ACTIVE DATABASES FOR COMMUNICATION NETWORK MANAGEMENT

Ouri Wolfson
Soumitra Sengupta
Yechiam Yemini

Columbia University
Dept. of Computer Science
Technical Report CUCS-059-90

ACTIVE DATABASES FOR COMMUNICATION NETWORK MANAGEMENT[1]

Preliminary Version

Ouri Wolfson
Soumitra Sengupta
Yechiam Yemini

Distributed Computing and Communication Lab.
450 Computer Science Bldg.
Columbia University
New York, NY 10027

ABSTRACT

This paper has two purposes. First is to propose new database language-features for systems used in real-time management. These features enable the specification of change-traces, events and correlation among events, and they do so in a declarative set-oriented fashion. Second is to introduce network management as an important and interesting application of active distributed databases.

1. Introduction

Generally speaking, database systems provide the capability of data definition and data manipulation. The data dictionary often provides meta-data concerning the data-definition. However, existing systems do not provide meta-data concerning the data-manipulation. An example of such meta-data is the sequence of the values of a certain attribute in the past hour. Such a sequence represents the way the data-manipulation has changed the value of the attribute. Moreover, currently there is no general language in which the user may request monitoring of database-changes. In this paper we propose such a language. Triggers in traditional database systems serve the purpose of monitoring database changes, but their functionality is limited. For example, our language allows the user to request that whenever the value of a certain attribute changes, the old value is appended to a trace. This may be requested for each object in a class, or just for objects that satisfy some selection predicate. Furthermore, a separate trace may be requested for each object, or several objects (e.g. all the ones with COLOR="yellow") may be monitored in the same trace. Another example of a new language feature is event correlation. For example, the user may request a certain action when two different changes occur simultaneously, or when one occurs before the other but the events are

1. This research was supported in part by DARPA research grant #F-29601-87-C-0074, by the Center for Advanced Technology at Columbia University NYSSTF-CAT(89)-5 and NYSSTF CU01207901, and by NSF grant IRI-90-03341.

not more than 50 seconds apart. Furthermore, this requirement may apply to the transaction time, or the occurrence time. As we shall demonstrate, our proposed language features are particularly important in database management systems that receive an automatic inflow of data from various sensors, in real-time.

More precisely, we propose three new language features. They all address temporal monitoring of database changes. First, the specification of basic events. An example of a basic event that can be specified using the proposed language feature is the following. "There exist tuples that satisfy a certain predicate, and this condition persists for a period of 10 minutes". Second, the specification of correlated events. A correlated event is triggered by the occurrence of several events that satisfy certain temporal constraints. Third, the specification of trace-collections. A trace records the way an attribute changes over time, and a trace-collection is a set of traces, all of which have the same characteristics.

Our proposal is independent of the underlying data model; it may be object-oriented or relational.

We demonstrate the use of the proposed features in network management. This is an application of active distributed databases for real-time system management. The system in this case is a communication network. Network management has recently emerged as a very active area of research and development ([P88]). It is mainly concerned with functions to handle faults and performance bottlenecks in very large (hundreds of thousands of nodes) communication networks ([CDFS88], [B89]). Network management, in general, consists of two activities: monitoring and controlling the network. Data concerning the real-time operation of the network is continuously flowing to the (distributed) network management system. Therefore, often what is being monitored and controlled is a data-model of the network, and the operations on the data are translated into operations in the physical network. Monitoring the network means "watching" for certain important phenomena, or events, to occur. Network control means the activation of processes that change the status of the network. We feel that by incorporating the proposed language features, a knowledge-base management system, of the type emerging as the next generation database systems ([U89, C*89, S90, S*90]), can serve as a network management system that is more advanced than the existing technology in network management.

The rest of this paper is organized as follows. The next section discusses the components of the network management system, namely the network database and the functions for fault and performance management. Section 2 is not strictly necessary for the reader interested solely in the new language

features proposed in this paper. In section 3 we discuss the specification of basic and correlated events. In section 4 we discuss the specification of traces and trace collections. In network management, traces often enable prediction of faults and performance bottlenecks. In section 5 we return to events, and discuss events that are data patterns in traces. In section 6 we compare this work to the relevant literature, and in section 7 we conclude and discuss future work.

2. The Network Management Components

A Network Management System consists of the *network database,* and a set of *management functions.* The NMS manages the network database the same way a database management system manages the data, and additionally it executes management functions, either automatically, or in response to requests by human operators.

2.1 The Network Database

There are two types of data in the network database. One is the *configuration* data, and the other is the *history* data. The configuration data represents the current status of the network, and is further divided into *static* and *dynamic* configuration. The static configuration database consists of the representation of permanent objects in the network, such as computers, communication links, software layers, groups of nodes, local area subnetworks, users; and, obviously, the relationships among these objects. The dynamic configuration database consists of the representation of transient objects, such as virtual circuits, user-sessions, delays, routing tables, etc. The configuration database is used for normal network operation (such as message routing), in addition to its usage for fault management.

The history database consists of information about the evolution of the network and its status over time (possibly in summary form), such as the virtual circuits between a certain pair of processors between 3pm and 5pm, the number of packets transmitted by certain processors in each minute for the last hour, etc. This information is generated by statistical tests (see next subsection), is mainly based on the dynamic configuration database, and is used for fault and performance management and for capacity planning.

2.2 The Network Management Functions

There are three basic types of management functions: actions, alerts, and inferences. *Actions* are management functions that execute in the real network. Each action is associated with one or more network objects. There are two types of actions: interventions and tests.

An *intervention* is an action aimed at overcoming a problem, by changing something in the way the network, or some of its components, operate. Several examples of interventions are: "Reboot a computer", "deactivate the interface between a gateway and a local area network", "call the technician indicating a faulty equipment", and "change some parameters of a protocol layer".

A *test* is an action that monitors the network. In contrast to an intervention, it is nonintrusive, i.e., it does not change the operation of the network. There are two types of tests. One is a definitive test, and the other is a statistical test. A *definitive test,* when applied to a network object, say a modem, determines, whether or not the object is functioning properly. A definitive test, usually supplied by the equipment manufacturer, may be associated with an intervention; the intervention is invoked if the test determines improper functioning.

Statistical tests are used when definitive tests are insufficient for problem diagnosis and isolation. A *statistical test,* when applied to an object or a set of objects, generates a trace (a history-database object) that can be analyzed for an abnormal pattern. An example of a statistical test, say MESSAGE_TIME(A, B), is one that generates a record for each message from the processor at network address A to the processor at network address B; the record specifies how long it took until the message was acknowledged.

An *alert* is a management function that indicates an abnormal condition. In this paper we discuss two types of alerts. An alert triggered by a state of the network (as reflected in the database), or an alert triggered by one or more tests. An example of an alert triggered by the network state is OVERLOAD: "the delay on 20% of the communication links exceeds 5 seconds". An example of an alert, called, say, SURGE(A, B), that is triggered by the test MESSAGE_TIME(A, B), is the following: "there is an increase of more than 10 seconds in the acknowledgement time of two consecutive messages from A to B". Generally, an alert may be triggered by multiple tests, and multiple alerts may be triggered by the same test. Therefore, there is a many to many relationship between tests and alerts.

An *inference* is a management function which maps one or more alerts to a set of tests, alerts, and interventions. For example an inference may map the alert SURGE(A, B) to the set of definitive tests

UP_DOWN(Z), for each processor Z that is on a path from *A* to *B*. Because of space limitations we will not discuss inferences in this paper, but will just mention that they can be performed using existing knowledge-base technology.

A *fault management state* is the collection of tests, interventions, and alerts, that are active in the network at some point in time. *Fault management* is the continuous activity of mapping from the current management state to the next, such that faults and performance bottlenecks in the network are either predicted and prevented, or, detected, isolated and eliminated.

3. Events

Most alerts in network management are simply events. We distinguish between two types of events: basic or correlated. A correlated event is a combination of multiple events; it occurs if they all occur simultaneously. One of the primary purposes of such grouping is to enable correlation of alerts. So, for example, the user may specify that if alerts A and B occur at the same time, and alert C does not occur, then alerts A and B are probably related to the same problem-source; the occurrence of A and B, but not C, will then be called D, which the inference mechanism may then treat differently than either A or B.

In order for an event to *occur*, i.e., be noticed (otherwise, as far as the network management system is concerned, it has not occurred), it must be *specified* and its monitoring must be *activated*. Event monitoring consumes resources, therefore, monitoring a specified event may be active or inactive. Next we discuss the specification of events.

3.1 Basic Events

A *basic event* is either: 1. A data-pattern in the network database; or 2. A data manipulation operation, namely a retrieve, add, delete, or update of the network database; or 3. Calendar-time.

A *data-pattern* event occurs when a certain data-pattern appears in the database; for example, when the delay on 20% or more of the communication links exceeds 5 seconds (OVERLOAD). A data-pattern event is specified using a data-retrieval operation, that supposedly executes continuously. The event occurs when the retrieval returns at least one element, and for this reason event of this type is also called a *nonempty-retrieval* event. For example, assume that there is a relation LINKS, that has a tuple for each communication-link in the network, and that one of the attributes of this relation is DELAY (on the link).

The nonempty-retrieval event OVERLOAD can be specified by the following data retrieval statement, which, for the sake of simplicity, is written in SQL:

```
SELECT      X = Y/Z
FROM
WHERE       0.2  ≤
            [SELECT Y=COUNT(*)
             FROM  LINKS
             WHERE  DELAY > 5 ]
                /
            [SELECT Z=COUNT(*)
             FROM  LINKS ]
```

In the case of retrieval from object classes, an object-oriented retrieval language (e.g., the one in [CCCTZ90]) can be used for specifying data-pattern events.

A parameter of a data-pattern event is the following: PERSISTENCE ≥ 'integer'. It indicates that the event is to occur only if the data-pattern persists in the database for an interval of time that not lower than 'integer'. For example, if PERSISTENCE ≥ 10minutes is associated with the above data-retrieval statement then the resulting event, that we call PERSISTENT-OVERLOAD, will occur when the delay on 20% or more of the communication links exceeds 5 seconds, and this condition persists for more than 10 minutes. Note that it is not required that the percentage of slow links remains constant for 10 minutes, just that the percentage exceeds 20% for this duration. The purpose of this parameter is to enable ignoring data-patterns that are transient.

A *data-manipulation* event occurs when a certain data-manipulation operation is performed in the system. It is obviously specified by a data-manipulation operation. For example,

ADD to LINKS, and

OLD LINKS, WHERE DELAY > 5.

are two events. The first occurs when some tuples are added to the relation LINKS, the other occurs when the tuples of LINKS that have a DELAY > 5 are either replaced or deleted. In general, we adopt the definition of an event in [SJGP90] to serve as our data-manipulation event. Thus, a data-manipulation event consists of an operation, i.e. ADD, DELETE, REPLACE, RETRIEVE, OLD (DELETE or REPLACE) or NEW (ADD or REPLACE), together with an object class or a subset of the class (e.g.

LINKS WHERE DELAY > 5).

A data-pattern event can obviously implemented by a data-manipulation event. For example, PERSISTENT-OVERLOAD can be implemented by examining each modification to LINKS, and when the 20% limit is exceeded, verifying that the condition persists. However, we feel that often the data-pattern event provides a higher level of abstraction (as is the case for PERSISTENT-OVERLOAD).

The specification of a *Calendar-time* event consists of a time hierarchy value. Three examples of time-hierarchy values are: 1 min, 3 hours, and 12am January 8. For the first specification, the event occurs every minute; for the second, every 3 hours; and for the third, every year on Jan. 8th at 12am.

With each basic event is associated a *transaction time* and a *valid time*. Intuitively, the transaction time is the time at which the network management system becomes aware of the event, and the valid-time is the time at which the event occurs in the real world. So, for example, the transaction time of OVER-LOAD is the time at which the data-retrieval operation succeeds, and the valid time is the time at which the number of links that have a delay greater than 5 seconds exceeds the 20% threshold. When our discussion applies to both, the transaction time and the valid time we will refer to the *occurrence time* of the event.

Now we will be precise about the transaction and valid times for the different types of basic events. For a calendar-time event, the valid time is identical to the transaction time, and is derived from the specification. So, for example, assuming that the event specified as 3 hours is activated at 1pm, its occurrence-times are 1pm, 4pm, 7pm, etc. For a data-pattern event or a data-manipulation event, the transaction time is the time at which the data-retrieval succeeds or the data manipulation event completes, and these are obviously available to the database management system. If PERSISTENCE $\geq v$ is specified for the data-pattern event, then the transaction time increases by v. In other words, if the pattern is required to persist for at least some time interval, then, assuming that it does, the transaction time is the time at the end of the interval.

For a data-pattern event or a data-manipulation event, the valid time refers to an attribute of the data being retrieved or manipulated. Presumably the attribute stores the time at which the event occurred in the real world, and so it must be specified. For example, suppose that the relation LINKS has a STATUS attribute, that indicates 'up' or 'down', and a TIME attribute which indicates when the status became 'up' or 'down'. Then consider the data-manipulation event:

NEW LINKS WHERE STATUS='down'.

If *valid-time* = TIME is specified, then the valid time of the event is taken from the attribute TIME of the tuple being added or replaced. However, since multiple objects may be retrieved or manipulated, then some aggregate function (average, minimum, or maximum) of all the attribute values must be specified (e.g. valid time = max(TIME)). For a data-pattern event, if PERSISTENCE is specified then the valid time is the value of the aggregate function at the end of the time interval.

3.2 Correlated Events

Basic events can be grouped into correlated events. For example, a correlated event may occur if the basic event OVERLOAD (a data-pattern event) occurs at the same time as the basic event 12am (a calendar-time event). Formally, a *correlated* event is a disjunction of conjunctions of events, and is specified using rules. With a correlated event specification, two parameters are specified to capture the temporal relationships among the events in the specification. These parameters, namely time order and time constraints, will be discussed after the presentation of rules.

Rules

An *atom* is an event symbol (e.g. OVERLOAD), possibly preceded by the symbol "~"; if it is, then the atom is *negative,* otherwise it is *positive.* A *rule* consists of a positive atom designated as the *head,* and a set of one or more atoms, designated as the *body.* The body of a rule must contain at least one positive atom. The head is the *correlated* event being defined by the rule, and its symbol must be different than that of a basic event. An event in the body of a rule may be basic or correlated. Syntactically, the head and the body of a rule are separated, in the Prolog tradition, by the symbol ":-" . A rule has the following semantics. If all the positive-atom events in the body occur simultaneously, and at that time none of the negative atom events occurs, then the correlated event in the head occurs. In practice, simultaneity means a default time-interval of some length $\varepsilon>0$.

Consider the following correlated event:

OVERLOAD-AT-12 :- OVERLOAD, 12am.

This event occurs if the OVERLOAD event occurs in conjunction with the 12am calendar-time event.

If there is another basic event, UNDERUTILIZED which means that 20% of the links are underutilized, then the following event:

OVERLOAD-UNDERUTILIZED :- OVERLOAD-AT-12, UNDERUTILIZED.

may be specified. OVERLOAD-UNDERUTILIZED occurs if 20% of the links are overloaded at 12am, and the same fraction of the links is underutilized at the same time.

To demonstrate the use of negative atoms, consider the event

D-NEG :- OVERLOAD-AT-12, ~UNDERUTILIZED.

It occurs if 20% of the links are overloaded at 12, but UNDERUTILIZED does not occur then.

It is possible to define a correlated event consisting of a disjunction of two events, *A* and *B*, by having two rules with identical heads, and with the bodies *A* and *B*, respectively. For example, the event OVERLOAD-*OR*-12, that occurs when OVERLOAD occurs, or at 12am, which ever is first, can be specified by the following two rules:

OVERLOAD-*OR*-12 :- OVERLOAD.

OVERLOAD-*OR*-12 :- 12am.

The transaction (valid) time of a correlated event is the last transaction (valid) time of a positive event in the body of the rule. Using the parameter *Delay* (say, Delay = 5 minutes) the transaction-time of the correlated event can be postponed.

Temporal Order

The events represented by the positive atoms in the body of the rule may be required to occur in a certain temporal order, say G. If so, then this is specified by the keyword *order* = *G* associated with the rule. G is a directed acyclic graph, which in our case represents the time-precedence requirements for the specification of the correlated event. The nodes of G are the positive atoms in the body of the rule, and the arcs represent the time-precedence requirements. If there is a path from node *a* to node *b*, then *a* is required to occur before *b*; only then the event in the head of the rule can occur. If there is no path from *a* to *b*, nor from *b* to *a*, then there is no requirement as to the order in which these two events occur.

For example, consider the rule specifying the event OVERLOAD-UNDERUTILIZED, and assume that *order* = *G* is appended to the body of the rule. Suppose that G is the graph:

OVERLOAD-AT-12 → UNDERUTILIZED

Then the event OVERLOAD-UNDERUTILIZED occurs only if the event OVERLOAD-AT-12 occurs before the event UNDERUTILIZED. If G does not have have any arcs, then OVERLOAD-UNDERUTILIZED occurs if both events in the body occur, regardless of the order.

Generally, there may be a required order for transaction times, and a required order for valid times.

Temporal-Constraints

The events represented by the atoms in the body of the rule are required, by the keyword *time-constraints*, to satisfy certain temporal constraints, *C*. This is specified as *time-constraints* = *C* associated with the rule, where *C* is a set of constraints. Each constraint is represented by a subset, *S*, of the atoms in the body of the rule, and a time hierarchy value, *v*. *S* must contain at least one positive atom. The constraint indicates that in order for the correlated event to occur, the time-interval of length *v* starting at the occurrence of the first event from *S*, will contain the occurrence time of all the positive events in *S*, and will not contain the occurrence time of the negative events. For example, consider the definition of OVERLOAD-AT-12. A constraint, {OVERLOAD, 12am} = 5 seconds, says that OVERLOAD-AT-12 will occur only if OVERLOAD and 12am are at most 5 seconds apart. For another example, consider the rule specifying D-NEG. A constraint:

{OVERLOAD-AT-12, ~UNDERUTILIZED} = 5 seconds

associated with the rule says that D-NEG occurs if OVERLOAD-AT-12 occurs, and UNDERUTILIZED does not occur within 5 seconds after the occurrence of OVERLOAD-AT-12. The intersection of two constraints may be nonempty.

The time-interval of a temporal constraint often cannot be determined a priori. Then a procedure, *P*, may be specified instead of a time-interval. Then *P* is invoked at the first occurrence of an event in the body of the rule, and it will compute the interval for the constraint (possibly by communicating to the network operator).

Generally, there may be set of temporal constraints for transaction times, and a set of temporal constraints for valid times.

Notice that the two rule parameters; order and time-constraints, are independent of each other. For example, the *order* = *G* parameter may be specified independently of whether or not *time-constraints* = *C* is specified. If both are specified, then any set of constraints is consistent, and any combination of values for *C* and *G* is consistent. By consistency we mean that there are occurrence times for the (positive and negative) events in the body, such that the precedence order specified is satisfied, and so are all the time-constraints. This is proven in the appendix.

Variables

Each data-pattern and data-manipulation event is associated with a variable that is instantiated when the event occurs. The variable is instantiated to the set of tuples whose retrieval or manipulation triggered the event. For example, the variable associated with OVERLOAD denotes the percentage of links that have a long (> 5) delay. The variable associated with

ADD LINKS WHERE STATUS='down'.

is the set of tuples being added. The variables associated with events may be used to further constrain the occurrence of composite events to the cases in which the variables associated with the positive events in the body of a rule satisfy a certain predicate. For example, suppose that the variable associated with UNDERUTILIZED denotes the percentage of links that is underutilized. Then the correlated event defined as

OU :- OVERLOAD(*X*), UNDERUTILIZED(*Y*), *X* < *Y*.

occurs only if the percentage of underutilized links exceeds the percentage of overloaded links. In general, the variables associated with the positive events can be used to construct relational algebra expressions and arithmetic expressions; in turn, these can be combined into predicates using set- and arithmetic- comparison operators. Then the correlated event occurs only if the relational and arithmetic predicates so constructed are satisfied. The variable associated with a data-pattern event that has a PERSISTENCE interval is instantiated at the end of the time-interval.

The correlated event at the head of the rule may also be associated with a variable, e.g. X, as follows:

OU(X) :- OVERLOAD(X), UNDERUTILIZED(Y), $X < Y$.

In general, the variable associated with the correlated event in the head of the rule, is some relational algebra expression involving the variables associated with the positive events in the body of the rule.

4. Traces

We model a statistical test in network management by a trace. A *trace* is a sequence that tracks the changing of an attribute value of an object, and it is part of the history database. Each member of a trace is a pair, (t,a), where t defines the position of the member in the trace (may be a time-stamp or an ordinal number), and a is a attribute-value. For example, a trace may track the acknowledgement time of messages from processor 101 to processor 102.

Each trace has an identifier, i, that uniquely identifies the trace (e.g. 101,102). i may be an object-id, or a relational key. Each trace also has a start and a stop point, each of which is defined by an event.

It is relatively simple to specify a trace, however, often the user will want to define a set of traces, all of which have the same characteristics. For example, suppose that the user wants to specify a statistical test, that, say, monitors the acknowledgement time of messages, from each IBM processor to each DEC processor. Therefore the topic of the rest of this section is specification of a *trace collection.* Intuitively, each trace collection is defined with respect to a *monitored class,* or relation scheme, i.e., a set of objects that is being monitored. Each trace records the changes in an attribute of the monitored class called the *monitored attribute.* Usually, each trace records the sequence of changes to the monitored-attribute for one object in the monitored-class, but it can record the sequence of changes to more than one object. For example, there may be a trace for the "yellow" objects and one for the "green" objects, although there may be more than one object of the same color. Then the "yellow" trace consists of the values of the yellow objects, and if two such objects are changed simultaneously, then the two values are recorded in the trace in a nondeterministic order. An object which is neither green nor yellow will not have its monitored attribute traced. In other words, the traces (or their identifiers) partition the set of monitored objects. The changes to the monitored attribute in each partition are recorded in one trace, and one (possibly empty) partition consists of the objects that are not traced at all.

A trace collection is specified and (de)activated. Next we formally present the specification parameters for a trace collection, and then we shall provide a detailed example.

Monitored class: The class of objects being monitored. It may be an extensional object class, for example the class of nodes in the network, or an intentional (also called virtual, or view) object class.

Monitored Attribute: This is an attribute name in the monitored class. It identifies the attribute whose change is being tracked.

Trace-identifiers class: The key-word OBJECT-ID, or an intentional or extensional class of trace-identifiers.

Trace identifier: A subset of the attributes of the monitored class. Should be specified only if the Trace-identifiers class is not OBJECT-ID.

Sampling event: An basic or correlated event. Upon its occurrence the monitored attribute is sampled.

Change only: An indication to ignore sampling if the current value of the attribute is identical to the one in the previous occurrence of the sampling event.

Time-stamp : "Yes" or "No". Indicates whether the position of each member in a trace is a time-stamp or an ordinal number.

Start event: An event which determines when the trace begins.

Stop event: An event which determines when the trace ends.

Now we discuss the above parameters. The operational semantics of the trace-collection specification are as follows. At each point in time there is an instance (set of objects), *S*, of Trace-identifiers. For the example in the beginning of this section, (101,102) is a member of the Trace-identifiers class. If the Trace-identifiers class is 'OBJECT-ID', then *S* is the set of object identities of the monitored class. Otherwise it is an (extensional or intentional) relation. Whenever the sampling event occurs, each object, *o*, in the monitored class is examined; if *o*'s Trace identifier (the attributes {FROM, TO} in the message-acknowledgement example) appears in *S*, then the current value of the monitored attribute is

appended to the end of the identified trace. If Change-only is 'on', then appending is done only if the current value of the attribute is different than the one in the previous occurrence of the sampling event.

Notice that the instance of the Trace-identifiers class may change over time. For example, the color 'red' may be added to the class of trace-identifiers. In this case, a trace is started. In general, the addition of a trace-identifier to the class starts a trace, and its deletion stops it. Also, if the trace identifier value of an object, *o*, changes, e.g. from yellow to green, then the monitoring of *o* switches traces.

For a comprehensive example of a trace-collection specification, consider the problem of tracing the acknowledgement time of each message between a DEC and an IBM computer. The example is somewhat complicated, to demonstrate various subtleties of the trace collection specification. Suppose that there exist two extensional object classes: 1. PROCESSOR, having among others, the attributes ID (e.g. 117658), NETWORK_ADDR (e.g. A12), and TYPE (e.g. DEC), and 2. A wrap-around relation MESSAGE, having, among others, the attributes SOURCE_ADDR, DEST_ADDR, and ACK_TIME. The relation MESSAGE is assigned a fixed amount of storage, and addition of tuples wraps-around. When a message is acknowledged, the elapsed-time is recorded in the attribute ACK_TIME of MESSAGE. Notice therefore, that there may be more than one tuple having the same pair of SOURCE_ADDR and DEST_ADDR values.

The trace-collection, that we call MESSAGE_TIME, is specified as follows.

Monitored-class: MESSAGE.

Monitored-attribute: MESSAGE.ACK_TIME.

Trace-identifier: SOURCE_ADDR, DEST_ADDR.

Trace-identifier class: Is a relation, having a tuple for each pair of addresses such that the first is an IBM and the second is a DEC. The relation can be defined as a view, using SQL, as follows.

```
SELECT SOURCE_ADDR = p1.NETWORK_ADDR,  DEST_ADDR = p2.NETWORK_ADDR
FROM   PROCESSOR p1, PROCESSOR p2,
WHERE  p1.TYPE=IBM, p2.TYPE=DEC
```

Sampling-event: NEW tuple of MESSAGE.

Change-only: On.

Time-stamp: Yes

Start-event: OVERLOAD

Stop-event: OVERLOAD + 1hr.

With each trace-collection specification we associate a *corresponding* object-class, i.e., a class having the same name as the collection (e.g. MESSAGE_TIME). Each object in this class represents an activation of the trace-collection. The class has a set-attribute, S, that contains the traces. For example, the trace for a pair of addresses, $A23$ and $B45$, in an activation, say 324, of the trace collection MESSAGE_TIME, is denoted MESSAGE_TIME (ID=324, S = (SOURCE_ADDR=A23, DEST_ADDR=B45)).

Next we argue that there is a need for an additional parameter in the trace-collection specification. Suppose that the trace-collection MESSAGE_TIME, as defined above, is active. If the type of the processor at address A55 changes from IBM to HP, then each trace in which the surrogate value is A55,x is either disabled or erased for the test MESSAGE_TIME. If it is disabled and subsequently the type of the processor at address A55 changes back to IBM, then each disabled trace can be enabled, namely resumed. Otherwise, new traces having the surrogate values A55,x are started. Namely, the trace collection has an additional attribute:

Status = either 'resume' or 'anew'.

Each one of these two options may result in a different trace collection.

This also indicates that an object class corresponding to a trace-collection which is specified with Status = 'resume', should have two set-attributes: S.ENABLED and S.DISABLED. At any point in time, S.ENABLED and S.DISABLED are disjoint sets of traces.

5. Trace-based Events

We have mentioned that in network management alerts may be triggered by the evolution of the network-configuration instance over time. Now that we have defined trace collections, we can precisely explain this statement. Similarly to basic events that are represented by nonempty retrievals from the network database, basic events may also be represented by nonempty retrievals from the history database. The

history database is the set of active traces, and each trace is simply a relation with two attributes, time point (t) and monitored attribute (a). Consequently, nonempty retrievals from traces are not different than any other retrieval, and can be defined as basic events.

For example, consider the trace collection MESSAGE_TIME. Following is the specification of an event, called SURGE, that occurs when two messages in a trace, that are acknowledged within 5 seconds one from the other, have an increase in the attribute value (acknowledgement time) which is greater than 17.

```
SELECT SOURCE, DEST
FROM MESSAGE_TIME p1, MESSAGE_TIME p2
WHERE p1.ID = p2.ID, and p1.S = p2.S, and p2.t < p1.t + 5, and p2.a > p1.a + 17
```

Clearly, more than one data-pattern event can be specified for a trace collection, and such an event may be triggered by multiple trace collections. In other words, the relationship between trace collections and events is many-to-many.

6. Comparison to Relevant Literature

The work on active databases (e.g. [C*89]), on triggers (e.g. [DB87]), and on rule-languages (e.g. [KMS88, S89, SJGP90, WF90, KDM88]) has addressed the specification of events. However, data-pattern events and temporal constraints arising in real-time systems have not been discussed in these works. The specification of such constraints has been the topic of this paper. As we have seen, even the definition of basic events is more complex when one wants to smooth out the effects of transient conditions (as enabled by the PERSISTENCE parameter). On the other hand, the work on real-time databases (e.g. [So88, KSS90]) concentrates on concurrency control issues, a topic which is outside the scope of this paper.

The present proposal of traces is related to work on temporal databases (e.g. [SS87, SS88, Sn88, SG89]). However, the emphasis in that research is on the *manipulation* of traces, whereas we discussed their specification. This work is also related to temporal logic (e.g. [P77]), maintenance of temporal databases in Artificial Intelligence (e.g. [D89]), and temporal extensions to the relational model (e.g. [Sn87]). However, the emphasis in such research is on the *inference* of temporal knowledge, rather than the *specification* of temporal constraints.

7. Conclusion and Future Work

We considered database management systems that receive an automatic inflow of data from various sensors, in real-time. We argued that for such applications new database-language features are necessary, for specification of treatment of such data (e.g. create traces) and actions to be taken upon certain arising conditions (events). We proposed language features for specifying basic events, correlated events and trace collections. The emphasis of the proposal is on the specification of temporal constraints that pertain to data-manipulation operations. We demonstrated the applicability of these language features using network management. We feel that this is an emerging important application of knowledge-base and active database technologies.

We showed how statistical-tests, alerts, and correlation among alerts, can be represented as trace collections, events, and correlated events, respectively.

In the future we intend to extend the proposed language into a complete event-trace model, in which the basic building blocks are events and traces, as schemas and tuples are in the relational model. The work in [SS87, SS88] can serve as a starting point for a language to manipulate events and traces. We will study the processing of such a language, particularly in a distributed environment. We will also study the static analysis of an event-trace specification to answer questions such as: can this correlated event occur, or, what is the maximum time between the occurrence of two correlated events, or, what is the minimum length of a trace that guarantees that no events based on the trace will be lost.

References

[B89] F.E. Boland, Ed., "Working Implementation Agreements for Open Systems Interconnection Protocol", IEEE Computer Society Press, CA, 1989

[BFKM86] L. Brownston, R. Farrell, E. Kant, and N. Martin, "Programming Expert Systems in OPS5: an Introduction to Rule-Based Programming", Addison-Wesley, 1986.

[CCCTZ90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanaka, and R. Zicari, "Integrating Object Oriented Data Modelling with a Rule-Based Programming Paradigm", Proc. of the ACM-Sigmod International Conf. on Management of Data, 1990.

[CDFS88] J.D. Case, J.R. Davin, M.S. Fedor, M.L. Schoffstall, "Simple Network Management Protocol", Request for Comments 1067, Network Information Center, SRI International, Menlo Park, CA, September 1988.

[C*89] S. Chakravarthy et. al., "HiPAC: A Research Project in Active, Time-Constrained Database Management", TR XAIT-89-02, Xerox Advanced Information Technology.

[D89] T. Dean, "Using Temporal Hierarchies to Efficiently Maintain Large Temporal Databases", JACM 36(4), Oct. 1989.

[DB87] M. Darnovsky and J. Bowman, "TRANSACT-SQL USER'S GUIDE" Document 3231-2.1 Sybase Inc., 1987.

[DE88] L. M. L. Delcambre and J. N. Etheredge, "A Self Controlling Interpreter for the Relational Production Language", Proc. of the ACM-Sigmod International Conf. on Management of Data, 1988.

[KDM88] A. KOTZ, K. Dittrich and J. Mulle "Supporting Semantic Rules by a Generalized Event/Trigger Mechanism", Proc. of the EDBT'88, Springer Verlag LNCS 303.

[KSS90] H. Korth, N. Soparkar, A. Silberschatz, "Triggered Real-Time Databases with Consistency Constraints", Proc. VLDB, Aug. 1990.

[P77] A. Pnueli, "The Temporal Logic of Programs", Proc. 18th Symposium on FOCS, IEEE 1977.

[P88] C. Partridge, Ed., Special Issue on Network Management, IEEE Network 2(2), March 1988.

[S89] T. Sellis, Ed., Special Issue on Rule Management and Processing in Expert Database Systems, SIGMOD RECORD, 18(3), Sept. 1989.

[So88] S. H. Son, ed., ACM SIGMOD RECORD, Special Issue on Real-Time Database Systems, Mar. 88.

[Sn87] R. Snodgrass, "The Temporal Query Language TQuel", ACM Trans. on Database Systems, 12(2), June 1987.

[Sn88] R. Snodgrass, ed., Data Engineering, Special Issue on Temporal Databases, Dec. 1988.

[S90] M. Stonebraker, ed., IEEE Trans. on Data and Knowledge Engineering, 2(1), Special Issue on Database Prototype Systems, March 1990.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, "On Rules, Procedures, Caching and Views in Database Systems", Proc. of the ACM-Sigmod International Conf. on Management of Data, 1990.

[S*90] M. Stonebraker, L. A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, D. Beech "Third Generation Data Base System Manifesto", Memorandum UCB/ERL M90/28 UC Berkeley.

[SS87] A. Segev and A. Shoshani, "Logical Modeling of Temporal Data", Proc. of the ACM-Sigmod International Conf. on Management of Data, 1987.

[SS88] A. Segev and A. Shoshani, "The Representation of a Temporal Data Model in the Relational Environment", 4th Inter. Conf. on Statistical and Scientific Data Management, June 1988.

[SG89] A. Segev and H. Gunadhi, "Event-Join Optimization in Temporal Relational Databases", Proc. VLDB, Aug. 1989.

[U89] J.D. Ullman, "Principles of Database and Knowledge-Base Systems", Vols. 1 and 2, Computer Science Press, 1989.

[WF90] J. Widom, S. Finkelstein, "Set-Oriented Production Rules in Relational Database Systems", Proc. of the ACM-Sigmod International Conf. on Management of Data, 1990.

APPENDIX

We will prove that any set of temporal constraints is consistent with any order graph. Let A be the set of atoms appearing in the body of an instantiated rule (a rule in which all variables are replaced by constants). OVERLOAD and ~OVERLOAD(0.3) are examples of atoms. We assume that an event and its negation do not both appear in the body of the rule. The *occurrence time* is a function that maps each atom $a \in A$ to a nonnegative real number, $o(a)$. Intuitively, the occurrence time of a negative atom is the occurrence of the

positive event. Denote by $P \subseteq A$ the subset of positive atoms. Let E be a set of ordered pairs, each of which has two members of P. The pair (P,E) is an *order* graph if it is acyclic. A *constraint*, c, is a pair (s,b), where s is a subset of A which contains at least one positive atom, and b is a positive real number. b is called the *interval* of the constraint. The next proposition states that for any given set constraints and for any order-graph, there is an assignment of occurrence times that satisfies the constraints and the order.

Proposition: Let C be a set of constraints, and $G=(P,E)$ an order graph. Then there exists an occurrence-time function, o, such that: (1) For any pair of positive atoms, e and f, and for any constraint $c = (s,b)$ of C such that e and f are in s, $|o(e)-o(f)| \leq b$; and (2) For any negative atom, e, and for any constraint $c = (s,b)$ of C such that e is in in s, there is a positive atom, $f \in s$, such that $o(e)-o(f) > b$; and (3) For any pair of positive atoms, e and f, if $(e,f) \in E$, then $o(e) < o(f)$.

Proof: The proof is constructive, demonstrating an occurrence time function with the desired properties. Intuitively, we "squeeze" all the positive atoms in the smallest interval, that starts from 0, and postpone all the negative atoms until very late. Formally, let us first assign occurrence time to the positive atoms. Let b_1 be the smallest bound of a constraint in C. For an atom, $a \in P$, that does not have any predecessors in G, $o(a)$ is 0. For the other positive atoms we assign occurrence times as follows. Let a be an atom for which all the immediate predecessors in G, have been assigned occurrence times; assume that m is the maximum occurrence time of such a predecessor. Then we assign $o(a) = m + \varepsilon$ where $0 < \varepsilon < b_1 - m$. Since G is acyclic, after at most $|P|$ iterations, all the positive atoms are assigned occurrence times. Let b_2 be the biggest bound of a constraint in C. All the negative atoms of A are assigned the occurrence time $2 \cdot b_2$.

Now it is easy to see that our occurrence time function satisfies the three requirements of the proposition. The first requirement is satisfied since all occurrence times of the positive atoms are within the smallest interval. The second requirement is satisfied since the occurrence time of a positive atom is smaller than b_1, and $2 \cdot b_2 - b_1 \geq b_2$. The third requirement is satisfied by the way the occurrence-time function, o, was defined. []

If the occurrence time of an atom must be a natural (rather than real) number, then the above proposition does not hold. For example, if G is the graph $a \rightarrow b \rightarrow c \rightarrow d$, then the single constraint $o(d)-o(a) < 2$ cannot be satisfied.