Invariance, Maintenance and other declarative objectives of triggers – a formal characterization of active databases

Mutsumi Nakamura
Department of CSE
University of Texas at Arlington
Arlington, TX 76019, USA
nakamura@cse.uta.edu

Chitta Baral
Department of CSE
Arizona State University
Tempe, AZ 85287, USA
chitta@asu.edu

Abstract

In this paper we take steps towards a systematic design of active features in an active database. We propose having declarative specifications that specify the objective of an active database and formulate the correctness of triggers with respect to such specifications. In the process we distinguish between the notions of 'invariance' and 'maintenance' and propose four different classes of specification constraints. We also propose three different types of triggers with distinct purposes and show through the analysis of an example from the literature, the correspondence between these trigger types and the specification classes. Finally, we briefly introduce the notion of k-maintenance that is important from the perspective of a reactive (active database) system.

1 Introduction and Motivation

Many commercial database systems (such as Oracle, Sybase, IBM's DB2-V2, etc.) and the database standard SQL3 incorporate *active features* – namely *constraints* (also referred to as *integrity constraints*) and *triggers*. Due to these active features explicit update requests to the database may have several consequences from the request being refused (as it may violate 'integrity constraints'), to the request being fulfilled with slight changes (as modified through 'before triggers'), to additional changes triggered by cascade deletes and inserts used in the processing of some constraints and/or firing of 'after triggers'.

Although originally, integrity constraints were thought of as *declarative* constraints about database states and defined which database states were valid and which were not, with the presence of cascade operations in the SQL3 constraints and the use of after triggers to maintain the integrity of the data, there is currently little tradition (except in [CF97] and few other cases) of using or following standard software engineering practices of separating specification from implementation when designing and developing active databases. This means that often active database developers do not even *specify* what the purpose of the active features of their database are. Thus there is no way to *verify* the correctness of the active features. We believe this is one of the reasons why many companies balk at using the active features of a database.

Our goal in this paper is to take steps towards developing a systematic approach to the design of active databases. In the process we will develop several language constructs that can be used in specifying the purpose of an active database; formulate the correctness of the procedural triggers with respect to declarative specifications; and develop guidelines that match the procedural aspects with the declarative aspects.

One major hindrance in this pursuit has been the multitude of syntax and semantics (and their complexity) associated with the various different implementation of active rules [WC96, Pat98] and the complexity of their semantics. In this paper we will follow the SQL3 standard (and the DB2-V2 implementation) to some extent and make certain simplifications.

1.1 The declarative notions

The basic goal of the active features of a database is to constrain the evolution of the database. Based on analyzing a large class of active database examples, we have identified four kind of constraints: *state invariance constraints; state maintenance constraints (or quiescent state constraints); trajectory invariance constraints; and trajectory maintenance constraints.*

In the above constraints there are two dimensions: (i) state vs trajectory (ii) invariance vs maintenance. Intuitively, in state constraints we are concerned about the integrity about particular states, while the trajectory constraints focus on the trajectory of evolution of the database. On the other hand, invariance constraints worry about all states of the database, while the maintenance constraints focus only on the quiescent states.

Definition 1 (State Constraints) *[ADA93]* A *state constraint* γ_s on a database scheme R, is a function that associates with each database r of R a boolean value $\gamma_s(r)$. A database r of R is said to *satisfy* γ_s if $\gamma_s(r)$ is true and is said to *violate* γ_s if $\gamma_s(r)$ is false. In the former case, it is also said that γ_s holds in r. A database r is said to satisfy a set of state constraints if it satisfies each element of the set. □

Definition 2 (Trajectory Constraints) A *trajectory constraint* γ_t on a database scheme R, is a function that associates with each database sequence Υ of R a boolean value $\gamma_t(\Upsilon)$. A database sequence Υ of R is said to *satisfy* γ_t if $\gamma_t(\Upsilon)$ is true and is said to *violate* γ_t if $\gamma_t(\Upsilon)$ is false. In the former case, it is also said that γ_t holds in Υ. A database sequence Υ is said to satisfy a set of trajectory constraints if it satisfies each element of the set. □

Often static integrity constraints are expressed through sentences in propositional logic or first-order predicate calculus while we need temporal operators to express trajectory constraints. We further discuss this in Section 3.

1.2 The procedural features of an active database

In SQL3 (and DB2-V2) the active features are: *Constraints; Before triggers; and After triggers.*

The constraints in DB2-V2 are of the kinds: NOT NULL constraints, column defaults, unique indexes, check constraints, primary key constraints, and foreign key constraints. Among these, the NOT NULL constraints, unique indexes, check constraints, primary key constraints and some of the foreign key constraints (with NO ACTION or RESTRICT in the action part) *refuse updates* that violate the constraints. These correspond to the state invariance constraints mentioned in the previous section.

On the other hand column default constraints and the foreign key constraints with CASCADE or SET NULL in the action part accept the updates but make additional changes. The former correspond to the state invariance constraints, while the later correspond to the state maintenance constraints.

The before triggers act on the update request directly (instead of the updated database) and modify it if necessary while the after triggers are triggered by the update request and can either refuse the update (through a rollback) or force additional changes. Here, the former can implement state and trajectory invariance constraints, while the later can implement any of the four types of constraints.

From the above analysis, it seems that certain specifications such as a state maintenance constraint can be implemented in multiple ways, through a DB2-V2 constraint or through after triggers. But the trigger processing architecture treats DB2-V2 constraints very differently from after triggers. Thus it becomes very difficult to formulate and verify the correctness of the DB2-V2 (or SQL3) active features with respect to specifications mentioned in Section 1.1.

We propose a different class of active features that are close to the SQL3 features, but that are *distinct* in terms of their goals. Our class consists of three kind of procedural features (triggers): *refusal triggers*, *wrapper triggers*, and *maintenance triggers.*

Intuitively, *the refusal triggers* when triggered refuse the update that caused the triggering. Thus refusal triggers can express not only after triggers with refuse actions, but also NOT NULL constraints, unique indexes, check constraints, primary key constraints, and foreign key constraints with NO ACTION or RESTRICT in the action part. The *wrapper triggers*, wrap the update request by additional changes and thus can express both before triggers and column default constraints. The *maintenance triggers*[1] trigger additional updates and thus can express both after triggers with similar purpose, and foreign key constraints with CASCADE or SET NULL in the action part.

[1] In Section 4 we will further divide maintenance triggers to two classes: short-term and long-term. This becomes necessary when we need to worry about reactive response to update requests.

Our division of the triggers into the above three classes makes them distinct in terms of what they set out to achieve. This is different from the active features in DB2-V2 and SQL3 where there is overlapping of goals making it difficult in designing active databases and formulating their correctness.

2 Actions, Events and Triggers

In this section we describe the necessary mechanism for reasoning about actions and events which we will then use to formulate correctness of triggers with respect to declarative specifications.

2.1 Actions and effects

Intuitively, an action when executed in a world changes the state of the world. In databases, an action can take several meanings; from the basic *insert*, *delete* and *update* actions to SQL update statements. In this paper by an action we will usually refer to an uninterruptable transaction.

To specify the effects of an action on a database we borrow constructs from the specification language $\mathcal{A}$ [GL93] and our earlier work in [BL96, BLT97]. In the following by a fluent we will mean a database fact, and by a fluent literal we will mean either a database fact or its negation. Effects of actions are specified through effect axioms of the following form:

$$a(\overline{X}) \textbf{ causes } f(\overline{Y}) \textbf{ if } p_1(\overline{X_1}), \ldots, p_n(\overline{X_n}) \tag{2.1}$$

where $a(\overline{X})$ is an action and $f(\overline{Y}), p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$ are fluent literals ($n \geq 0$). $p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$ are called *preconditions*. The intuitive meaning of (2.1) is that in any state of the active database execution in which $p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$ are true, the execution of the action $a(\overline{X})$ causes $f(\overline{Y})$ to be true in the resulting state. A word of caution is needed regarding the *safeness* [Ull88] of variables in the *causal law*. The preconditions $p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$ will be evaluated as regular queries in the database and $a(\overline{X})$ is an action that could be invoked by a user or an active rule. Thus, variables appearing in $\overline{Y}$ or in any negated fluent in the preconditions must also appear in one of the positive fluents in the precondition. If there are variables in $\overline{X}$ that do not appear in any of the positive fluents in the preconditions these arguments must be ground at the time of the invocation of the action, otherwise there will be an error in the execution. Moreover, the variables are *schema variables*, and intuitively an effect axiom with variables represents the set of ground effect axioms where the variables are replaced by ground terms in the domain.

Two effect axioms with preconditions $p_1, \ldots, p_n$ and $q_1, \ldots, q_m$ respectively are said to be *contradictory* if they describe the effect of the same action a on complementary fs, and $\{p_1, \ldots, p_n\} \cap \{\overline{q_1}, \ldots, \overline{q_m}\} = \emptyset$

A *state* is a set of fluent names. Given a fluent name f and a state σ, we say that f *holds* in σ if $f \in \sigma$; $\neg f$ *holds* in σ if $f \notin \sigma$. A *transition function* is a mapping Φ of the set of pairs (a, σ), where a is an action name and σ is a state, into the set of states.

A collection of effect axioms (EA) for various actions in our world – with no contradictory effect axioms in them, define a transition function from the set of actions and the set of database states to the set of database states.

For every action a and every state σ,

$$\Phi(a, \sigma) = (\sigma \cup \sigma') \setminus \sigma'',$$

where σ' (σ'') is the set of fluent names f such that EA includes an effect proposition describing the effect of action a on f (respectively, $\neg f$) whose preconditions hold in σ.

We now show how the effect of simple actions such as *insert*, *delete* and *update* can be specified using effect axioms.

$$insert(R(t)) \textbf{ causes } R(t) \tag{2.2}$$

$$delete(R(t)) \textbf{ causes } \neg R(t) \tag{2.3}$$

$$update(R(t), R(t')) \textbf{ causes } R(t') \textbf{ if } R(t) \tag{2.4}$$

$$update(R(t), R(t')) \textbf{ causes } \neg R(t) \textbf{ if } R(t) \tag{2.5}$$

We now show how we can specify the effect of actions corresponding to more complex transactions:

Example 1 Consider another transaction a_2 from [Cha96]:

UPDATE parts
SET qonorder = qonhand,
    qonhand = qonorder
WHERE partno = 'P207';

Its effects can be described in our language through the following effect propositions:

a_2 **causes** $parts(P207, Descr, qonhand, qonorder)$ **if**
$parts(P207, Descr, qonorder, qonhand)$

a_2 **causes** $\neg parts(P207, Descr, qonhand, qonorder)$ **if**
$parts(P207, Descr, qonhand, qonorder)$ □

To reason about the effect of a sequence of actions on a database σ, we need to extend the function Φ, to allow sequence of actions as its first parameter. This extension is defined as follows:

- $\Phi([], \sigma) = \sigma$, and
- $\Phi([\alpha|a], \sigma) = \Phi(a, \Phi(\alpha, \sigma))$.

2.2 Events and ECA rules

Triggers (or active rules) in active databases are normally [WC96] represented as a triple consisting of *events, conditions and actions*. In most active database architectures, the sequence of actions that have been executed since the last evaluation point are evaluated to decide on what events have taken place. These events together with the valuation of the condition with respect to the current database state determine whether a particular ECA active rule should be triggered or not.

Different active databases allow different event sets and have different ways of evaluating the events. In the simplest case, the events can be the set of *inserts* and *deletes* explicitly performed by the last action. On the other hand, in Starburst [Wid96] events are defined in terms of the net effects of a sequence of transitions. To allow the flexibility of defining a set of events and computing them from a sequence of actions we use the notion of event definitions from [BLT97].

An *event definition* proposition is an expression of the form:

$$e(\overline{X}) \textbf{ after } a(\overline{W}) \textbf{ if } e_1(\overline{Y_1}), \ldots, e_m(\overline{Y_m}), q_1(\overline{Z_1}), \ldots, q_n(\overline{Z_n}) \tag{2.6}$$

where $e(\overline{X}), e_1(\overline{Y_1}), \ldots, e_m(\overline{Y_m})$ are event literals[2] and $q_1(\overline{Z_1}), \ldots, q_n(\overline{Z_n})$ are fluent literals. This proposition says that the execution of the action $a(\overline{W})$ ordered in a state in which each of the fluent literals $q_i(\overline{Z_i})$ is true *and* each of the event literals $e_j(\overline{Y_j})$ is true generates the event literal $e(\overline{X})$ if the event literal is positive, or removes the event from the set of current events if the event literal $e(\overline{X})$ is negative. If the execution is ordered in a state in which some of the $q_i(\overline{Z_i})$ or $e_j(\overline{Y_j})$ does not hold then (2.6) has no effect. Each of the (schema) variables appearing in $\overline{X}$ or in a negated event or fluent literals, has to appear either in $\overline{W}$ or in a positive event/fluent literal.

The default assumption is that the event *persists* from one state to another, with two possible exceptions: either the event is *consumed* by an active rule (see below), or the event is removed by an action based on the specification of an event definition. For example, if we have an expression $\neg e_1$ **after** a_1, the execution of the action a_1 will cause the event e_1 not to be present in the resulting state. Hence, the meaning of "an event is true in a given state" is: the event was induced (i.e. generated) in some state prior to the given one and the event persisted, or the event was induced by an execution of an action in the previous state.

Example 2 (Events in Starburst) In Starburst net effects (or events) are expressed in words through the following conditions:

- If a tuple is inserted and then updated, it is considered an insertion of the updated tuple.

[2] Like fluent literals, an event literal is an event or its negation.

- If a tuple is updated and then deleted, it is considered as a deletion of the original tuple.
- If a tuple is updated more than once, it is considered as an update from the original value to the newest value.
- If a tuple is inserted and then deleted, it is not considered in the net effect at all.

These four premises can be encoded through event definitions as follows:

$$\begin{array}{rl}
e_add(H) & \textbf{after } \ upd(G,H) \textbf{ if } \ e_add(G) \\
\neg e_add(G) & \textbf{after } \ upd(G,H) \textbf{ if } \ e_add(G) \\
e_del(G) & \textbf{after } \ del(F) \textbf{ if } \ e_upd(G,F) \\
\neg e_upd(G,F) & \textbf{after } \ del(F) \textbf{ if } \ e_upd(G,F) \\
e_upd(G,I) & \textbf{after } \ upd(H,I) \textbf{ if } \ e_upd(G,H) \\
\neg e_upd(G,H) & \textbf{after } \ upd(H,I) \textbf{ if } \ e_upd(G,H) \\
\neg e_add(G) & \textbf{after } \ del(G) \textbf{ if } \ e_add(G)
\end{array} \tag{2.7}$$

In the above example, at the first glance it appears that our notation is more verbose than the original rules. For each of the first three rules we needed two event definition propositions. This is because *we assume that events have inertia.* This assumption actually cuts down in writing individual event definition propositions encoding the persistence of each event due to actions that do not affect it. For example we do not need to explicitly write:

$$e\_add(H) \textbf{ after } \ del(G) \textbf{ if } \ e\_add(H), H \neq G$$

We characterize events using the function Ξ whose input is a set of events, a state, and an action and the output is a set of events. More formally, let $E^+(E,\sigma,a) = \{$ e : there is an event definition proposition of the form e **after** a **if** $e_1,\ldots e_m, q_1,\ldots,q_n$ where $e_1,\ldots,e_m$ hold in E and $q_1,\ldots,q_n$ hold in σ $\}$; and let $E^-(E,\sigma,a) = \{$ e : there is an event definition proposition of the form $\neg e$ **after** a **if** $e_1,\ldots e_m, q_1,\ldots,q_n$ where $e_1,\ldots,e_m$ hold in E and $q_1,\ldots,q_n$ hold in σ $\}$. $\Xi(E,\sigma,a)$ is then defined as follows:

$$\Xi(E,\sigma,a) = (E \cup E^+(E,\sigma,a)) \setminus E^-(E,\sigma,a)$$

To be able to compute events with respect to a sequence of actions we extend Ξ as follows:

- $\Xi(E,\sigma,[\,]) = E$, and
- $\Xi(E,\sigma,[\alpha|a]) = \Xi(\Xi(E,\sigma,\alpha),\Phi(\alpha,\sigma),a)$.

2.3 Characterizing database evolution due to ECA rules

As mentioned in Section 1.2, we have three kinds of triggers: wrapper triggers, refusal triggers and maintenance triggers. We represent each of them through ECA rules but distinguish them by the action part. In wrapper triggers the action part is a *wrapping function* ω which maps an action sequence and a database state to an action sequence. Intuitively, for a single action a, by $\omega(a, \sigma) = a'$ we mean that a' is the action obtained by wrapping a with ω in state σ. In refusal triggers the action part is the special action $REFUSE$ and in maintenance triggers the action part could be an arbitrary sequence of actions. Thus an ECA rule is a triple $\langle e, c, \alpha \rangle$, where e is an event in our language, c is a temporal formula about the database history, and α is either a wrapping function, the special action REFUSE, or a sequence of actions. Often we will represent a single action as the ECA rule $\langle \emptyset, True, a \rangle$.

In this subsection our goal is to give a formal characterization of the evolution of a database due to a sequence of actions in presence of a set of ECA rules. In our characterization we strive to keep a balance between not making the semantics too complicated and at not losing expressibility. We now give an intuitive description of our characterization.

Intuitively, after the action sequence (with necessary modifications due to wrapper triggers) is executed the set of events corresponding to that sequence of actions are evaluated. Then the ECA rules that match with the events are identified. We assume (as in many implemented systems [Cha96]) that there is a total ordering among the ECA rules with the condition that refusal triggers have higher priority than maintenance triggers. Using this total ordering a priority list of the identified ECA rules is created. Then the condition parts of the ECA rules in the priority list are evaluated in the order of their priority and if the condition evaluates to be true, the action part is executed. Since the action part may trigger additional ECA rules, an important concern is how these ECA rules are assimilated into the already existing prioritized list of ECA rules. Two straightforward approaches are to view the list as a stack where newly triggered ECA rules are pushed onto the top of the stack, or to view the list as a queue where newly triggered ECA rules are put at the end of the queue. In both cases, among the newly added rules, the wrapper triggers have the highest priority, the refusal triggers have the second highest priority and the maintenance triggers have the lowest priority.

So after the execution of the action part of the currently considered ECA rule, the newly triggered ECA rules are put into the priority list and the evaluation of the ECA rules in the modified list are again done based on their priority. This loop of executing the action part of the currently chosen ECA rule, updating the list of ECA rules, and evaluating the list to find the next ECA rule, continues until the list is empty. During the execution when faced with a trigger whose action part is REFUSE, the database is rolled back.

We now formally define the function $\Psi(\sigma, \alpha, List)$, where σ is a database state, α is a sequence of actions and $List$ is a prioritized list of ECA rules that are yet to be processed, and the output of the function is a sequence of database states. Once we define this function, the evolution of a database state σ due to an action sequence α, can then be expressed by $\Psi(\sigma, \alpha, [\,])$. (For lack of space we only consider the simple case where there are no triggers with REFUSE in their action part.)

Definition 3 **[Evolution due to actions and triggers]**

1. $\Psi(\sigma, \alpha, List) = \sigma$ if σ is a state, α is an empty sequence, and $List = [\,]$.
2. $\Psi(\sigma, \alpha, List)$ is an empty list if σ is undefined.
3. $\Psi(\sigma, \alpha, List) = \sigma \circ \Upsilon$ if
   (a) $\sigma' = \Phi(\omega(\alpha), \sigma)$, where ω is the composition of the wrapping functions of the before triggers triggered by the events in $\Xi(\emptyset, \sigma, \alpha)$. (If there are no before triggers triggered by the events in $\Xi(\emptyset, \sigma, \alpha)$ then ω is the identity function; i.e., $\forall \alpha . \omega(\alpha) = \alpha$).
   (b) $List_1$ is the list obtained by adding the new ECA rules triggered by the events in $\Xi(\emptyset, \sigma, \omega(\alpha))$ to $List$ and *adjusting the priorities*,
   (c) eca is the ECA rule with the highest priority in the priority list $List_1$,
   (d) α' is the action part in eca,
   (e) $List_2 = List_1 \setminus \{eca\}$, and
   (f) $\Psi(\sigma', \alpha', List_2) = \Upsilon$. □

Because of the second condition above, when $\Phi(\omega(\alpha), \sigma)$ is undefined we obtain Υ as an empty list, and then $\Psi(\sigma, \alpha, List)$ is a sequence of length one with σ as the only element. Rollbacks can be accounted for by having an additional parameter in Ψ which stores the initial state, where the database should be rolled back to when a trigger with REFUSE in its action part is triggered.

2.4 Correctness of ECA rules

Our next step is to formally define when a set of ECA rules are correct with respect to invariant and maintenance constraints. For state maintenance constraints, intuitively, the correctness means that the ECA rules force the database to evolve in such a way that the final state that is reached is a state where all the state maintenance constraints are satisfied. For state invariant constraints, intuitively, the correctness means that the ECA rules force the database to evolve in such a way that the state invariance constraints are satisfied in all states of the trajectory.

Since our ultimate goal is to be able to use this definition to verify the correctness, we add another dimension to the definition: *the class of exogenous actions that we consider;* where exogenous actions are the actions that outside users are allowed to execute on the database. It should be noted that the action part of the ECA rules may have actions other than the exogenous actions.

We now formally define correctness with respect to state invariant and maintenance constraints.

Definition 4 Let Γ_{si} be a set of state invariant constraints, Γ_{sm} be a set of state maintenance constraints, A be a set of exogenous actions, and T be a set of ECA rules. We say T is correct with respect to $\Gamma_{si} \cup \Gamma_{sm}$ and A, if for all database states σ where the constraints in Γ_{si} and Γ_{sm} hold, and for all action sequences α consisting of exogenous actions from A,

- all the states in the sequence $\Psi(\sigma, \alpha, [\,])$ satisfy the constraints in Γ_{si}; and
- the last state of the evolution given by $\Psi(\sigma, \alpha, [\,])$ satisfies the constraints in Γ_{sm}. □

To expand the Definition 4 to define correctness with respect to trajectory constraints we need to consider a larger evolution window where the database evolves through several exogenous requests each consisting of a sequence of (exogenous) actions. For this we use the notation σ_α to denote the last state of the evolution given by $\Psi(\sigma, \alpha, [\,])$. We use the notation $\sigma_{(\alpha_1, \alpha_2)}$ to denote the last state of the evolution given by $\Psi(\sigma_{\alpha_1}, \alpha_2, [\,])$, and similarly define $\sigma_{(\alpha_1, \ldots, \alpha_i)}$.

Definition 5 Let Γ_{si} be a set of state invariant constraints, Γ_{sm} be a set of state maintenance constraints, Γ_{ti} be a set of trajectory invariant constraints, Γ_{tm} be a set of trajectory maintenance constraints, A be a set of exogenous actions, and T be a set of ECA rules. We say T is correct with respect to $\Gamma_{si} \cup \Gamma_{sm} \cup \Gamma_{ti} \cup \Gamma tm$ and A, if for all database states σ where the constraints in Γ_{si} and Γ_{sm} hold, and for all action sequences $\alpha_1, \ldots, \alpha_n$ consisting of exogenous actions from A,

- all the states in the sequences $\Psi(\sigma, \alpha_1, [\,])$, $\Psi(\sigma_{\alpha_1}, \alpha_2, [\,])$, $\ldots$, $\Psi(\sigma_{(\alpha_1, \ldots, \alpha_{n-1})}, \alpha_n, [\,])$ satisfy the constraints in Γ_{si};
- all the states $\sigma_{\alpha_1}, \ldots, \sigma_{(\alpha_1, \ldots, \alpha_n)}$ satisfy the constraints in Γ_{sm};
- the trajectory obtained by concatenating $\Psi(\sigma, \alpha_1, [\,])$ with $\Psi(\sigma_{\alpha_1}, \alpha_2, [\,])$, $\ldots$, $\Psi(\sigma_{(\alpha_1, \ldots, \alpha_{n-1})}, \alpha_n, [\,])$ satisfy the constraints in Γ_{ti}; and
- the trajectory $\sigma, \sigma_{\alpha_1}, \ldots, \sigma_{(\alpha_1, \ldots, \alpha_n)}$ satisfies the constraints in Γ_{tm}. □

Example 3 Consider the relational Schema:

$Employee(Emp\#, Name, Salary, Dept\#)$
$Dept(Dept\#, Mgr\#)$

We have two state maintenance constraints:
(i) If (e, n, s, d) is a tuple in *Employee* then there must be a tuple (d', m') in

Dept such that $d = d'$.
(ii) If (d, m) is a tuple in *Dept*, then there must be a tuple (e', n', s', d') in *Employee* such that $d = d'$ and $m = e'$

The only allowable exogenous action is $del(Employee(E, N, S, D))$.

The set of maintenance triggers that can be shown to be correct with respect to the above maintenance constraints and exogenous actions consists of the following trigger.

• For any Delete (e, n, s, d) from *Employee*, if (d, e) is a tuple in *Dept*, delete that tuple from *Dept* and delete all tuples of the form (e', n', s', d') from *Employee*, where $d = d'$. □

We can now make the formal claim that the above maintenance triggers are correct with respect to the above mentioned state maintenance constraints and exogenous actions.

3 Elaborating on our abstractions

In Section 1.1 we defined state constraints and trajectory constraints as boolean functions on database states and sequences of database states respectively. Our next concern is how to represent such functions parsimoniously. One approach is to use logical constructs. In this section we introduce several language constructs that we proposed to use in specifying state and trajectory constraints and show their use through examples.

We start with a description of the mail order business active database from [Cha96]. To save space and to make it readable without knowing the syntax of triggers in DB2-V2, we describe the triggers of this active database in words, and not in the syntax of DB2-V2.

3.1 The tables

The five tables that are mentioned in the database in [Cha96] and their attributes are:

Cust(C#, Cname, Caddr, Baldue, Creditlmt)
Suppl(S#, Sname, Saddr, Amtowed)
Inv(It#, Iname, S#, Qonhand, Unitsalpr, Qonorder, Unitorderpr, Orderthreshold, Minorder)
Purch(Orddate, Ordtime, S#, It#, Qordered, Dtrecvd, Qrcvd, Unitpr)
Sales(Sldate, Sltime, C#, It#, Qsold, Unitpr, Totalsale)

3.2 A subset of the triggers

Due to lack of space we only consider two of the eight triggers given in [Cha96], and identify the state and trajectory constraints corresponding to these triggers.

- (PT1: a wrapper trigger)
  When *inserting* into the *Purch table* modify the tuples (to be inserted)

so that for any It#, the values for S# and Unitpr are the values for S# and Unitorderpr for that It# in the Inv table. (Note that because of the constraints associated with the Purch table that allow Orddate and Ordtime to get the current date and time by default, It# and Qordered are the only pieces of information required to do insertions into the Purch table.)

- (PT2 – a maintenance trigger)
  After *inserting* an order for an It# to *Purch*, update the Inv table by increasing the Qonorder (in the tuple with that It#) by Qordered.

3.2.1 The corresponding constraints

We first list the constraints in a high level language that we developed and then explain the meaning of the constructs in this language.

- (C1) **ForAll** $It\#.\ Inv.S\# = Purch.S\#$ **is invariant**
- (C2) **ForAll** $It\#.\ Inv.Unitorderpr = Purch.Unitpr$ **is invariant**
- (C3) **newtuple** *Purch* **requires** Orddate = Currentdate and Ordtime = Currenttime
- (C4) **ForAll** $It\#.\ Purch.Sum(Qordered) - Purch.Sum(Qrcvd)$ $= Inv.Qonorder$ **is maintained**

Among the above constraints, the first two are state invariant constraints, the second is a trajectory invariant constraint, and the third is a state maintenance constraint. These constraints can be specified in first-order logic with temporal and aggregate constructs. We specify them using such constructs below with the assumption that all free variables are universally quantified and all the existentially quantified variables are denoted by underscores "_".

- (C1') $(Inv(It\#, \_, S_1, \_, \_, \_, \_, \_, \_) \wedge Purch(\_, \_, S_2, It\#, \_, \_, \_, \_))$ $\Rightarrow (S_1 = S_2)$
- (C2') $(Inv(It\#, \_, \_, \_, \_, \_, UOP_1, \_, \_) \wedge Purch(\_, \_, \_, It\#, \_, \_, \_, UP_2))$ $\Rightarrow (UOP_1 = UP_2)$
- (C3') $(\neg Purch(OD, OT, S\#, It\#, \_, \_, \_, \_) \wedge$ **nexttime** $(Purch(OD, OT, S\#, It\#, \_, \_, \_, \_))) \Rightarrow$ **nexttime** $(OD = date \wedge OT = time)$
- (C4.1') $R_1(It\#, Sum\_Qord) = {}_{It\#}\mathcal{G}_{\mathbf{Sum}\ Qordered}(Purch)$

  (C4.2') $R_2(It\#, Sum\_Qrcvd) = {}_{It\#}\mathcal{G}_{\mathbf{Sum}\ Qrcvd}(Purch)$

  (C4') $(quiescent \wedge R_1(It\#, Sum\_Qord) \wedge R_2(It\#, Sum\_Qrcvd) \wedge$ $Inv(It\#, \_, \_, \_, \_, Qonorder, \_, \_, \_)) \Rightarrow (Sum\_Qord - Sum\_Qrcvd = Qonorder)$

The first order formulas (C1') and (C2') are low level representations of the state invariant constraints (C1) and (C2) respectively. The temporal formula (C3') is a low level representation of the trajectory invariant constraints (C3) and the temporal operator **nexttime** in (C3') has the usual FTL (future temporal logic) [CT95] meaning. Next we have the formulas (C4.1'), (C4.2') containing grouping aggregation expressions using the notation[3] from the text book [SKS96], and (C4') which are a low level representation of the state maintenance constraint (C4). Note the difference between (C4') and (C1'-C2'). Since the former is a maintenance constraint, we use the proposition *quiescent* in the left hand side of the implication, meaning that the implication only holds in quiescent states. On the other hand the implications in (C1'-C2') must hold in all states.

Proposition 1 Let DB be the schema declaration in Section 3.1, and the only allowable exogenous action is 'Insert into Purch with Dtrecvd and Qrcvd as null, and Qordered as a positive value'. Then in the context of DB the set of triggers {PT1,PT2}, is correct w.r.t. the set of constraints {C1, C2, C3, C4}, and the above mentioned exogenous action. $\square$

4 Interrupting exogenous updates

So far we have (implicitly) assumed that if new exogenous update requests come in when the active database system is in the midst of processing ECA rules due to a previous exogenous update, the new requests are kept in hold until the processing (due to the previous update) comes to an end. Such an assumption is perhaps acceptable when the exogenous updates are not that frequent and/or trigger processing is not that time consuming, and *there is no guaranteed quality of service requirement.*

With the popularity of e-commerce where updates to the database would often be due to e-transactions over the web, companies may require a guaranteed quality of service requirement. In particular, they may require *immediate response to requests.* In such a case, it may be a good idea to partition *maintenance triggers* to two kinds *short term* and *long term*, with the idea that in order *to give reactive response to new update requests, processing of long term maintenance triggers may be postponed in favor of processing the new update request.*

[3] In this notation the general form is: ${}_{G_1,G_2,\ldots,G_n}\mathcal{G}_{F_1\ A_1,F_2\ A_2,\ldots,F_m\ A_m}(E)$, where E is any relational-algebra expression, $G_1,\ldots,G_n$ constitute a list of attributes on which to group, each F_i is an aggregate function, and each A_i is an attribute name. The meaning of the operation is defined as follows. The tuples in the result of expression E are partitioned into groups such that:

(i) All tuples in a group have the same values for $G_1,\ldots,G_n$.
(ii) Tuples in different groups have different values for $G_1,\ldots,G_n$.

The groups now can be identified by the values of the attributes $G_1,\ldots,G_n$ of the relation, and for each group $(g_1,\ldots,g_n)$, the result has a tuple $(g_1,\ldots,g_n,a_1,\ldots,a_m)$ where, for each i, a_i is the result of applying the aggregate function F_i on the multi-set of values for the attribute A_i in the group.

The formulation of correctness in such a case becomes tricky, and we have made a small start in that direction. In this we only consider condition-action triggers, and consider all triggers to be *long term*. Before we get to our definition of correctness in such cases, we have the following notation. Let T be a set of condition-action triggers, and σ be a database state. By $\Xi_T(\sigma)$ we denote the action of the trigger which has the highest priority among the triggers whose conditions are satisfied in σ. We also have the following additional notations:

- $\Xi_T^0(\sigma) = \Xi_T(\sigma)$ and $\sigma_T^0 = \sigma$.
- $\Xi_T^{k+1}(\sigma) = \Xi_T(\sigma_T^{k+1})$ and $\sigma_T^{k+1} = \Phi(\Xi_T^K(\sigma), \sigma_T^k)$.

Definition 6 (k-maintenance) Let T be a set of condition-action triggers, Γ be a set of long term maintenance constraints, S be a set of states, and A be a set of allowable exogenous actions.

By $Closure(S, T, A)$ we denote the smallest set of states that is a superset of S and that satisfies the properties that if $\sigma \in S$, then for an exogenous action a from A, $\Phi(a, \sigma) \in S$, and $\Phi(\Xi_T(\sigma), \sigma)) \in S$.

We say T k-maintains the maintenance constraints Γ from S and A, if for each state σ in S, the sequence $\sigma_T^0, \ldots, \sigma_T^k$ satisfies Γ. $\square$

Intuitively, the notion of k-maintenance means that the active database system will get back to consistency (with respect to Γ) if it is given a window of opportunity of processing k triggers without any outside interference in terms of new update requests.

An important aspect of such a notion of k-maintainability is that in reactive (active database) systems, if we know that our system is k-maintainable, and each transition takes say t time units, then we can implement a transaction mechanism that will regulate the number of exogenous actions allowed per unit time to be $\frac{1}{k \times t}$. On the other hand, given a requirement that we must allow m requests (exogenous actions) per unit time, we can work backwards to determine the value of k, and then find a set of triggers to make the system k-maintainable.

5 Conclusion and future work

In this paper we have taken several steps towards the systematic design of active features in an active database. The main steps that we have taken are identifying a few constructs for specification, classifying triggers into distinct classes based on their purpose, linking the trigger classes with the specification classes, formulating correctness of triggers with respect to a given specification, elaborating our formulation through examples and briefly introducing the notion of k-maintainability.

Due to space limitations we were not able to detail our formulation (especially, the prioritization used in defining Ψ and the differentiation between row and statement triggers) and show the design methodology with respect to a large example. In the full version we will show how our formulation in this paper can be used in systematically developing the triggers for the complete example in [Cha96], starting from a specification which is not given in [Cha96]. Our main future work will be to develop composition methods and theorems so that given sets of triggers T_1 and T_2 that are correct with respect to specifications S_1 and S_2 respectively, we can construct triggers that are correct with respect to $S_1 \cup S_2$. We also plan to identify additional specification constructs with matching trigger sub-classes, and further elaborate on our notion of k-maintainability.

References

[ADA93] P. Atzeni and V. De Antonellis. *Relational database theory.* The Benjamin/Cummings publishing company, 1993.

[BL96] C. Baral and J. Lobo. Formal characterization of active databases. In *Proc. of International Workshop on Logic in Databases – LID'96 (LNCS 1154)*, pages 175–195, 1996.

[BLT97] C. Baral, J. Lobo, and G. Trajcevski. Formal characterization of active databases: Part II. In *DOOD 97*, 1997.

[CF97] S. Ceri and P. Fraternali. *Designing database applications with objects and rules – the IDEA methodology.* Addison-Wesley, 1997.

[Cha96] D. Chamberlin. *Using the new DB2: IBM's Object-relational database system.* Morgan Kaufmann, 1996.

[CT95] J. Chomicki and D. Toman. Implementing temporal integrity constraints using an active dbms. *IEEE transactions on knowledge and data engineering*, 1995.

[GL93] M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17(2,3,4):301–323, 1993.

[Pat98] N. Paton. *Active rules in database systems.* Springer-Verlag, 1998.

[SKS96] A. Silberschatz, H. Korth, and S. Sudershan. *Database System Concepts.* McGraw Hill, 3rd edition, 1996.

[Ull88] J. Ullman. *Principles of Database and Knowledge-base Systems, volume I.* Computer Science Press, 1988.

[WC96] J. Widom and S Ceri, editors. *Active Database Systems - Triggers and Rules for advanced database processing.* Morgan Kaufmann, 1996.

[Wid96] J. Widom. The Starbust rule system. In J. Widom and S Ceri, editors, *Active Database Systems*, pages 87–110. Morgan Kaufmann, 1996.