# The Situation Manager Rule Language

Asaf Adi, Opher Etzion

IBM Haifa Research Lab
Haifa University Campus, Haifa 31905, Israel
{adi, etzion}@il.ibm.com

**Abstract.** This paper presents the "*situation manager*" rule language. The *situation manager* is a tool that includes both a language and an efficient run-time execution mechanism, aimed at reducing the complexity of active applications. It follows the observation that in many cases, there is a gap between current tools that enable to react to a single event (following the ECA: Event-Condition-Action paradigm), and the reality, in which a single event may not require any reaction, but the reaction should be given to patterns over the event history. The concept of **situation** presented in this paper, extends the concept of **composite event**, in its expressive power, flexibility, and usability. This paper motivates the work, surveys other efforts in this area, and discusses the situation manager's rule language.

## 1. Introduction

In recent years, a substantial amount of work has been invested in systems that either react automatically to actual changes (reactive systems), or to predicted changes in their environment (proactive systems). These systems perform actions or signal alerts in response to the occurrence of events that are signaled when changes in the environment occur (or inferred). Such systems are used in a wide spectrum of areas and include command and control systems, active databases, system management tools, customer relationship management systems and e-commerce applications.

A central issue in reactive and proactive systems is the ability to bridge the gap between the events that are identified by the system and the **situations** to which the system is required to react. Some examples, from various areas, of situations that need to be handled are:

- A client wishes to activate an automatic "buy or sell" program, if a security that is traded in two stock markets, has a difference of more than five percent between its values in the markets, such that the time difference between the reported values is less than five minutes ("arbitrage").
- A customer relationship manager wishes to receive an alert, if a customer's request was reassigned at least three times.
- A groupware user wishes to start a session when there are ten members of the group logged in to the groupware server.

**Fig. 1.** Situation.

There are a variety of tools that have been constructed to provide work environment for event driven applications. The work described in this paper has been motivated by the observation that most of the contemporary tools can react to the occurrence of a single event. In many applications (including all the examples shown above) the customer wishes to react to the occurrence of a *situation***,** which is a semantic concept in the customer's domain of discourse. The syntactic equivalent of a situation is a (possibly complex) pattern over the event history. Thus, there is a gap between applications' requirements and the capabilities of the supporting tools, resulting in excessive work. This paper aims at bridging this gap and saving the excessive work. It should be noted that the "pattern over the event history" may in some cases be only an approximation of the actual situation, or express the situation with some level of uncertainty. In this paper we have made the simplified assumption of equivalence between these two terms. Some tools and some research prototypes approach this difficulty by providing a mechanism for the definition of *composite events* that are detected when a predicate over the event history is satisfied. However, previous research was focused on specific fields such as active database [3][9][17] and network management [14][16]. It resulted in partial solutions that have limited expressive power and can only be used in these specific domains by systems to which they were specially designed. Moreover, these prototypes are not able to fully express some of the fundamentals of a situation definition:

1. The events that can participate in situation detection.
2. The context during which situation detection is relevant.
3. The impact of the semantic information that is reported with events on situation detection (i.e. the semantic conditions that must be satisfied in order to detect a situation).
4. The decision possibilities about the reuse of event instances that participated in situation detection. The decision is whether, and on which conditions, the event instance is "consumed" and cannot be used for the detection of other situations.

In this paper we present the *Situation Manager* rule language. The situation manager is a part of *Amit* (Active Middleware Technology) framework. Amit is both an application development and run-time control tool that is intended to enable fast and reliable development of reactive and proactive applications. The situation manager is a run-time monitor that receives information about the occurrence of events, detects the situations to which applications are required to react, and repots the detected situations to subscribers, typically other applications. It moves the responsibility of situation detection from the application to a high level tool and bridge the gap between the application and the situations to which it requires to react. It provides a general solution (i.e. a solution that is practical in many domains) that can express the fundamentals of a situation definition that were describes above.

This paper reports on the situation concept and the rule language that is used to express it. Section 2 reviews some previous work that was done in order to define the semantics of composite events. Section 3 describes the concept of event that is the basic building block of the situation language. Section 4 describes the concept of event group, a semantic equivalence among events. Section 5 describes the concept of lifespan, a temporal context during which situation detection is relevant. Section 6 describes the concept of event collection, a collection of events that are relevant for situation detection. Section 7 describes to concept of situation and how events, event

groups, event collections, and lifespans are used during situation composition. Section 8 concludes the paper.

Examples from the domain of e-commerce application (stock market) follow through the paper the different elements of situation definition. The language is expressed using XML (Extensible Markup Language).

## 2. Related Work

Contemporary **commercial systems** do not support composite events. However, they support triggers as specified in the SQL3 standard [11]. A trigger in SQL3 is an ECA rule that is activated by a database state transition and has an SQL3 predicate as a condition and a list of SQL3 statements as an action. Commercial databases that support triggers include DBMS products such as DB2, Oracle, Sybase and Informix.

Research on complex events for **active databases** is quite comprehensive and several research prototypes have been proposed, most of them base their event composition capabilities upon some kind of event algebra.

1. ODE [9] is limited to database events only. It detects composite events over an event history that contains all event occurrences (i.e. ODE can not express time interval during which situation detection is relevant) and forbids the reuse of event instances (i.e. events are always consumed). Although semantic information is reported with events in ODE, this information can only be used to impose some filtering conditions (masks) and equality conditions (parameters) on events that participate in an event expression (composite event).
2. Snoop [3] supports both database event and external events (the semantics of external events are not described). It has limited expressive capabilities for the definition of time internals using the operators A, A*, P, and P* in association with a parameter context. Parameter contexts describe some decision possibilities for event reuse (consumption). However, Snoop cannot express all possibilities of event reuse (consumption) policies. Although semantic information is reported with events in Snoop, this information cannot be used during event composition (it can be used in the condition part of the ECA rule).
3. Zimmer's and Unland's model [17] supports both database event and external events. It does not define the time interval during which situation detection is relevant and supports only few, predetermined, event reuse (consumption) policies. Semantic information is reported only with database event. This information can only be used to impose equality conditions on composing events.

Additional research prototypes of complex events for active databases including EXACT [6], REACH [18], ACOOD [2], ROCK & ROLL [7], Chimera [12], and REFLEX [13] do not offer new functionality. Other prototypes offer new functionality by introducing new operators. These include HiPAC [5], NAOS [4], and SAMOS [8] that deals with the detection of complex events using colored petri-nets in addition to the introduction of a new operator. Additional prototypes that are not based on event algebra, but on functional programming and real time logic include

PFL [15] that is based on functional programming; JEM [10], that is based on the logic RTL (Real Time Logic); and ADL [1].

Event correlation (network management) systems (HP openView Event Correlation Services [14], SMARTS InCharge [16], VERITAS NerveCenter [19]) are designed to handle only network events. Their expressive power is limited to the network management domain and they do not aim at providing a general (domain independent) solution that supports the fundamentals of a situation definition we described earlier.

We have shown that none of these prototypes and systems is a comprehensive solution that satisfies the requirements we have defined. We have shown that these solutions suffer from a deficiency in their expressional power, inaccuracies in their semantics and a centric approach that prevent the possibility to use these solutions in most of the real world applications. We have shown that a comprehensive solution to these problems is needed.

## 3. Event

An event is a significant (in some domains) instantaneous (happens in a specific point in time) atomic (happens completely or not at all) occurrence. We distinguish between *concrete* events and *inferred* events. Concrete events are those that happen in reality, usually as a result of a change in an object's state. Examples are: a person entering the meeting room; a light in the third floor of the building is turned on. Inferred events do not happen in the physical reality, but can be logically concluded by viewing the world's state (context) and the history of concrete event occurrences. An inferred event represents the occurrence of a significant situation in the physical reality. Examples are: all the invitees have already arrived to the meeting room (the meeting can start); the electricity load in the third floor is too high (electric outage may occur). We define two classes of events accordingly:

- *External events* are those, usually concrete events, which are pushed into the situation manager by external sources in runtime. These include sensors, other applications and human sources.
- *Internal events* are inferred events that are signaled by the situation manager when it detects the occurrence of a situation.

An event, either external or internal, is represented by an *event instance* that contains the necessary information about the event. This information includes: the occurrence time of the event, data that is relevant to applications that react to the event, and additional data that is needed in order to decide if a situation (inferred event) has occurred.

An *event class* describes the common properties of a similar set of event instances on an abstract level. It defines the type of information that can be associated with the event (attributes) and its behavior (methods).

An event class has the properties that are described bellow.

1. A *class name* identifies the class. It must be unique with respect to the set off all event class names.

2. *A set of superclasses* describes the class's direct ancestors. A class inherits the attributes and methods of its superclasses.
3. *A set of attributes* describes the information that is associated with the event.

### 3.1. Attributes

Most contemporary systems view event information as a string, without semantic meaning. In our model an event is represented as a structured set of attributes. An event attribute maps an event class to either an object class or to a collection of classes that are not necessarily events; the latter class describes the schema of the event class. Each attribute has a unique name, a type, and a default value. These properties are further detailed bellow.

1. An *attribute name* identifies the attribute. It must be unique with respect to the set of all attribute names defined in the class.
2. An *attribute type* determines the attribute's possible values. It can be either a primitive attribute or a reference to an object. Primitive attributes include number, string, boolean, and chronon[1]. The value of a reference attribute, in contrast to that of a primitive type, is a reference to an object that is the actual value of the attribute. This object can be an event instance or any other data structure.
3. A *default value* determines the attribute's value if it is not reported with the event. It is a complex expression that may involve other event attributes and external information (a database is an example). A default value can be used to define derived attributes.

### 3.2. The Class *event*

The class *event* is the root of the class hierarchy (i.e. it is an ancestor of every event). It defines attributes that are common to all event classes.

- *eventTime* is a chronon that describes the timestamp in which the event occurs in reality as reported by the event source.
- *detectionTime* is a chronon that describes the timestamp in which the event is detected by an application. Delays in event reporting and events that are reported not according to their real order result in a distinction between the *eventTime* and *detectionTime*
- *certainty* is a number that quantifies the certainty level of the event.
- *source* is string that uniquely describes the event source
- *expiration* is number that describes the maximal period in which the event is relevant. An event expires when the information associated with it is not relevant anymore (e.g. an event that reports that a disk utilization is 95 percent

[1] A *chronon* is a non-decomposable time interval of some fixed, minimal duration.
Data models may represent a time line by a sequence of non-decomposable, consecutive time intervals of identical duration. These intervals are termed chronons. A data model will typically leave the particular chronon duration unspecified, to be fixed later by the individual applications, within the restrictions posed by the implementation of the data model [20].

that expires after 30 seconds). When an event instance expires all uncompleted computations in which it participates should ignore it. The default value for this attribute is *null*; it designates that the event is never expired. Descendant classes can override this attribute default value.

After instantiation, the information that is associated with the event is either reported by an event source, or determined by a default value, is immutable. This is because an event, and the information associated with it, describes something that happens instantaneously, or an instantaneous step (start of an activity); such information does not change after the event occurrence.
The event class *quote* describes event instances that are signaled when a stock is quoted regardless to the source of the event (stock market).

```
<event name = "quote" >
    <attribute name = "symbol"
        type = "string" />
    <attribute name = "lastTrade"
        type = "number" />
    <attribute name = "change"
        type = "number" />
</event>
```

The event class *quote* has three attributes: *symbol*, *lastTrade*, and *change*. The attribute *symbol* describes the stock symbol, the attribute *lastTrade* describes the stock value, and the attribute *change* describes the change in the stock value since the previous quote.
Two examples of event instances associated with the *quote* event class are:

- An event instance that quotes IBM's stock with the values: *symbol* = ibm, *lastTrade* = 114.25, *change* = 0.75.
- An event instance that quotes a trade in Intel's stock with the values: *symbol* = intc, *lastTrade* = 26.85, *change* = -1.11.

The event class *fullQuote*, that extends the class *quote*, describes a stock quote in more details than the class *quote*. It has three attributes that describe the stock's volume and the stock's bid and ask values in addition to the attributes defined in the class *Quote*.

```
<event name = "fullQuote"
    extends = "quote" >
    <attribute name = "volume"
        type = " number " />
    <attribute name = "bid"
        type = "number" />
    <attribute name = "ask"
        type = "number" />
</event>
```

**Fig. 2.** Event.

## 4. Event Group

*Event group* is a collection of semantically associated event instances (e.g. *quote* events that reports about the same stock belong to the same event group). Event groups are used to match different event instances that refer to the same entity or concept. An event group divides the situation detection process to numerous separate independent detection process (denote partitions); one partition for every event group.

An *event group class* defines a set of grouping expressions that semantically associate event instances with an event group or several event groups according to specific semantics (examples are: all quotes with the same symbol, all quotes with high trade volume). An event group class has the properties that are described bellow.

1. *A class name* identifies the class. It must be a legal identifier that is unique with respect to the set of all event group class names.
2. *Extension policy* describes how to handle event instances that are not explicitly associated with an event group. *Relaxed* extension policy (default) designates that these event instance are implicitly associate with every event group. *Strict* extension policy designates that these event instances are not associated with an event group.
3. *A set of grouping expressions* describes the association rules of an event instance with event groups.

### 4.1. Grouping Expression

A grouping expression associates event instances with event groups. It is a ternary relation of an event, a condition, and a group value (or an expression). An event instance, that is classified to the event and satisfies the condition, is associated with (belong to) the event group that is described by the group value (the group value is a result of a grouping expression over the information that is associated with the event instance).

*Event group class* defines a set of grouping expressions that semantically associate event instances with an event group or several event groups according to specific semantics (examples are: all quotes with the same symbol, all quotes with high trade volume). An event group class has the properties that are described bellow.

1. *The event name* is the name of an event class.
2. *The condition* specifies the conditions that an event instance must satisfy in order to belong to an event group. The event group is (later) determined by evaluating the group value. It is a predicate over the event attributes and external data. The default condition is *true*.
3. *The group value* is a constant or an expression over event attributes. This value, or the expression's result, describes the event group to which event instances that are classified to the event and satisfy the condition belong.

Several grouping expression (tuples of event, condition, and value) can be defined for a single event group class. An event instance that satisfies the conditions of several grouping expression (and has several group values) belongs to several event groups. An event instance that does not satisfy the conditions of any grouping expression is treated according the event group's extension policy.

The event group class *symbol*, partitions the event instances into event groups according to the stock's symbol. It defines that all quotes (event instances) that refer to the same stock belong to the same event group. The group value for such group is the stock's symbol.

```
<group name = "symbol">
    <event name = "quote" >
        < groupingExpression value = "symbol" />
    </event>
</group>
```

The event group class *tradeVolume*, partitions the event instances into event groups according to the quote's volume. It defines that quotes (event instances) are partitioned into three groups. Quotes with high tradability that have a volume that is higher than 100, quote with average tradability that have a volume between 30 and 120, and quotes with low tradability that have a volume that is less than 50. The group values are *high*, *average*, and *low* correspondingly.

```
<eventGroup name = "tradeVolume"
    extension = "strict">
    <event name = "fullQuote" >
        <groupingExpression value = "'high'"
            condition = "volume &gt; 100" />
        <groupingExpression value = "'average'"
            condition = "volume &gt; 30 and volume &lt; 120" />
        < groupingExpression value = "'low'"
            condition = "volume &lt; 50" />
    </event>
</eventGroup>
```

A quote (an event instance) can belong to both "high" and "average" event groups, if it has a volume between 100 and 120, or to both "moderate" and "low" event groups, if it has a volume between 30 and 70.
A quote (an event instance) that is not classified to *FullQuote* does not belong to an event group (for example an instance of the event *tradeStrat*) because the extension policy is strict and a grouping expression is not associated with such events.

**Fig. 3.** Event Group.

## 5. Lifespan

A lifespan is the temporal context during which situation detection is relevant. The lifespan is an interval bounded by two events called *initiator* and *terminator*. An occurrence of an initiator event initiates the lifespan and an occurrence of a terminator event terminates it. The initiator and terminator can be external events, internal events, or system events such as system startup and system shutdown.

A *lifespan class* describes the common properties of a similar set of lifespans on an abstract level. It defines the set of events that can initiate a lifespan, the set of events that can terminate it, the conditions for the lifespan initiation and termination, and its maximal length. A lifespan class has the properties that are described bellow.

1. *A class name* identifies the class. It must be unique with respect to the set of all lifespan class names.
2. *A list of initiators* describes the conditions for lifespan initiation.
3. *A list of terminators* describes the conditions for lifespan termination.

Note that more than one lifespan of the same class may be open simultaneously, if two initiator events have occurred before a terminator event, depending on the conditions for initiation.

A lifespan has its own semantics, which may be independent from the semantics of a specific situation. In fact, a single lifespan can be a relevant context for the detection of multiple situations. Example is the lifespan *tradingDay*, which starts when the event *tradeStart* occurs and ends when the event *tradeEnd* occurs. This lifespan is a relevant time window for numerous situations. Moreover, the conditions for lifespan initiation and termination are not influenced by the specific situations that are relevant during the lifespan.

### 5.1. Initiator

A lifespan is initiated by an occurrence of an *initiator* when an event, either external or internal, occurs or (if defined this way) when the situation manager starts to run (i.e. system startup). An initiator is a ternary relation: an event name, a condition, and a correlation code. The event name identifies the event whose occurrence, if the condition and the correlation code are satisfied, initiates a lifespan.

1. *The event name* is a name of an event class.
2. *The condition* specifies threshold condition that the event must satisfy in order to initiate the lifespan. It is a predicate over the initiating event attributes and external data (a database query is an example). The default condition is *true*.
3. *The correlation code* determines the lifespan duplication policy. There are two possible correlation codes: *add* and *ignore*. If the correlation code is *ignore*, a new lifespan is initiated, only if a lifespan of the same class is not already open. If the correlation code is *add*, a new lifespan is opened while any existing lifespans remain open.

Multiple tuples that consists of an event class, a condition, and a correlation code can be defined for a single lifespan class. This allows different instances of the same lifespan to be initiated by different events and under different conditions. Note that an event occurrence can initiate lifespans by satisfying only a single initiation tuple (the first in the order of definition), although it may satisfy the conditions of numerous initiation tuples.

### 5.2 Terminator

A lifespan remains open since its initiation time until it is either terminated by an occurrence of a *terminator* or it expires. The terminator defines whether lifespan instances are terminated after a period of time, by event occurrences, or both; under which conditions; and in case of multiple lifespan instances, which lifespans are terminated. A terminator is either an expiration interval (that determines that maximal length of the lifespan) or a ternary relation: an event name, a condition, and a quantifier. The event name identifies the event whose occurrence, if the condition is satisfied, terminates a lifespan. The quantifier determines which open lifespans are terminated.

1. *The event name* is the name of an event class.
2. *The condition* specifies threshold condition that the event must satisfy in order to terminate the lifespan. It is a predicate over the terminating event attributes and external data. The default condition is *true*.
3. *The quantifier* determines which open lifespans are terminated. There are three possible quantifier values: *first*, *last* and *each*. If the quantifier is *first,* the oldest lifespan is terminated; if the quantifier is *last,* the newest lifespan is terminated; and if the quantifier is *each,* all the open lifespans are terminated.

Multiple instances of the tuple that consists of event class, threshold conditions, and quantifier, can be defined for the same lifespan. This allows a lifespan to be terminated by different events and under different conditions and makes it possible to define lifespans that represent time intervals in which situations are relevant in reality.

A broker wishes to identify situations regarding IBM's stocks. The context in which such situations may occur is associated with lifespans (time intervals) that start when an IBM's stock is quoted, if no such lifespan is already open and end after 20 time units.

```
<lifespan name = "IBM">
    <initiator name = "quote"
        condition = "symbol = 'IBM' "
        correlate = "ignore" />
    <terminator interval = "20" />
</lifespan>
```

Below is a scenario of event occurrences and their influence on the lifespan initiation:

1. An IBM's stock quote event initiates a new lifespan.
2. A HP's stock quote event is ignored.
3. An IBM's stock quote event that occurs less than twenty time unites since the first event is ignored.
4. An IBM's stock quote event that occurs more than twenty time units since the first event, initiates a new lifespan.

**Fig. 4.** Lifespan.

## 6. Event Collection

An *event collection* designates the event instances that are considered for situation detection. They are evaluated if they occur during the time that the context (*lifespan*) that is associated with the situation is active. These event instances, denoted *candidates*, are partitioned into *candidate lists*. A candidate list is a collection of event instances (candidates) that are classified to the same class, have the same role in situation detection (e.g. in a situation that identifies arbitrage deals events that report about quotes from one stock market have one role, and events that report about quotes from other stock market have other role), satisfy the same filtering conditions, and share the same decision possibilities about the reuse of event instances that participate in situation detection.

An *event collection class* describes the common properties of a similar set of event collections on an abstract level. It defines the classification of the collected events into candidate lists. An event collection class has the properties that are described below.

1. A *class name* identifies the class. It must be a legal identifier that is unique with respect to the set of all event collection class names.
2. A *set of candidate lists*.

Note that an event collection is relevant only if it is associated with a context (lifespan and event group). It is because the collected event instances share the same semantics (event group) and occur while the associated lifespan is active.

### 6.1. Candidate List

A *candidate* is an event instance that has an impact upon situation detection. In order to decide if a situation occurred in reality, all candidates must be monitored. Moreover, it is sufficient to base the decision upon this event instance only. A candidate list is a collection of candidates that share the same properties (event class, role, filter conditions, decision possibilities about reuse of event instance that participated in situation detection) that effect situation detection. A candidate list is a tuple: an alias (the candidate list name), an event name, a threshold condition, an override condition, and a retain condition. An event name identifies event instances, that if satisfy the threshold condition are included in the candidate list; the override condition determines if a new instance override existing candidates in the list; and the retain condition determines if an instance that triggered a situation can be reused.

1. The *alias* identifies the candidate list. It must be unique with respect to the set of all aliases defined in the event collection class.
2. *The event name* is the name of an event class
3. *A threshold condition* specifies the condition that an event instance must satisfy to be considered a candidate. It is a predicate over the event attributes and an external data. The default threshold condition is *true*.
4. *An override condition* specifies the condition that if satisfied, a new event instance overrides existing candidates in the list. It is a predicate over the event attributes and an external data. The default threshold condition is *false*.

5. *A retain condition* specifies the condition that if satisfied, a candidate that triggered a situation, can be used again in situation detection.

Note that an event instance can be collected in several candidate lists within an event collection. However, the properties of each candidate list can be different, thus an event instance can be considered a candidate only in some of these candidate lists.

In order to decide if a situation occurs event instance in all or some of the candidate list of an event collections are evaluated.

A broker wishes to identify situations regarding stocks that have high trade volume and stocks that have low trade volume. The required event collection is consisting of two candidate lists. One contains quotes with high volume and the other contains quotes with low volume.

```
<eventCollection name = "quoteVolume" >
    <candidiate name = "fullQuote"
        alias = "highVolume"
        threshold = "volume &gt; 100" />
    <candidiate name = "fullQuote"
        alias = "lowVolume"
        threshold = "volume &lt; 100" />
</eventCollection >
```

**Fig. 5.** Event Collection.

## 7. Situation

An event is a significant instantaneous atomic occurrence. We distinguish between *concrete* events and *inferred* events. Concrete events are those that happen in reality, usually as a result of a change in an object's state. Inferred events do not happen in the physical reality, but can be logically concluded by viewing the world's state (context) and the history of concrete event occurrences. An inferred event represents the occurrence of a significant situation in the physical reality.

A *situation* defines the inference logic. It defines the methodology for the detection of an (inferred) event instance that represents a significant situation (i.e. the information that is associated with it and its detection time). A situation has the properties that are described below.

1. A *situation name* identifies the situation. It must be is unique with respect to the set off all situations.
2. A *lifespan name* identifies the lifespan that defines the situation's temporal context.
3. An *event group* identifies the event group that defines the situation's semantic connotation.
4. An *event collection name* identifies the event collection that defines the event instances that are considered for situation detection.
5. A *situation expression* defines the conditions for situation detection, and the event instances that caused it.

6. A *triggering expression* identifies the inferred event (or events) and defines its associated information.

The lifespan and event group determines the situation context. The lifespan is the time interval during which situation detection is relevant, thus only event instances that occur while the lifespan is open are considered for the situation. The event group determines the situation semantic connotation, thus only event instance that belong to the same event group are considered for the situation (including those event that determine if the lifespan is open). Note that if multiple lifespans are open simultaneously, or if several event groups are defined, a distinct situation is detected for each combination of a lifespan and an event group (context). The detection of a situation in one context does not influence the detection of a situation in other context. Accordingly, the decision if an event instance is a candidate of the situation (i.e. the evaluation of the event collection) is performed in each context separately.

Nested situation can also be defined. Situation nesting can be preformed by specifying an inferred event as a candidate in an event collection. We denote the situation that triggered the inferred event a *sub situation* and the situation that uses the inferred event as a candidate a *nested situation*. Note that both situations (the sub situation and the nested situation) can be detected in the same context using the same event collection, in different contexts using the same event collection, in the same context using different event collections, or in different contexts using different event collections.

### 7.1. Situation Expression

A *situation expression* is an expression over event classes that determine situation occurrences as a function of an event collection instance and a context. A situation expression consists of a combination of an *operator* and *qualifiers*, a *predicate* (applicable for certain operators), and *detection mode*. The combination of an operator and qualifiers designates an event pattern; the predicate designates a condition over the events in the pattern results in tuples of event instances that could have cause the situation; and the detection mode determines if a situation can be detected during the context (*immediate*) or at the end of it (*deferred*)

#### 7.1.1. Qualifiers

Situation expression qualifiers designate a selection strategy when several candidates exist in a candidate list of an event that is in the domain of a situation expression operator. We denote these events, *operands* of the situation expression. A qualifier is applied to every operand and has seven possible values: *first*, *last*, *each*, *min*, *max*, *not* and, *never*.

1. *first*(*E*) – selects the first (oldest) candidate (or candidates) in *E*'s candidate list (instances effectively classified to *E* that occurred while the context was active, satisfied the threshold condition, and are not obsolete).
2. *last*(*E*) – selects the last (most recent) candidate (or candidates) in *E*'s candidate list.
3. *each*(*E*) – selects all candidates in *E*'s candidate list.

4. *min*(*E*, *exp*) – selects the candidate (or candidates) with the minimal result for the associated expression, *exp*, in *E*'s candidate list.
5. *max*(*E*, *exp*) – selects the candidate (or candidates) with the maximal result for the associated expression, *exp*, in *E*'s candidate list.

The qualifiers *not* and *never* are different from the other qualifiers in the sense that they do not selects candidates but designates if a candidate does not exists.

6. *never*(*E*) – designates if a candidate of *E* did not exist since the beginning of the context
7. *not*(*E*) – designates if a candidate of *E* does not exist since the occurrence of the previous operand (if the operator imposes operands' order, and *E* is not the first operand), or a candidate of *E* does not exist (otherwise).

Note that a qualifier is applied on an operand before the situation's operator is considered.

### 7.1.2. Operators

A situation expression operator designates an event pattern. The operators are classified into five groups: *joining operators*, *selection operators, assertion operators, aggregation operators*, and *temporal operators*.

1. Joining operators
   1.1. The operator *conjunction*($E_1$, $E_2$, …, $E_k$), $k \geq 2$ designates a conjunction of events $E_1 \ldots E_k$ with no order importance.
   1.2. The operator *disjunction*($E_1$, $E_2$, …, $E_k$), $k \geq 2$ designates a disjunction of events $E_1 \ldots E_k$.
   1.3. The operator *sequence*($E_1$, $E_2$, …, $E_k$), $k \geq 2$ designates an ordered conjunction of events $E_1 \ldots E_k$ where $E_i$ precedes $E_{i+1}$ for each $i<k$.
   1.4. The operator *strictSequence*($E_1$, $E_2$, …, $E_k$), $k \geq 2$ designates an ordered conjunction of events $E_1 \ldots E_k$ where $E_i$ precedes $E_{i+1}$ and no other (candidate) events occurs between $E_i$ and $E_{i+1}$ for each $i<k$.
   1.5. The operator *simultaneous*($E_1$, $E_2$, …, $E_k$), $k \geq 2$ designates a conjunction of events $E_1 \ldots E_k$ when events $E_1 \ldots E_k$ occurs simultaneously.
   1.6. The operator *aggregation*(($E_1$, $exp_1$), ($E_2$, $exp_2$)…, ($E_k$, $exp_k$), *havingPredicate*), $k \geq 1$ designates an aggregation (sum; count; average, minimum, maximum of associated expression) over events $E_1 \ldots E_k$. It triggers an inferred event each time an event (of $E_1 \ldots E_k$) becomes a candidate, or a candidate becomes obsolete if the aggregation over the existing candidates satisfy the having predicate.
2. Selection operators
   2.1. The operator *first*($E_1$, $E_2$, …, $E_k$), $k \geq 1$ designates the first event (or events) of $E_1 \ldots E_k$.
   2.2. The operator *until*($E_1$, $E_2$), designates the occurrences of $E_1$ that occurred before an occurrence of $E_2$.
   2.3. The operator *since*($E_1$, $E_2$), designates the occurrences of $E_1$ that occurred after an occurrence of $E_2$.
   2.4. The operator *range*(*E*) designates an occurrence of *E* that satisfies the predicate, if the previous occurrence of *E* (if there was a previous occurrence) did not satisfy it.

Selection operators are always evaluated immediately, thus the *detection mode* modifier does not influence the inferred events, triggered by situations based on these operators.

A broker wishes to identify situations regarding each stock traded in a stock market during a single trading day (*tradingDay* lifespan, *symbol* event group). He is interested in stocks that have high trade volume and stocks that have low trade volume (*quoteVolume* event collection).
He wishes to identify a situation in which a quote is trades in high volumes consecutively (i.e. three consecutive high volume quotes without low volume quotes between them). This situation requires the *strictSequence* operator with three *highVolume* operands. The operands have the qualifier *last* in order to identify only the last three consecutive high volume quotes.

```
<situation name = "threeConsecutiveHighVolumeQuotes"
    lifespan = "tradingDay"
    eventGroup = "symbol"
    eventCollection = "quoteVolume" >
    <situationExpression>
        <strictSequence>
            <event name = "highVolume" qualifier = "last"/>
            <event name = "highVolume" qualifier = "last"/>
            <event name = "highVolume" qualifier = "last"/>
        </strictSequence>
    </situationExpression>
    ...
</situation>
```

The broker from the previous examples wishes to identify additional situation in the same context, using the same event collection.
He wishes to be notified whenever the average value of high volume stocks (since the beginning of the trading day) is higher than 500. Such situation is identified using the *totalAggregation* operator.

```
<situation name = "averageValue"
    lifespan = "tradingDay"
    eventGroup = "symbol"
    eventCollection = "quoteVolume" >
    <situationExpression>
        <aggregation having = "average &gt; 500">
            <event name = "highVolume" qualifier = "each"
                   expression = "highVolume.lastTrade"/>
        </aggregation >
    </situationExpression>
    ...
</situation>
```

**Fig. 6.** Joining operator.

The broker from the previous example wishes to identify additional situation in the same context, using the same event collection.
He wishes to identify situations in which a stock is starting to be traded with extremely high volumes. Such situation occurs when a stock is traded with extremely high volumes, after it was traded with lower volumes for a certain period of time.

```
<situation name = "extremelyHighVolumePeriod"
    lifespan = "tradingDay"
    eventGroup = "symbol"
    eventCollection = "quoteVolume" >
    <situationExpression>
        <range predicate = "highVolume.volume > 1000">
            <event name = "highVolume" qualifier = "last"/>
        </range >
    </situationExpression>
    ...
</situation>
```

**Fig. 7.** Selection operator.

3. Assertion operators
   3.1. The operator *never*($E$) designates an event that never occurred during a context.
   3.2. The operator *sometimes*($E$) designates an occurrence of an event during a context.
   3.3. The operator *last*($E_1$, $E_2$, …, $E_k$), $k \geq 1$ designates the most recent event (or events) of $E_1 \ldots E_k$.
   3.4. The operator *min*(($E_1$, $exp_1$), ($E_2$, $exp_2$)…, ($E_k$, $exp_k$)), $k \geq 1$ designates the event (or events) with the minimal result for their associated expression, $exp_i$.
   3.5. The operator *max*(($E_1$, $exp_1$), ($E_2$, $exp_2$)…, ($E_k$, $exp_k$)), $k \geq 1$ designates the event (or events) with the maximal result for their associated expression, $exp_i$.
   3.6. The operator *unless*($E_1$, $E_2$), designates the occurrence of $E_1$, and the non occurrence of $E_2$ during a context. An event (situation) inferred using the *unless* operator is equivalent to an event (situation) inferred using the operator *conjunction* over $E_1$, $E_2$', where $E_2$' is itself an inferred event triggered by the situation *never*($E_2$).

Assertion operators are evaluated at the end of a context, thus the *detection mode* modifier does not influence the inferred events, triggered by situations based on these operators.

The broker from the previous examples wishes to identify additional situation in the same context, using the same event collection.
He wishes to identify the high volume quote with the minimal value for each stock in each trading day. Such situation is identified using the *min* operator at the end of each trading.

```
<situation name = "minValueHighVolume"
        lifespan = "tradingDay"
        eventGroup = "symbol"
        eventCollection = "quoteVolume" >
    <situationExpression>
        <min>
            <event name = "highVolume" qualifier = "each"
                    expression = "highVolume.lastTrade"/>
        </min >
    </situationExpression>
    ...
</situation>
```

**Fig. 8.** Assertion Operator

4. Temporal operators
   4.1. The operator *at*(*timePattern*) designates the points in time that correspond to the specified time pattern. A time pattern is a sting, formatted *dd/mm/yyyy hh:mm:ss.mmm*, that can contain wildcards. For example, a situation with the operator *at* with a time pattern **/11/2001 00:00:00.000 is triggered at the beginning of every day during November 2001.
   4.2. The operator *after*(*E*, *t*) designates an interval of *t* time units since the occurrence of *E*.
   4.3. The operator *every*(*t*) designates a repeating interval of *t* time units.
   Temporal operators are always evaluated immediately, thus the *detection mode* modifier does not influence the inferred events, triggered by situations based on these operators.

### 7.2. Triggering Expression

A *triggering expression* triggers an inferred event whenever a situation is detected. It consists of *event name*, a set of *selectors*, and a set of *resolvers*. The event name identifiers the triggered inferred event; the selectors determined the amount of inferred event instance, and a *resolver* determines the value of an inferred event instance attribute.

#### 7.2.1. Selectors

A situation expression determines when a situation occurs, by selecting tuples of event instances that could have cause the situation. Several such tuples may be selected by a situation expression (example is a *conjunction* of $E_1$ and $E_2$, both with *each* qualifier, that identifies two tuples ($e_{11}$, $e_{21}$) and ($e_{12}$, $e_{21}$) after the occurrence of event instances $e_{11}$, $e_{12}$, $e_{21}$ in the specified order). Selectors determine the amount of

inferred events (up to one inferred event for each tuple) by selecting event tuples on which an inferred event instance is based. There are six possible selectors:

1. *first*(*E*) – selects the tuple (or tuples) where the first (oldest) event (or events) of *E* exists.
2. *last*(*E*) – selects the tuple (or tuples) where the last (most recent) event (or events) of *E* exists.
3. *min*(*E*, *nExp*) – selects the tuple (or tuples) where the event that has the minimal result for the associated numeric expression, *nExp*, exists.
4. *max*(*E*, *nExp*) – selects the tuple (or tuples) where the event that has the maximal result for the associated numeric expression, *nExp*, exists.
5. *each*(*E*) – selects a tuple for each (different) event of E.
6. *set*(*E*) – selects a single tuple that consists a set of *E*'s events.

The order of selectors definition is important. Each selector designates a new set of tuples, imposed by its selection strategy, that is the base tuple set for the consecutive selector.

#### 7.3.1. Resolvers

A resolver determines the value of one's of the inferred event attributes. It designates a function from an event tuple (one of the event tuples designated by the selectors) to an attribute type. The function may involve attributes of events in the tuple (entries that holds a single event) or aggregation (min, max, count, sum, average) over event sets (entries that hold several events, designated by the *set* selector) and the event group value.

The broker from the previous examples wishes to identify additional situation in the same context, using the same event collection.
He wishes to know the maximal volume distance of each stock during a trading day (a maximal volume distance of a stock is the difference between its highest quote volume and its lowest quote volume).
This situation is detected at the end of the context (deferred detection mode) by a conjunction of high volume and low volume quotes.

```
<situation name = "maxVolumeDifference"
    lifespan = "tradingDay"
    eventGroup = "symbol"
    eventCollection = "quoteVolume"
    detectionMode = "deferred" >
    <situationExpression>
        <conjunction>
            <event name = "highVolume" qualifier = "each"/>
            <event name = "lowVolume" qualifier = "each"/>
        </conjunction >
    </situationExpression>
    <triggeringExpression>
        <selector event = "highVolume" selector = "max"
            expression = "highVolume.volume"/>
        <selector event = "lowVolume" selector = "min"
            expression = "lowVolume.volume"/>
        ...
    </triggeringExpression>
</situation>
```

An event is trigger by the triggering expression only once. Its information is based on a pair of quotes, the quote with the highest volume and the quote with the lowest volume. For example, if IBM quotes during a trading day, in order of occurrence were:

1. symbol = ibm, lastTrade = 114.25, volume = 103 (high volume).
2. symbol = ibm, lastTrade = 115.74, volume = 107 (high volume).
3. symbol = ibm, lastTrade = 112.53, volume = 99 (low volume).
4. symbol = ibm, lastTrade = 112.94, volume = 102 (high volume).
5. symbol = ibm, lastTrade = 114.75, volume = 98 (low volume).

The set of high volume – low volume pairs that are detected at the end of the trading day are: {(1,3), (1,5), (2,3), (2,5), (4,3), (4,5)}. Without selectors an event will be triggered by the detection of the situation six times (one triggered instance for every such pair).
The selectors select only a single pair and as result only the event is triggered only once.
The first selector selects pairs that have the maximal volume value for the high volume event. It select pairs (2,3) and (2,5). The second selector then selects pairs that have the minimal volume value for the low volume event, from the pairs selected by the first selector. It selects the pair (2,5). The event that is triggered by this situation is based on the information in the second and fifth event in the example.

**Fig. 9.** Selector.

The situation from the previous example triggers the event *maxVolumeDifference*. This event has two attributes *symbol* and *difference*. The attribute *symbol* designates a stock symbol, and the attribute *difference* designates the maximal volume difference in the stock.

```
<event name = "maxvolumeDifference"
    <attribute name = "symbol" type = "maxVolumeDifference"/>
    <attribute name = "difference" type = "number"/>
</event>
```

The resolver element defines the value of the trigger event attributes. The first attribute, *symbol*, is equal to the event group value, and the second attribute, *difference*, is equal to the difference of the volume of the two instances selected by the selector.

```
<situation name = " maxVolumeDifference"
    lifespan = "tradingDay"
    eventGroup = "symbol"
    eventCollection = "quoteVolume"
    detectionMode = "deferred" >
    <situationExpression>
        <conjunction>
            <event name = "highVolume" qualifier = "each"/>
            <event name = "lowVolume" qualifier = "each"/>
        </conjunction >
    </situationExpression>
    <triggeringExpression event = "maxVolumeDifference">
        <selector event = "highVolume" selector = "max"
            expression = "highVolume.volume"/>
        <selector event = "lowVolume" selector = "min"
            expression = "lowVolume.volume"/>
        <resolver attribute = "symbol"
            expression = "eventGroupValue"/>
        <resolver attribute = "difference"
            expression = "highVolume.volume – lowVolume.volume"/>
    </triggeringExpression>
</situation>
```

**Fig. 10.** Resolver.

## 8. Conclusions

This paper has presented the "*situation manager*" rule language. It is a general markup language for active rules that introduce a combination of powerful event algebra with semantic notations such as context (lifespan), and semantic association (event group). The "situation manager" has been implemented in Java, and is being integrated with various products and services of IBM.

There is a substantial amount of further research that is being carried out now. It deals with areas such as: extending the situation manager's operators from temporal to spatio-temporal, adding uncertainty consideration, adding inference mechanism to derive rules out of a model, and dealing with "deep" temporal issues.

## 9. References

[1] Behrends-H. "Simulation-based Debugging of Active Databases." Proceedings of IEEE International Workshop on Research Issues in Data Engineering: Active Databases Systems. Feb. 1994; Houston, TX, USA. IEEE Comput. Soc. Press, 1994. 172-180.

[2] Berndtsson-M. "ACOOD: an Approach to an Active Object Oriented DBMS" Master's thesis, Department of Computer Science, University of Skovde, Sweden. 1991.

[3] Chakravarthy-S, and Mishra-D. "Snoop: an expressive event specification language for active databases." Data and Knowledge Engineering 14.1 (1994): 1-26.

[4] Collet-C, and Coupaye-T. "Composite events in NAOS." Proceedings of the 7th International Conference on Database and Expert Systems Applications, DEXA. Sept. 1996; Zurich, Switzerland. Springer Verlag, 1996. 244-253.

[5] Dayal-U, Buchmann-A, and Chakravarthy-U. "The HiPAC Project." Active Database Systems: Triggers and Rules For Advanced Database Processing Morgan Kaufmann, 1996. 177-206.

[6] Diaz-O, and Jaime-A. "EXACT: an extensible approach to active object-oriented databases." VLDB Journal. 6.4 (1997): 282-295.

[7] Dinn-A, Paton-NW, Williams-MH, and Fernandes-AAA. "An Active Rule Language for ROCK & ROLL." Proceedings of the 14th British National Conferenc on Databases. July 1996; Edinburgh, UK. Springer Verlag, 1996. 36-55.

[8] Gatziu-S, and Dittrich-KR. "Events in an active object-oriented database system." proceedings of the 1st International Workshop on Rules in Database Systems. Sept. 1993; Edinburgh, UK Springer Verlag, 1994. 23-29.

[9] Gehani-NH, Jagadish-HV, and Shmueli-O. "Composite event specification in active databases: model and implementation." Proceedings of 18th International Conference on Very Large Data Bases. Aug. 1992; Vancouver, BC, Canada. Morgan Kaufmann, 1992. 23-27.

[10] Guangtian-Liu, Mok-AK, and Konana-P. "A unified approach for specifying timing constraints and composite events in active real-time database systems." Proceedings of 4th IEEE Real-Time Technology and Applications Symposium. 1998; Denver, CO, USA. IEEE Comput. Soc. Press, 1998. 199-208.

[11] Kulkarni-K, Mattos-NM, and Cochrane-R. "Active Database Features in SQL3." Active Rules in Database Systems. Springer Verlag, 1999. 197-219.

[12] Meo-R, Psaila-G, and Ceri-S. "Composite Events in Chimera." Proceedings of 5th Conference on Extended Database Technology (EDBT`96). March 1996; Avignon, France. Springer Verlag, 1996. 56-78.

[13] Naqvi-W, and Ibrahim-MT. "EECA: An Active Knowledge Model." Proceedings of 5th International Conference on Database and Expert Systems Applications. Sept. 1994; Athens, Greece. Springer Verlag, 1994. 380-389.

[14] Sheers-KR. "HP OpenView event correlation services." Hewlett Packard Journal. 47.5 (1996): 31-42.

[15] Swaup-R, Alexandra-P, and Carol-S. "PFL: An Active Functional DBPL." Active Rules in Database Systems. Springer Verlag, 1999. 297-308.

[16] Yemini-SA, Kliger-S, Mozes-E, Yemini-Y, and Ohsie-D. "High speed and robust event correlation." IEEE Communications Magazine. 34.5 (1996): 82-90.

[17] Zimmer-D, and Unland-R. "A General Model for Specification of the Semantics of Complex Events in Active Database Management Systems." C-LAB Report. 1998.
[18] Zimmermann-J, and Buchmann-A. "REACH." Active Rules in Database Systems. Springer Verlag, 1999. 263-277.
[19] "VERITAS NerveCentertm" VERITAS Software http://eval.veritas.com/webfiles/docs/ NCOverview.pdf
[20] The Consensus Glossary of Temporal Database Concepts. http://www.cs.auc.dk/~csj/Glossary.