Event Database Processing

Joris Mihaeli, Opher Etzion
IBM Haifa Research Lab
Haifa University Campus
Mount Carmel, 31905
Haifa, Israel
{jorism, opher}@il.ibm.com

Abstract: The purpose of the current work is to explore and improve the analysis of event data stored in event repositories, enabling the application of specialized event algebra operators over the event data. We consider an event data model and apply an event specification language for detection of situations (patterns) over the event history. We introduce layered information architecture and consider implementation issues. Finally, we consider a proper query execution order for the detection process.

1. Introduction

The events occurring in particular application areas can be stored in specialized repositories in order to provide a basis for more thorough and complex interpretation and correlation between the events. Consequently, it is feasible to perform offline querying over the event history and detection of particular patterns leading to the discovery of additional (composite) events.

There are several products and projects that support event stores, such as IBM's Tivoli Enterprise Console [5] or the ongoing work on realizing the Common Event Infrastructure [2]. The different event repositories use either data management systems – relational or object-oriented DBMS, or dedicated storage facilities. The DBMS-based approaches use standard languages – SQL or OQL. However, the query capabilities are rather general, treating the event data as standard relational or object-oriented data without support of more complex pattern detection. The aim of the current work is to augment the capabilities of the standard relational systems, applying notions from event specification languages. These languages have been primarily proposed and investigated in the area of active databases [10], [11], [13], [14], [15], and [22]. The active mechanisms in these systems serve to detect and react to updates in the database that may jeopardize the data integrity, or to execute some business-related application logic. The event languages support event algebra operators that can be applied over the streaming events in real time.

The main idea we pursue is to add event algebra operations to the relational algebra in order to be able to detect more complex patterns (composite events) in the event databases. We realize event algebra operators using standard SQL language and adding programming logic wherever the expressive power of SQL is not sufficient.

The particular event specification language that we apply was designed and is being applied in various application areas for complex real-time event processing [1], [8].

In this paper we consider a generalized event database, review a particular event definition language and propose information architecture for the event query processor component. Then we consider system architecture and implementation issues. Finally, we review related results in several research areas.

2. Event Database

In general, an event is a message about some activity of interest, and it belongs to an event type. An event type has some similarity to an object type, encapsulating a number of attributes. The attributes of a particular event (or an event type instance) uniquely identify the activity of interest (its parameters). The events belong to particular event types, which are specializations of higher event types. All the event types inherit from a base event class. The base event type consists of base attributes, such as the time point at which the event occurs (or is observed), a unique event identifier, an event type name, the identity of the event source, etc.

The events that may have happened can be primitive event types and composite event types. A primitive event is assumed to be instantaneous and atomic – it cannot be further decomposed and it either happens or doesn't happen. Every primitive event occurs at a particular time point, and we assume here that time consists of a linearly ordered infinite number of discrete time points. All the events are considered ordered in time and occurring at different time points. They don't overlap because they have no duration.

The composite events (alternatively: situations) are compound events that encapsulate primitive and composite events. The compound events span some time interval during which the constituent events happen. We will, however, associate them with a single time point of occurrence. A composite event is considered to occur when the last primitive or composite event that defines it takes place. Thus, we may assume that the composite events also have no duration and occur at the time point of the last event from the particular combination that it is defined of. Furthermore, we define the composite events as regular events that belong to and have attributes of an event type. Another aspect of the composite events is that they can be detected during particular time intervals (that may include parts or the whole event history). This means that the composite events are valid during particular time intervals.

It is important to be able to distinguish between events that occur at particular time points, to relate events and time intervals during which they occur, and to reason about the various time intervals. As said earlier, the events occur at particular time points, have no duration, and the time points are linearly ordered. Thus event A can occur *before* event B (A < B), while event B happens *after* event A, or *follows* event A (B > A). The different events cannot overlap.

The time intervals include set of time points between their starting and ending time points and have duration. An event A can happen *during* time interval I or *outside* of it. The time intervals have duration, and consequently there exist more complex relationships between them. In order to reason about the different time intervals we apply the interval algebra, proposed by J. Allen [9]. This work introduced taxonomy of relations between temporal intervals that is complete for time intervals over a linear set of time points. According to the taxonomy between two intervals I and J, there may be 13 possible relationships:

I *precedes* J - I happens before J and there is non-empty interval separating them
I *meets* J - I happens before J, but there is no non-empty interval between them
I *starts* J - I has the same starting point as J, but ends earlier than J
I *overlaps* J - I starts before J, and J ends after I
I *equals* J - I and J start and end simultaneously
I *finishes* J - I starts after J, but has the same end point
I *during* J - I starts after J, and finishes before J.

Additional relationships are the converse relationships of all the aforementioned relationships (except *equals,* which is symmetric). The 13 interval relationships are mutually exclusive.

An event database stores event instance tuples that contain data from the received message data about the activities of interest in the particular application area. The events have a flat structure, and have a unique name and attributes that can be standard or user-defined. The event tuples may reflect several temporal dimensions like occurrence time, detection time, and/or transaction time (the time when the event data is stored in the database). The detection time may not reflect the actual order of occurrence of the events as they may be received at a different order, and also because the events may occur in distributed environments and the sensors may have different settings. On the other hand, the transaction time sequence is truly a linear temporal order because the events are stored one at a time.

The different event types constitute a generalization hierarchy (Fig.1). The root event type of the hierarchy contains common event attributes, while the children contain the specific user attributes for the various event types that can be defined for the particular application area. The common attributes may include an event instance unique identifier, an event detection time, a transaction time. Other common attributes may include the event source and the actual component where the event occurred, event status, severity code, and priority. The conceptual model defines an event type generalization hierarchy that is specific for the particular application area, and contains two additional non-overlapping specializations: primitive/composite and retained/consumed.

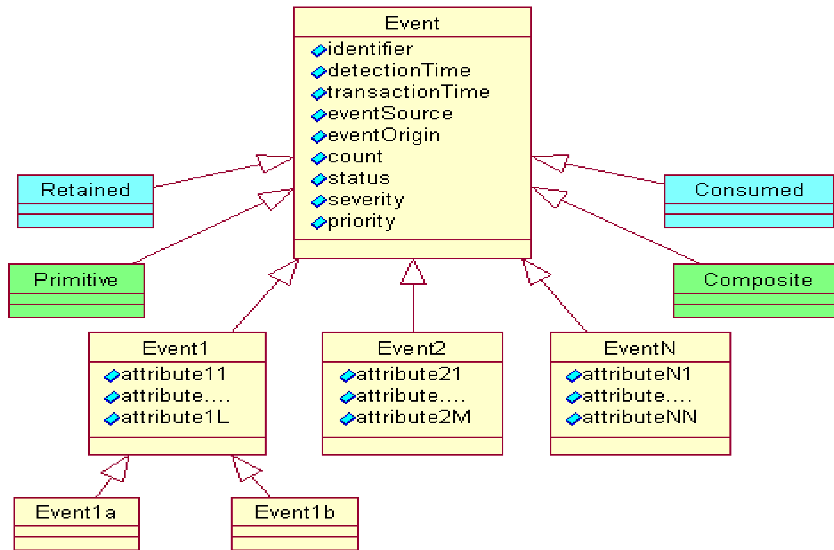

Fig.1. Event database conceptual model

A similar approach is used in the Tivoli Event Console [5]. The supported event model forms a nested multi-level generalization hierarchy. The root class of the hierarchy contains the common attributes of all the event classes. The classes at the lower levels inherit the common attributes, and the attributes of their ancestors. Some of the attributes can be redefined, or dropped. The events have a flat data structure.

Another approach is pursued in the Common Base Event (CBE) data model [2], [3]. CBE assumes that the events can have a hierarchical nested structure of the extended data elements (user-attributes). Thus, CBE may include also aggregation hierarchies for particular event types. Such an assumption leads to a more complex database structure in which the events, having aggregations, are realized with additional dependant tables, connected via PK/FK relationships to the main event table.

In the following considerations we assume that base tables are defined for each of the event types. The root relation contains the events' common attributes. This approach allows performing temporally oriented operations only on the root relation, while searching for specific user attribute values in the specialization relations. In addition to the base tables, we define relational views for the event classes of the conceptual model. The relational views allow developing generalized algorithms, implementing the event specification language constructs that are independent from the concrete physical database structure.

3. Event Query Processor

The event database contains the event data in multiple relations, one of which also has a timestamp column reflecting the temporal dimension of the events. The standard means of the relational DBMS allow the specification and execution of general ad-hoc queries over the event data. However, more specific queries, including a search for specific patterns and combinations of events, are difficult to achieve. This is due to the fact that the event data has to be processed in sequence, while the SQL language is a set-oriented language. In addition, the event data has certain temporal aspects that are not supported in the standard SQL (SQL-92) provided by the most current commercial DBMS products [12], [21].

We consider the event specification language that is to be supported by the event query processor. The event query processor parses the event definition specifications of the Amit event rule engine, builds the appropriate SQL logic, and queries the event database. In case of detected situations, it inserts composite events in the database and takes appropriate actions.

3.1. Basic Constructs of Amit

Amit uses XML-based specifications that define the basic event-related entity types of Amit – events, situations, lifespans, and keys. In the following, we briefly examine the definition constructs of Amit. More elaborate description can be found in [1], [8].

3.1.1. Events

The events are the basic underlying entities in Amit. Each event instance belongs to an event type. The optional "reference" element points to a set of event types that relate to the given event type. Amit supports several relationship types: generalized, cross-section, influenceOn, and dependOn.

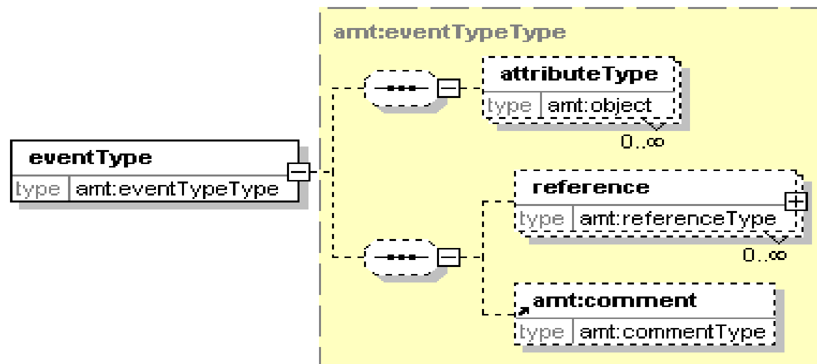

Fig.2. Event type definition

3.1.2. Keys

Keys can be defined both for lifespans and situations (described below). The "key" element allows relating particular event instances (forming groups) according to application specific criteria. Such criteria can be, for example, relating event instances to particular entities in the application area, particular applications, sources, etc.

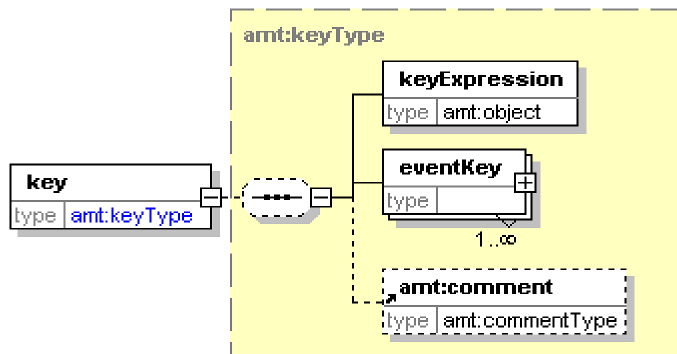

Fig.3. Key type definition

Each key has a unique name. A key value can be either an attribute, or an expression involving specific event attribute values. The "keyExpression" element allows specifying the data type of the common event attributes and, optionally, the expression to be calculated.

3.1.3. Lifespans

The lifespan construct is used in Amit to define intervals (sequences of event instances) during which specific situations are valid and might be detected. A lifespan has one or more initiators, zero or more terminators, and may have related keys.

The lifespan initiator specifies that the situations, valid for the lifespan, may occur from the beginning of the event history (startup element), or with the occurrence of a given event type instance. The lifespan initiator may also define selection conditions for the event instances.

The lifespan terminator can be particular event type instances, a time interval after the beginning of the lifespan, or a particular point in time. There may be several different possible terminators specified. Alternatively, a lifespan may continue indefinitely. For each event type that may terminate a lifespan an "eventTerminator" element is defined. It may define also selection conditions to comply with. The "eventTerminator" element specifies whether the event instances that are considered during a lifespan are to be discarded from further considerations after the termination of the lifespan and, in case of multiple initiated lifespans, it specifies which lifespans should be terminated – first, last, or all of them.

The lifespans may have related keys. A key, related to a lifespan, can specify conditions for the lifespan initiator(s), terminator(s), and eventually for the related situation's operands. If a key is introduced, then only event instances that have the same key value can initiate and/or terminate a lifespan.

3.1.4. Situations

The situation definitions are the main construct in the event specification language. A detected situation may fire predefined external actions. A detected situation is reported in Amit as a new event and inserted in the input queues. This allows the chained (or nested) execution of several situation operators. (Amit allows single operator per situation definition.) A "situation" element defines a unique name of the situation, and the interval during which the situation is valid.

Other features of the situation definition include operator and situation attributes. A situation is defined using "situationAttribute" elements in the situation definition construct to specify event attributes and values, and by "attributeType" elements in the corresponding event definition to specify attributes' data types. The situations can have the following detection mode types:

- immediate – a situation is fired immediately after detection;
- delayed – situation detection is immediate but is reported only at the end;
- deferred – situation detection is performed only at the end of the lifespan over all the event instances.

In the following sections we consider the main situation operators of Amit. Every operator type may have multiple operands. The operands specify the event instances which are considered, and conditions for these instances.

3.1.4.1. Joining Operators – ALL, SEQUENCE

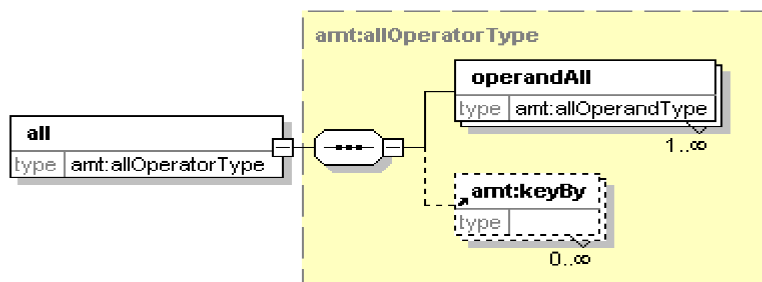

Fig.4. Operator ALL definition

The ALL operator defines a situation where all the event types that are looked after have event instances during the situation lifespan. This is a conjunction of event occurrences without requiring some specific order between them.

The “operandAll” elements define every participating event type. It may also specify filtering conditions for the participating event instances, and determine what to do when multiple event instances of the event type are encountered – select the first, last or all the instances. Additional more sophisticated conditions may specify whether a new event instance overrides the previous occurrences, and whether matching event instances are to be reused for further detection for the same situation or not.

The SEQUENCE operator has a similar definition. It defines a conjunction of event occurrences in a particular order. It can also define conditions for the participating events.

3.1.4.2. Counting Operators – ATLEAST, ATMOST, Nth

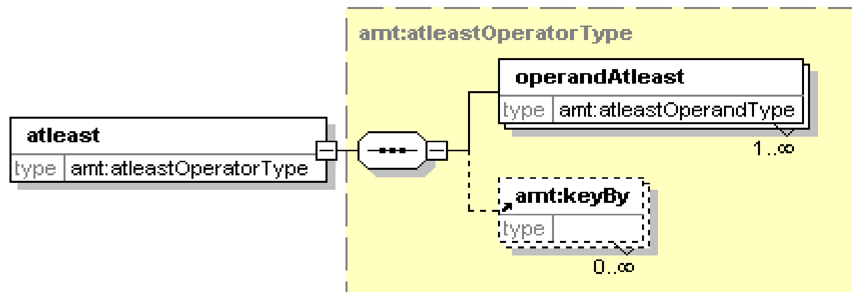

Fig.5. Operator ATLEAST definition

The operator ATLEAST defines a situation where at least N event instances satisfy some conditions during its lifespan. It also specifies the minimum number of event occurrences. The “operandAtleast” element defines expected event instances. In addition to the previously described options, it can define different weights for the different event types and which event types are to be considered for the detection process.

Similar definitions have the operators ATMOST and Nth. The operator ATMOST defines a situation in which there are at most N event instances that satisfy the specified conditions. The operator Nth defines a situation according to which an N in a succession event occurrence is detected in the event history. The “operandAtmost” and “operandNth” elements define the participant event types.

3.1.4.3. Absence Operators – NOT, UNLESS

The operator NOT defines situations where no event type occurrences have occurred during a given lifespan (monitoring interval) in the event history.

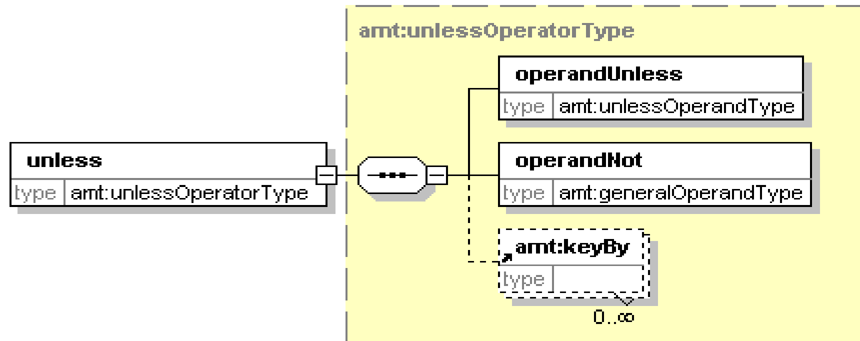

Fig.6. Operator UNLESS definition

The operator UNLESS defines situations where at least one instance of the first event type occurred, and no instances of the second event type occurred during the specified lifespan.

3.1.4.4. Temporal Operators – EVERY, AFTER, AT

The operator EVERY defines a situation that occurs every N time units during the specified lifespan. The operator AFTER defines a situation which takes place N time units after an event occurrence has happened. The operator AT defines a situation that is to occur at a given point in time.

3.2. Architecture

Following is the general architecture of the Amit system, including the event query processor component.

Fig.2 presents the overall architecture of Amit, including EQP. The Definition Manager (DM) has a separate parser dedicated for EQP – to discover errors and unsupported parameters. DM will also invoke the Amit parser to discover inconsistencies related to Amit. Then the definitions are passed to EQP for further processing. EQP constructs relevant lifespan views over the event database, builds and executes queries for the defined situations, and implements additional logic as necessary. The detected situations are passed to the Amit dispatcher for possible further processing.

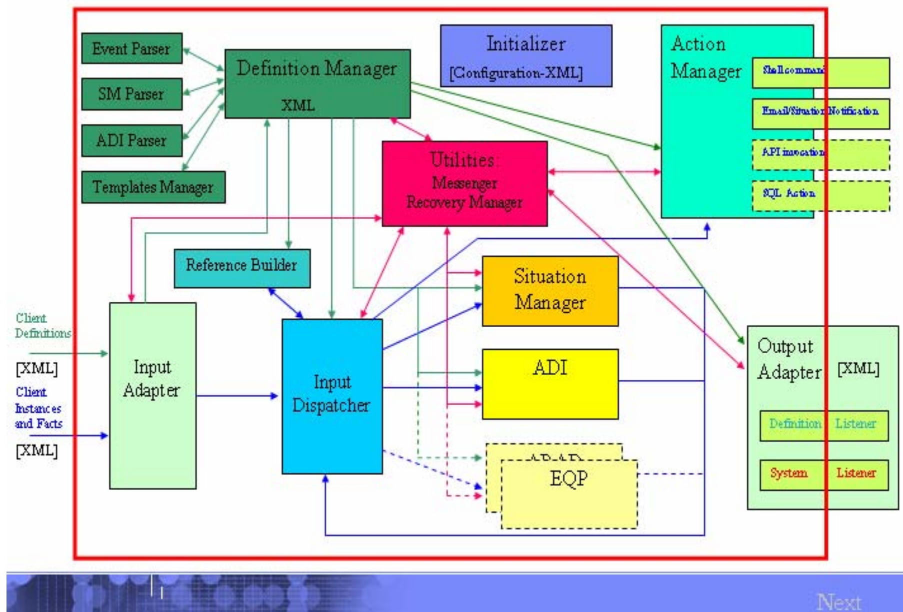

Fig.7. Architecture of Amit

Fig.3 presents the layered information architecture of the event query processor. Each of the layers is independent of the adjacent layers above and below it, which allows transparent modification and further development. The bottom layer consists of a set of base tables for the root class and the descendent classes, a set of referential integrity constraints over the tables, and a set of views.

The next layer consists of a set of views that present the relevant lifespan subsets of the event data. Each lifespan has an initiating and terminating event or time point, thus having starting and ending time points as a time interval. Between the initiating and terminating events there is a sequence of events that belong to the time interval and are a subset of all the events in the database. These subsets are dynamic and can change with the detection of new events. We realize the defined lifespans as dynamically formed views over the event database.

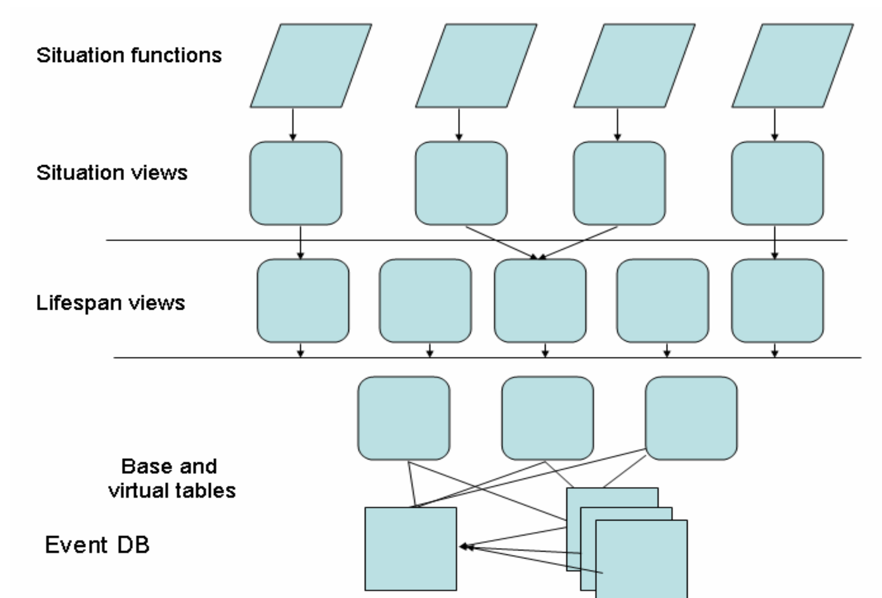

Fig.8. Information architecture

The top level of the information architecture contains the situations of Amit. The situations are defined as views (select statements) over the underlying lifespan views. The situation views comprise the part of the Amit situations that is purely SQL-oriented. It filters the lifespan views and presents further refined subsets of the event instances. Over these subsets are realized situation functions that detect possible composite events. The functions comprise the event algebra oriented logic that augments the relational algebra operations. In case of such detection, relevant event instances are inserted in the event database and defined actions are taken.

4.3. Implementation Issues

4.3.1. Lifespans

The Amit lifespans correspond to time intervals consisting of discreet time points in which the event instances occur. We define such lifespans using database views, referenced while executing the situation queries. Using views allows defining virtual tables containing relevant subsets of the event instances that can change dynamically with the detection of new composite events.

The variety of initiators and terminators allows us to express very complex conditions in determining a lifespan for a given query. In all cases, we have to determine the minimal time point at which one of the several possible initiators, as well as the minimal time point at which one of the terminators are encountered in the event history. In order to define a lifespan we have to determine starting and terminating

time points, and a set of relevant event instances. We use transactionTime for temporal dimension, but it is plausible to use detectionTime.

Here follows a sample lifespan definition. The definition doesn't use keys. It presents a time interval that starts with the first occurrence of the initiating event and ends with the first occurrence of the terminating event or 90 minutes after the start of the lifespan.

```
<lifespan name="L_90AfterActivity" updateDefinition="add">
    <initiator>
        <eventInitiator name="CustomerBuy" correlate="ignore"/>
        <eventInitiator name="CustomerSell" correlate="ignore"/>
    </initiator>
    <terminator>
        <eventTerminator name="CustomerBuyl" quantifier="first" />
        <eventTerminator name="CustomerSell" quantifier="first" />
        <expirationInterval timeInterval="5400000" />
    </terminator>
</lifespan>
```

The lifespan is realized as a view using common table expressions.

```
CREATE VIEW AMT.L_90AfterActivity AS
WITH
  T (MinTime) AS
    (
     SELECT MIN (AMT.V_Event.tempDimension)
     FROM AMT.V_Event, AMT.V_CustomerBuy, AMT.V_CustomerSell
     WHERE AMT.V_Event.eventName IN ('CustomerBuy', 'CustomerSell')
    ),

  T1 (MaxTime1) AS
    (
     SELECT MIN (AMT.V_Event.tempDimension)
     FROM AMT.V_Event, AMT.V_CustomerBuy, AMT.V_CustomerSell
     WHERE AMT.V_Event.eventName IN ('CustomerBuy', 'CustomerSell')
     AND AMT.V_Event.tempDimension > (SELECT MinTime FROM T)
    ),

  T2 (MaxTime2) AS
    (
     SELECT MinTime + 00000000009000.000000
     FROM   T
    )

SELECT * FROM AMT.V_Event
WHERE tempDimension >= (SELECT MinTime FROM T)
```

```
 AND    tempDimension <= (SELECT MaxTime1 FROM T1)
 AND    tempDimension <= (SELECT MaxTime2 FROM T2);
  -- Duration format:yyyymmddhhmmss.zzzzzz
```

Here the first table expression shows the timestamp of the first event occurrence of event types: CustomerBuy and CustomerSell. This will be the start of the lifespan. The second expression defines the timestamp of the first event occurrence of types CustomerBuy or CustomerSell after the beginning of the lifespan. The last expression defines a timestamp that is 90 minutes after the start of the lifespan. The end of the lifespan is defined by the minimum of the two timestamps just defined.

A more complicate case is when a lifespan has also keys defined. Here keys are used to define that only particular event instances from the initiator and terminator event types can initiate and terminate a lifespan.

4.3.2. Situations

Each situation definition consists of three distinct parts:

1) Header - defines the situation identifier (name), and a lifespan during which the situation is valid. It also defines the detection mode over the event instances.
2) Situation operator with additional qualifiers
3) Composite event data

The specifics of EQP impose a "deferred" detection mode – the detection is done at the end of each lifespan taking into account all the event instances.

The situation operator has two components. One of them serves as an additional filter over the lifespan view. It defines conditions on all the participating event instances, and conditions per specific event type. The other component relates to the specifics of the event algebra that we realize in EQP. This includes the concrete event algebra operators (situation operators), and the event types we are looking for. Here we relate also additional parameters that specify further conditions for the candidate event instances. Amit supports the following event algebra operators:

1) Conjunction – *all* (E1, E2… En)
2) Sequence – *sequence* (E1, E2… En)
3) Weighted conjunctions – *atleast* (n, E1, E2, …En), *atmost* (n, E1, E2, …En), *Nth* (n, E1, E2 …En)
4) Negation – *not* (E1, E2… En)
5) Absence - *unless* (E1, E2)

Additional temporally oriented operators are *every* (t), *after* (t), and *at* (t) and statistics oriented operators report (), crosses (), and percentage ().

In considering the specifics of the event algebra operators to be realized in EQP, we have to consider typical of the active databases such aspects as event instance

selection, and event instance consumption policies. In Amit we can specify the first, last, or all candidate instances for a specific event type that satisfy conditions to be selected. The event instance consumption policy defines whether we consider the event instances that were used to detect a given situation further in the detection process, or drop them.

The third part of the situation definitions defines the composite event that may be announced to external consumers (if not defined as internal) and be inserted in the database. As noted above, every detected situation is considered a composite event. The concrete composite event instances are constructed on the basis of the set of event instances that the detected situation is comprised of.

Let us consider some of the situation operators of Amit and describe possible detection logic. The operator ALL defines a conjunction of events without a particular order in their occurrence. Each participating event type is defined via an operandAll element. The ALL operator can be realized following the steps below:

1. Check whether situation ALL for N event types exists - if the number of events equals N, then situation ALL is detected.

```
SELECT  COUNT (DISTINCT eventName) AS C
   FROM S_View
  WHERE eventName IN (operandAll.eventType1… operandAll.eventTypei);
```

2. Find the event instances of each event type with the lowest timestamps

```
SELECT * FROM
      (SELECT X.*, RANK () OVER
            (PARTITION BY name
             ORDER BY transactionTime ASC)
       AS R1 FROM SituationView X) AS XXX
WHERE R1 = 1
    AND eventName IN (operandAll.eventType1… operandAll.eventTypei)
ORDER BY tempDimension;
```

The operator SEQUENCE defines a conjunction of events with a particular order in their occurrence. In order to realize it, the following steps are performed:

1. Select all instances ordered by their timestamps
2. Transform the ordered set name into an alphabet sequence.
3. Build regular expressions
4. Detect patterns in the alphabet sequence.

The counting operators have an additional attribute ("quantity") that specifies the required weight of event instances.

The considerations so far are based on the assumption that the situation definition language of Amit is to be used to analyze the event data. In addition, situation procedures described above can be built as user-defined functions (UDF) and invoked directly in SQL statements.

4.3.3. Query Execution Order

The defined situations are processed as queries over the database, and after detecting valid situations EQP inserts composite events into the event database. Depending on the event consumption policies EQP may also drop from further consideration the event instances that triggered a situation detection (constitute the compound composite event). Consequently, the order of execution of the situation queries is important for the correct detection of composite events in the event history.

Of further consideration is the number of query executions. If there are no situations in the event histories, the number of executions is equal to the number of the defined situations. If there are detected situations, it will be necessary to re-execute some of the queries. A brute force approach would be to re-execute all the queries in their temporal order as long as there are detections. We could, however, reduce the number of executions following some more elaborate strategy. The algorithm for determining the proper query execution order must satisfy the following goals:

1. Ensure detection of all possible situations over the event history.
2. Minimize the overall number of query executions

Intuitively, the order of execution depends on the time order of the temporal intervals over which the situations are defined, and their interval relationships. (In addition it is possible to assign priorities to the situations and execute the queries accordingly, but this is orthogonal to the current considerations.) The algorithm for query execution ordering consists of ordering the defined lifespans in ascending order of their starting time points, and their end time points. In order to reduce the number of query executions, we use the interval relationships between the related lifespans forming groups based on the relationships. This effectively builds a lifespan graph where the nodes are the defined lifespans and the edges are the interval relationships. Then an execution path is established traversing the graph.

5. Related Work

Our work is based on the results in event specification languages that were proposed in the area of active databases. Rich event algebras have been proposed for a number of systems, including SAMOS [13], ODE [14], [15], Snoop [10], [11], etc. An excellent survey of the semantic issues involving composite events was done by Zimmer and Unland [22]. Using a formal meta-model, the authors point to a number of semantic inconsistencies and ambiguities in the proposed event specification languages. All these works are aimed at realizing event languages that are processed at run time by the active database systems to detect dynamic composite events.

Sequence databases are a related area. There are several proposals to extend the database query languages with means to search for sequential patterns. Examples are the time series systems, such as the time-series data blades in Informix [4], and the analytic functions in Oracle [6], [7]. Sequences are supported as user-defined types

(UDF) and stored in tuple fields. Users are provided with a library of functions that can be invoked from SQL queries. However, the functions provided are rather general and do not support a more complex and dedicated class of queries. Another approach seeks to extend SQL to support sequenced data. Such extensions were first proposed for the PREDATOR system (SEQUIN [19], [20]). SRQL [16] extended the relational algebra with sequence operators for sorted relations, and added constructs to SQL for querying sequences. Then, more powerful extensions for pattern recognition over sorted relations were proposed for SQL-TS [17], [18]. It also proposed techniques for sequence queries optimizations.

These works are aimed at finding more general solutions for sequence data. They do not take into account the specifics of the event data. Our results are based on the results in the event specification languages and applied for detection of composite events in event databases. In addition to searching for particular event patterns (situations), we realize such typical for the active databases policies as event instances selection and consumption policies. Thus the results are more focused and more suitable to realize offline complex event processing.

6. Summary

This work has shown the motivations for augmenting relational algebra operators with event algebra operators. We realize an event query processor that accepts as input the event rule definitions of the Amit event processing engine. In order to design generalized algorithms we define a general event database structure. Then we introduce layered information architecture for the query processor and consider the logic to implement the different Amit constructs. Finally we consider the implications of detecting composite events on the situation queries execution order, and define an approach based on the interval relationships between the situation lifespans.

References

1. Amit User's Guide, version 2.0, November, 2003.
2. Common Event Infrastructure, IBM, 2003
3. Common Situation Data Format: The Common Base Event– proposal to OASIS, 2003
4. IBM Informix Time Series Data Blade Module, User Guide, V.4.0, 2001
5. IBM Tivoli Enterprise Console – documentation http://publib.boulder.ibm.com/tividd/td/EnterpriseConsole3.9.html
6. Oracle8 Time Series Data Cartridge, White Paper, February 1998
7. Migrating from Oracle Time Series to Oracle Analytic Functions: Time Series Functions, White Paper, November 1999
8. Adi, A., Etzion, O.: "Amit – The Situation Manager", The VLDB Journal, to be Published
9. Allen, J.: "Maintaining Knowledge about Temporal Intervals" Comm. ACM, Vol. 26, Num. 11, November 1983, pp. 832-843
10. Chakravarthy, S., Mishra, D.: "Snoop: An Expressive Event Specification

Language for Active Databases", Tech. Report, UF-CIS-TR-93-007, 1993
11. Chakravarthy, S., Krishnaprasad, V., Anwar, A., Kim, S.: "Composite Events for Active Databases: Semantics, Contexts and Detection", VLDB, 1994, pp. 606-617
12. Date, C., Darwen, H., Lorentzos, N.: "Temporal Data and the Relational Model", Elsevier Science, 2003
13. Gatziu, S., Dittrich, K.: "Events in an Active Object-Oriented Database System", Intl. Workshop on Rules in Database Systems, Edinburgh, 1993
14. Gehani, N., Jagadish, H., Shmueli, O.: "Composite Events Specification in Active Databases: Model and Implementation", VLDB, 1992, pp. 327-338
15. Gehani, N., Jagadish, H., Shmueli, O.: "Event Specification in an Active Object-Oriented Database", SIGMOD, 1992, pp. 81-90
16. Ramakrishnan, R. et al.: "SQRL: Sorted Relational Query Language", SSDBM 1998, pp. 84-95
17. Sadri, R., Zaniolo, C., Zarkesh, A., Adibi, J.: "Optimization of Sequence Queries in Database Systems", 2001
18. Sadri, R.: "Optimization of Sequence Queries in Database Systems", PhD Thesis, UCLA, 2001
19. Shadri, P., Livny, M., Ramakrishnan, R.: "Sequence Query Processing" , SIGMOD, 1994, pp. 430-441
20. Shadri, P., Livny, M., Ramakrishnan, R.: "SEQ: A Model for Sequence Databases", ICDE, 1995, pp. 232-239
21. Snodgrass, R.: "Developing Time-Oriented Database Applications in SQL", Morgan Kauffman Publishers, 2000
22. Zimmer, D., Unland, R.: "On the Semantics of Complex Events in Active Data Base Management Systems", ICDE, 1999, pp. 392-399