] \models$ **sometime** φ **until** τ *iff there exists an* $i \in I(\underline{\sigma})$*:* $i \leq \mu\tau$ *and* $[\underline{\sigma}_i, \theta] \models \varphi$

(5) *Additional quantifiers:*

$[\underline{\sigma}, \theta] \models$ **from** α **holds** φ *iff* $\mu\alpha = \infty$ *or* $[\underline{\sigma}_{\mu\alpha}, \theta] \models \varphi$.

$[\underline{\sigma}, \theta] \models$ **always-from** α **holds** φ *iff for all* $i \in I(\underline{\sigma})$ *with* $(i = 0$ *or* $[\underline{\sigma}_{i-1}, \theta] \models \neg\alpha)$ *holds* $[\underline{\sigma}_i, \theta] \models (\alpha \;\Rightarrow\; \varphi)$.

Object quantifications $(\forall, \exists)$ over possible objects (from the universe) and actual objects (from a single state) can be defined in the usual way. The following section works for temporal formulas which may indeed have arbitrary predicate logic formulas (possibly including object quantifiers) as basic constituents, but only propositional or temporal connectives as defined above. The quantifiers (5) are introduced here for convenience only; actually, they are special cases of (4c) (see [18]).

These definitions guarantee the correctness of temporal recursion rules [21] for nonempty state sequences. These rules split a temporal formula into a present nontemporal part to be checked in the first state and into a future temporal part with the **next**-operator to be checked in the tail of a state sequence.

Lemma 4 *The following equivalences hold for arbitrary temporal formulas* φ *and* τ *in state sequences* $\underline{\sigma}$*, provided* $\underline{\sigma}$ *is nonempty.*

(i)	**always** φ	$\iff$	$\varphi \wedge$ **next always** φ
(ii)	**sometime** φ	$\iff$	$\varphi \vee$ **next sometime** φ
(iii)	**always** φ **before** τ	$\iff$	$\tau \vee (\varphi \wedge$ **next** (**always** φ **before** $\tau))$
(iv)	**sometime** φ **before** τ	$\iff$	$\neg\tau \wedge (\varphi \vee$ **next** (**sometime** φ **before** $\tau))$
(v)	**always** φ **until** τ	$\iff$	$(\varphi \wedge \tau) \vee (\varphi \wedge$ **next** (**always** φ **until** $\tau))$
(vi)	**sometime** φ **until** τ	$\iff$	$\varphi \vee (\neg\tau \wedge$ **next** (**sometime** φ **until** $\tau))$

Here, proofs for infinite state sequences that rely on the existence of a first state only

[1] $\equiv$ stands for syntactic identity.

can be literally adapted from [18] to nonempty state sequences.

Whereas in theory dynamic integrity constraints have to be checked in state sequences for all assignments of possible keys to their object variables, in practice ordinary constraints affect only objects which exist in the database and that only during their time of existence. Since objects may be inserted and deleted during database runtime, their existence intervals, the so-called object life cycles [25], form subsequences of the complete database state sequence. If the same key values are deleted and re-inserted, the constraint refers to all such life cycles.

Definition 5 *Let φ be a temporal formula without possible quantifiers $\underline{\forall}, \underline{\exists}$ over objects (but maybe over data), but with free object variables $X = (x_1, \ldots, x_m)$. An* existence-restricted *integrity constraint with* body *φ is of the form*

$$\psi \equiv \textbf{during-existence}(X) : \varphi.$$

A state sequence $\underline{\sigma}$ is admissible *with regard to the constraint ψ iff φ is valid in every existence interval $\underline{\sigma_\theta}$ of each object combination $\theta = (t_1, \ldots, t_m)$ that can be assigned to the variables X. (An existence interval $\underline{\sigma_\theta}$ of an object combination ($m > 1$) begins when all objects in θ have been inserted, and it ends when the first object in θ is deleted.) Here, only* relevant *object combinations with nonempty existence intervals need to be considered.*

In the following, we will restrict ourselves to existence-restricted integrity constraints since they comprise most of the constraints relevant for practice.

2.2 Examples

To illustrate our techniques, we give a short extract of a database scheme describing an offer-and-order management (primary key attributes are underlined):

CUSTOMER(*cno*,*cname*, *account*)
SUPPLIER(*sno*, *sname*, *address*)
OFFER(*sno*, *part*, *price*)
ORDER(*ord_no*, *cno*, *sno*, *part*, *delivered*)

The following static constraint restricts single database states:

Example 6 *"For every order there must exist a matching offer." (referential integrity)*

$\rho \equiv \forall$ *ord: ORDER* $\exists$ *of: OFFER (ord.sno = of.sno* $\wedge$ *ord.part = of.part)*

It can be re-written as a dynamic constraint STC according to the pattern STC $\equiv$ **always** *ρ stating that ρ must hold in every database state.*

Example 7 *The following dynamic constraint DYC1 describes a restriction on state sequences concerning object pairs from relations OFFER and ORDER. From the instant of time when there exists an order for an offer, the price for this offer must not increase as long as the order for this offer exists. Furthermore the order must be delivered sometime during that time. (The deletion of the order might be connected to the creation of an invoice which is not specified here.)*

$$\textbf{during-existence}\,(of : OFFER, ord : ORDER)\ \underline{\forall} p : integer$$
$$of.part = ord.part \wedge of.sno = ord.sno \wedge of.price = p$$
$$\Rightarrow\ (\textbf{always}\ of.price \leq p \wedge \textbf{sometime}\ ord.delivered)$$

Example 8 *Since the primary key of ORDER is* ord_no *we additionally have to require explicitly that the initial values of the attributes* cno, ..., part *remain unchanged. We do so by specifying the following dynamic constraint DYC2:*

$$\textbf{during-existence}(ord : ORDER)\ \ \underline{\forall} c : integer,\ \ldots,\ \underline{\forall} p : integer$$
$$ord.cno = c \wedge \ldots \wedge ord.part = p\ \Rightarrow\ \textbf{always}\ (ord.cno = c \wedge \ldots \wedge ord.part = p)$$

2.3 Transition Graphs

In order to monitor dynamic integrity constraints in state sequences at database runtime, we have proposed in [21,18,22,25] to utilize transition graphs. They can be constructed from the temporal formulas sketched above at schema definition time and they reduce analysis of state sequences to checks on state transitions. In the appendix an algorithm is described to construct transition graphs as discussed in the sequel.

Definition 9 *A* transition graph $T = \langle V, E, F, \nu, \eta, v_0 \rangle$ *for a temporal formula* φ *consists of*

- *a directed graph* $\langle V, E \rangle$ *having a finite set* V *of nodes and a set* $E \subseteq V \times V$ *of edges,*
- *a set of* final nodes $F \subseteq V$*,*
- *a* node labelling ν *with temporal formulas,*
- *an* edge labelling η *with nontemporal formulas,*
- *and an* initial node $v_0 \in V$ *with* $\nu(v_0) = \varphi$*.*

For simplification reasons, we will work here with deterministic graphs only:

Definition 10 *A transition graph* T *is called* deterministic *iff for each node* v *with* d *outgoing edges labelled with* $\beta_1, \ldots \beta_d$ *holds:*

$$\beta_i \wedge \beta_j\ \Leftrightarrow\ \textbf{false}\ \ \textit{for all}\ i, j \in \{1, ..., d\}, i \neq j,$$

i.e. labels of different outgoing edges exclude each other.

A transition graph can be used to analyze a state sequence by searching for a corresponding path, whose edge labels successively hold in the states of the sequence. For that, the node which has been reached by an object combination θ must be "marked": A node v' is marked after passing a state σ iff there exists an edge $e = (v, v')$ such that the edge label $\eta(e)$ is valid in σ. If no edge satisfies this condition, no node gets marked. Since the graph was assumed to be deterministic, at most one outgoing edge can apply in each state. That graph processing, of course, starts in the initial node v_0. A marking must be maintained separately for each object combination θ relevant to the constraint. If we keep θ fixed for a moment, we can rely on the relationships between graphs and formulas reported below.

Definition 11 *A finite state sequence $\underline{\sigma}$ of states $\langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$ is called* accepted *up to state σ_n by a transition graph T iff there is a node marked after passing state σ_n. A finite state sequence is* completely accepted *by T iff a final node is marked after the last state.*

To guarantee that accepted state sequences correspond to admissible sequences concerning the constraint, certain requirements must hold for the node and edge labels of the transition graph. Intuitively, the nodes must be labelled by temporal formulas indicating what remains to be monitored in the future (as it is the case for the initial node which is labelled with the constraint body φ), whereas the edges are labelled by nontemporal formulas which have to be checked in the next state such that the right target node gets marked.

Definition 12 *A transition graph T is* correct *iff the following equivalence holds (in nonempty state sequences) for each node v with outgoing edges $e_k = (v, v_k), k = 1, ..., d$:*

$$\nu(v) \;\equiv\; \bigvee_{k=1}^{d} (\eta(e_k) \wedge \mathbf{next}\ \nu(v_k))$$

T is strongly correct *iff it is correct and for each node v with labelling $\nu(v)$ holds*

$$v \quad \textit{is final} \quad \textit{iff} \quad \underline{\lambda} \models \nu(v)$$

The definition supposes that each node label is valid in a nonempty state sequence iff there exists at least one outgoing edge whose label is valid in the first state and whose corresponding target node label is valid in the tail sequence. This tail sequence may, of course, become empty; then a final node must have been reached. Now we can state the following theorem:

Theorem 13 *Let φ be a temporal formula, T a correct transition graph for φ, and $\underline{\sigma}$ a state sequence. φ is potentially valid in a finite prefix of $\underline{\sigma}$ (i.e. there exists a continuation to a state sequence where φ is valid) iff that prefix is accepted by T and*

the formula in the marked node is satisfiable. If T is strongly correct, then φ is valid in $\underline{\sigma}$ iff $\underline{\sigma}$ is completely accepted by T and $\underline{\sigma}$ is finite.

The graph constructions are sketched in the appendix and extensively discussed in [17,18] with the additions how to distinguish final nodes in [22]. They deliver strongly correct transition graphs (as shown there).

In order to monitor existence-restricted constraints we can use the transition graph for the constraint body and check acceptance of the (usually finite) existence intervals.

Corollary 14 *Let T be a strongly correct transition graph for φ. $\underline{\sigma}$ is admissible with regard to* **during-existence**$(X) : \varphi$ *iff every existence interval $\underline{\sigma_\theta}$ is completely accepted for each relevant object combination θ.*

Thus we must keep track of the marked node, respectively situation for each object (combination) θ – at each state during its (joint) lifespan; this will be done in a new relation called *situation relation* with the scheme

$$SITUATION_\varphi(\underline{\langle \textit{ primary keys of involved objects } \rangle},\ node)$$

Such relations can be derived directly from the underlying constraint and transition graph, respectively. In Section 5 we will discuss some cases where such a situation relation for a transition graph is not necessary and where information about object situations can be derived from already stored information. Please note that theoretically markings are needed even for the combinations of those objects with all data fitting to data variables in the constraint body. In [15,16], however, techniques have been developed to work with finite representations of data combinations to avoid infinitely many tuples in the above relation. Our examples are restricted to such cases where only one unique data value has to be kept for each object combination as "historical information". Its determination will be indicated in the edge labels by a "**set**" clause. This value must be represented by an additional non-key attribute in the relation above.

Example 15 *Applying the graph construction algorithm in the appendix, the transition graph in Figure 1 can be constructed for the dynamic constraint DYC1 from example 7. v_0 is the initial node of the transition graph and final nodes are indicated.*

The nodes carry the following labels as temporal formulas which remain to be monitored in the future (depending on the situation of an order/offer pair):

v_0 : *of.part* $=$ *ord.part* $\wedge$ *of.sno* $=$ *ord.sno* $\wedge$ *ord.price* $= p$
$\Rightarrow$ (**always** *of.price* $\leq p \wedge$ **sometime** *ord.delivered*)
(initial node, labelled with constraint body)
v_1 : **always** *of.price* $\leq p \wedge$ **sometime** *ord.delivered*
v_2 : **always** *of.price* $\leq p$ *(final node)*

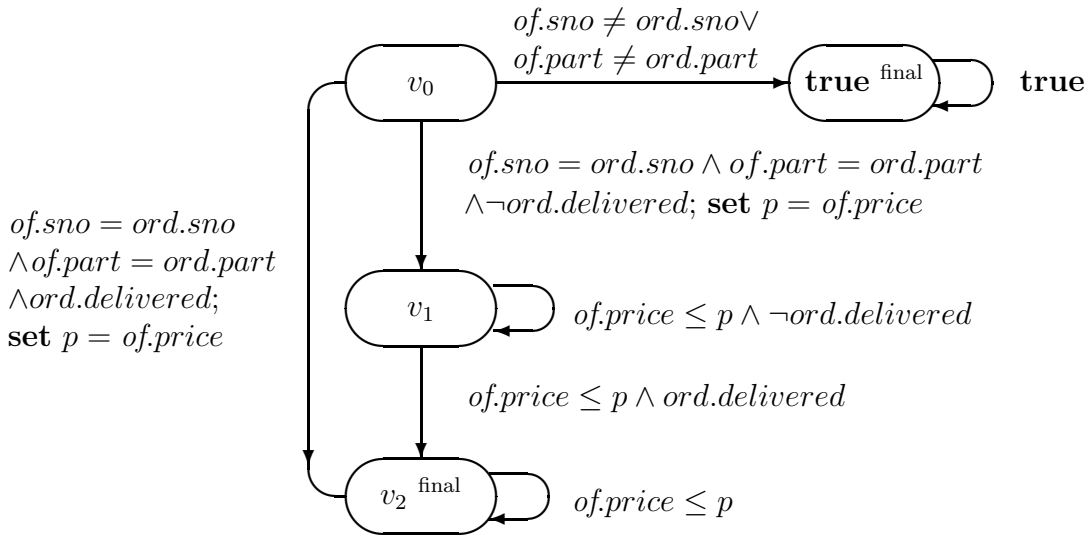


Fig. 1. Transition graph for *DYC1*

With those labels, the graph is strongly correct; later we will not need the node labels any longer. Intuitively, the nodes correspond to following situations:

- v_0 : *on creation of a matching offer/order pair (typically the insertion of an order corresponding to an offer, since offers may exist without orders but not vice versa – compare STC in Example 6)*
- v_1 : *offer under price limit, order before delivery*
- v_2 : *offer still under price limit, but after delivery (necessary for keeping the price till order deletion, maybe connected with an invoice creation)*

The corresponding situation relation has the following scheme where p is the initial price of an offer:

SITUATION_DYC1(sno, part, ord_no, p, node)

In Section 2.4 we will describe how to use the graph for monitoring constraint DYC1.

In addition to single graph constructions, there are techniques to combine graphs so that complete life cycles concerning all constraints on the same objects could be derived and taken as the basis for later constraint analysis and trigger generation.

Most transition graphs have a property that will be very helpful for trigger simplification:

Definition 16 *A transition graph is called iteration-invariant iff there exists a loop labelled with β for each node v such that for each ingoing edge β' from a node v' to v holds $\beta' \Rightarrow \beta$ (Figure 2).*

The above example graph (Fig. 1) obviously is iteration-invariant, i.e. the current situation will not change if the same condition holds as on entering the situation. In fact, all temporal formulas without **next** lead to iteration-invariant transition graphs,

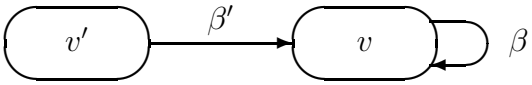


Fig. 2. Iteration-invariant graph

as shown in [17,18]. Since the sequence even of independent state transitions can hardly be controlled in real (multi-user) database systems, it seems to be reasonable not to refer absolutely to the next state. Typical transitional constraints can be expressed using **always** instead of **next**, like "prices can never increase" instead of "prices cannot increase from one to the next state".

2.4 Monitoring Algorithm

Algorithmically, monitoring a dynamic integrity constraint in a nonempty state sequence works as follows: For each substitution, the monitor follows corresponding correct transition graphs on each state transition to check whether the current prefix is accepted. In this case the constraint is not violated up to the current state but can be violated in subsequent future states. The constraint then is said to be *partially valid*. For an infinite state sequence, only constraint violations can be determined, but for a complete finite state sequence, full validity of the constraint can be decided: this is checked at the end of the sequence by looking for a marked final node.

In order to monitor existence-restricted constraints the transition graphs corresponding to their bodies need only be followed during the existence intervals of inserted objects (see Corollary 14), but not on the entire state sequence.

We now give a basic (non-optimized) algorithmic scheme for monitoring that will serve as a guideline for the later trigger generation presented in Section 4.

Algorithm 17 *We assume that a strongly correct and deterministic transition graph T_φ with final node set F has been constructed for each existence-restricted constraint with body φ. Let $SITUATION_\varphi(\theta, v)$ denote the schema of the relation where the marked node v for a substitution θ (object-combination) concerning a constraint φ is stored. The following procedure* **MONITOR** *has to be called after each transition from an old database state σ_{old} into a state σ, including the initialization into $\sigma = \sigma_0$ (σ_{old} undefined) as well as the exit after the last state (σ undefined). The phrase "kind of substitution" is used to determine whether a substitution has been inserted or deleted or whether it remained actual on that transition.*

procedure MONITOR *:*
  / global variables: σ_{old}, σ */*
  for *each constraint φ and*
  for *each substitution θ with objects from $\sigma_{old} \cup \sigma$ only*
    do case *kind of substitution θ* **of**

inserted: *insert into* $SITUATION_\varphi$ *tuple* (θ, v_0)*;*
/* v_0 *is the initial node of* T_φ */
$CHECK\_ACCEPTANCE(\theta, v_0, \varphi)$
actual: $CHECK\_ACCEPTANCE(\theta, v, \varphi)$
deleted: $CHECK\_COMPLETE\_ACCEPTANCE(\theta, v, \varphi)$*;*
delete from $SITUATION_\varphi$ *tuple* (θ, v)

endproc

procedure $CHECK\_ACCEPTANCE(\theta, v, \varphi)$
/* *get a new situation by the transition rule* */
for *each outgoing edge* $e = (v, v')$ *from* v *in* T_φ
do if $\eta(e)$ *is valid in* σ *for the substitution* θ **then** *update* $SITUATION_\varphi$
set values (θ, v') *where values are* (θ, v)*;*
if *no edge label has been valid* **then** *ERROR("not accepted: no edge applicable")*

endproc

procedure $CHECK\_COMPLETE\_ACCEPTANCE(\theta, v, \varphi)$ *:*
/* *Let* F *be the set of final nodes of* T_φ*.* */
if $SITUATION_\varphi(\theta, v), v \notin F$ **then**
ERROR("no final node reached after final state")

endproc

To monitor the given constraint *DYC1*, situations and the initial price p of an offer have to be maintained for all orders and offers which belong to the database at some time. Such a situation needs only to be generated when an order corresponding to an offer has been inserted. It is stored until the order object is deleted from the database. Deletions are only allowed in node v_2 (indicating that the order has been delivered). The graph excludes constraint violations like transitions leading to a price increase, since all edges outgoing from situations v_1 and v_2 are labelled with $of.price \leq p$.

Please note that there is no need to consider arbitrary values of p (different from the initial price) for monitoring, since the usage of substitutions for data variables can be optimized to a certain extent [16].

3 Triggers

The rule language we will use for our triggers is based on the language used in Starburst, a prototype relational database system developed at the IBM Almaden Research Center, but can also easily adopted to other systems. The main advantages of the rule system in Starburst (which are also important for our trigger generation) are that triggers can be deferred to the end of a transaction and that priorities between triggers can be specified. A detailed description of Starburst and the integrated production rule system can be found in, e.g., [12,35]. Here we focus on the features

relevant to our trigger generation.

Since we are interested in integrity maintenance, and transactions shall be integrity preserving units, we consider only triggers which are deferred to the end of transactions. Transactions are sequences of elementary database operations like insert-, update-, or delete-operations on relations. A transaction T determines a set Δ_T of operation instances. In Δ_T only the transaction net effect appears, i.e.

- if a tuple is updated several times in T, only the composite update appears in Δ_T,
- if a tuple is inserted and then updated in T, only the insertion of the updated tuple appears in Δ_T,
- if a tuple is updated and then deleted in T, only the deletion of the original tuple appears in Δ_T, and
- if the same tuple is deleted and then inserted in T (or vice versa), nothing appears in Δ_T.

We assume that the DBMS documents the net effect in *transition tables*. These logical tables reflect the changes that have occurred during the transaction, and which inherit all attributes from the corresponding base relations. For every relation in the database scheme there exist four transition tables: **inserted, deleted, old_updated** and **new_updated. inserted** lists those tuples of the relation in the current state which were inserted into the relation by the transaction (**deleted** by analogy). The two tables **old_updated** and **new_updated** contain those tuples of the relation of the previous state of a transaction and of the current state where at least one of the specified columns has been updated. We will make use of these two tables in checking dynamic constraints, since they allow to access the difference between the pre- and poststate of a transaction and may herewith improve efficiency on checking constraints.

The syntax for specifying a trigger (synonymously called a rule) is as follows:

create rule <*name*> **on** <*relation*>
when <*triggering-operations*>
    [**if** <*condition*>]
then <*list of actions*>
[**precedes** <*rule-list*>]
[**follows** <*rule-list*>]

Each rule is defined on a base relation. The clauses in square brackets above indicate optional parts of a rule definition. The **when** clause specifies one or more triggering operations. A rule is activated after the end of a transaction if at least one of the specified triggering operations **inserted, updated** or **deleted** has occurred on the relation during the transaction and has participated in its net effect, i.e. the corresponding transition table is not empty. The triggering operation **updated** may include a list of columns of the relation. Then at least one update on one of the listed

columns must have occurred in the transaction net effect to trigger the rule.

The **if** clause of a rule specifies a condition which is evaluated once if a rule is triggered. A condition may be any **select** statement and is evaluated to true (after triggering the rule) if the statement produces at least one tuple (instead of the **select** command also the SQL predicate **exists** can be used). We then say the rule is fired. If the condition is omitted the rule is always fired. If the rule is fired the **then** clause, which may contain a list of actions, is executed. An action can be any database operation, i.e. select, insert, update and delete statement, including a rollback. Actions are executed sequentially in the listed order.

The optional **precedes** and **follows** clause list other defined rules and are used to specify rule priorities. When a rule R specifies a rule R' in its **precedes** list, this indicates that if both rules can be fired at the same time, R precedes R'. In contrast, if a rule R specifies a rule R' in its **follows** list, R' will be considered before R if both are fired at the same time.

The condition and the action parts of a rule may refer to the current database state, i.e. the state when the rules are triggered. They may refer also to transition tables mentioned above. If a rule on a relation p specifies **inserted** as its triggering operation, the transition table **inserted** used in the condition and/or action part refers to the logical table containing all tuples that were inserted into the relation p. The transition tables **new_updated** and **old_updated** contain the current respectively original values of tuples updated in the relation p by the triggering transaction. Note that also in transition tables only the net effect appears.

Examples of rules are given in the following section where we use them for monitoring integrity constraints.

Rules are processed automatically at the end of each transaction (*deferred triggers*). Rule processing typically starts with the end of a user- or application generated transaction T before committing T. A transaction T performs a state transition from an initial state σ to a new state σ', denoted as $\sigma \xrightarrow{T} \sigma'$. If a rule R is triggered in σ' and its condition is evaluated to true, the action of the rule is executed. The execution of this action is considered as a continuation T_R of the transaction T leading to a new state σ'' ($\sigma' \xrightarrow{T_R} \sigma''$). If any rule executes a rollback the entire transaction aborts, leading to the initial state σ. Otherwise the entire transaction $T \circ T_R \circ \ldots$ commits when rule processing terminates, i.e. when no rule is fired after a transaction. For more details on the semantics of rule execution in Starburst, in particular how transition tables are maintained during rule execution, we refer to [36].

4 Deriving Triggers

In this section we describe the main procedures to derive the triggers that ensure the constraints specified in the database scheme. The techniques presented here work for dynamic constraints and their corresponding transition graphs, respectively, as well as for simple static constraints.

Given a transition graph, the goal now is to derive triggers which react on all operations at runtime that may violate the respective constraint. In the following we first discuss techniques for static constraints (compare [3] for more detailed investigations) and then develop our approach for dynamic constraints.

4.1 Static Constraints

On the assumption that a static constraint is valid before transaction execution the condition has to be checked after committing a transaction only if operations occurred in the transaction net effect that may invalidate the constraint. To determine whether an operation can invalidate a constraint the indifference-, insertion-, deletion- and modification-rules described in [20] can be used: If a constraint remains invariant under the operation (for arbitrary arguments), this operation cannot invalidate the constraint. For example, our referential integrity constraint (Example 6)

$$\rho \equiv \forall ord : ORDER\ \exists of :\ OFFER\ (ord.sno = of.sno \wedge ord.part = of.part)$$

can never be violated by any operation on the relation *CUSTOMER*, since the relation is not mentioned in the constraint. But also a few operations on the mentioned relations can never invalidate this constraint, e.g., any deletion on *ORDER* or any update on the attribute *price* of *OFFER*. All other operations, however, may be critical for that constraint.

As mentioned in Section 2.3, static constraints are implicitly prefixed by the temporal quantifier **always**. The corresponding transition graph consists of a single node and a loop labelled with the constraint ρ:

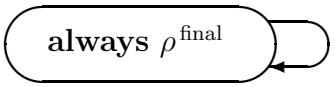


Figure 3: Transition graph for static constraint ρ

Remember that the condition at the loop must hold after every state transition. Since one can assume the condition to be valid in the state before the transition, the condition has to be checked after a transition that includes possible invalidating operations.

Obviously, an operation which may violate this constraint is an insertion into the relation *ORDER*. Thus the following trigger is needed (among others) to enforce the constraint:

create rule *ins_ORDER* **on** *ORDER*
when inserted
if exists (**select** $*$ **from** *ORDER ord*)
    where not exists (**select** $*$ **from** *OFFER of*
        where *ord.sno* = *of.sno* **and** *ord.part* = *of.part*)
then rollback

The condition to be checked in the trigger is the violation condition of the constraint, stating that there exists an order with no corresponding offer. A closer look at the arguments of the insert operation shows that the constraint can be violated only by such tuples of the relation *ORDER* that have been inserted into the relation by the transaction, since for all "old" tuples in *ORDER* the constraint was assumed to be valid before. Consequently the constraint has to be checked only for the new tuples of *ORDER* which we can find in the transition table **inserted**. The condition to be checked by the trigger then can be modified as follows, thus reducing the necessary database accesses and improve checking:

if exists (**select** $*$ **from inserted** *neword*)
    where not exists (**select** $*$ **from** *OFFER of*
        where *neword.sno* = *of.sno* **and** *neword.part* = *of.part*)

The underlying technique of specializing constraints to such tuples that were affected by the transaction was introduced in [24] and has been extended by [13] and other authors. The rules of [20] described how to syntactically preselect completely uncritical operations. All these approaches utilize (more or less explicitly) the principle that the constraint or parts of it remain invariant under certain operations for all or for certain arguments. We will utilize this principle for dynamic constraints, too.

Given such trigger patterns, a database designer will often tend to modify the rollback reaction such that constraint violations get repaired actively. Automatic generation of active triggers, however, is not considered in this paper. First successful attempts in that direction have been made in [1,8] for active databases, in [32,28] in the context of object-oriented databases and in [23] in context of deductive databases.

4.2 Dynamic Constraints

Since dynamic constraints are specified by temporal formulas which normally are interpreted in state sequences rather then in single states, maintaining such constraints depends upon unknown future states as well as entire subsequence of states. These problems can be avoided by using transition graphs described in Section 2.3:

They reduce analysis of state sequences to state transitions and herewith to checking changing static conditions.

In the following we describe how to use transition graphs for deriving triggers which monitor arbitrary dynamic constraints. First, we investigate which operations are uncritical for a constraint such that no triggers are needed for them.

In contrast to static constraints it does not make sense to assume a dynamic constraint to be valid before a transaction. As we have already seen for the monitoring algorithm we must look at the situation that has been reached by the objects or object combinations under consideration. Since the corresponding transition graph describes their life cycles corresponding to the constraint, situations are represented by nodes and characteristic conditions for those situations can be found in the loop label of each node. Remember the example graph of Section 2.3 (Fig. 3).

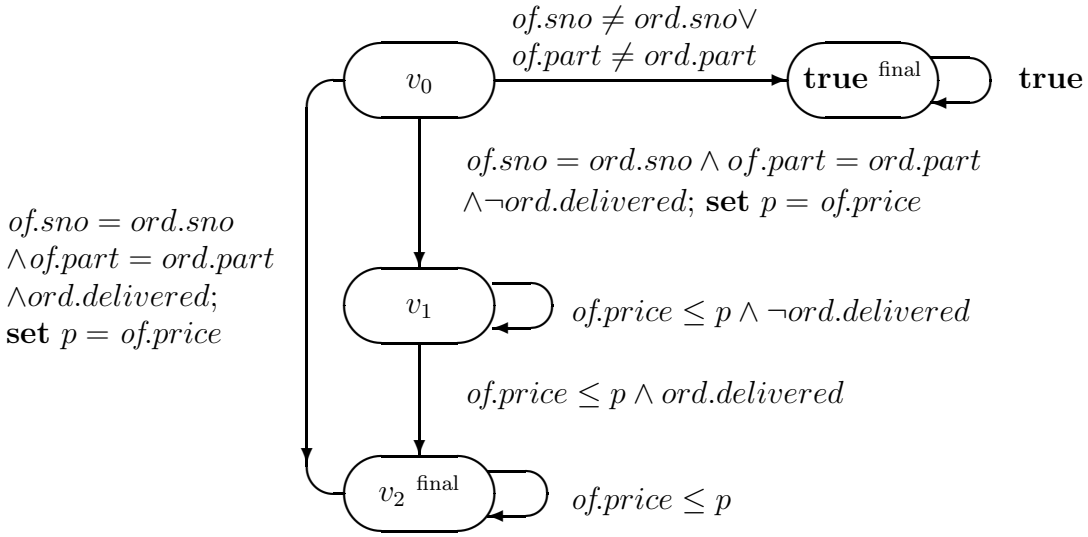


Fig. 3. Transition graph for *DYC1*

The life cycle of an object combination θ (here the primary key values of a tuple *of* from the relation *OFFER* and of a tuple *ord* from the relation *ORDER*) begins by entering the graph via an edge which leaves the initial node v_0 and whose label is valid in the first state. Thus these edges have to be checked only if at least one object of the combination has been inserted during the current transaction. In our example, inserting an order which is not immediately delivered and which corresponds to an offer (that may be inserted by the same transaction) leads to entering the the situation node v_1; hence the insertion is valid concerning constraint *DYC1*. Thus we need triggers for insertions of orders and offers.

Now assume that an object combination (*of, ord*) has reached a situation node v_i $(i = 1, 2)$ in its life cycle. This node must have been entered via one ingoing edge valid in the previous state when that situation was computed. Since we consider iteration-invariant graphs only, there must be a loop at v_i and the loop label must be a logical implication of all ingoing edge labels (see Definition 16). So this label

has been valid in the previous state and can be assumed to hold before the current transaction.

The subsequent state transition is now acceptable iff there exists one outgoing edge from v_i which is valid in the new state (compare procedure CHECK_ACCEPTANCE in Section 2.4). If the loop label at v_i is invariant under the transaction, the situation remains invariant, too, since the transition graph is deterministic. Thus an operation in that transaction is uncritical for the constraint in situation v_i, if the loop label is invariant under the operation; then no trigger needs to be derived for that operation and for that situation. For situation nodes labelled **true** nothing needs to be checked at all since the loop label **true** is valid always.

Again critical, however, may be deletions on the objects under consideration. They are valid only if a final node has been reached before (compare procedure CHECK_COMPLETE_ACCEPTANCE in Section 2.4). Edge labels need not be checked for them.

If the loop label invariance described above holds for an operation at each node of a transition graph, the operation is completely uncritical for the corresponding constraint so that no trigger is needed. Otherwise, we have to generate triggers depending on operations and situations. We will discuss this aspect in more detail in Section 5.

We generalize and formalize our observations above into the following rules for deriving trigger patterns.

Rules 18 *Let v_i denote a node in the transition graph for an integrity constraint of the pattern*

$$\textbf{during-existence}(x_1\colon R_1, \ldots, x_m\colon R_m)\colon\ \varphi$$

concerning combinations of tuples from the relations R_1 to R_m. Outgoing edge labels at v_i are denoted by β^i for the loop label and $\beta^i_1, \ldots, \beta^i_d$ for all other edge labels like in Figure 4.

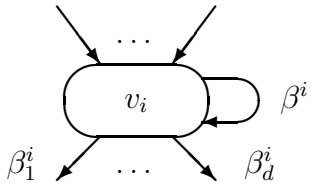


Fig. 4. Node v_i with ingoing and outgoing edges

Let furthermore SITUATION$_\varphi$ denote the situation table corresponding to the transition graph for the constraint above.

for each *node v_i* **do**
  for each *operation ω on relation R'* **do**
    if *v_i is the initial node v_0* **and** *$\omega \equiv$* insert into $R_j, i \in \{1, \ldots, m\}$*:*

create rule $\varphi\_v_0\_ins$ **on** R_j
when inserted
if exists
  select $*$ **from** $R_1\, x_1, \ldots, R_{j-1}\, x_{j-1}$, **inserted** x_j, $R_{j+1}\, x_{j+1}, \ldots, R_m\, x_m$
  where not $(\beta^0 \vee \beta_1^0 \vee \ldots \vee \beta_d^0)$
then rollback

rule (1)

else if $\omega \equiv$ delete from R_j, $j \in \{1, \ldots, m\}$ **and** v_i *is a non-final node*

create rule $\varphi\_v_i\_del$ **on** R_j
when deleted
if exists
  select $*$ **from deleted** x_j, $SITUATION_\varphi$ *sit*
   where $<$ *matching condition on primary key values* $>$
   and *sit.node* $= v_i$
then rollback

rule (2)

else if β^i *is not invariant under the operation* ω *on* R'

create rule $\varphi\_v_i\_\omega$ **on** R'
when ω
if exists
  select $*$ **from** $R_1\, x_1, \ldots, R_m\, x_m$, $SITUATION_\varphi$ *sit*
   where $<$ *matching condition on primary key values* $>$
   and *sit.node* $= v_i$ **and not** $(\beta^i \vee \beta_1^i \vee \ldots \vee \beta_d^i)$
then rollback

rule (3)

fi
od; od.

Depending on the node type and operation a trigger is derived to check whether, depending on the current situation, at least one outgoing edge label is valid in the new state (rule 1 and 3) or to check whether deletions are permitted in case of non-final nodes (rule 2). In the negative case, a rollback of the triggering transaction is induced. To check outgoing edge labels at the initial node v_0 only for object combinations containing newly inserted objects, we directly refer to the transition table **inserted** in the **if** clause of rule 1. We directly refer to the transition table **deleted** in rule 2, since we have to check whether the current situation corresponds to a final node only for those combinations where an object was deleted from. To refer to situations reachable in object life cycles we utilize the situation relation $SITUATION_\varphi$ as introduced in Section 2.3.

In addition to the primary key values of the objects and their current situation this

relation can also contain attributes for storing (historical) data like the price of a part when inserting an order to an offer as in the previous example. In Section 4.3 we will describe how to set such data but here they are assumed to be already stored.

Applying the rules above to each constraint leads to a complete, but yet more optimizable set of triggers which ensure that no transaction can invalidate any constraint.

We now give examples of applying the rules to the transition graph of the constraint *DYC1* (the disjunction of edge labels are simplified if possible): [3]

Example 19 *Trigger to check validity of delete operations on the relation ORDER:*

create rule *DYC1_v1_del* **on** *ORDER*
when deleted
if exists (**select** $*$ **from deleted** *ord, SITUATION_DYC1 sit*
where *ord.ord_no* = *sit.ord_no* **and** *sit.node* = *'v1'*)
then rollback

The violation condition states that a tuple from the relation ORDER has been deleted by the triggering transaction (and hence is stored in the transition table **deleted***) and which had node v1, i.e. a non-final node, as its current situation when deleted. In this case the triggering transaction will be rolled back.*

Example 20 *Trigger to check validity of an update operation on the relation OFFER:*

create rule *DYC1_v1_upd* **on** *OFFER*
when updated(*price*)
if exists (**select** $*$ **from** *OFFER of, SITUATION_DYC1 sit*
where (*sit.sno* = *of.sno* **and** *sit.part* = *of.part*
and *sit.node* = *'v1'* **and not** *of.price* <= *sit.p*))
then rollback

There is no reference to ORDER in the violation condition, since the disjunction of edge labels at node v1 simplifies to of.price $\leq p$*. Since the loop label remains invariant for all tuples whose price has not been updated, we may restrict the trigger to the new attribute values of the updated tuples: OFFER of can be replaced by* **updated_new** *of.*

Example 21 *The trigger for updating the delivery status on the relation ORDER reads similar:*

create rule *DYC1_v1_upd* **on** *ORDER*

[3] Transformation of edge labels, i.e. formulas of the first-order predicate logic, into equivalent SQL-expressions are done intuitively here but could also be automated by appropriate mechanisms. Compare, e.g., [11] who gives translation rules from tuple calculus into SQL.

when updated(*delivered*)
if exists (**select** $*$ **from** *ORDER ord, SITUATION_DYC1 sit, OFFER of*
where (*sit.ord_no = ord.ord_no*
and *sit.sno = of.sno* **and** *sit.part = of.part*
and *sit.node = 'v1'* **and not** *of.price <= p*))
then rollback

Although the loop label at v_1 (of.price $\leq$ p **and not** *delivered) is not invariant, we can utilize that a part of it, namely of.price $\leq$ p is invariant under the update, so that this condition need not to be checked by the trigger as well. Thus the trigger can be omitted!*

Abstracting from the latter examples, we can give a general rule for trigger optimizations:

Remember that we assume that the loop label β at node v_i has been valid for all object combinations at situation v_i. Thus any subcondition β' of β specialized to such object combinations and appearing in the disjunction of outgoing edge labels $(\beta \vee \beta_1 \vee \ldots \vee \beta_d)$ can be deleted from the condition of trigger $\varphi\_v_i\_\omega\_R'$, if β' is invariant under the operation ω.

Above optimizations have utilized this rule.

4.3 Computation of New Object Situations

If none of the derived triggers causes a rollback of the triggering transaction, the transaction is admissible with regard to all specified constraints. Hence we know that for each object or object combination having a situation at node v there exists a valid outgoing edge label in the new state, or a deletion has occurred together with a final node. We now consider how to manage situations of objects or object combinations and how to handle **set** clauses as part of edge labels.

For an object in a situation v_i, the situation may change on a state transition since an outgoing edge from the node v_i in the transition graph to a node different from v_i is valid in the new state. The situation (i.e. marked node for this object) then has to be updated to comply the description of admissible life cycles of objects in transition graphs as described in Section 2.3. This update has to take place on the extra relation $SITUATION_\varphi$, but it need not be done by the user or the transaction but can be done by triggers as well which are derivable from the corresponding transition graph. Please note that these extra relations are only used by our triggers. Thus they are invisible to the user.

Triggers to compute new object situations are similar to those for checking edge labels as described in the previous section. Since we know after execution of the checking

triggers that for an object in a situation v_i at least one outgoing edge is valid in the new state, we have to check only which edge label is valid and then have to set the corresponding new situation in the situation relation $SITUATION_\varphi$. For this purpose, we generate d different triggers for node v_i if there are d outgoing edges not including the loop label β^i. These triggers also depend on the node type and operation under consideration.

Rules 22 *An object combination which enters a transition graph (being inserted) does not have any situation before. Its primary key values and its first situation, which is given by the target node v' of the valid edge label β' at the initial node v_0, have to be inserted into the relation $SITUATION_\varphi$. Similar to rule 18(1) we generate triggers according to the following pattern:*

create rule $\varphi\_v_0\_v'\_ins$ **on** R_j
when inserted
then insert into $SITUATION_\varphi$(<*primary key values*>, v')
  select < *primary key values* > **from** $R_1\, x_1, \ldots,$ **inserted** $x_j, \ldots, R_m\, x_m$
  where β'

For objects that already have a situation v in the graph, the new situation depends on which outgoing edge to a node $v'(v' \neq v)$ has a label β' valid in the new state. Similar to rule 18(3) the following triggers are generated:

create rule $\varphi\_v\_v'\_\omega$ **on** R'
when ω
then update $SITUATION_\varphi$ *sit* **set** *sit.node* $= v'$
  where *sit.node* $= v$ **and**
        exists (**select** $*$ **from** $R_1 x_1, \ldots, R_m x_m$
              where < *matching condition with sit* > **and** β')

Remember that situations whose loop label is valid in the new state remain unchanged so that no triggers are needed for loops. If the target node v' is a **true** *node, the old situation tuples may simply be deleted, since the situation would not change in the future.*

For objects deleted by the transaction and having a final node v as the current situation, storing situations is not needed any longer since their life cycle corresponding to the constraint has been finished. They only have to be deleted from the relation $SITUATION_\varphi$ by analogy to rule 18(2):

create rule φ_v_del **on** R_j
when deleted
then delete from $SITUATION_\varphi$ *sit*
  where *sit.node* = *v*
  and exists (**select** * **from deleted** x_j
    where *sit.*<*primary key values*> = x_j.<*primary key values*>)

Edge labels which have a **set** clause cause extensions to the insert or update operations on the relation $SITUATION_\varphi$ in the following way (here for an insertion on *ORDER* in our example *DYC1*, assuming that the edge label from v_0 to v_1 is valid):

Example 23

create rule *DYC1_v0_v1_ins* **on** *ORDER* **when inserted**
then insert into *SITUATION_DYC1*(*sno, part, ord_no, p, 'v1'*)
  select *of.sno, of.part, ord_no, price* **from** *OFFER of,* **inserted** *neword*
  where (*of.sno* = *neword.sno* **and** *of.part* = *neword.part*
    and not *neword.delivered*)

Given an order/offer combination having node v_1 as their current situation, updating the delivery status of an offer from **false** to **true** leads to entering the situation node v_2. This new situation of the combination can be computed by the following trigger generated by rule 18(2):

Example 24

create rule *DYC1_v1_v2_upd* **on** *ORDER* **when updated**(*delivered*)
do update *SITUATION_DYC1 sit* **set** *node = 'v2'*
  where *sit.node = 'v1'* **and exists**
  (**select** *ord_no* **from updated_new** *updord*
    where *updord.ord_no* = *sit.ord_no* **and** *updord.delivered* = **true**)

Given a static constraint ρ, there is no need to store any situation since in the corresponding graph only one situation occurs. This reduces trigger generation to rule (3) in Section 4.2 (without using situations) and thus includes the known treatment of static constraints.

During runtime all triggers computing new situations of objects have to be executed **after** checking validity of the transaction by triggers of Section 4.2. As described in Section 3, the Starburst rule system provides a suitable mechanism to order triggers by the use of the **precedes** and **follows** clauses. Hence the set of all triggers checking invalidating operations has to be listed in the **precedes** list of the triggers above.

5 Integrating Trigger Generation into Database Design

The presented approach for deriving integrity checking and object situations maintaining triggers can easily be integrated into the database design. Aside from modelling database structures, e.g., relational schemata, the specification of static and dynamic integrity constraints and their transformation into integrity maintaining triggers therein becomes a main design task (Fig. 5).

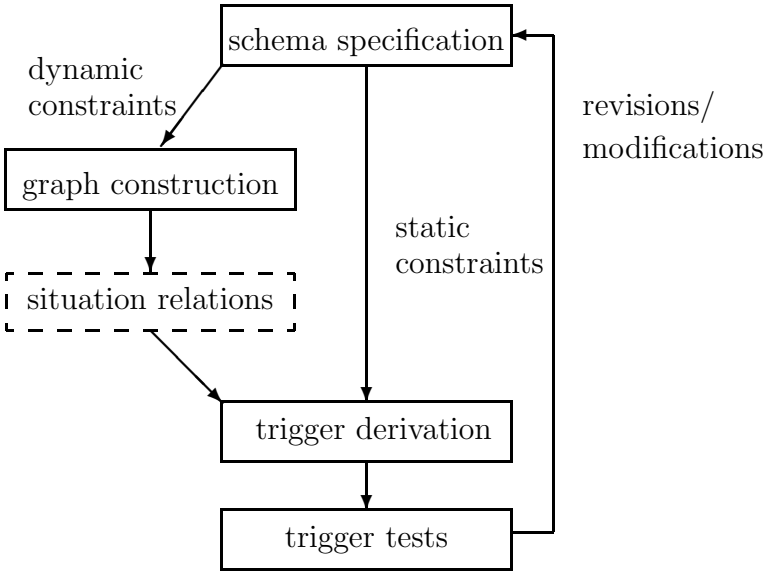


Fig. 5. Process of trigger derivation

The derived triggers allow to test the original (typically descriptive) integrity specification for its operational behavior. If this does not match the designer's expectations the constraints have to be revised, and certain design and transformation steps have to be repeated.

In this section we focus on two important interactive design tasks involved in the trigger derivation process. First, we describe how to exploit certain techniques to avoid extra relations for storing object situations. Secondly, we shortly sketch how derived triggers can be enriched to capture more meaning on constraint violations at runtime.

5.1 Simplifications on Storing Object Situations

As discussed in the previous sections, object situations need to be stored in extra generated situation relations for each transition graph in order to keep track of marked nodes at runtime. Storing and maintaining object situations, however, can produce a large overhead in checking constraint violations by triggers. Time consuming joins need to be performed on object and situation relations in order to determine object

situations and thus which edges have to be valid and which nodes get marked, respectively. An important design task thus is to exploit certain properties of transition graphs corresponding to dynamic integrity constraints in order to avoid such extra relations. We will discuss these properties in the sequel.

For transition graphs which describe the life cycle and admissible state sequences of single objects (i.e. no object combinations) the current situation as well as historic information for the objects can completely be stored in the object relation itself. The relation then is extended by respective attributes and no external situation relation is needed for this transition graph. If, however, these objects are involved in many dynamic integrity constraints the number of attributes of this object relation can increase drastically. Thus the designer has to choose for each such transition graph whether to extend the object relation or to generate an extra situation relation. If it is known in advance that later transactions may often violate the underlying constraint, object situations (and historic information) should preferably be stored in the object relation itself.

Another important feature of extending object relations is, of course, that at runtime the user can easily query object situations, e.g., in order to plan future transactions which correctly continue a life cycle. Such an access to situational information, of course, requires that the designer has given some application–oriented interpretation, at least an appropriate naming, to node numbers etc. Nevertheless, respective implementations of object relations and of explicit situation relations in a DBMS should guarantee that only the derived triggers are permitted to modify object situations and historic information.

For a certain class of transition graphs any situation relation can be avoided, namely if the loop labels at each node exclude each other [18]. This criterion holds for transition graphs that take single objects as well as object combinations into account. For example, assume that for our graph constructed from constraint *DYC1* (Figure 1) the loop at node v_2 has the label $of.price \leq p \wedge ord.delivered$ (this implicitly holds due to Definition 16). Then the loop labels at node v_1 and v_2 obviously exclude each other, i.e. there exists no order/offer pair where the loop label at node v_1 and the label at node v_2 holds. In order to determine the current situation of an order/offer pair it is sufficient to check whether the order is delivered or not. Additionally, the initial price of an offer can be stored in the relation *ORDER*. All information about current situations and historic information are available in the relations *ORDER* and *OFFER*. Thus time consuming joins for matching conditions on primary key values using the situation relation as described in rules 18 and 22 are superfluous.

Example 25 *Assume the formula at the loop label at node v_2 as mentioned above. Furthermore, suppose that the relation ORDER additionally stores the initial price of an offer as follows*

ORDER(<u>ord_no</u>, *cno, sno, part, delivered, initprice*).

The following trigger checks the validity of an update on the relation offer.

create rule *DYC1_v1_upd'* **on** *OFFER*
when updated(*price*)
if exists (**select** ∗ **from** *ORDER ord,* **updated_new** *of*
where (*of.sno* = *ord.sno* **and** *of.part* = *ord.part*
and not *ord.delivered* **and not** *of.price* <= *ord.initprice*))
then rollback

In contrast to the respective trigger in Example 21, a join over the relations ORDER and the transition table **updated_new** *(containing only the new values of updated tuples of the relation OFFER) is sufficient.*

Note that due to the simplifications on storing and computing object situations as described above the rules 18 and 22 have to be modified appropriately.

5.2 Extending Derived Triggers

The action parts of our derived integrity checking triggers consist only of a simple rollback of the triggering transaction. In view of rising acceptance and demand for automated integrity control, however, this is not sufficient for future users. The triggers should at least be enriched (at design time) by actions and procedures which visualize the detected violations of a constraint and violating tuples, respectively.

Example 26 *According to the checking trigger presented in Example 21 the following extended trigger stores violating tuples in an extra relation <Name> and informs the user:*

create rule *DYC1_v1_upd* **on** *OFFER*
when updated(*price*)
if exists (**select** ∗ **from** *OFFER of, SITUATION_DYC1 sit*
where (*sit.sno* = *of.sno* **and** *sit.part* = *of.part*
and *sit.node* = *'v1'* **and not** *of.price* <= *sit.p*))
then insert into *violate_DYC1_OFFER*
select ∗ **from updated_new** *of*
where exists (**select** ∗ **from** *SITUATION_DYC_1 sit*
where *sit.sno=of.sno* **and** *sit.part=of.part*
and *sit.node='v1'* **and not** *of.price* ≤ *sit.p*);
dbms_message (*'constraint DYC1 violated by update on OFFER;*
violating tuples are stored in VIOLATE_DYC1_OFFER');

Complex ad-hoc transactions can violate several different constraints. Hence it is also desirable first to check all constraints for violations, then to print out violating tuples/operations of the triggering transaction, i.e. those involved in the net effect,

and finally to perform a rollback in case of inconsistencies. Therefore checking triggers, in addition to the storage clause in Example 26 above have to set, e.g., a rollback flag. This flag, again, can be checked using a single trigger which follows all integrity checking triggers and precedes all triggers maintaining object situation. If the flag is set, the trigger performs a rollback.

6 Conclusions

In this paper we have presented general rules how to derive integrity monitoring triggers from dynamic integrity constraints represented by transition graphs and also how to refine rules according to graph properties to get more efficient triggers. Transition graphs describe life cycles of database objects that are admissible with regard to the constraints. They reduce integrity monitoring to checking changing static conditions according to life cycle situations. Thus triggers have to be generated from these graphs which depend not only on the operations that occur in the net effect of a transaction, but also for the situations that have been reached by the objects mentioned in the constraints. Fortunately, many of these combinations prove to be uncritical by considering node types, by identifying preconditions in the graphs which are characteristic for the situations, and by utilizing such (sub)conditions that remain invariant under database operations. Here the property of transitions graphs to be iteration-invariant has been helpful. Further simplifications to improve efficiency on integrity checking rely on identifying objects not affected by the transaction and not involved in potential constraint violations since corresponding condition instances are invariant. In addition to the constraint checking triggers, additional – active – triggers have been derived to maintain the information on life cycle situations which is needed for constraint checking.

As mentioned at the end of Section 4.1 the designer often tends to modify the rollback reaction such that constraint violations can get repaired actively. In [7,8] we have presented a framework for specifying the reactive behavior on violations of static integrity constraints. Repair actions on constraint violations can be specified declaratively and are transformed into respective repairing triggers. Our future work includes the extension of such active reactions to violations of dynamic integrity constraints. In particular, the problem arises that **sometime** ... **before** constraints may require a timely activation of certain transactions before the end condition occurs (which may be some unpreventable instant of real time). Using the current approach, a designer will become aware of such critical cases when analyzing the transitional behavior, but then he/she will typically tend to weaken constraints and to introduce some explicit exception management. It remains to be investigated whether using other non-standard logics like, e.g., deontic logic [37] for constraint specification might lead to deriving some trigger-based support of handling non-normative behavior, as temporal logic does for handling history-dependent behavior.

Although the rules presented may serve as a guideline for manual design, there is a need for tools that control completeness of trigger generation and help the designer in finding simplifications as mentioned in Section 5.1. In [9] we have presented a design environment called ICE ("Integrity-Centered Environment") which supports the designer by suitable tools. Our tool box for database design supports dynamic integrity constraints in particular. In addition to the construction of transition graphs from temporal formulas, transformation of these graphs into pre-specified transactions according to [19] and into triggers according to this paper are the main lines of the project. Tests for invariants and automatic detection of simplifications are subject to current implementations. Since the system is to deliver a prototype database, we want to check the performance of maintaining integrity constraints that has been reached by our generation and simplification patterns.

References

[1] S. Ceri, P. Fraternali, S. Paraboschi, L. Tanca: Automatic Generation of Production Rules for Integrity Maintenance. *ACM Transactions on Database Systems 19:3 (September 1994)*, 367–422.

[2] J. Chomicki: History-less Checking of Dynamic Integrity Constraints In *Eighth International Conference on Data Engineering - 1992*, 557–564, IEEE Computer Society Press, 1992

[3] S. Ceri, J. Widom: Deriving Production Rules for Constraint Maintenance. In D. McLeod, R. Sacks-Davis, H. Schek (eds.), *Proceedings of the 16th Int. Conf. on Very Large Data Bases - 1990*, 566–577, Morgan Kaufmann Publishers, 1990.

[4] C. Date: *An Introduction to Database Systems, Vol. II*, Addison-Wesley, Reading (Mass.), 1983.

[5] K. Eswaran, D. Chamberlin: Functional Specifications of a Subsystem for Data Base Integrity. In *Proc. Int. Conf. on Very Large Data Bases 1975*, 48–68, 1975.

[6] P. W. Grefen, P. M. Apers: Integrity Control in Relational Database Systems - An Overview. *Data & Knowledge Engineering 10:2 (March 1993)*, 187–223.

[7] M. Gertz: On Specifying the Reactive Behavior on Constraint Violations. Technical Report 02/93, Institut für Informatik, Universität Hannover, 1993

[8] M. Gertz: Specifying Reactive Integrity Control for Active Databases. In J. Widom, S. Chakravarthy (eds.), *RIDE'94 - Fourth International Workshop on Research Issues in Data Engineering*, 62–70, IEEE Computer Society Press, 1994.

[9] M. Gertz, U. W. Lipeck: ICE: An Environment for Integrity-Centered Database Design. In U. W. Lipeck, G. Vossen (eds.), *Formale Grundlagen für den Entwurf von Informationssystemen - GI-Workshop, Tutzing*, Technical Report 03/94, Institut für Informatik, Universität Hannover, 1994.

[10] G. Gardarin, M. Melkanoff: Proving Consistency of Database Transactions. In *Proc. Int. Conf. on Very Large Data Bases 1979*, 291–298, 1979.

[11] M. Gogolla: *An Extended Entity-Relationship Model - Fundamentals and Pragmatics*. LNCS 767. Springer-Verlag, Berlin, 1994.

[12] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson: Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering 2:1 (March 1990)*, 143 – 160.

[13] A. Hsu, T. Imielinski: Integrity Checking for Multiple Updates. In *Proc. ACM SIGMOD Int. Conf. on Management of Data 1985, Austin*, 152–168, 1985.

[14] M. Hammer, D. McLeod: Semantic Integrity in a Relational Database System. In *Proc. Int. Conf. on Very Large Data Bases 1975*, 25–47, 1975.

[15] K. Hülsmann, G. Saake: Representation of the Historical Information Necessary for Temporal Integrity Monitoring. In F. Bancilhon, C. Thanos, D. Tsichritzis (eds.), *Advances in Database Technology — EDBT '90*, 378–392, LNCS 416, Springer-Verlag, Berlin, 1990.

[16] K. Hülsmann, G. Saake: Theoretical Foundations of Handling Large Substitution Sets in Temporal Integrity Monitoring. *Acta Informatica 28 (1991)*, 365–407.

[17] U.W. Lipeck, D. Feng: Construction of Deterministic Transition Graphs from Dynamic Integrity Constraints. In *Proc. 14th Intern. Workshop on Graph-Theoretic Concepts in Computer Science*, 166–179, LNCS 344, Springer-Verlag, Berlin, 1988.

[18] U. W. Lipeck: *Dynamische Integrität von Datenbanken: Grundlagen der Spezifikation und Überwachung* (Dynamic Database Integrity: Foundations of Specification and Monitoring, in German). Informatik-Fachberichte 209. Springer, Berlin, 1989.

[19] U. W. Lipeck: Transformation of Dynamic Integrity Constraints into Transaction Specifications. *Theoretical Computer Science 76 (1990)*, 115 – 142.

[20] T.-W. Ling, P. Rajagopalan: A Method to Eliminate Avoidable Checking of Integrity Constraints. In *Proc. IEEE Trends & Applications Conference: Making Database Work*, 60–69, 1984.

[21] U.W. Lipeck, G. Saake: Monitoring Dynamic Integrity Constraints Based on Temporal Logic. *Information Systems 12:3 (1987)*, 255–269.

[22] U. W. Lipeck, H. Zhou: Monitoring Dynamic Integrity Constraints on Finite State Sequences and Existence Intervals. In J. Goers, A. Heuer, G. E. Saake (eds.), *Proc. 3rd Workshop on Foundations of Models and Languages for Data and Objects*, Informatik-Bericht 91/3, TU Clausthal, 115–130, 1991.

[23] G. Moerkotte, P. C. Lockemann: Reactive Consistency Control in Deductive Databases. *ACM Transactions on Database Systems 16:4 (December 1991)*, 670–702.

[24] J.-M. Nicolas: Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica 18 (1982)*, 227–253.

[25] G. Saake: Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering 6 (1991)*, 47–73.

[26] S. Schwiderski, T. Hartmann, G. Saake: Monitoring Temporal Preconditions in a Behaviour Oriented Object Model. *Data & Knowledge Engineering 14 (1994)*, 143–186.

[27] M. Stonebraker, G. Kemnitz: The Postgres Next-Generation Database Management System. *Communications of the ACM 34:10 (October 1991)*, 78 – 92.

[28] K.-D. Schewe, B. Thalheim: Achieving Consistency in Active Databases. In J. Widom, S. Chakravarthy (eds.), *RIDE'94 - Fourth International Workshop on Research Issues in Data Engineering*, 71–76, IEEE Computer Society Press, 1994.

[29] E. Simon, P. Valduriez: Design and Implementation of an Extendible Integrity Subsystem. In *Proc. ACM SIGMOD Int. Conf. on Management of Data 1984* 9–17, 1984.

[30] D. Stemple, S. Mazumdar, T. Sheard: On the Modes and Meaning of Feedback to Transaction Designers. In U. Dayal, I. Traiger (eds.), *Proc. ACM SIGMOD Int. Conf. on Management of Data 1987, San Francisco*, 374–386, ACM Press, 1987.

[31] D. Toman, J. Chomicki: Implementing Temporal Integrity Constraints Using an Active DBMS. In J. Widom, S. Chakravarthy (eds.), *RIDE'94 - Fourth International Workshop on Research Issues in Data Engineering*, 87–95, IEEE Computer Society Press, 1994.

[32] S. D. Urban, A. P. Karadimce, R. B. Nannapaneni: The Implementation and Evaluation of Integrity Maintenance Rules in an Object-Oriented Database In *Eighth International Conference on Data Engineering - 1992*, IEEE Computer Society Press, 565–572, 1992

[33] A. I. Wasserman: Behavior and Scenarios in Object-Oriented Development. *Journal of Object-Oriented Programming 4:9 (February 1992)*, 61–64.

[34] J. Widom (ed.): Special Issue on Database Constraint Management. *Data Engineering Bulletin 17:2 (June 1994).*

[35] J. Widom, R. J. Cochrane, B. Lindsay: Implementing Set-Oriented Production Rules as an Extension to Starburst. In G. M. Lohmann, A. Sernadas, R. Camps (eds.), *Proceedings of the 17th Int. Conf. on Very Large Data Bases - 1991*, Morgan Kaufmann Publishers, 1991.

[36] J. Widom, S. J. Finkelstein: Set-Oriented Production Rules in Relational Database Systems. In H. Garcia-Molina, H. V. E. Jagadish (eds.), *Proc. ACM SIGMOD Int. Conf. on Management of Data 1990*, 259–270, ACM Press, 1990.

[37] R. Wieringa, J.-J. Meyer, H. Weigand: Specifying Dynamic and Deontic Integrity Constraints. *Data & Knowledge Engineering 4:2 (August 1989)*, 157–189.

[38] H. Zhou: *Überwachung dynamischer Integritätsbedingungen in vergangenheitsbezogener und gemischter temporaler Logik.* (Monitoring Dynamic Integrity Constraints in Past and Mixed Temporal Logic, in German) Technical Report 01/94, Institut für Informatik, Universität Hannover, February 1994.

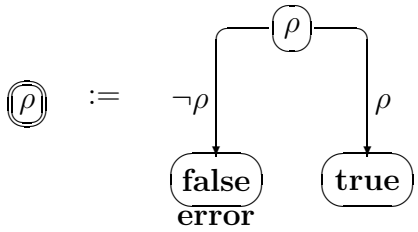
Appendix: Construction of Transition Graphs

Given a temporal formula φ, the following algorithm constructs a *deterministic transition graph* for φ in a bottom-up manner corresponding to the composition of φ from subformulas: The graph is built by stepwise composition or manipulation of smaller graphs for subformulas. Essentially, the algorithm defines such operators on transition graphs that correspond to the logical and temporal operators.

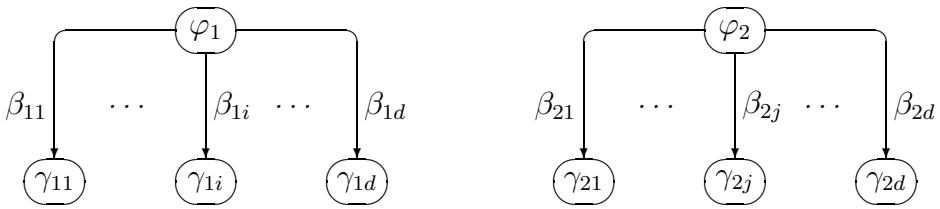
Algorithm 27 *The transition graph for φ, denoted in the diagrams by a double-circled node labelled φ, is constructed inductively according to the structure of φ:*

(0) For $\varphi \equiv$ **true** *or $\varphi \equiv$* **false** *we simply define:*

(Loops at nodes labelled with **true** *or* **false** *will be omitted in the sequel.)*
For other nontemporal formulas ρ:



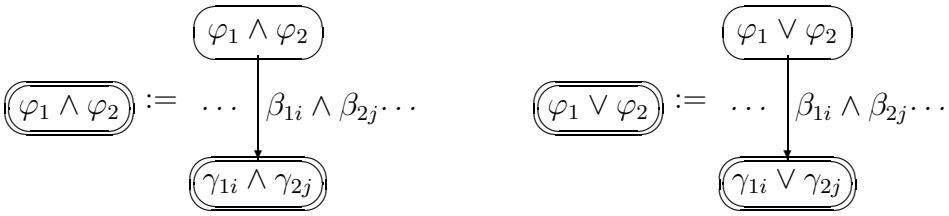
(1ff.) Now let φ be composed of a logical or temporal operator and formulae φ_1, φ_2. The following system of (recursive) rules defines corresponding operators on standard transition graphs. We inductively assume that such graphs have already been constructed for φ_1 and φ_2. Let the outgoing edges of their initial nodes and the target nodes be labelled as follows:

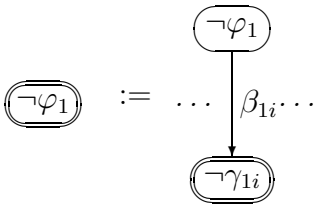


(Note that a target node may coincide with the source node.)

In particular, we have transition graphs for the γ-formulas, too: the subgraphs reachable from the corresponding labelled nodes.

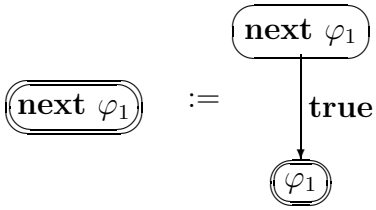
(1) In the case $\varphi \equiv \varphi_1 \wedge \varphi_2$, a node with label "$\varphi_1 \wedge \varphi_2$" is generated which has an outgoing edge labelled "$\beta_{1i} \wedge \beta_{2j}$" leading into the transition graph for $\gamma_{1i} \wedge \gamma_{2j}$ for each pair $(i,j), i = 1,..,d/\ j = 1,..,e$. That graph can in turn be constructed by applying this rule recursively. (For $\varphi \equiv \varphi_1 \vee \varphi_2$ and $\neg\varphi$ by analogy.) We abbreviate these rules to following diagrams:



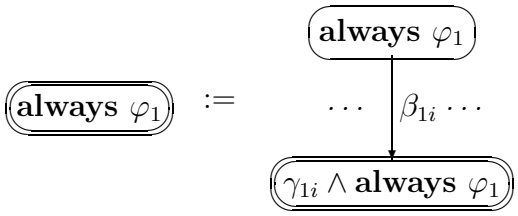


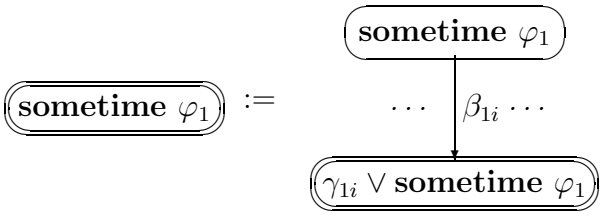
Rules for all other cases are given by diagrams only:

(2)

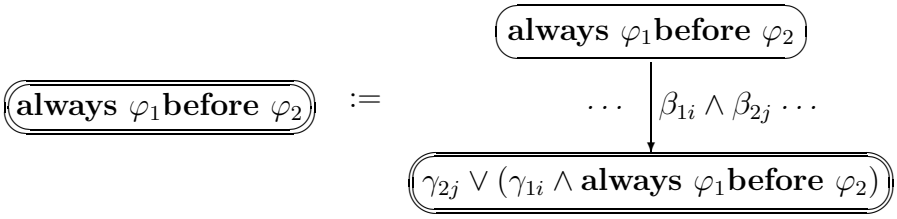


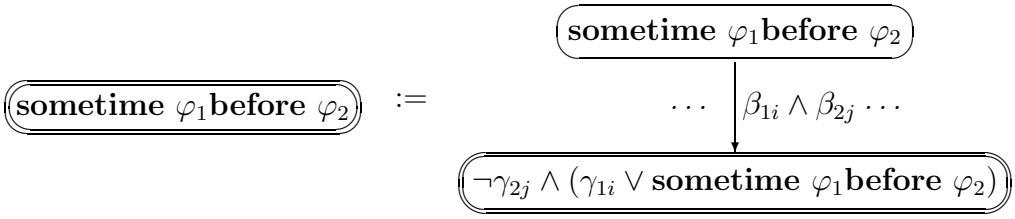
(3)

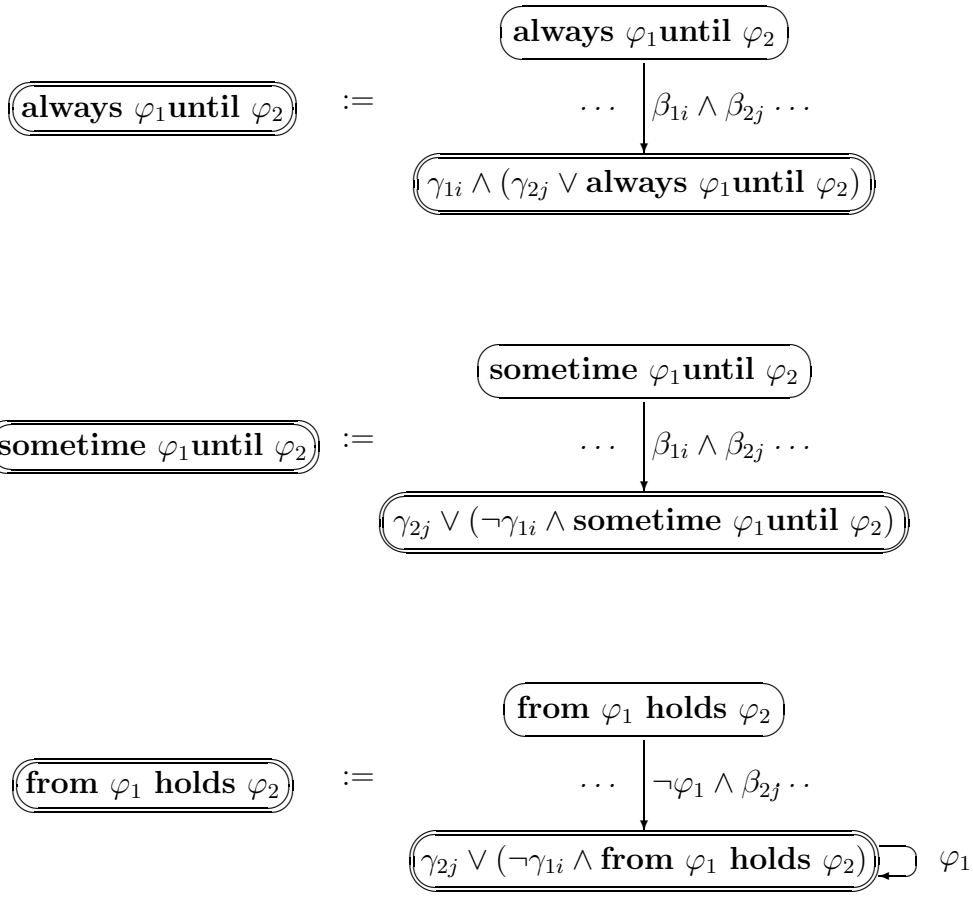

always φ_1
always φ_1
:=
$\cdots$
β_{1i} $\cdots$
$\gamma_{1i} \wedge$ always φ_1


sometime φ_1
sometime φ_1
:=
$\cdots$
β_{1i} $\cdots$
$\gamma_{1i} \vee$ sometime φ_1

(4)


always $\varphi_1$before φ_2
always $\varphi_1$before φ_2
:=
$\cdots$
$\beta_{1i} \wedge \beta_{2j}$ $\cdots$
$\gamma_{2j} \vee (\gamma_{1i} \wedge$ always $\varphi_1$before $\varphi_2)$


sometime $\varphi_1$before φ_2
sometime $\varphi_1$before φ_2
:=
$\cdots$
$\beta_{1i} \wedge \beta_{2j}$ $\cdots$
$\neg\gamma_{2j} \wedge (\gamma_{1i} \vee$ sometime $\varphi_1$before $\varphi_2)$



The latter rule applies for nontemporal φ_1 only. In the general case, **from** φ_1 **holds** φ_2 *must be reduced to the equivalent* **always** $(\neg\varphi_1 \vee \varphi_2)$ **until** φ_1.

always-from φ_1 **holds** φ_2 *can be equivalently replaced by*

$$(\varphi_1 \;\Rightarrow\; \varphi_2) \wedge \textbf{always}\ (\neg\varphi_1 \;\Rightarrow\; \textbf{from}\ \varphi_1\ \textbf{holds}\ \varphi_2).$$

Even the direct recursions above, i.e. applications of operators to the same arguments like in "$\gamma_{1i} \wedge$ **always** φ_1" are well-defined, since any construction of one new "level" of the graph needs to know only the outermost level of the argument graph, which is just defined by the respective rule.

During the construction steps, the following meta-rules must always be observed: Only such edges are included whose labels cannot be transformed into **false** by means of propositional laws. Each new node or edge label is simplified according to propositional equivalences as far as possible; then outgoing edges (of the same node) and nodes with propositionally equivalent labels are merged. (Node merging includes appropriate edge changes.) For these manipulations, the entire subgraph constructed so

far has to be taken into consideration as a global context. — Finally, the error node can be deleted.

Proposition 28 The algorithm above terminates and constructs a correct and deterministic transition graph for each propositional-temporal formula φ (compare Def. 10, 12).

Hints to the proof ([17,18]): All properties can be shown by induction on formula structure and thus on application of rules in the algorithm: Termination is guaranteed by the principle of earliest simplification, since only finitely many different node labels can be generated for every operator. Determinism is obviously propagated from basic graphs by using either original edge labels or conjunctions. Thus, given partitions are retained or refined. The correctness criterion can be concluded from laws of propositional logic, from commutability of next and logical operators, and in particular from the temporal recursion laws for the temporal operators. ∎

In order to determine the final nodes F in transition graphs concluding the graph construction, the following test function for node labels can be derived from the semantics of formulas in the empty state sequence (Def. 3):

Definition 29 *Let Φ be the set of temporal formulas and $\alpha, \varphi, \tau, \rho \in \Phi$, ρ being a nontemporal atom. The* finality function $\mathcal{L}$*:* $\Phi \to \{\mathbf{true}, \mathbf{false}\}$ *is recursively defined as follows:*

$$
\begin{aligned}
\mathcal{L}(\rho) &:= \rho \equiv \mathbf{true} \\
\mathcal{L}(\neg\varphi) &:= \neg\mathcal{L}(\varphi) \\
\mathcal{L}(\mathbf{next}\ \varphi) &:= \mathcal{L}(\varphi) \\
\mathcal{L}(\varphi \wedge \tau) &:= \mathcal{L}(\varphi) \wedge \mathcal{L}(\tau) \\
\mathcal{L}(\varphi \vee \tau) &:= \mathcal{L}(\varphi) \vee \mathcal{L}(\tau) \\
\mathcal{L}(\mathbf{always}\ \varphi) &:= \mathbf{true} \\
\mathcal{L}(\mathbf{sometime}\ \varphi) &:= \mathbf{false} \\
\mathcal{L}(\mathbf{always}\ \varphi\ \mathbf{before}\ \tau) &:= \mathbf{true} \\
\mathcal{L}(\mathbf{sometime}\ \varphi\ \mathbf{before}\ \tau) &:= \mathbf{false} \\
\mathcal{L}(\mathbf{always}\ \varphi\ \mathbf{until}\ \tau) &:= \mathbf{true} \\
\mathcal{L}(\mathbf{sometime}\ \varphi\ \mathbf{until}\ \tau) &:= \mathbf{false} \\
\mathcal{L}(\mathbf{from}\ \alpha\ \mathbf{holds}\ \varphi) &:= \mathbf{true}
\end{aligned}
$$

This function tests whether a formula is valid in the empty sequence or not.

Lemma 30 For any nontemporal formula ψ holds $\mathcal{L}(\psi) = \mathbf{true}$ iff $\underline{\lambda} \models \psi$.

Corollary 31 After applying the finality test to all node labels, a strongly correct graph (see Def. 12) has been obtained.