TEMPORALLY ACTIVE DATABASES := ACTIVE DATABASES + TIME

by

Alex Tuzhilin
Assistant Professor
Information Systems Department
Leonard N. Stern School of Business
New York University
New York, NY 10003

December 1991

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-91-43

Temporally Active Databases := Active Databases + Time

Abstract

A method of adding time to active databases is described in this paper. This is achieved by incorporating operators of temporal logic, temporal actions, and different temporal clauses into the Event-Condition-Action model of a rule. In addition, a temporal recognize-act cycle is described and new temporal conflict resolution strategies are proposed. A conflict avoidance strategy for temporal rules is also described.

1 Introduction

There has been much work done recently on studying active databases [dMS88, MD89, WF90, SJGP90, HCKW90, GJ91, SPAM91]. However, few of the existing proposals deal with the temporal domain. In the initial description of the HiPAC project [Dea88], the importance of supporting temporal constructs is explicitly stated. However, in the subsequent description of HiPAC system [MD89], the authors do not specify how to incorporate time into their model. Another active database, Ode [GJ91], supports timed triggers. Once activated, a timed trigger must fire within some time period; otherwise a timeout action, if any, is executed. Nevertheless, both systems provide only initial approaches to the subject of incorporating time in active databases.

On the other hand, many applications of active databases deal with the temporal domain. For example, in a banking application, we may need to state a rule that if a customer deposits an out-of-state check to his/her account, it cannot be withdrawn for 7 working days. Also in a stock trading system, we may want to say that if a stock has been steadily declining in the past 5 hours, then it should be sold. In addition, we also believe that the support for time in active databases can be useful in manufacturing, communication, process control, and command and control applications.

In this paper, we study *temporally active databases.* We assume that rules contain predicates that change over time, and also include actions that occur over some periods of time. For example, the following statement

> after a customer deposited a check to his/her bank account and if the customer had a good banking record in the past, then clear the check within the next 24 hours.

can be expressed as a temporal rule:

after	**insert** deposited_checks(customer, check)
if	not **sometime_past** bad_check(customer)
then	**insert_sometime** (within 24_hours) cleared_checks(customer, check)

where **sometime_past** `P(x)` is a *temporal operator* that is true when the *temporal predicate* `P(x)` was true sometime in the past, and **insert_sometime** `(within T) P(x)` is a *temporal action* that says that `x` is inserted in predicate `P` within the next `T` time units. This rule says that if the **insert** operation occurred on the relation `deposited_checks` in the *past*, and if the customer has not written bad checks in the past, then the **insert** operation will occur on the relation `cleared_checks` within the next 24 hours.

As illustrated by this example, we consider three temporal categories: predicates, events and actions. *Temporal predicates*, such as `deposited_checks` and `cleared_checks`, change over time, and their instances form a temporal database [CW83, Sno87, Gad88, TC90]. *Temporal events*, such as **insert** `deposited_checks(customer, check)`, occur *instantaneously* in time and correspond to the events in active non-temporal databases [dMS88, MD89, WF90, SJGP90, GJ91, SPAM91]. However, temporal actions differ from the actions in the non-temporal case. A *temporal action* is an update operation with some temporal constraint attached to it. For example, the action **insert_sometime** `(within 24_hours) cleared_checks(customer, check)` inserts the tuple `(customer, check)` into relation `cleared_checks` *sometime* within the next 24 hours, and the action **insert_and_keep** `(for T = 6_months) CD(customer, account, amount)` inserts a new certificate of deposit in the `CD` relation and *keeps* it there for six months (this means that no deletions of the newly inserted tuple can occur for six months). The exact semantics of temporal actions will be defined in Section 4.

In the previous example, the temporal clause **after** says that the rule can fire only *after* some temporal event or action occurred in the past, e.g. after the tuple `(customer, check)` has been inserted into the predicate `deposited_checks`.

In general, temporal rules check if certain conditions hold at present or held in the past, and also if certain temporal events or actions happen at present or occurred in the past. If all the conditions stated in the rule are true then the temporal rule "fires" and schedules some temporal actions in the future. By allowing temporal actions and operators of temporal logic [Kro87] to appear in the antecedent part of a rule and by supporting additional temporal clauses, such as **before**, **after**, and **while**, we extend the traditional ECA model of a rule [MD89] to the *Action-Event-Condition-Action (AECA)* model of a temporal rule in active databases.

We make the following contributions in this paper. First, we extend the ECA model of an active rule to incorporate time. This is achieved by supporting temporal operators, temporal actions, and **before**, **after**, and **while** clauses in the antecedent part of a rule, and temporal actions in the consequent part of a rule. Second, we describe a temporal recognize-act cycle including some temporal conflict resolution strategies and a method of compressing relevant parts of the past database history to the present. Finally, we describe a method that avoids conflicts among rules for the temporal conflict resolution strategies presented in the paper.

We are interested in the *real time* active databases. This means that it is generally not known when an update operation will actually occur once an update is issued. For example, it is not known when a customer record will be updated by an ATM machine because other users are trying to access database concurrently. This gives rise to a hard problem of meeting real-time temporal constraints, such as assuring that the customer record will be updated within 5 seconds.

Since the main objective of this paper is to describe the syntax and the semantics of temporally active rules, we do not address the problem of real-time scheduling of temporal actions to meet various temporal constraints. Therefore, we make a simplifying assumption that all the database updates occur *instantaneously.* In the ATM application, this amounts to an instantaneous update of the customer record assuming there are no conflicting actions scheduled before. This assumption is valid if times specified in temporal actions are much greater than an average time it takes to do an update. For example, if a temporal action deposits a check into a bank account and keeps it for 3 business days, then the amount of time it takes to perform the deposit transaction (usually a few seconds) is negligible in comparison to three days. Furthermore, the assumption is also valid if we deal with a single user database that does not require concurrency control. In this case, a database is usually updated faster than in the case when a concurrency control mechanism is in place.

Another problem which we do not address in the paper is the issue of integration of transactions and temporal rules. Since both transactions and rules occur in time, it becomes an interesting problem how to integrate them in a coherent manner. We leave the problems of real-time scheduling and transaction support as a topic of future research and will briefly touch upon them in Section 7.

2 Background: Some Concepts from Temporal Logic

The syntax of a predicate temporal logic is obtained from the first-order logic by adding various future temporal operators such as **sometime_future** ($\diamond$), **always_future** ($\square$), **next** ($\circ$), **until** and their past "mirror" images **sometime_past** ($\star$), **always_past**($\blacksquare$), **previous** ($\odot$), and **since** to its syntax [Kro87]. For example, **sometime_future** A is true now, if A is true at *some* time in the future, and **always_future** A is true now if A is *always* true in the future. Note that function symbols are allowed in temporal logic formulas since they are based on first-order logic. The temporal logic consisting of the temporal operators listed above is called the *US* temporal logic [GM91][1].

However, we will also consider other temporal operators in this paper, such as **before**, **after**, **while** [Kro87], and *bounded* temporal operators [Tuz91b]. The meaning of some of the bounded temporal operators is presented in Fig. 1[2]. Furthermore, we will consider several *derived* bounded operators that are obtained from the bounded operators by making specific assumptions about the bounds for these operators. The meaning of some of the derived operators for the bounded operator **sometime_future** is presented in Fig. 2. The meanings of derived operators for bounded operators **sometime_past**, **always_future**, and **always_past** is defined similarly to the meanings of derived operators for **sometime_future** presented in Fig. 2. Furthermore, it follows from the

[1] It stands for the operators *Until* and *Since*.

[2] Note that times T_1 and T_2 in the second operator in Fig. 1 are reversed because we assume that T_2 occurred *before* T_1 in the past.

sometime_future (from T_1 to T_2) A :	is true now if A will be true at some time in the future between times T_1 and T_2
sometime_past (from T_1 to T_2) A :	is true now if A was true at some time in the past between times T_2 and T_1
always_future (from T_1 to T_2) A :	is true now if A will always be true in the future between times T_1 and T_2
always_past (from T_1 to T_2) A :	is true now if A was always true in the past between times T_2 and T_1

Figure 1: Bounded Operators of Temporal Logic.

sometime_future (for T) A :	**sometime_future** (from 0 to T) A
sometime_future A :	**sometime_future** (for ∞) A
sometime_future (at T) A :	**sometime_future** (from T to T) A
sometime_future (at next) A :	**sometime_future** (at 1) A

Figure 2: Derived Operators for the Bounded Operator **sometime_future**.

results in [Kro87] and [Kam68] that the temporal operators considered in this paper have the same expressive power as the operators of the *US* logic.

3 Action-Event-Condition-Action Model of a Temporal Rule

Rules in active databases are based on the Event-Condition-Action (ECA) model [MD89]. To incorporate time into the structure of a rule, we extend this model to the *Action-Event-Condition-Action (AECA)* model. To define the AECA model, we introduce some preliminary concepts first.

In active non-temporal databases, there are two primary types of actions: insertions and deletions[3]. Temporally active databases have a richer set of actions. If T_1 and T_2 are temporal variables, constants, or expressions, if $T_1 \leq T_2$, if *now* is the time when an action occurs, and if P is a temporal predicate, then we consider the following set of *basic* temporal actions:

1. **insert_and_keep** (from T_1 to T_2) $P(x_1, \ldots, x_n)$. This action says that the tuple $(x_1, \ldots, x_n)$

[3]We assume that an update can be defined a delete followed by an insert.

is inserted in predicate P at time $now + T_1$ and *is kept* there until the time $now + T_2$. This action guarantees that the tuple $(x_1, \ldots, x_n)$ cannot be removed from predicate P between the times $now + T_1$ and $now + T_2$.

2. **insert_sometime** (from T_1 to T_2) $P(x_1, \ldots, x_n)$. This action says that the tuple $(x_1, \ldots, x_n)$ is inserted in predicate P *sometime* between $now + T_1$ and $now + T_2$. It does not specify precisely at what time the insertion will take place. Different methods of insertion will be described in Section 4.2 when semantics of temporal rules will be defined.

3. **delete_and_keep** (from T_1 to T_2) $P(x_1, \ldots, x_n)$. This deletion action is similar to **insert_and_keep.** Tuple $(x_1, \ldots, x_n)$ is deleted from predicate P at time $now + T_1$, and it cannot be reinserted back until time $now + T_2$.

4. **delete_sometime** (from T_1 to T_2) $P(x_1, \ldots, x_n)$. This deletion action is similar to **insert_sometime**, but deals with deletions instead of insertions.

As was mentioned in the introduction, we make a simplifying assumption that all the insertions and deletions in the database happen *instantaneously*. For example, if the action **insert_and_keep** (from T_1 to T_2) $P(x_1, \ldots, x_n)$ is scheduled from time $now + T_1$ to time $now + T_2$, then we assume that the tuple $(x_1, \ldots, x_n)$ will *actually* be inserted into predicate P at time $now + T_1$ (and will stay there until $now + T_2$).

We derive additional temporal actions from these four basic actions as follows:

- **insert_and_keep** (for T) $P(x_1, \ldots, x_n)$ as **insert_and_keep** (from 0 to T) $P(x_1, \ldots, x_n)$
- **insert_and_keep** (forever) $P(x_1, \ldots, x_n)$ as **insert_and_keep** (from 0 to ∞) $P(x_1, \ldots, x_n)$; this temporal operator corresponds to the standard necessity operator $\Box$ of temporal logic
- **insert** (at T) $P(x_1, \ldots, x_n)$ as **insert_and_keep** (from T to T) $P(x_1, \ldots, x_n)$
- **insert** $P(x_1, \ldots, x_n)$ as **insert** (at 0) $P(x_1, \ldots, x_n)$
- **insert** (at next) $P(x_1, \ldots, x_n)$ as **insert** (at 1) $P(x_1, \ldots, x_n)$

We define other derived temporal actions, such as **insert_sometime** (for T) $P(x_1, \ldots, x_n)$, **insert_sometime** (forever) $P(x_1, \ldots, x_n)$, **delete_and_keep** (for T) $P(x_1, \ldots, x_n)$, **delete_and_keep** (forever) $P(x_1, \ldots, x_n)$, **delete** (at T) $P(x_1, \ldots, x_n)$, **delete** $P(x_1, \ldots, x_n)$, similarly to the derived insertion actions. Note that the action **insert_sometime** (forever) corresponds to the standard possibility operator $\diamond$ of temporal logic.

A rule in an active non-temporal database has an Event-Condition-Action structure [MD89] and does not support actions in its antecedent part. We extend this concept for temporally active rules to make actions depend not only on the events and conditions but also on other actions. We define a rule of the Action-Event-Condition-Action (AECA) type as

[**if**	conditions]
[**when**	events]
[**while**	actions]
[**before**	events | actions]
[**after**	events | actions]
then	combination of actions

where `conditions` is a conjunction of literals and past temporal literals, `events` is a conjunction of events, and `actions` is a conjunction of actions. We defined conditions and actions already; so, we define events now.

An *event* can be of two types. First, it can be a usual insert or delete event as defined in a non-temporal case [MD89, WF90, SJGP90], such as `insert P(x,y,z)` or `delete Q(x,y,z)`. Second, it can be a *beginning* or an *end* of a temporal action. For example, `begin.`**insert_and_keep** `(from` T_1 `to` T_2`)` $P(x_1, \ldots, x_n)$ is an event indicating that the action **insert_and_keep** `(from` T_1 `to` T_2`)` $P(x_1, \ldots, x_n)$ has just started. This event occurs at the time when the action begins, and it happens instantaneously, as all events do. Similarly, `end.`**insert_sometime** `(from` T_1 `to` T_2`)` $P(x_1, \ldots, x_n)$ is an event indicating that the action **insert_sometime** `(from` T_1 `to` T_2`)` $P(x_1, \ldots, x_n)$ has just finished. It occurs at the time when the action ends, and it also happens instantaneously. An example of an event associated with the end of a temporal action will be presented in Example 2.

Temporal clauses are divided into antecedent and consequent clauses. **If**, **when**, **before**, **while**, and **after** are antecedent clauses and **then** is a consequent clause. **If** clause tests if some conditions involving the present and the past instances of predicates hold, **when** clause tests if certain events occur at the moment, **while** clause tests if certain temporal actions are happening at present (e.g. while the meeting lasts, keep the lights on), **after** clause tests if certain events or actions happened in the past, and **before** clause tests if certain events or actions have not happened yet. Examples of these clauses will be provided below. We assume that the antecedent clauses refer to the past and to the present and the consequent clause refers to the present and the future. In case that both the antecedent and the consequent clauses refer to the present the rule is reduced to the standard non-temporal rule.

Temporal actions are combined together in the **then** clause of a rule either in a sequential or a parallel fashion. If two actions A_1 and A_2 are combined *sequentially* with the ";" operator, i.e.

$A_1; A_2$, then it means that the action A_2 is executed immediately after the action A_1 is finished. Furthermore, two actions A_1 and A_2 are executed *concurrently* if they are combined with the *parallel* operator "$||$", i.e. $A_1||A_2$. We define an *update* as a delete sequentially followed by an insert.

To illustrate temporal rules described above, consider an example of a banking application. Let *customer(cname,caddr)* be a relation describing the list of customers of a bank and *account(cname, type, id, balance)* be a relation describing the accounts opened at a bank by its customers. Then the following rules illustrate various features of the AECA model of a temporal rule.

Example 1 When a customer opens a 6 month CD account, he/she cannot close it for 6 months.

if customer(cname,caddr)
when **insert** account(cname, "CD-6", cd_number, amount)
then **insert_and_keep** (for 180_days) account(cname, "CD-6", cd_number, amount)

This example shows the Event-Condition-Action structure of a rule. Also, it shows the application of the **insert_and_keep** operator. It says that the tuple `(cname, ''CD-6'', cd_number, amount)` is inserted in the relation `account` now and will be kept there for the next 180 days. Finally, the rule shows an example of an event **insert** in the **when** clause.

□

The next example illustrates the AECA model of a temporal rule, the sequential operator in the **then** clause, and the use of the end-of-action operator.

Example 2 After a CD has expired, see if there is a better CD. If there is one, invest the same amount in it as in the expired CD.

If `interest_rates(CDtype,rate,period)` is a relation specifying interest rates and maturity periods of different types of CDs then the rule can be expressed as

after **insert_and_keep** account(cname, CDrate, CDnumber, amount)
if interest_rates(CDtype,newrate,period) and newrate > CDrate
then **delete** account(cname,CDrate, CDnumber, amount);
insert_and_keep (for period) account(cname, best(newrate), newnumber(), amount)

where the function `best` selects the best CD rate out of the list of existing rates, and the function `newnumber()` assigns a number to the new CD. This rule says that *after* the **insert_and_keep** activity is finished (i.e. the CD expired), and if there are CD's with better rates, then select the best CD out of them, delete the record for the old CD and insert the record for the newly selected CD.

This rule illustrates the AECA model of a temporal rule by referring to past actions in its antecedent part (the **after** clause)[4]. This rule also illustrates the use of the sequential operator ";". It says that the action **insert_and_keep account** must follow the action **delete account**. In this simple example, the first action occurs instantaneously. However, the sequence of actions can occur over time in general.

The same rule can be expressed somewhat differently. Instead of saying "after action **insert_and_keep** account," we can say "after *the end* of action **insert_and_keep** account" using the end operator described above, i.e.

after end.**insert_and_keep** account(cname, CDrate, CDnumber, amount)

□

The next example shows the use of the bounded temporal operator **insert_sometime**.

Example 3 If a monthly statement is sent to a credit card customer, then he or she will pay the bill sometime within 7 to 30 days (in this simplified example we assume that the customer must pay the total amount within the specified period).

If `billing(cname, billing_period, amount_due)` is a relation describing the bills sent to customers over various billing periods and `payments(cname, billing_period, amount_due)` is the relation describing customer payments then the rule can be expressed as

when **insert** billing(cname, billing_period, amount_due)
then **insert_sometime** (from 7 to 30) payments(cname, billing_period, amount_due)

□

The next example illustrates the use of parallel and sequential operators and the past temporal operators in "if" conditions.

Example 4 If a customer opens an account and he or she has never had an account with the bank in the past then create a new record for the customer (otherwise an old record will be updated).

If `application(cname,caddr,checking,chid,chdep,saving,sid,sdep)` is a relation describing the information a customer provided on the application, then the rule can be expressed as

[4] Notice that the action in the **after** clause is specified without temporal bounds `from` T_1 `to` T_2 since in this case we just want to know if the action ever occurred in the past. In general, we allow both bounded and unbounded actions in **before** and **after** clauses.

```
if      application(cname,caddr,checking,chid,chdep,saving,sid,sdep) and
        not sometime_past customer(cname,prev_addr)
then    insert (at next) customer(cname,caddr);
        (insert (at next) account(cname, checking, chid, chdep) ||
        insert (at next) account(cname, saving, sid, sdep))
```

This rule illustrates several points. First, the temporal operator **sometime_past** determines if the customer had an account in the past. Second, the sequential composition operator ";" says that the customer information is inserted into his or her checking and savings accounts right after the customer record is created. Finally, the data is inserted into the savings and the checking accounts *concurrently.*

□

4 Execution of Temporally Active Rules

In this section, we describe the recognize-act cycle for temporally active rules. As in the non-temporal case, the cycle consists of the matching, conflict resolution and execution steps. In the matching step, the antecedents of the rules are matched against the current and the past states of the database and against the previous events and actions, and the set of actions to be scheduled for the execution is determined. In the conflict resolution step, conflicts are resolved among the conflicting actions, and the selected actions are scheduled for the execution. Finally, the scheduled actions are executed in the execution step. The detailed description of the temporal recognize-act cycle will be presented in Section 4.2.

The temporal recognize-act cycle differs from the non-temporal case in the following respects. First, the matching is done not only against the current state of the database but also against its past history because conditions in the **if** clause can have past temporal operators and because the **after** clause also refers to the past. Second, for *any* type of an interpreter, including the sequential one such as the OPS5 interpreter, there will always be conflicts between INSERTs and DELETEs because the conflicting actions are scheduled at *different* moments of time.

In this paper, we make an assumption, standard for active databases, that, once a tuple is inserted in the database, it is kept there until it is explicitly deleted from it.

In the matching part of the cycle, the entire past history of the database has to be examined in general. Clearly, this makes the whole approach impractical in the database context. To "save" it, we first describe a method that *compresses* the "relevant" parts of the past history to the present so that the matching part of the cycle can be performed efficiently.

4.1 Compressing the Past History to the Present

The main idea of the compression technique is to replace arbitrary temporal rules with an "equivalent" set of rules containing only the **previous** operator. Since this operator refers only to the preceding time moment, there is no need to search an arbitrarily distant past in this case.

Given a set of temporal rules, we introduce a set of auxiliary predicates and modify temporal rules using the following guidelines. Each expression of the form **sometime_past** $P(x_1, \ldots, x_n)$ in a rule gives rise to predicate $P'(x_1 \ldots, x_n)$ and the rule

if $P(x_1, \ldots, x_n)$
then **insert** $P'(x_1 \ldots, x_n)$

The predicate $P'(x_1 \ldots, x_n)$ will remain true forever because it will remain true until it is explicitly deleted (which will never happen). Therefore, $P'(x_1 \ldots, x_n)$ is true if and only if **sometime_past** $P(x_1, \ldots, x_n)$ is true. For this reason, we replace all the occurrences of the expression **sometime_past** $P(x_1, \ldots, x_n)$ in rules with an equivalent predicate $P'(x_1 \ldots, x_n)$.

Example 5 To illustrate the conversion process, consider the rule from Example 4. It is converted to rules:

if application(cname,caddr,checking,chid,chdep,saving,sid,sdep) and
not past_customer(cname,prev_addr)
then **insert** (at next) customer(cname,caddr);
(**insert** (at next) account(cname, checking, chid, chdep) ||
insert (at next) account(cname, saving, sid, sdep))
if customer(cname,caddr)
then **insert** past_customer(cname,caddr)

In this example, `past_customer(cname,caddr)` will remain true since the first time the tuple `(cname,caddr)` was inserted in it.

□

Similarly, each expression of the form **always_past** $P(x_1, \ldots, x_n)$ gives rise to predicate $P''(x_1, \ldots, x_n)$ and an additional rule:

if $P''(x_1, \ldots, x_n)$ and not $P(x_1, \ldots, x_n)$
then **delete** $P''(x_1, \ldots, x_n)$

The value of P'' at the initial moment of time is equal to the value of P at that time. Again, once a tuple is inserted in P'', it will always stay there until it is explicitly deleted from the relation. Therefore, $P''(x_1, \ldots, x_n)$ is true at some time if and only if **always_past** $P(x_1, \ldots, x_n)$ is true at that time.

An expression of the form **sometime_past** (for T) $P(x_1, \ldots, x_n)$ gives rise to predicate $P''(x_1, \ldots, x_n)$, an auxiliary predicate $P'(x_1, \ldots, x_n, t)$, and the following rules:

if $P(x_1, \ldots, x_n)$
then **insert** $P'(x_1, \ldots, x_n, T)$; **insert** $P''(x_1, \ldots, x_n)$

if $P'(x_1, \ldots, x_n, t)$
then **delete** (at next) $P'(x_1, \ldots, x_n, t)$ ||
insert (at next) $P'(x_1, \ldots, x_n, t-1)$

if $P'(x_1, \ldots, x_n, 0)$
then **delete** $P''(x_1, \ldots, x_n)$; **delete** $P'(x_1, \ldots, x_n, 0)$

if $P'(x_1, \ldots, x_n, T)$ and $P'(x_1, \ldots, x_n, t)$ and $0 \leq t < T$
then **delete** $P'(x_1, \ldots, x_n, t)$

Notice that in this last case, we explicitly used function symbols in rules (decrementing T by 1).

Elimination of other past temporal operators, such as **always_past**, **sometime_past** (from T_1 to T_2), is done in a similar way and is omitted because of space limitations.

Finally, past temporal *actions* can be encoded with present events. For example, the temporal clause **after insert_and_keep** $P(x_1, \ldots, x_n)$ can be replaced with the temporal clauses

when end.**insert_and_keep** $P(x_1, \ldots, x_n)$
then **insert** $P'(x_1, \ldots, x_n)$

Conversion of other temporal clauses, such as **after insert_sometime** $P(x_1, \ldots, x_n)$, **after delete_and_keep** $P(x_1, \ldots, x_n)$, is done in a similar manner.

4.2 Temporal Recognize-Act Cycle

After the relevant past has been compressed to the present as explained in Section 4.1, we are ready to describe the temporal recognize-act cycle. A temporal recognize-act cycle is described in Fig. 3. It consists of matching, conflict resolution, and execution steps. We describe each step in turn now.

In the first step, antecedents of temporal rules are matched against the present and the past state of the database and against the present and past events and actions. Since the relevant past was compressed to the present with the method described in Section 4.1, this means that the rules

1. Match antecedents of all the rules against the current and compressed relevant past states of the database. Determine the actions S to be scheduled for the execution.
2. Find conflicts among actions in S and between actions in S and previously scheduled actions and resolve these conflicts. Schedule for the execution those actions in S that passed the conflict resolution phase.
3. Select the scheduled actions with the shortest remaining time and execute them.

Figure 3: Temporal Recognize-Act Cycle

have to be matched only against the current state of the database. Therefore, the matching can be done *exactly* as for the non-temporal case. This means that *any* non-temporal interpreter for existing rule-based systems can be used for that purpose including the interpreters described in [BFK86, KT89, WF90, SJGP90, HCKW90, TK91].

In Step 2 of the cycle, conflicts between temporal actions generally scheduled at different periods of time are resolved. For example, one action can be **insert_and_keep** (from 20 to 40) $P(a_1, \ldots, a_n)$ and another **delete_and_keep** (from 15 to 25) $P(a_1, \ldots, a_n)$. Notice that this type of conflict differs from the conflicts among rules in the non-temporal case. In the non-temporal case, conflicts are resolved among the tuples instantiated at the *same* time moment. For example, OPS5 interpreter selects only one instantiated tuple out of the set of instantiated tuples. Once this tuple is selected, there can be no conflicts, and the interpreter proceeds to execute the rule. In the temporal case, the conflicts still can exist among actions scheduled at *different* moments of time, even if the interpreter selects only one instantiated tuple at a time. For example, the action **delete_and_keep** (from 15 to 25) $P(a_1, \ldots, a_n)$ could be scheduled at time $t = 8$ and the action **insert_and_keep** (from 20 to 40) $P(a_1, \ldots, a_n)$ at time $t = 12$. We describe different conflict resolution strategies in Section 4.3, including a new conflict resolution strategy arising from the fact that actions occur over a period of time. Once the conflicts are resolved, the remaining actions are scheduled to be executed at some future time moments.

In the selection stage of Step 3, we find all the scheduled actions with the minimal "remaining" time. For example, if the current time is $t = 12$, and the action **delete_and_keep** (from 15 to 25) $P(a_1, \ldots, a_n)$ was scheduled (at time $t = 15$), and there are no actions scheduled between times $t = 12$ and $t = 15$, then select that action for the execution together with other actions scheduled at that time.

In the execution stage of Step 3, the actions selected in the previous stage of Step 3 are executed in real time. At this stage, actual insertion and deletion commands are issued by the interpreter and later performed by the transaction manager. In this paper, we made a simplifying assumption that we do not consider time delays associated with concurrent executions of multiple transactions, and, therefore, assume that insertions and deletions occur instantaneously. This assumption significantly simplifies the execution of scheduled actions. Nevertheless, we do not describe the real-time scheduler in this paper because we plan to do it in a subsequent paper when we relax the simplifying assumption stated above and consider concurrent transactions.

4.3 Conflict Resolution Strategies

In this section, we describe how conflicts can be resolved for temporally active rules. First, we describe various semantics of conflicts and then the methods to resolve them.

4.3.1 Semantics of Conflicts

If two potentially conflicting actions are of the type *keep*, i.e. are **insert_and_keep** and **delete_and_keep**, then the conflict occurs when their time intervals intersect. Specifically, **insert_and_keep** (from T_1 to T_2) $P(a_1, \ldots, a_n)$ conflicts with **delete_and_keep** (from T_3 to T_4) $P(a_1, \ldots, a_n)$ if and only if intervals $[T_1, T_2]$ and $[T_3, T_4]$ intersect.

For conflicts between actions of type *keep* and *sometime*, e.g. when **insert_and_keep** conflicts with **delete_sometime**, we consider the following two types of conflicts. Let *keep* action occur over the interval $[T_1, T_2]$, and *sometime* action occur over the interval $[T_3, T_4]$.

The *intersection semantics* of conflicts says that the two actions conflict when intervals $[T_1, T_2]$ and $[T_3, T_4]$ *intersect*. Intuitively, it says that if a *keep* action overlaps with a *sometime* action then the *sometime* action cannot be scheduled at any arbitrary time in the interval $[T_3, T_4]$ and must be restricted to some smaller time domain. However, the programmer who wrote the temporal rules, may count on the fact that the *sometime* action can occur *anywhere* in $[T_3, T_4]$. To assure his/her expectations, the programmer may select the intersection semantics.

The *containment semantics* of conflicts says that the two actions conflict when interval $[T_1, T_2]$ *contains* interval $[T_3, T_4]$. Intuitively, it says that if *keep* action is scheduled during the whole time interval of *sometime* action, then the *sometime* action cannot occur at *any* point in this time interval. Clearly, this means that *sometime* action is invalid, and the two actions conflict.

The last type of conflict occurs between two *sometime* actions. In this case, we also consider two types of semantics for conflicts. As in the previous case, if two *sometime* actions occur at time intervals $[T_1, T_2]$ and $[T_3, T_4]$ then they conflict if

1. intervals $[T_1, T_2]$ and $[T_3, T_4]$ intersect

2. $T_1 = T_2 = T_3 = T_4$.

4.3.2 Conflict Resolution

In the previous section, we identified situations when conflicts occur between temporal actions. In this section, we present methods for resolving these conflicts.

There have been several conflict resolution strategies proposed for non-temporal active databases. One such strategy orders rules (either partially or totally) according to their precedence. Then the qualifying rules with the highest precedence are selected. This is the conflict resolution strategy adopted in Starburst [HCL+90] and POSTGRES [SJGP90]. The conflict resolution strategy of OPS5 is based on several tuple selection criteria that take into account structural properties of rules and recency of tuple insertions into the database [BFK86]. If all these criteria fail to resolve the conflict, a single instantiation is chosen at random. Still another conflict resolution strategy initially proposed in [KT89] and later extended in [TK91] operates on the consequent part of a rule. It assumes that the insertion of a tuple has a precedence over its deletion if the database does not contain the tuple and the deletion has a precedence over the insertion if the tuple exists in the database. The intuitive justification for this strategy is presented in [TK91]. Furthermore, de Maindreville and Simon [dMS88] describe a conflict resolution strategy (within a rule), such that if an INSERT conflicts with a DELETE, then both actions are canceled. Finally, Ioannidis and Sellis [IS89] describe some conflict resolution strategies for rules assigning values to virtual attributes. Furthermore, they classify three types of conflicts: conflicts occurring either at the rule or the antecedent or the consequent levels [IS89].

These non-temporal conflict resolution strategies are also applicable to temporally active databases. Furthermore, the conflict resolution strategy of OPS5 based on the recency of tuple insertions into the database can be supported in a more direct way for the temporal case.

In addition to the strategies borrowed from the non-temporal case, we propose the following *temporal* conflict resolution strategy **TCRS** in which conflicts are resolved at the consequent level (using terminology of [IS89]):

> If the actions of two rules conflict, then select the action of the rule that fired first. If both rules are fired at the same time then apply any conflict resolution strategy for the non-temporal case, e.g. cancel the conflicting actions or select the conflicting action from the rule with the higher precedence.

For example, if rule R_1 fired the action **insert_and_keep** (from 10 to 20) $P(a_1, \ldots, a_n)$ at time $t = 5$ and the rule R_2 fired the action **delete_and_keep** (from 15 to 25) $P(a_1, \ldots, a_n)$ at time

$t = 8$ then the first action has a precedence over the second action because rule R_1 was fired before rule R_2.

Intuitively, the **TCRS** strategy says that once an action is scheduled for a future execution, the commitment is made to execute it at some later time, and the scheduled action cannot be canceled[5].

5 Conflict Avoidance Strategies

It was shown in [TK91] for the non-temporal case that it is possible to avoid conflicts by writing an equivalent set of non-conflicting rules. In this section, we extend this idea to the temporal domain.

As an initial approach to the problem, we impose the following restrictions on the structure and semantics of rules. First, we assume that the consequent part of a rule contains a *single* action. Second, we restrict our consideration only to the intersection semantics of conflicts between *keep* and *sometime* actions as defined in Section 4.3.1. Third, we assume that rules contain only **if** and **then** clauses (i.e. no **when**, **before**, and **after** clauses). We are currently working on the ways to relax these three assumptions.

Each conflict resolution strategy gives rise to its own set of equivalent non-conflicting rules. To be specific, we selected the temporal conflict resolution strategy **TCRS** described in Section 4.3.2. However, as will be pointed out later, the same approach is applicable to some other conflict resolution strategies considered in that section.

In the rest of this section, we describe a method that replaces two conflicting rules with an equivalent set of non-conflicting rules. Let rule R_1 be

if $Q_1(x_1, \ldots, x_n, T_1, T_2)$
then **insert_and_keep** (from T_1 to T_2) $S(x_1, \ldots, x_n)$

and rule R_2 be

if $Q_2(x_1, \ldots, x_n, T_3, T_4)$
then **delete_and_keep** (from T_3 to T_4) $S(x_1, \ldots, x_n)$

In these rules, $Q_1(x_1, \ldots, x_n, T_1, T_2)$ and $Q_2(x_1, \ldots, x_n, T_3, T_4)$ are *conditions* rather than predicates. They define conjunctions of predicates, optionally preceded by negations and by temporal operators. Furthermore, we assume that T_1, T_2, T_3, and T_4 do not change over time and that

[5]Notice that we do not consider concurrency control here. Therefore, this kind of commitment is different from the commitment of transactions.

$T_1 < T_2$ and $T_3 < T_4$.

Let ΔT be the (absolute) time difference between the times rules R_1 and R_2 are fired. For example, if rule R_1 is fired at time $t = 10$ and rule R_2 at time $t = 15$ then $\Delta T = 5$.

Then rules R_1 and R_2 conflict in the following situations:

1. Let R_1 be fired before R_2 and $T_4 < T_1$. Then R_1 and R_2 conflict iff $T_1 - T_4 \leq \Delta T \leq T_2 - T_3$.
2. Let R_1 be fired before R_2, $T_1 \leq T_4$ and $T_3 \leq T_2$. Then R_1 and R_2 conflict iff $0 \leq \Delta T \leq T_2 - T_3$.
3. Let R_2 be fired before R_1 and $T_2 < T_3$. Then R_1 and R_2 conflict iff $T_3 - T_2 \leq \Delta T \leq T_4 - T_1$.
4. Let R_2 be fired before R_1, $T_3 \leq T_2$ and $T_1 \leq T_4$. Then R_1 and R_2 conflict iff $0 \leq \Delta T \leq T_4 - T_1$.
5. Let R_1 be fired at the same time as R_2. Then R_1 and R_2 conflict iff $T_3 \leq T_2$ and $T_1 \leq T_4$.

These five possibilities cover all the conflicting situations because in the two remaining cases (a) when R_1 is fired before R_2 and $T_2 < T_3$, and (b) when R_2 is fired before R_1 and $T_4 < T_1$, the rules do not conflict.

We replace the rules R_1 and R_2 with the set of equivalent rules for the conflict resolution strategy **TCRS** in two stages. In the first stage, we replace them with the set of *pseudo-rules* that are not directly supported by the syntax of the rules as defined in Section 3. In the second stage, we replace each pseudo-rule with the set of real temporally active rules. Finally, we show that the resulting set of rules is equivalent[6] to rules R_1 and R_2 and does not have conflicts.

The pseudo-rules that replace rules R_1 and R_2 in the first stage are shown in Fig. 4. To simplify the notation, we drop arguments in expressions for S, Q_1, and Q_2. We always assume that S has arguments $S(x_1, \ldots, x_n)$, Q_1 arguments $Q_1(x_1, \ldots, x_n, T_1, T_2)$, and Q_2 arguments $Q_2(x_1, \ldots, x_n, T_3, T_4)$.

Pseudo-rule 1 says that if rule R_2 is fired now and rule R_1 was fired in the past so that the two rules conflict, then rule R_1 has a precedence over rule R_2 and, therefore, the pseudo-rule 1 performs insertion. It corresponds to the conflict situation 1 described above. Similarly, pseudo-rule 2 says that if R_2 is fired now and R_1 in the past so that the two rules do not conflict then the action of rule R_2 is carried out. Pseudo-rules 1 and 2 are applicable to the cases when $T_4 < T_1$. Pseudo-rules 3 and 4 are similar to pseudo-rules 1 and 2 but are applicable to the cases when $T_3 \leq T_2 \wedge T1 \leq T4$. Pseudo-rules 3 corresponds to the conflict situation 2. Pseudo-rules 5, 6, 7, and 8 are similar to the first four pseudo-rules but take care of the situation when rule R_1 is fired now and rule R_2

[6]Two sets of rules are *equivalent* if, for any initial state of the database, they always produce the same sequences of predicates over time. Formal details of this definition can be found in [TK91].

1.	**if**	$Q_2\wedge$ **sometime_past**(from $T_1 - T_4$ to $T_2 - T_3$) $Q_1 \wedge T_4 < T1$
	then	**insert_and_keep** (from T_1 to T_2) S
2.	**if**	$Q_2 \wedge \neg$ **sometime_past**(from $T_1 - T_4$ to $T_2 - T_3$) $Q_1 \wedge T_4 < T1$
	then	**delete_and_keep** (from T_3 to T_4) S
3.	**if**	$Q_2 \wedge \neg Q_1 \wedge$ **sometime_past**(from 0 to $T_2 - T_3$) $Q_1 \wedge T_3 \leq T_2 \wedge T1 \leq T4$
	then	**insert_and_keep** (from T_1 to T_2) S
4.	**if**	$Q_2 \wedge \neg Q_1 \wedge \neg$ **sometime_past**(from 0 to $T_2 - T_3$) $Q_1 \wedge T_3 \leq T_2 \wedge T1 \leq T4$
	then	**delete_and_keep** (from T_3 to T_4) S
5.	**if**	$Q_1\wedge$ **sometime_past**(from $T_3 - T_2$ to $T_4 - T_1$) $Q_2 \wedge T_2 < T_3$
	then	**delete_and_keep** (from T_3 to T_4) S
6.	**if**	$Q_1 \wedge \neg$ **sometime_past**(from $T_3 - T_2$ to $T_4 - T_1$) $Q_2 \wedge T_2 < T_3$
	then	**insert_and_keep** (from T_1 to T_2) S
7.	**if**	$Q_1 \wedge \neg Q_2\wedge$ **sometime_past**(from 0 to $T_4 - T_1$) $Q_2 \wedge T_3 \leq T_2 \wedge T_1 \leq T_4$
	then	**delete_and_keep** (from T_3 to T_4) S
8.	**if**	$Q_1 \wedge \neg Q_2 \wedge \neg$ **sometime_past**(from 0 to $T_4 - T_1$) $Q_2 \wedge T_3 \leq T_2 \wedge T_1 \leq T_4$
	then	**insert_and_keep** (from T_1 to T_2) S
9.	**if**	$Q_2 \wedge$ **sometime_past**$Q_1 \wedge T_2 < T_3$
	then	**delete_and_keep** (from T_3 to T_4) S
10.	**if**	$Q_2 \wedge \neg$**sometime_past**$Q_1 \wedge T_2 < T_3$
	then	**delete_and_keep** (from T_3 to T_4) S
11.	**if**	$Q_1 \wedge$ **sometime_past**$Q_2 \wedge T_4 < T_1$
	then	**insert_and_keep** (from T_1 to T_2) S
12.	**if**	$Q_1 \wedge \neg$**sometime_past**$Q_2 \wedge T_4 < T_1$
	then	**insert_and_keep** (from T_1 to T_2) S
13.	**if**	$Q_1 \wedge Q_2 \wedge T_3 \leq T_2 \wedge T_1 \leq T_4$
	then	do nothing

Figure 4: Pseudo-rules that Avoid Conflicts.

was fired in the past. Pseudo-rules 5 and 7 correspond to the conflict situations 3 and 4 described above. Furthermore, pseudo-rules 9 and 10 say that if $T_2 < T_3$ and rule R_2 is fired now then there can be no conflict between inserts and deletes. Therefore, the deletion operation from rule R_2 is carried out in these two pseudo-rules. Pseudo-rules 11 and 12 take care of the similar situation but pertaining to rule R_1. Finally, pseudo-rule 13 is a "vacuous" rule saying that if rules R_1 and R_2 are fired at the same time and conflict then, based on the conflict resolution strategy **TCRS**, insert and delete operations are canceled. Notice that if we adopt any other conflict resolution strategy for the case when the two rules are fired simultaneously, all we have to do is to change the pseudo-rule 13.

The rules presented in Fig. 4 are pseudo-rules because most of them have a temporal operator in front of an *expression* or a negation in front of an expression. For example, the statement **sometime_past** (from 0 to $T_4 - T_1$) $Q_2(x_1, \ldots, x_n, T_3, T_4)$ is illegal in our language.

In stage 2 of the conversion process, the pseudo-rules from Fig. 4 are replaced with legal temporally active rules. To illustrate the replacement process, consider the pseudo-rule 1 in Fig. 4, where Q_1 is a conjunction of some temporal literals $P_1 \wedge \ldots \wedge P_n$. The pseudo-rule can be replaced with the following set of real rules, where R is an auxiliary temporal predicate:

if $Q_2 \wedge$ **sometime_past**(from $T_1 - T_4$ to $T_2 - T_3$) $R \wedge T_4 < T1$
then **insert_and_keep** (from T_1 to T_2) S

if Q_1
then **insert** R

if **previous** $P_i \wedge \neg P_i$ (for $i = 1, \ldots, n$)
then **delete** R

The last two rules make the expression Q_1 to be equivalent to predicate R. The same technique that replaces pseudo-rules with a set of real rules is applicable to other pseudo-rules in Fig. 4.

Combining all these observations together, we state the following result:

Theorem 1 *For the conflict resolution strategy* **TCRS**, *conflicting rules R_1 and R_2 can be converted into an equivalent set of non-conflicting rules.*

Sketch of Proof: By inspection, conflicting pseudo-rules in Fig. 4 are mutually exclusive. Furthermore, if we disjunct all the "if" parts of these pseudo-rules then we can simplify the resulting expression to $Q_1 \vee Q_2$. This means that the pseudo-rules from Fig. 4 are collectively exhaustive: they cover all the possibilities when rules R_1 and R_2 can fire. Furthermore, these pseudo-rules are designed so that they resolve conflicts exactly as the conflict resolution strategy **TCRS** would do.

□

Although the rules in Fig. 4 are designed for the conflict resolution strategy **TCRS**, they can be changed to accommodate other conflict resolution strategies considered in Section 4.3.2.

6 Related Work

An idea to add time to active databases was first expressed in [Dea88]. However, the subsequent description of the HiPAC project [MD89] does not specify how to do this. Also, Ode [GJ91] supports timed triggers in the way described in the introduction. Timed triggers are similar to our **insert_sometime** and **delete_sometime** temporal actions. However, temporal rules presented in this paper also support other types of temporal actions in the consequent part of a rule as well as in the antecedent part. In addition, they support temporal predicates, different types of temporal operators, and the AECA model of a rule.

Temporally active databases are also related to temporal logic programming (TLP) systems, such as Templog [AM89], MetateM [BFG+89], Temporal Prolog [KKN+90] and PTL [Tuz91b]. Both approaches combine rules and temporal logic. However, they differ in the same way as production systems differ from deductive databases. Temporally active databases deal with events and actions, and TLP systems with facts. Furthermore, the two systems have different types of semantics. For example, the concept of conflict resolution is not applicable to TLP systems at all.

Furthermore, temporally active databases are related to the requirements specification language Templar [Tuz91a]. As in the case of temporal logic programming, both systems use rules and temporal logic. However, rules described in this paper deal with database updates, whereas Templar rules specify some high-level requirements specification activities which can consist of smaller subactivities. This means that Templar does not deal with the issues of scheduling, resolving conflicts and executing temporal rules, as the system described in this paper does.

Finally, the technique compressing the relevant parts of past history of a database to the present state was proposed by other researchers for some temporal logic programming systems. Kato et al [KKN+90] describe a method that converts a temporal logic program with past necessity, **previous**, **after**, **next**, future necessity, **atnext**, and **until** operators to an equivalent temporal logic program with only the **previous** operator. Similarly, Baudinet converts Templog programs to its equivalent fragment TL1 containing only the **next** operator [Bau89]. Similar observation was made by Chomicki [Cho91] when he described a method to compress the past history of a temporal database to the present state for a set of dynamic integrity constraints. In this paper, we extended the approach from [KKN+90] to temporally active databases.

7 Conclusions and Future Work

In this paper, we studied temporally active databases. They differ from active non-temporal databases in several ways. First, actions in temporally active databases can occur over some periods of time. Second, antecedents of temporal rules examine both the current state of the database and its past history, as well as the past actions and events. Third, consequents of temporal rules specify actions that will occur in the future, assuming that antecedents are true.

To support these additional characteristics of temporally active databases, we proposed to extend the traditional ECA model to a more general Action-Event-Condition-Action (AECA) model. AECA model of a temporal rule differs from the ECA model in that it supports actions both in the antecedent and the consequent part of a rule, in that it admits additional temporal clauses, such as **before**, **after**, and **while**, and in that it supports operators of temporal logic.

Addition of time to active databases also affects the recognize-act cycle. To make the recognize-act cycle practical, we described a method that compresses the past history of the database to the present. We also adjusted non-temporal conflict resolution strategies to incorporate time. Furthermore, we described the types of conflicts that can occur in the temporal case and proposed a method to resolve them. Finally, the temporal dimension requires new conflict avoidance methods, and we described an initial solution to this problem.

In future research we plan to extend the conflict avoidance strategy presented in this paper to a more general setting when other clauses, such as **when**, **while**, **before**, and **after**, are allowed in a rule, and when the containment semantics is assumed for the conflicts between *keep* and *sometimes* actions.

We also plan to work on adding a transaction model to temporal rules and on the integration of temporally active databases with real-time concurrency control. This will enable us to support atomicity of transactions and multiple users of active databases. To support real-time concurrency control, we have to find ways to integrate scheduling of future actions with the real-time transaction processing to be able to meet real-time deadlines set by the programmer writing rules. We believe that the work on the real-time transaction processing [AGM88, KSS90] can serve as a starting point for that.

8 Acknowledgments

The author wishes to thank H. V. Jagadish for discussions of some of the issues in this paper and also for providing many useful comments about an earlier draft of the paper. He is also gratful to Jim Clifford for reading an earlier draft of the paper.

References

[AGM88] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. In *International Conference on Very Large Databases*, pages 1–12, 1988.

[AM89] M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8:277–295, 1989.

[Bau89] M. Baudinet. Temporal logic programming is complete and expressive. In *Symp. on Principles of Programming Languages*, pages 267–280, 1989.

[BFG+89] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A framework for programming in temporal logic. In *Stepwise Refinement of Distributed Systems*, pages 94–129. Springer-Verlag, 1989. LNCS 430.

[BFK86] L. Brownston, R. Farrell, and E. Kant. *Programming Expert Systems in OPS5: an Introduction to Rule-Based Programming.* Addison-Wesley, 1986.

[Cho91] J. Chomicki. History-less checking of dynamic integrity constraints. Unpublished manuscript, 1991.

[CW83] J. Clifford and D. S. Warren. Formal semantics for time in databases. *TODS*, 8(2):214–254, 1983.

[Dea88] U. Dayal and et al. The HiPAC project: Combining active databases and timing constraints. *ACM SIGMOD Record*, 17(1):51–70, 1988.

[dMS88] C. de Maindreville and E. Simon. Modelling non deterministic queries and updates in deductive databases. In *International Conference on Very Large Databases*, pages 395–406, 1988.

[Gad88] S. K. Gadia. A homogeneous relational model and query languages for temporal databases. *TODS*, 13(4):418–448, 1988.

[GJ91] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *International Conference on Very Large Databases*, 1991.

[GM91] D. Gabbay and P. McBrien. Temporal logic and historical databases. In *International Conference on Very Large Databases*, 1991.

[HCKW90] E.N. Hanson, M. Chaabouni, C.H. Kim, and Y.W. Wang. A predicate matching algorithm for database rule systems. In *Proceedings of ACM SIGMOD Conference*, pages 271–280, 1990.

[HCL+90] L. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, 1990.

[IS89] Y.E. Ioannidis and T.K. Sellis. Conflict resolution of rules assigning values to virtual attributes. In *Proceedings of ACM SIGMOD Conference*, pages 205–214, 1989.

[Kam68] H. Kamp. *On the Tense Logic and the Theory of Order.* PhD thesis, UCLA, 1968.

[KKN+90] D. Kato, T. Kikuchi, R. Nakajima, J. Sawada, and H. Tsuiki. Modal logic programming. In *VDM and Z - Formal Methods in Software Development.* Springer-Verlag, 1990. LNCS 428.

[Kro87] F. Kroger. *Temporal Logic of Programs.* Springer-Verlag, 1987. EATCS Monographs on Theoretical Computer Science.

[KSS90] H.F. Korth, N. Soparkar, and A. Silberschatz. Triggered real-time databases with consistency constraints. In *International Conference on Very Large Databases*, pages 71–82, 1990.

[KT89] Z. M. Kedem and A. Tuzhilin. Relational database behavior: Utilizing relational discrete event systems and models. In *Proceedings of PODS Symposium*, 1989.

[MD89] D. McCarthy and U. Dayal. The architecture of an active, object-oriented database system. In *Proceedings of ACM SIGMOD Conference*, 1989.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, cashing and views in database systems. In *Proceedings of ACM SIGMOD Conference*, pages 281 – 290, 1990.

[Sno87] R. Snodgrass. The temporal query language TQuel. *TODS*, 12(2):247–298, 1987.

[SPAM91] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: an architecture for transforming a passive DBMS into an active DBMS. In *International Conference on Very Large Databases*, 1991.

[TC90] A. Tuzhilin and J. Clifford. A temporal relational algebra as a basis for temporal relational completeness. In *International Conference on Very Large Databases*, pages 13–23, 1990.

[TK91] A. Tuzhilin and Z. M. Kedem. Modeling dynamics of databases with relational discrete event systems and models. Working Paper IS-91-5, Stern School of Business, NYU, 1991.

[Tuz91a] A. Tuzhilin. Templar: A knowledge representation language for requirements specifications. Working Paper IS-91-27, Stern School of Business, NYU, 1991.

[Tuz91b] A. Tuzhilin. Temporal logic as a simulation language. In *Proceedings of the International Conference on Artificial Intelligence and Simulation*, New Orleans, Louisiana, April 1991.

[WF90] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of ACM SIGMOD Conference*, pages 259 – 270, 1990.