

Amit - The Situation Manager

ASAF ADI

OPHER ETZION

IBM Research Laboratory in Haifa

Email: opher@il.ibm.com

Phone: +972-4-829-6455

Fax: +972-4-829-6116

Abstract:

This paper presents the “situation manager”, a tool that includes both a language and an efficient run-time execution mechanism, aimed at reducing the complexity of active applications. This tool 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, however 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 both the language and the execution model.

Keywords: *active technology, active databases, high-level languages, composite events.*

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.
- A network manager wishes to receive an alert, if the probability that the network will be overloaded in the next hour is high.

Figure 1 – example of possible situations

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 [7][18][36] and network management [31][35], and 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 a situation detection.
2. The context during which a 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 alternatives 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*, 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 reports the detected situations to subscribers, typically other applications. It transfers the responsibility of situation detection from the application to a high level tool and bridges 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. The situation manager's high-level architecture is described in figure 2.

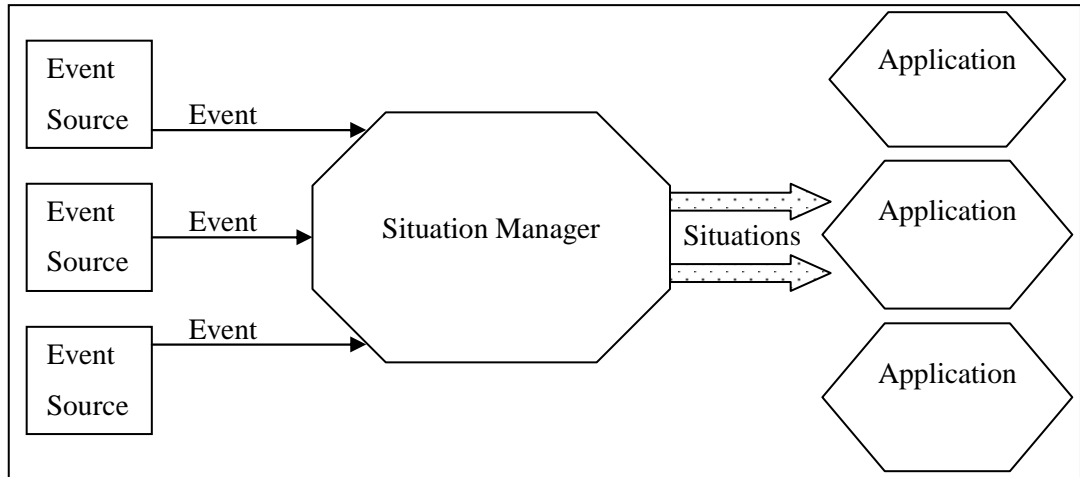


Figure 2 – the situation manager's high level architecture

This paper reports on the situation concept and its implementation in the framework of the Amit project. In section 2, we show the situation definition language; in section 3 we describe algorithms and data structures that are used by the situation manager during the detection process; in section 4 we present Amit's performance measurements, and in section 5 we review some previous work aimed to define the semantics of composite events and compare it to Amit. Section 6 concludes the paper.

2. The Situation definition language

This section describes the features of the situation definition language, followed by examples from the domain of e-commerce application (stock market). The language is implemented using XML (Extensible Markup Language); its DTD (Document Type Definition), which describes a definition meta-data, is specified in appendix A.

Section 2.1, event definition, describes the concept of event that is the basic building block of the situation language. Section 2.2, lifespan definition, describes the concept of lifespan, which is the temporal context during which situation detection is relevant. Section 2.3, situation definition, describes to concept of situation and how events, keys, and lifespans are used during situation composition. Section 2.4, key definition, describes the concept of key, a semantic equivalence among events.

2.1. Event definition

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 type* describes the common properties of a similar set of event instances on an abstract level. It defines a schema of attributes that are instantiated in runtime when an event actually occurs and describes the information that is associated with the event. This information is pushed into the situation manager, if the event is external and is calculated by it, if the event is internal.

A *trade-start* event and a *trade-end* event occur when a trading day in a specific stock exchange starts or ends respectively. The schema of these events has one attribute: *stock-exchange*.

A *stock-quote* event, a *bond-quote* event, and an *option-quote* event occur when a specified stock/bond/option is quoted in a specific stock exchange. The schema of these events has four attributes: *stock-exchange*, *symbol*, *value*, and *change* (in value since last quote).

An example for a *stock-quote* event instance is a tuple (NasdaqNM, TEVA, \$65.75, +4.89%) that was quoted on Nov 7, 2000, 15:25. Another example is a tuple (Berlin, TEVA, \$62.85, +5.04%) that was quoted on Nov 7, 2000, 08:31.

Figure 4 – example of event definition

There is a distinction between the time in which the event happens in reality (*event time*) and the time it is detected by the system (*detection time*). This phenomenon may be a result of delays in event reporting that is inflicted by synchronization problems in distributed environment (network overload) and inaccuracies in sensor readings. This phenomenon and situations in which the situation manager, a sensor, or a network line is down, may result in the detection of situations that have not occurred in reality or in missing the detection of situations that occurred in reality [6][14]. The consequences of this issue are beyond the scope of this paper.

The set of event types Σ_E is a finite set $\Sigma_E = \{E_1, E_2, \dots, E_n\}$, $n \geq 0$. An event type E is a tuple $E = (id, atts)$ where id is a unique identifier (event name) such that $\forall E_i, E_j \in \Sigma_E, i \neq j : E_i.id \neq E_j.id$. and $atts = \{att_1, att_2, \dots, att_n\}$, $n \geq 0$, is a finite set of attributes. An attribute att is a tuple $att = (id, type)$ where id is a unique identifier (attribute name) such that $\forall E \in \Sigma_E, \forall att_i, att_j \in E.atts, i \neq j : att_i.id \neq att_j.id$ and $type$ is an attribute type, $type \in \{number, boolean, string\}$.

Figure 5 – formal definition of event type

2.2. Lifespan definition

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 type* describes the common properties of a similar set of lifespans on an abstract level. It defines the set of events that can initiate the lifespan, the set of events that can terminate it, and the conditions for lifespan initiation and termination. Note that more than one lifespan of the same type 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 *trading-day*, which starts when the event *trade-start* occurs and ends when the event *trade-end* 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.

The notion of lifespan was not formally defined in previous works. It was usually simulated by the operator sequence with three operands, whose first and last operands can be looked as an initiator and terminator of the time interval, in which the second operand, usually a complex event, can occur. Note that a simulation of lifespan using the sequence operator covers a single lifespan's initiation and termination policy. Snoop [7] defines the special operators A, A*, P, P* to emulate the semantics of lifespans. However, this emulation is only partially analogous to the notation of lifespan and has different semantics. It strongly couples the lifespan with a specific composite event and the decision whether to initiate or terminate the lifespan with a parameter context. Note that a parameter context covers a single lifespan's initiation and termination policy. However, in many cases such emulations cannot represent the notation of lifespan at all.

A broker wants to run an automatic buy & sell in New York stock exchange only. He wants to detect situations during the NYSE trading day that requires the activation of such a program.

- Models that emulate the notation of lifespan using the sequence operator can only detect the required situations at the end of the trading day. However, at the end of the trading day, the knowledge that the situations represent is not relevant anymore.
- Models that emulate the notation of lifespan using special operators like Snoop's [7] A, A*, P, and P* can detect the required situations. However, they also detect composite events that represent situations that did not occur in reality because events signaling the starting and ending of trading days in different stock markets interleave. In this case all possible combinations of composite events must be detected and filtered in the condition part of the ECA rule. This results in a substantial superfluous computation.

Figure 6 – example of lifespan management

2.2.1. Lifespan initiation

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). The lifespan type defines whether lifespan instances are initiated by system startup, by event occurrences or both and under which conditions (i.e. the lifespan's initiators). The conditions that an event instance must satisfy in order to initiate a lifespan include threshold conditions on the event instance itself and a correlation code that 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 type is not already open. If the correlation code is *add*, a new lifespan is opened while any existing lifespans remain open. Multiple values of the tuple that consists of event type, threshold conditions, and correlation code, may be defined for the same lifespan. This allows a lifespan to be initiated by different events and under different conditions, and enables the definition of lifespans that represent time intervals in which situations are relevant in reality. Note that an event occurrence can initiate only a single lifespan of the same type, although it may satisfy the conditions of more than one initiation tuple defined by that type.

A broker wishes to identify situations regarding IBM's stocks and options that are traded in the New York stock exchange. These situations are relevant in time intervals that start when an IBM's option is quoted or when IBM's stock is quoted, if no situation is already being evaluated in such a time interval. All open lifespans are discarded if a situation is detected or 60 minutes after their initiation.

```
lifespan = "example 7"  
initiator = event: "option-quote"  
           where: "symbol = IBM and stock exchange = NYSE"  
           correlate: "add"  
initiator = event: "stock-quote"  
           where: "symbol = IBM and stock exchange = NYSE"  
           correlate: "ignore"  
terminator = event: "detected situation"  
            where: "symbol = IBM and stock exchange = NYSE"  
            termination type: "discard"  
            quantifier: "each"  
terminator = expiration interval: "60 minutes"  
            termination type: "discard"
```

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

1. An IBM's stock-quote event from New York initiates a new lifespan.
2. An IBM's option-quote event from Berlin is ignored.
3. An IBM's stock-quote event from New York is ignored.
4. An IBM's option-quote event from New York initiates a new lifespan.

Two instances of this lifespan, initiated by events one and four, are open simultaneously. The situation is detected in each one of these time intervals separately. When a situation is detected, with no importance to the lifespan in which it was detected, all open lifespans are discarded.

Figure 7 – example of lifespan initiation

2.2.2. Lifespan termination

A lifespan remains open since its initiation time until it is either terminated by an occurrence of a *terminator* or it expires. The lifespan type 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. The termination type also determines the conditions that an event instance must satisfy in order to terminate a lifespan. The conditions include threshold conditions on the event instance itself, a quantifier that determines which open lifespans are terminated, and a termination type that

specifies whether situations that are detected during the lifespan are discarded. 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. The termination type specifies if a situation that is detected during (or at the end of) the lifespan and was not reported should be discarded (*discard* termination type) or not (*terminate* termination type).

Multiple instances of the tuple that consists of event type, threshold conditions, termination type, 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.

The set of lifespan types Σ_L is a finite set $\Sigma_L = \{L_1, L_2, \dots, L_n\}$, $n \geq 0$. A lifespan type L is a tuple $L = (id, inits, terms)$ where id is a unique identifier (lifespan name) such that $\forall L_i, L_j \in \Sigma_L, i \neq j : L_i.id \neq L_j.id$; $inits = \{init_1, init_2, \dots, init_n\}$, $n > 0$, is a finite set of initiators; and $terms = \{term_1, term_2, \dots, term_n\}$, $n \geq 0$, is a finite set of terminators. An initiator $init$ is a tuple $init = (id, correlation, cond)$ where id is the identifier of the initiating event or the symbol startup (if the lifespan is initiated at startup) such that $\forall L \in \Sigma_L, \forall init \in L.inits, \exists E \in \Sigma_E : init.id = E.id$ or $init.id = startup$; $correlation \in \{add, ignore\}$ is a correlation code; and $cond$ is a predicate over the initiating event attributes.

A terminator $term$ is a tuple $term = (id, quantifier, termType, cond)$ where id is the identifier of the terminating event or an expiration interval (if the lifespan is terminated after a specific period of time) such that $\forall L \in \Sigma_L, \forall term \in L.terms, \exists E \in \Sigma_E : term.id = E.id$ or $term.id$ is an expiration interval; $quantifier \in \{first, last, each\}$, $termType \in \{terminate, discard\}$ is a termination type; and $cond$ is a predicate over the terminating event attributes.

Figure 8 – formal definition of lifespan

2.3. Situation definition

We have defined the inferred event (situation) to be the occurrence of a significant situation that does not happen explicitly in the physical reality, but can be logically inferred by viewing the world's state and the history of concrete event occurrences. A situation accordingly defines the set of events, both internal and external, that need to be evaluated, and the conditions they must satisfy in order to determine if a significant situation (inferred event) occurred in reality, and an internal event must be signaled. A situation also defines the information

associated with an internal event based on the information associated with the specific event instances that triggered it.

Formally a situation S is a function from a set of event types Σ_E to an event type E , where the domain is the set of events that need to be evaluated in order to decide if a situation occurred, and the range is the internal event that is triggered when the situation is detected.

We use the term situation to refer to both an inferred event and its definition. However, the intent is usually clear from the context.

The decision process, whether a situation occurred or not, is divided into three phases. Each phase is based upon one dimension of the situation definition.

1. The collection phase - event instances that play any role in the situation are collected.
2. The detection phase - event instances, whose occurrence entail detection of the situation, are selected.
3. The consumption phase - event instances that participate in the situation are removed from the collection.

Below is a short description of the decision process.

<p>While the lifespan is open, instances of events that participate in the situation are collected (1). If the conditions for situation occurrence have been met, then a subset of the event instances in the collection from those that caused the detection of the situation are selected (2). These instances are consumed and removed from the collection (3).</p>
--

Figure 9 – decision process for situation detection

2.3.1. Collection phase

A *candidate* is an event instance that has an impact upon the situation detection.

In order to decide if a situation occurred in reality, all candidates must be monitored. Moreover, it is sufficient to base this decision upon these event instances only. The definition of this candidates' collection should include the following information:

1. The time interval during which the situation detection is relevant.

A lifespan is the time interval during which a situation detection is relevant, thus, every situation is bounded to a lifespan type and only event instances that occur while the lifespan is open are considered for the situation. Note that if multiple lifespans of the same type are open

simultaneously, the situation is evaluated in each one separately, thus the detection of a situation in one lifespan does not influence the detection of a situation in other lifespans. Accordingly, the decision if an event instance is a candidate of the situation is done in each lifespan separately.

2. The types of events that can participate in the situation.

We defined a situation S as a function from a set of event types Σ_E to an event type E . We call an event that in the domain of the situation function (i.e. an event that must be evaluated in order to decide if a situation occur) an *operand* of the situation.

3. The conditions that event instances must satisfy in order to participate in the collection.

An event instance that occurs while the lifespan is open must satisfy some conditions in order to be considered as candidate. These conditions are defined for each operand of the situation and may differ among operands. They include threshold conditions on the event instance itself and override conditions that determine the influence of the new candidate on other candidates of the operand. This means that existing candidates of the operand are removed from the collection if the new candidate satisfies the override conditions.

The candidates in the collection are associated with the operand they belong to and form a separate *candidate list* for each operand. Note that a situation can have more than one operand of the same type. In this case the decision whether an event instance belongs to the candidate list of an operand is done separately for each operand, thus an event instance can be a candidate of one operand only, of some operands, or all operands.

The situation *portal-collapse* is influenced by the quotes of Yahoo and Lycos. It is detected if within a five minutes time window since the value of one of these companies increases, the value of Yahoo decreases in more than one percent in a single quote and the value of Lycos decreases in more than two percents in a single quote (i.e. high tendency for decreases).

A new lifespan is opened for this situation whenever Yahoo or Lycos stock increases, thus multiple lifespans of this situation can be open simultaneously, in each one the situation is detected separately.

This situation has two operands, both have the same event type: *stock-quote*. The first operand has a threshold condition that allows only quotes of Yahoo that decrease in more than one percent to be candidates. The second operand has a threshold condition that allows only quotes of Lycos that decrease in more than two percent to be candidates.

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

1. A Yahoo stock is quoted with an increase in its value. A new lifespan is opened.
2. A Lycos stock is quoted with a decrease of three percents in its value. The instance is a candidate of the second operand.
3. A Lycos stock is quoted with an increase in its value. A second lifespan is opened.
4. A Yahoo quote is quoted with a decrease of half percent in its value and ignored.
5. A Yahoo quote is quoted with a decrease of two percents in its value. The instance is a candidate of the first operand in both lifespans but the situation is detected only in the first one.

Figure 10 – example of collection phase

2.3.2. Detection phase

The decision process, whether a situation occurred, can be performed immediately when an event that is a candidate of the situation occurs, or when the lifespan of the situation is terminated. Similarly the report on the situation can be delayed until the lifespan terminates. These possibilities are captured by a *detection mode* that can have one of these values:

1. *Immediate* - the conditions for situation detection are evaluated immediately when a new event occurs. The situation is reported immediately when it is detected.
2. *Delayed* - the conditions for situation detection are evaluated immediately when a new event occurs. The situation, if detected, is reported at the end of the lifespan.
3. *Deferred* - the conditions for situation detection are evaluated at the end of the lifespan. The situation, if detected, is reported immediately.

The decisions, whether a situation occurred and which event instance actually triggered it, are based upon a combination of an operator, a condition, and a set of quantifiers. The quantifiers designate a selection strategy when multiple occurrences of event instances of the same type that satisfy the conditions are possible. A quantifier is applied to every operand and has six possible values: *first*, *strict first*, *last*, *strict last*, *each*, and *strict each*.

1. *First* - selects the first instance of the operand that satisfies the conditions.
2. *Strict first* - selects the first instance of the operand if it satisfies the conditions.
3. *Last* - selects the last instance of the operand that satisfies the conditions.
4. *Strict last* - selects the last instance of the operand if it satisfies the conditions.
5. *Each* - selects all the instances of the operand that satisfy the conditions.
6. *Strict Each* - selects all the instance of the operand if all of them satisfy the conditions.

Note that if event instances that satisfy the conditions cannot be selected for every operand, the situation may not be detected, depending on the situation's operator. Our model supports numerous operators that are classified to several groups.

1. Joining operators: *all* & *sequence*
 - The operator *all*(E_1, E_2, \dots, E_k) designates a conjunction of events $E_1 \dots E_k$ with no order importance.
 - The operator *sequence*(E_1, E_2, \dots, E_k) designates an ordered conjunction of events $E_1 \dots E_k$ where event E_i precedes event E_{i+1} .

The situation *portal-collapse* that was described in the previous example is defined using the operator *all*.

```

operator = "all"
detection mode = "immediate"
first operand = event: "stock-quote"
    threshold: "symbol = YHOO and change = -1"
    quantifier: "first"
second operand = event: "stock-quote"
    threshold: "symbol = LCOS and change = -2"
    quantifier: "first"

```

Its lifespan is

```

initiator = event: "option-quote"
    where: "symbol = YHOO or symbol = LCOS and change > 0"
    correlate: "add"
terminator = expirationInterval: "5 minutes"

```

The situation *last-increase* occurs if a stock increases at least two times within a lifespan. It is reported only once at the end of the lifespan with information based on the two last increases in the stock. This situation is defined using the operator *sequence* and a condition that imposes that the two quotes that trigger it have the same symbol.

```

operator = "sequence"
detection mode = "deferred"
first operand = event: "stock-quote" as: "first-quote"
    threshold: "change > 0"
    quantifier: "last"
second operand = event: "stock-quote" as: "second-quote"
    threshold: "change > 0"
    quantifier: "first"
condition = "first-quote.symbol = second-quote.symbol"

```

Figure 11 – example of joining operators

2. Counting operators: *atleast*, *atmost* & *nth*

Counting operators designate a conjunction of n weighted events. A weight, that can have a negative value, is associated with each event operand. A situation with a counting operator is triggered when the total weight of collected events satisfies the operator.

- The operator $atleast(n, E_1, E_2, \dots, E_k)$ designates a minimal conjunction of m events out of $E_1 \dots E_k$ with no order importance such that the total weight of the m events is more than n .

- The operator $atmost(n, E_1, E_2, \dots, E_k)$ designates a maximal conjunction of m events out of $E_1 \dots E_k$ with no order importance such that the total weight of the m events is less than n within the lifespan. A situation with this operator is always detected at the end of the lifespan (i.e. *deferred* detection mode).
- The operator $nth(n, E_1, E_2, \dots, E_k)$ designates a conjunction of m events out of $E_1 \dots E_k$ with no order importance such that the total weight of the m events is exactly n .

The situation *decrease-tendency* occurs if the amount of company's stocks that are traded in decrease is higher than then number of stocks that are traded in increase by more than ten. A quote with an increase in the stock value has a default weight of one and a quote with a decrease in the stock values has a weight of minus one. More than a single candidate of each operand is counted as determined by the quantifier *each*.

```
operator = "atleast 10"
detection mode = "immediate"
first operand = event: "stock-quote"
                threshold: "change < 0"
                quantifier: "each"
second operand = event: "stock-quote"
                threshold: "change > 0"
                weight: "-1"
                quantifier: "each"
```

The situation *low-tradability* occurs if not all of the company's securities are traded. In order to count only one candidate of each operand, the operands' quantifier is set to *last*.

```
operator = "atmost 3"
detection mode = "deferred"
first operand = event: "stock-quote"
                quantifier: "last"
second operand = event: "option-quote"
                quantifier: "last"
third operand = event: "bond-quote"
                quantifier: "last"
condition = "stock-quote.symbol = option-quote.symbol and
            stock-quote.symbole = bond-quote.symbole"
```

Figure 12 – example of counting operators

An operand default weight is one, thus in the default case the operator *atleast* designates a minimal conjunction of n events, the operator *atmost* designates a maximal conjunction of n events, and the operator *nth* designates a conjunction of exactly n events.

The decision whether more than a single candidate that is bounded to an operand is counted is determined by the operand's quantifier. The quantifiers *each* and *strict each* designate that more than one candidate is counted, while other quantifiers designate that only a single candidate is counted. Note that there is no restriction on the value of the enumerator n ; it can be greater than the number of operands k , equal to k , or less than k .

3. Absence operators: *not* & *unless*

- The operator *not*(E_1, E_2, \dots, E_k) designates that non of the events $E_1 \dots E_k$ has occurred within the lifespan.
- The operator *unless*(E_1, E_2) designates the occurrence of the first operand and the non occurrence of the second within the lifespan.

Situations with absence operator are always detected at the end of the lifespan (i.e. *deferred* detection mode).

The situation *strong-buy* occurs if a stock price only increases during a trading day (lifespan). This situation is defined using the *unless* operator and is not .

```

operator = "unless"
detection mode = "deferred"
lifespan = "trading_day"
first operand = event: "stock-quote" as: "quote-inc"
                threshold: "change > 0"
                quantifier: "first"
second operand = event: "stock-quote" as: "quote-dec"
                threshold: "change < 0"
                quantifier: "first"
condition = "quote-inc.symbol = quote-dec.symbol"

```

Figure 13 – example of absence operators

4. Temporal operators: *every*, *after* & *unless*

- The operator *every*(t) designates a period of $i*t$ time-units since the situation's lifespan initiation, where $i > 0$
- The operator *after*(E, t, c) designates a period of t time-units since the occurrence of E , where the correlation code c determines the

correlation between two instances of E that the time distance between their occurrences is less than t . There are three possible correlation codes: *add*, *ignore*, and *replace*.

- *Add* - both occurrences of E are considered.
 - *Ignore* - the first occurrence of E is considered and the second occurrence is ignored.
 - *Replace* - the first occurrence of E is ignored and the second occurrence is considered.
- The operator $at(tp)$ designates time points that match the time pattern tp . A time pattern is a time-stamp formatted $dd/mm/yyyy hh:mm:ss.mmm$, that can contain a wildcard, denoted "*", in its fields and matches time points that have any value in these fields. For example, the time pattern $*/11/2000 00:00:00.000$ matches the beginning of every day in November 2000.

Situations with absence operator are always triggered when they occur (i.e. *immediate* detection mode).

2.3.3. Consumption phase

A situation is usually repetitive (i.e. may occur more than once during its lifespan). However, a singular situation (i.e. may occur only once during its lifespan), denoted by the *repeat mode = once*, can be defined. Note that a repetitive situation that is detected in the *deferred* detection mode can be detected more than once at the end of the lifespan. In this case, the decision process whether a situation occurred or not, is preformed repetitively at the end of the lifespan until no more new situations are detected.

If the situation is repetitive, a decision, whether the event instances that triggered it can be considered again as candidates for the same situation, should be applied. This decision is determined by the situation's consumption policy. The consumption policy is defined by a condition that event instances that triggered the situation must satisfy in order to be considered again as a candidate of the situation. This condition, which is called *consumption condition*, can be defined for every operand. An event instance that triggered the situation and satisfies the consumption condition that is associated with its operand is removed from the

candidate list of that operand. If a consumption condition is not defined for an operand, event instances of this operand are always consumed (i.e. the default consumption condition is *true*).

The consumption modes that are defined in some composite event languages (e.g. snoop [7]) denote a predefined set of simple consumption conditions. These consumption modes can be defined in our model using a combination of a quantifier and consumption condition for each operand. Example is a situation that corresponds to a snoop [7] composite event in the *recent* parameter context that consumes the last instance of every operand when it occurs. This situation has a quantifier *last* and a *true* consumption condition for each operand.

Our model supports consumption policies that cannot be defined in existing composite event languages. These consumption policies take place when different operands have different consumption conditions or when the consumption conditions are not *true* or *false*.

The situation *increase-decrease* is detected for every pair of stock quotes where the first quote increases and the second quote decreases during a trading day (lifespan). This situation is defined using the *sequence* operator. The first operand (stock quote increases) has a quantifier *each* and a *false* consumption conditions in order to "remember" all the increases in the stock. The second operand (stock quote decreases) has a quantifier *last* and a *true* consumption conditions in order to consider only the last decrease and to avoid the multiple detection of the same pair. All pairs are reported at the end of the lifespan.

```
operator = "sequence"  
detection mode = "differed"  
lifespan = "trading_day"  
first operand = event: "stock-quote" as: "first-quote"  
    threshold: "change > 0"  
    quantifier: "each"  
    consumption condition: "false"  
second operand = event: "stock-quote" as: "second-quote"  
    threshold: "change > 0"  
    quantifier: "last"  
    consumption condition: "true"
```

Figure 14 – example of consumption phase

The situation *high-increase* occurs if a stock increases and has a volume that is higher in at least 20 points than the volume it had in the beginning of the trading day. This situation is defined using the operator *sequence* and a condition that imposes that the two quotes that trigger it have the same symbol and that the second quote is higher in more than 20 points than the first. The first operand has a quantifier *strict first* and a consumption condition *false* that imposes that only the first quote in the lifespan is considered for situation detection. The second operand has a threshold conditions that imposes that it increases, a quantifier *last*, an override condition *true*, and consumption condition *true* that impose that only the (currently) last quote is considered for situation detection and that each quote is considered only once.

```

operator = "sequence"
detection mode = "immediate"
lifespan = "trading_day"
first operand = event: "stock-quote" as: "first-quote"
                quantifier: "strict-first"
                consumption condition: "false"
second operand = event: "stock-quote" as: "second-quote"
                threshold: "change > 0"
                quantifier: "last"
                consumption condition: "true"
                override condition: "true"
condition = "first-quote.symbol = second-quote.symbol and
            first-quote.volume > second-quote.volume + 20 "

```

Figure 15 – example of consumption phase

2.3.4. Nested situations

The situation manager triggers an inferred event when it detects a situation. An inferred event, as any other event, can be used as operand in other situations. This capability enables the definition of *nested situations* – situations that are based on concrete and inferred (other situations) events. The inferred event is denoted an *inner situation*. Nested situations are roughly equivalent to operator composition [7][18] in the sense that a nested situation has an inferred event, that is triggered as a result of event composition as one of its operand. However, nested situations do not require that the same lifespan and operands' (events') selection and consumption policies are applied on the nested and inner situations. An example of situation nesting is presented figure 16.

A broker wishes to sell a stock if it has a decrease tendency and low tradability in a trading day. Situations that detected both cases are defined in figure 12. The situation *sell-stock* is a nested situation, based on the situations defined in figure 12. It correlates inferred events detected by the inner situations and checks if they both occurred on the same trading day.

```
operator = "all"  
detection mode = "immediate"  
lifespan = "trading_day"  
repeatMode = "once"  
first operand = event: "low-tradability"  
quantifier: "first"  
second operand = event: "decrease-tendency"  
quantifier: "first"  
condition = " low-tradability.symbol = decrease-tendency.symbol"
```

Figure 16 – example of a nested situation

2.4. Key Definition

Key is used to perform semantic matching of different events, by attribute's value. This is similar (semantically) to equi-join in relational databases.

A **key** denotes a semantic equivalence among attributes that belong to different events. For example the stock-exchange attribute in the stock-quote event, the stock-exchange attribute in the trade-start event, and the stock-exchange attribute in the trade-end event are semantically equivalent, in the sense that they refer to a stock exchange symbol.

Keys are used to match different event instances that refer to the same entity (New York stock exchange is an example of an entity that is referred by the stock-exchange attribute). A key divides a situation's detection process to numerous separate independent detection processes (denoted *partitions*), one partition for every group of semantically equivalent event instances. The partitioning can be performed in the lifespan level, denoted *global partitioning*; in the situation level, denoted *local partitioning*; or in both levels.

Global partitioning designates the partitioning of the situation's lifespan according to the values of the attributes defined by the dividing key, called a *global key*. If global partitioning is applied, a new lifespan is opened for every group of semantically equivalent event instances according to the lifespan definition, thus initiator and terminator events of one partition are not influenced by the existence

of open lifespans in other partitions. Moreover, an event instance that is associated with a situation operand is considered to be a candidate of the situation (if it satisfies the operand's threshold conditions) only in open lifespans that were initiated by an initiator that is semantically equivalent to the new event instance. Note that a global key must define a semantic equivalence among all the events that participate in the situation (i.e. initiators, terminators and operands). Local partitioning designates the partitioning of the situation within a lifespan according to the values of the attributes defined by the dividing key, called a *local key*. If local partitioning is applied, a separate detection process is performed for every group of semantically equivalent event instances according to the situation selection strategy (selection phase), composition strategy (detection phase) and consumption strategy (consumption phase). The decisions that are taken in each phase of the situation composition in one partition (i.e. which candidate to select? is the situation detected? which candidates to consume?) are not influenced by the existence of candidates in other partitions. Note that local partitioning is different from equality conditions on the situation operands, in the sense that it applies the consumption and selection policies to each partition separately. A local key defines a semantic equivalence among all the situation's operands.

The key *symbol* defines a semantic equivalence among *stock-quote*, *bond-quote*, and *option-quote* events using the *symbol* attribute. The key *stock-exchange* defines a semantic equivalence among *stock-quote*, *bond-quote*, *option-quote*, *trade-start*, and *trade-end* events using the *stock-exchange* attribute. The situation *stock-collapse* occurs if there are less than three decreases in a trading day. The *stock-exchange* key globally partitions this situation, thus a different lifespan is opened for every trading day in a different stock exchange. The *symbol* key locally partitions this situation, thus a different detection process is performed for every symbol (company) and for every trading day.

```

operator = "atmost 3"
detection mode = "deferred"
lifespan = "trading day"
global key = "stock-exchange"
local key = "symbol"
operand = event: "stock-quote"
threshold: "change > 0"
quantifier: "each"

```

Figure 17 – example of key definition

The set of keys Σ_K is a finite set $\Sigma_K = \{K_1, K_2, \dots, K_n\}$, $n \geq 0$. A key K is a finite set of pairs consisting of an event and one of its attributes. $K = \{eventAtt_1, eventAtt_2, \dots, eventAtt_n\}$, $n \geq 2$. A pair of an event and one of its attributes $eventAtt$ is a tuple $eventAtt = (eventId, attrId)$ where $eventId$ is an event identifier and $attrId$ is an attribute identifier such that

1. $\forall K \in \Sigma_K, \forall eventAtt \in K, \exists E \in \Sigma_E : E.id = eventAtt.eventId \wedge \exists attr \in E.attrs : attr.id = eventAtt.attrId$;
2. $\forall K \in \Sigma_K, \forall eventAtt_i, eventAtt_j \in K, i \neq j : eventAtt_i.eventId \neq eventAtt_j.eventId$; and
3. $\forall K \in \Sigma_K, \forall eventAtt_i, eventAtt_j \in K, i \neq j : typeOf(eventAtt_i.attrId) = typeOf(eventAtt_j.attrId)$ where $typeOf$ is a function that returns the type of the specified attribute.

Figure 18 – formal definition of key

The set of situation definitions Σ_S is a finite set $\Sigma_S = \{S_1, S_2, \dots, S_n\}$, $n \geq 0$. A situation S is a tuple $S = (eventId, operands, lifespanId, cond, detectionMode, globalKeys, localKeys)$ where

1. $eventId$ is the internal event that is triggered by S such that $\forall S \in \Sigma_S, \exists E \in \Sigma_E : E.id = S.eventId$
2. $operands = \{operand_1, operand_2, \dots, operand_n\}$, $n \geq 0$ is a finite bag of operands. An operand $operand$ is a tuple, $operand = (eventId, threshold, quantifier, override, consumption)$ where $eventId$ is an event identifier such that $\forall S \in \Sigma_S, \forall operand \in S.operands \exists E \in \Sigma_E : E.id = operand.eventId$; $threshold$ is a predicate over the operand's event attributes; $quantifier \in \{first, last, each, strict\ first, strict\ last, strict\ each\}$; $override$ is a predicate over the operand's event attributes; and $consumption$ is a predicate over the operand's event attributes.
3. $lifespanId$ is a lifespan identifier such that $\forall S \in \Sigma_S, \exists L \in \Sigma_L : L.id = S.lifespanId$
4. $cond$ is a predicate over the situation's operands' event attributes (includes the event algebra operator)
5. $detectionMode \in \{immediate, differed, delayed\}$
6. $globalKeys = \{keyId_1, keyId_2, \dots, keyId_n\}$, $n \geq 0$ is a finite set of key identifiers such that $\forall S \in \Sigma_S, \forall keyId \in S.globalKeys, \exists K \in \Sigma_K, \exists L \in \Sigma_L : (K.id = keyId \wedge L.id = S.lifespanId \wedge (\forall init \in L.inits, \exists eventAtt \in K.eventAttr : init.id = eventAtt.eventId) \wedge (\forall term \in L.terms, \exists eventAtt \in K.eventAttr : term.id = eventAtt.eventId \text{ or } term.id \text{ is an expiration interval}) \wedge (\forall operand \in S.operands, \exists eventAtt \in K.eventAttr : operand.eventId = eventAtt.eventId))$
7. $localKeys = \{keyId_1, keyId_2, \dots, keyId_n\}$, $n \geq 0$ is a finite set of key identifiers such that $\forall S \in \Sigma_S, \forall keyId \in S.localKeys, \exists K \in \Sigma_K : (K.id = keyId \wedge \forall operand \in S.operands, \exists eventAtt \in K.eventAttr : operand.eventId = eventAtt.eventId)$

Figure 19 – formal definition of situation

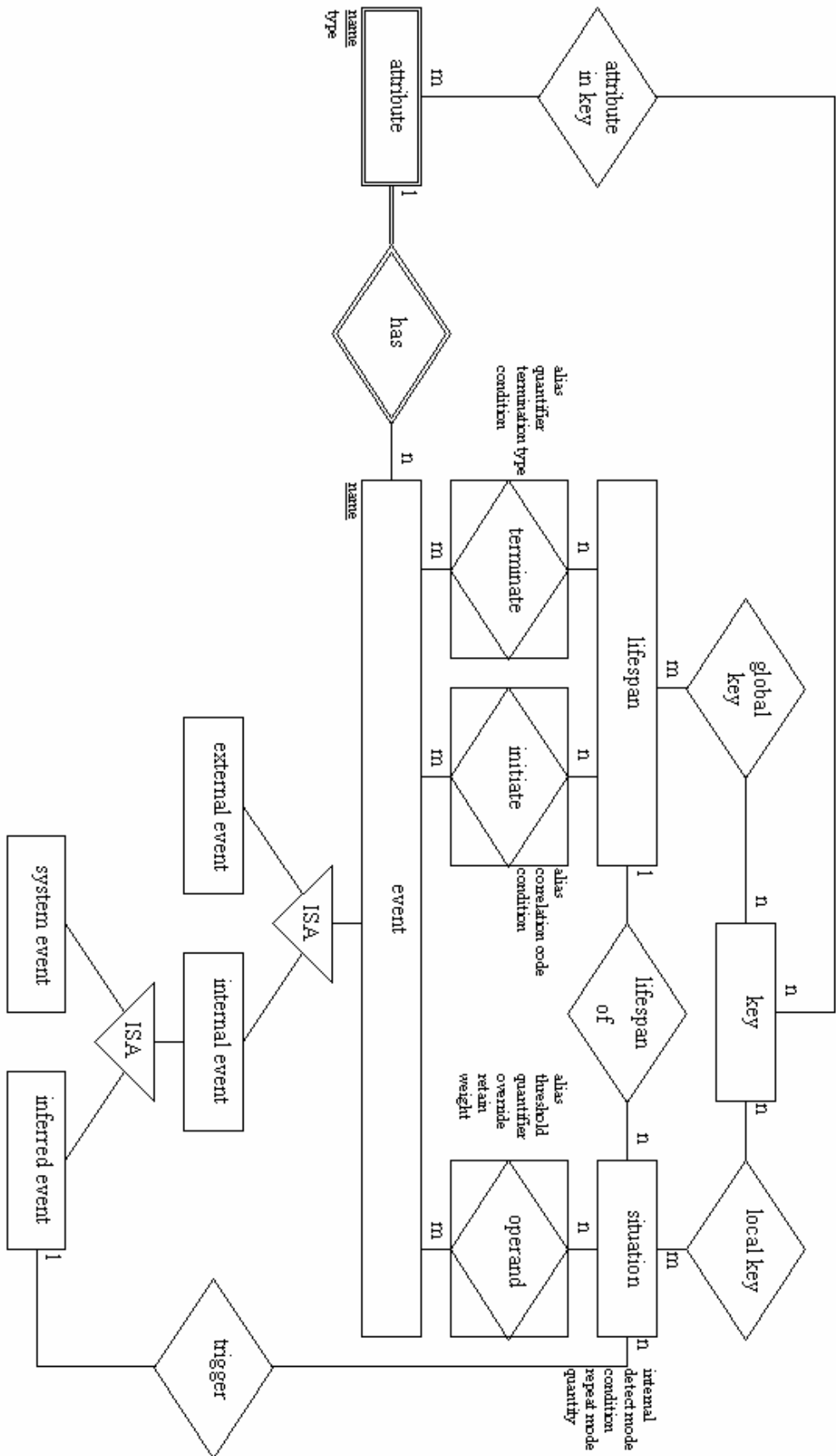


Figure 20 – entity relationship diagram of the main elements of the situation definition language

3. Data structures and algorithms

Situation composition is the process applied to detect the occurrence of a situation in reality. It is performed in three phases: the collection phase, the detection phase, and the consumption phase as described in section 2.3 - situation definition. Each phase of the composition process performs different tasks; however, all phases use the same data structure.

Section 3.1, data structures, describes the data structures that are used during the composition process; section 3.2, algorithms, describes the algorithms that are used in each phase of the composition process; section 3.3, best-case and worst-case scenario, describes the best-case and worst-case performance of the algorithms.

3.1. Data structures

The data structure, that is specific for a single situation, is classified to static and dynamic data structure. The static part maintains the situation's meta-data (definition), while the dynamic part represents knowledge about events that occur in runtime and affect the composition process. The dynamic part of the data structure consists of global and local partition tables that separate the event instances that affect the composition process to partitions, as defined by the station's global and local keys. The event instances, in each partition, are stored in *candidate lists*.

The situation's data structure is described in figure 21.

- *Global partition table* - A mapping between *global key values* and *global partitions*. A global key value is the information associated with event instances in attributes that are semantically equivalent, as defined by the situation's global key. A global partition contains open lifespans that have been initiated by events with a global key value that is associated with the global partition. If global partitioning (i.e. global key) is not defined, then there is a single global partition that holds all the open lifespans of the situation.
- *Local partition table* - A mapping between *local key values* and *local partitions* in a specific lifespan. A local key value is the information

associated with event instances in attributes that are semantically equivalent, as defined by the situation's local key. A local partition contains a set of candidate lists that contains event instances that match the local key value and are associated with the situation's operands. In a local partition, each candidate list is associated with a single operand of the situation and each operand is associated with a single candidate list. A candidate list contains event instances that satisfy the operand's threshold condition and where not consumed, sorted by the event's detection time. If local partitioning (i.e. local key) is not defined, then there is a single local partition that holds the candidate lists of the situation.

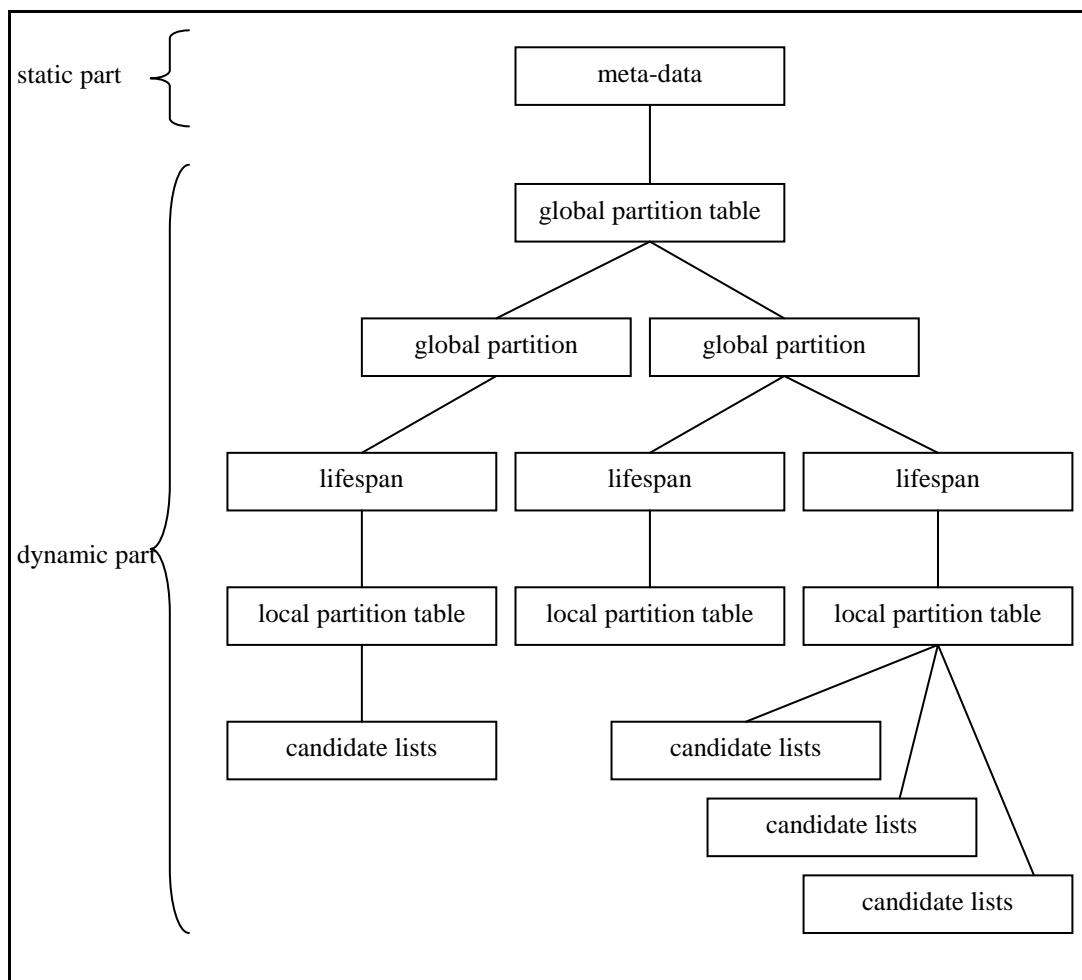


Figure 21 – situation's data structure

3.2. Algorithms

The composition process makes changes in the dynamic data structures according to event occurrences and the situation's meta-data. New event instances may initiate and terminate lifespans (lifespan management algorithms) and may be

added to the situation's candidate lists (collection phase algorithms). When the dynamic data structure is updated, the situation detection process (detection phase algorithms) can be applied. The detection process does not change the dynamic data structure, however, if a situation is detected, candidates may be consumed (consumption phase algorithms). When the dynamic data structure is updated again (after the consumption phase) a new event instance can be processed.

```

process event (event:  $e$ )
for each situation  $s$  that  $e$  might terminate
    call terminate lifespan ( $s, e$ )
for each situation  $s$  that  $e$  is associated with one of its operand
    call collect instance ( $s, e$ )
for each situation  $s$  that  $e$  might initiate
    call initiate lifespan ( $s, e$ )

```

Figure 22 – algorithm for process event

A single event instance cannot take part in a situation in different roles (i.e. it can not initiate and terminate the same lifespan, or participate in the situation and terminate or initiate the situation's lifespan) thus the evaluation order of the instance possible roles (initiation, termination, and collection) is irrelevant. The evaluation order that is preformed in the *process event* algorithm satisfies this requirement by giving priority to lifespan termination over instance collection.

3.2.1. Lifespan management algorithms

An event occurrence can result in the initiation of new lifespans and in the termination of existing ones. The initiation and termination of lifespan changes the dynamic part of the data structure. It can add or remove global partitions to the global partition table of numerous situations, if such tables exist, and change the amount of open lifespans for each situation, otherwise.

When an event instance that serves as a possible terminator occurs, the *terminate lifespan* procedure is evaluated. It is called for each situation that is associated with a lifespan that this event instance may terminate. This procedure removes open lifespans, and all the data that is associated with them, from the dynamic part of the situation's data structure and initiates the situation's detection process if the situation's detection mode is deferred (i.e. the situation is detected at the end of its lifespan). It tests the conditions for lifespan termination in all the situation's open

lifespans, if the situation does not have a global key, and open lifespans that belongs to the global partition that is associated with the global key value, if the situation defines a global key. An open lifespan is terminated if the event instance satisfies the conditions for its termination (i.e. the event instance satisfies the lifespan terminator's threshold condition and the lifespan matches the terminator's quantifier).

```

terminate lifespan (situation: s, event: e)
gv ← global key value of e as defined by s global key
gp ← global partition that is mapped to gv in the global partition table of s
for each lifespan l in gp
if e satisfies the conditions for l termination
    remove l from gp
    if s detection mode is deferred
        call detect situation (s, l)

```

Figure 23 – algorithm for lifespan termination

When an event instance that serves as a possible initiator occurs, the *initiate lifespan* procedure is evaluated. It is called for each situation that is associated with a lifespan that this event instance might initiate. This procedure adds new lifespans to the dynamic part of the situation's data structure. It tests the conditions for lifespan initiation in the global partition associated with the event instance's global key value, if the situation defines a global key or in the single global partition that exist, if the situation does not define a global key. A new lifespan is initiated if the event instance satisfies the conditions for its initiation (i.e. the event instance satisfies the lifespan initiator's threshold condition and the lifespan's correlation is satisfied).

```

initiate lifespan (situation: s, event: e)
gv ← global key value of e as defined by s global key
gp ← global partition that is mapped to gv in the global partition table of s
if e satisfies the conditions for lifespan initiation
    l ← new lifespan
    add l to gp

```

Figure 24 – algorithm for lifespan initiation

3.2.2. Collection phase algorithms

An event instance may be a candidate (i.e. influence the situation detection) in numerous situations, and for every situation it may be a candidate in each one of its lifespans. When an event instance occurs, the *add instance* procedure determines in which situations, in which global partitions (i.e. in which lifespans), and in which local partitions an event instance is considered as a candidate. This procedure is called for each situation for which the event is associated with one of its operands. It checks if the event instance is a candidate in every lifespan that belongs to the global partition that is associated with the event instance's global key value, if the situation has a global key. If the situation does not define a global key, it checks if the event instance is a candidate in all the lifespans of the situation. For each lifespan, the event instance is added to the candidate list of the operand that it is associated with its event class, if it satisfies the operand's threshold conditions. Note that if a local key is defined, the event instance is added (i.e. is a candidate) only to candidate lists that are in the local partition that is associated with the instance's local key value, in the local partition table of each lifespan. A situation detection process is performed, if the situation's detection mode is immediate or delayed (i.e. situation occurrence is detected immediately) and the event instance is added to at least one candidate list in the situation's lifespan.

```
add instance (situation: s, event: e)  
gv ← global key value of e as defined by s global key  
lv ← local key value of e as defined by s local key  
gp ← global partition that is mapped to gv in the global partition table of s  
for each lifespan l in gp  
    lp ← local partition that is mapped to lv in the local partition table of l  
    for each operand op in s  
        if e is associated with op in s  
            if e satisfies the threshold conditions of op  
                add e at the beginning of op candidate list in lp  
    if s detection mode is immediate or delayed and e was added to a candidate list  
        call detect situation (s, l, e)
```

Figure 25 – algorithm for collection phase

3.2.3. Detection phase algorithms

The detection process is performed when a lifespan is terminated, if the situation detection mode is deferred, or when a candidate event occurs, if the situation detection mode is immediate or delayed. In the first case, the detection process is applied to all local partitions that exist in the terminated lifespan. In the second case, the detection process is applied only on the local partitions in which the event becomes a candidate.

The detection process depends on the situation's operator. It applies different methods to detect situation with different operators.

```

detect situation (situation:  $s$ , lifespan:  $l$  [,event:  $e$ ])
 $lv \leftarrow$  local key value of  $e$  as defined by  $s$  local key or null if  $e$  is not specified
 $lp \leftarrow$  local partition that is mapped to  $lv$  in the local partition table of  $l$  if  $lv$  is not null
for each local partition  $lp$  in  $l$ , if  $lp$  is not null, or for  $lp$ 
    call detect [operator] situation ( $s$ ,  $lp$ )

```

Figure 26 – algorithm for detection phase

- **Joining operators (*all, sequence*)** – If the detection mode is immediate or delayed then a backtracking algorithm is applied on local partitions in which an event becomes a candidate; otherwise (detection mode is deferred), it is applied on all local partitions. If the operator is satisfied, the backtracking algorithm checks if the conditions applied by the operator, the where clause, and the operands' quantifiers are satisfied; selects the candidates that satisfy these conditions; and triggers the situation. This algorithm is based on the fact that the condition defined in the where clause can be converted to CNF (clause normal form). The CNF condition can further be converted to another form such that a clause C_i^* , $0 < i \leq k$, k is the number of operands, is a conjunction of all the clauses in the CNF condition that reference operands that are defined before the *ith* operand in the situation, and include a reference to the *ith* operand.

Note that if a CNF clause C_i references only a single operand, this clause can be removed from the condition and be added to the operand's threshold condition, if the operand's quantifier is relative.

This algorithm considers the conditions in their final form. It selects the first, the last or each candidate of every operand as defined by the

operand's quantifier (see section 2.3.2, Detection phase). It selects a candidate from the first operand, and then continues and selects a candidate from each of the ascending operands. A candidate that is selected for an operand k must satisfy the sub-condition for this operand C_k^* (also written sub-cond_{op}^* where op is the k th operand). If there are no candidates of the operand that satisfy the sub-condition, then the algorithm backtracks to the previous operand and selects another candidate that is associated with it. When a candidate is selected for every operand, an internal event, that represents the situation, is triggered using the values of the selected candidates. This algorithm considers only few combinations of candidates that may cause a situation and saves a lot of computation effort. It eliminates combinations of events that do not satisfy the conditions for situation composition. It does so by eliminating a combination if a prefix of k events in the combination does not satisfy the sub-condition C_k^* .

A situation S is triggered when a sequence of three events $E1$, $E2$, and $E3$ occur. Each event has a single attribute named x . $E1$ is the first operand of S , $E2$ is the second, and $E3$ is the third. The situation S is triggered only if the condition specified in the where clause is satisfied.

The condition in its original form is

$$E1.x > E2.x \text{ or } (E1.x = E2.x \text{ and } E2.x < E3.x \text{ and } E3.x > 5)$$

The condition in a CNF form is

$$(E1.x > E2.x \text{ or } E1.x = E2.x) \text{ and } (E1.x > E2.x \text{ or } E2.x < E3.x) \text{ and } (E1.x > E2.x \text{ or } E3.x > 5)$$

Where

$$C1 \equiv (E1.x > E2.x \text{ or } E1.x = E2.x)$$

$$C2 \equiv (E1.x > E2.x \text{ or } E2.x < E3.x)$$

$$C3 \equiv (E1.x > E2.x \text{ or } E3.x > 5)$$

The condition in its final form is

$$(E1.x > E2.x \text{ or } E1.x = E2.x) \text{ and } (E1.x > E2.x \text{ or } (E2.x < E3.x \text{ and } E3.x > 5))$$

Where

$$C^*1 \equiv \text{true}$$

$$C^*2 \equiv C1 \equiv E1.x > E2.x \text{ or } E1.x = E2.x$$

$$C^*3 \equiv C2 \wedge C3 \equiv (E1.x > E2.x \text{ or } E2.x < E3.x) \text{ and } (E1.x > E2.x \text{ or } E3.x > 5)$$

$$\equiv (E1.x > E2.x \text{ or } (E2.x < E3.x \text{ and } E3.x > 5))$$

Figure 27 – example of condition resolution

- **Counting operators (*atmost, atleast, nth*)** - If the total weight of candidates, satisfies the conditions applied by the operator (see section 2.3.2 Detection phase), a situation is triggered. A numerator totals the weight of candidates in each local partition. It increases when a new candidate is detected, and decreases when an existing candidate is consumed. When the detection process is applied, the enumerator is compared to the required number, as defined by the situation, and an internal event is triggered, if the conditions are satisfied.

If the conditions for situation composition are defined in the where clause of the situation, the backtracking algorithm that was described for joining operators is preformed. However, internal events are not triggered when the algorithm handles the last operand, but the candidates in the selection are accumulated. An internal event is triggered if the total number of accumulated candidate satisfies the counting operator.
- **Absence operators (*not, unless*)** - If there are no candidates that are associated with the situation's absence operands then a situation is triggered. A flag is raised if a candidate that is associated with an absence operand is detected. After the flag is raised, event instances are ignored by this situation thus no new candidates are considered. At the end of the lifespan (recall that absence situation are always detected in the deferred detection mode) an internal event is detected if the flag is not raised.
- **Temporal operators (*at, every, after*)** - A timer is used for the detection of temporal situations. The timer triggers an internal event that represents the situation, after the required time interval has passed. The request for the time notification is preformed when the situation's lifespan initiates for the *at* and *every* operators, when a candidate arrives for the *after* operator, and when the timer notification occurs for the *every* operator.


```

detect joining situation (situation:  $s$  local partition:  $lp$  [,operand  $op$ ] [,selection  $s$ ])
if  $op$  is last operand
     $s$  satisfies operator and conditions
    trigger internal event using  $s$ 
    return true
else
    return false
else
     $op \leftarrow$  next operand or first operand in  $lp$  if  $op$  is null
relative:
if  $op$  quantifier is relative first, relative last or relative each
     $c \leftarrow$  first/last unselected candidate of  $op$  thus sub-cond* $_{op}$  is satisfied by  $s \cup c$ 
    if not  $c$ 
        if situation was triggered
            return true
        else
            return false
    if call detect joining situation ( $s \cup c, lp, op, s$ )
        quantifier is first or last
        return true
    else
        goto relative
absolute:
if  $op$  quantifier is strict first, strict last, or strict each
     $c \leftarrow$  first/last unselected candidate of  $op$ 
if  $s \cup c$  satisfied sub-cond* $_{op}$ 
    if call detect joining situation ( $s \cup c, lp, op, s$ )
        quantifier is first or last
        return true
    else
        goto absolute
else
    return false
else
    return false

```

Figure 28 – algorithm for detection of situation with joining operator

3.2.3. Consumption phase algorithms

The consumption process is performed when the detection phase is finished, if the situation's detection mode is immediate or delayed. This process determines if candidates that actually caused the situation detection can be considered again in future detection processes. Since only a single detection process is applied if the situation's detection mode is deferred (i.e. at the end of the lifespan), the consumption process is applied only if the situation's detection mode is immediate or delayed. Candidates that caused the situation are accumulated during the detection process (i.e. when an internal event that represents the situation is triggered by a selection of candidate s , the candidates in s are accumulated). When the detection process is finished, the accumulated candidates are consumed if they satisfy the consumption conditions of the operand that they are associated with. Note that candidates are consumed and not event instance, that an event instance that is a candidate in multiple global and local partitions can be consumed in some partitions and not in others.

3.3. Best-case and worst-case scenarios

Situation definitions and reported event instances impact the performance (performance measurements are discussed in section 4) of the situation manager and the execution of the algorithms introduced in the previous section.

- A scenario where all reported event instances are not defined in Amit results in discarding all the event instances before they are considered for situation detection. The events are evaluated against a list of all event types and then discarded. The complexity in this case is $O(\log E_n)$ where E_n is the number of event types to evaluate a single event instance.
- A scenario on which a situation has several operands with a quantifier *each* and candidates that are never consumed (consumption condition is *false*) or overridden (override condition is *false*); is detected at the *immediate* detection mode; is relevant during a lifespan that is never terminated; has a *were* condition; and affected by all event instances results in exponential computation in the number of events. A new event instance always triggers evaluation of the situations that compares it

against all existing candidates (in order to evaluate the *where* condition). Since candidates are not consumed, a new event instance is checked against all previous events. The complexity in this case is $O(n^{op} \cdot l)$ where n is the number of event instance, op is the number of operands in the situation, and l is the number of open lifespans where the situation is relevant.

4. Performance measurements

The performance measurements goal is to estimate the incoming event rate that the situation manager can handle and compare it with other event management tools. A previous work that defines benchmarks of rules in active database [19] does not cover the functionality of our language. The incoming event rate that is required by an application varies among different applications. We identified several factors [23] that influence the performance of the situation manager.

1. The number of parallel open lifespans in a computation
2. The number of candidates (i.e. partially processed event instance in the situation manager that passed the threshold conditions and were not consumed).
3. The number of detected situations. Situation detection triggers an internal inferred event that is processed by the situation manager.

These factors differ from one application to the other. We defined several scenarios that describe typical applications. The results of these scenarios provide an estimation of the situation manager's performance in real life applications.

Section 4.1, scenarios, describes four scenarios. Section 4.2 results, describes the results of these four scenarios.

4.1. Scenarios

We define a set of fourteen event types, $\{E_1, E_2, \dots, E_{14}\}$, each event has a single attribute X that has a discrete value distributed evenly between 1 and 10. These events are used in order to define the situations in the scenarios. We defined ten sets of 100,000 event instances that are used to evaluate the performance of the situation manager.

Event Type	Distribution
E_1	0.2
E_2	0.2
E_3	0.1
E_4	0.1
E_5	0.1
E_6	0.1
E_7	0.1
E_8	0.05
E_9	0.02
E_{10}	0.02
E_{11}	0.0025
E_{12}	0.0025
E_{13}	0.0025
E_{14}	0.0025

Table 1 – event distribution

These event instances' sets are generated randomly using the distribution detailed in the table above.

4.1.2. Standby World

This is an empty scenario that does not define any situations. It gives an upper bound on the performance of the situation manager (i.e. the event rate that the situation manager can handle).

4.1.3. Noisy World

This is a light scenario that uses only low percentage (1%) of the event instances in order to decide if a situation occurs. The situations are not complex (i.e. no conditions, small number of lifespans are open simultaneously) and constructed from a small number of events.

4.1.3. Filtered World

This is a filtering scenario that uses high percentage (80%) of the event instances in order to decide if a situation occurs. However, high percentages of these instances (80%) are not relevant (i.e. do not satisfy the threshold conditions). The

situations are complex (i.e. conditions are applied, many lifespans are open simultaneously) and some are based on other situations (i.e. on internal events).

4.1.4. Complex World

This is a heavy scenario that uses high percentage (80%) of the event instances in order to decide if a situation occurs. The situations are complex (i.e. conditions are applied, many lifespans are open simultaneously), some based on other situations (i.e. on internal events).

<p><i>Situation s1</i></p> <p><i>operator = "sequence"</i></p> <p><i>detection mode = "immediate"</i></p> <p><i>first operand = event: "E₁₂"</i></p> <p><i>second operand = event: "E₁₃"</i></p> <p><i>lifespan = initiator: "E₁₁" correlation: "ignore"</i></p> <p><i>terminator: "E₁₄" quantifier: "each"</i></p>
<p><i>Situation s2</i></p> <p>¹<i>operator = "after 1000"</i></p> <p><i>detection mode = "immediate"</i></p> <p><i>first operand = event: "E₁₃"</i></p> <p><i>lifespan = initiator: "startup"</i></p> <p><i>no terminator</i></p>
<p><i>Situation s3</i></p> <p><i>operator = "all"</i></p> <p><i>detection mode = "deferred"</i></p> <p><i>first operand = event: "E₁₃"</i></p> <p><i>second operand = event: "E₁₄"</i></p> <p><i>local key : attribute: X</i></p> <p><i>lifespan = initiator: "E₁₁" correlation: "add"</i></p> <p><i>terminator: "E₁₂" quantifier: "each"</i></p>

Figure 29 – noisy world scenario

¹ S₂ is detected 1000 milliseconds after an instance of E₁₃ occurs

Situation s1

operator = "sequence"
detection mode = "immediate"
first operand = event: "E₁" threshold: "X > 7" alias: "E_{1A}"
second operand = event: "E₂" threshold: "X > 7"
third operand = event: "E₁" threshold: "X > 7" alias: "E_{1B}"
condition = "E_{1A}.X = E_{1B}.X"
lifespan = initiator: "E₅" correlation: "ignore" threshold: "X=3"
initiator: "E₆" correlation: "ignore" threshold: "X<2"
terminator: "E₇" quantifier: "first" threshold: "X = 7"

Situation s2

operator = "atleast 5"
detection mode = "immediate"
first operand = event: "E₂"
global key : attribute: X
lifespan = initiator: "E₆" quantifier: "add" threshold: "X = 4"
terminator: "E₈" quantifier: "each" threshold: "X = 9"

Situation s3

operator = "all"
detection mode = "immediate"
first operand = event: "S₁"
second operand = event: "E₃" threshold: "X > 7"
third operand = event: "E₄" threshold: "X > 7"
condition = "E₃.X = E₄.X"
lifespan = initiator: "E₅" correlation: "add" threshold: "X = 1"
terminator: "E₆" quantifier: "last" threshold: "X = 2"

Situation s4

operator = "not"
detection mode = "deferred"
first operand = event: "S₂"
lifespan = initiator: "E₃" correlation: "add"
terminator: "E₄" quantifier: "first"

Figure 30 – filtered world scenario

Situation s1

operator = "sequence"
detection mode = "immediate"
first operand = event: "E₁" alias: "E_{1A}"
second operand = event: "E₂"
third operand = event: "E₁" alias "E_{1B}"
condition = "E_{1A}.X = E_{1B}.X"
lifespan = initiator: "E₅" correlation: "ignore"
initiator: "E₆" correlation: "ignore"
terminator: "E₇" quantifier: "first"

Situation s2

operator = "atleast 5"
detection mode = "immediate"
first operand = event: "E₂"
global key : attribute: X
lifespan = initiator: "E₆" quantifier: "add"
terminator: "E₈" quantifier: "each"

Situation s3

operator = "all"
detection mode = "immediate"
first operand = event: "S₁"
second operand = event: "E₃"
third operand = event: "E₄"
condition = "E₃.X = E₄.X"
lifespan = initiator: "E₅" correlation: "add"
terminator: "E₆" quantifier: "last"

Situation s4

operator = "not"
detection mode = "deferred"
first operand = event: "S₂"
lifespan = initiator: "E₃" correlation: "add"
terminator: "E₄" quantifier: "first"

Figure 31 – complex world scenario

4.2. Scenarios' results

Measurements were performed on a Pentium IV 1.4 Ghz machine running windows 2000. The measurements started after the situation manger loaded the definitions and the set of event instances was generated (in memory). In runtime, the “client” thread (a java program that use the situation manager) sent event instances (one by one) to the situation manager by calling the situation manager’s API. The “client” thread yielded (the CPU) every 1000 sent events. Three parameters were monitored:

1. Number of incoming event and detected situations (i.e. number of processed events)
2. Execution time.
3. Internal execution statistics.

Table 2 presents the average results of performance measurements of 10 executions, each one with 100,000 events. The high number of processed event eliminates the affect of the sequence in which the events occurred (which is random in our case) on the measured results. The results are detailed in the table below.

	Standby world	Noisy world	Filtered world	Complex world
External events	100000	100000	100000	10000
Detected situations	0	112	30435	139111
Events + Situations	100000	100112	130435	239111
Performance Time (ms)	1372	1742	16503	124319
External events / sec	72887	57406	6060	804
Detected situations /sec	0	64	1844	1118
Events / sec	72887	57470	7903	1923
Candidates	0	1077	120662	2350754
Access to event information	0	992	1289131	11302695
Condition Performed	0	0	833487	15299472
Initiated lifespans	0	389	12543	36423
Terminated lifespans	0	387	11524	36280

Table 2 - performance measurements results

The performance measurements results show that

- a. The situation manager upper limit is about 70,000 events per second. This event rate is achieved if none of the incoming event instances are classified to events that take part in situation composition.
- b. The lower bound is about 2,000 events per second. This is considered as high performance relative to other solutions in the event composition/correlation/management spaces.
- c. The factors that significantly affect performance are
 1. The average number of parallel open lifespans (the situation manager's detection process is performed separately for each lifespan).
 2. The number of relevant event instances (i.e. partially processed event instances that passed the threshold conditions and were not consumed). A candidate represents a relevant event instance within a lifespan. A high number of candidates result from high number of relevant event instances, high number of parallel open lifespans, or both.
- d. There is not a decisive association between the number of candidates, the number of open lifespan, and the number of relevant events to the number of detected situations. The number of detected situations is also influenced by the situation's *operator* and *where* condition.

5. Related Work

We review prototypes and systems that support definition of composite events. These include prototypes from the active database domain, systems from the network management (i.e. event correlation) domain, and workflow management domain. We compare the situation manager definition language to the related work and show how it extends their semantics.

5.1. Active Database

Contemporary commercial systems do not support composite events. However, they support triggers as specified in the SQL3 standard [25]. 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.

5.1.1. ODE

ODE [18] is an active object oriented database that has been developed in Bell Laboratories and supports the specification and detection of composite events. Primitive events in ODE are triggered by the database and include object state events, method execution events, time events and transaction events. Composite events are specified as event expressions. An event expression is a mapping from a history h (sequence of primitive events) to another history, a subset of h , comprising of the points at which the event expression is satisfied. An event expression can be *NULL*, any primitive event a , or an expression formed using the operators \wedge , $!$, (not), *relative* and *relative+*. The semantics of ODE event expressions are defined as follows (E and F denote event expressions):

1. $E[null] = null$ for any event E , where *null* is the empty history.
2. $NULL[h] = null$.
3. $a[h]$, where a is a primitive event, is the maximal subset of h composed of all the occurrences of event a .
4. $(E \wedge F)[h] = E[h] \cap F[h]$.
5. $(!E)[h] = (h - E[h])$.
6. $relative(E, F)[h]$ are the event occurrences in h , at which F is satisfied, assuming that the history started immediately following some event occurrence in h at which E takes place. Formally, $relative(E, F)[h]$ is defined as follows. Let $E^i[h]$ be the i^{th} event occurrence in $E[h]$; let h_i be obtained from h by deleting all events that occurred before $E^i[h]$. Then $relative(E, F)[h] = \bigvee_i F[h_i]$ where i ranges from 1 to the cardinality of $E[h]$.
7. $relative+(E)[h] = \bigvee_{i=1}^{i=\infty} relative^i(E)[h]$ where $relative^1(E) = E$ and $relative^i(E) = relative(relative^{i-1}(E), E)$.

ODE implements composite event detection using finite state automata. This is because composite events can be expressed as regular expressions.

Amit extends the semantics of ODE in several aspects:

1. ODE does not support the operators *atleast*, *atmost*, *nth*, *at*, *after*, and *every*.

2. ODE cannot express the information reported with detected composite events, thus limiting the expressiveness of nested situation.
3. ODE has limited expressive capabilities for the definition of time intervals during which event composition is relevant using the operator *relative*(E, F) that designates the occurrence of *F* after an occurrence of *E* (initiator) and *before*(E) that designates any event before E (terminator).
4. ODE does not support selection of event instances (quantifiers)
5. ODE does not support reuse policies of event instances (i.e. events are always consumed).
6. ODE makes limited usage of the semantic information that is reported with events during event composition. It allows some filtering conditions (masks) and equality conditions (parameters) on events that participate in an event expression (composite event).

Tables 3 shows how ODE operators can be expressed in Amit

5.1.2. Snoop

Snoop [7] has been developed at the University of Florida. It is an expressive event specification language for active database, implemented in the Sentinel object oriented database [5]. Events in Snoop are atomic occurrences and include database events, explicit (also called external or abstract) events and temporal events. Events in Snoop, both primitive and composite, have a schema of parameters (attributes) associated with them. This schema describes additional information on the event that can be used only during the condition part of the ECA rule. A composite event in Snoop is defined by applying an event operator to component events that are either primitive events or composite ones.

Consequently, an event is a function from the time domain onto the boolean values. Snoop supports the disjunction, conjunction and sequence operators in addition to the following operators.

1. $\text{Any}(m, E_1, E_2, \dots E_n)$, where $m \leq n$, occurs when m distinct events out of the n events occur. $\text{Any}(m, E^*)$ specifies m distinct occurrences of an event E .
2. The aperiodic event $A(E_1, E_2, E_3)$ is signaled each time E_2 occurs during the closed interval defined by the occurrence of E_1 and E_3 . The event

$A^*(E_1, E_2, E_3)$ occurs only once when E_3 occurs and accumulates the parameters for each occurrence of E_2 .

3. The periodic event $P(E_1, t[:parameters], E_3)$ where $t[:parameters]$ is a constant time increment with optional parameter list. It occurs every t time units, starting when E_1 occurs and ending after E_3 , and collects the specified parameters. The commutative version of P , $P^*(E_1, t[:parameters], E_3)$ occurs only once when E_3 occurs. The specified parameters are collected and accumulated at the end of each period and made available when P^* occurs.

Snoop introduces the notation of parameter contexts (analogous to the notation of consumption modes introduced in HiPAC [11]) for the purpose of capturing application semantics for computing the parameters (of composite events) when they are not unique. Four contexts are introduced.

1. *Recent*: In this context, only the most recent occurrences of each E_i that have started the parameter computation are taken into account for computing the parameters of E . When E occurs, the composite event is signaled and all the occurrences that are used in the parameter relation are deleted.
2. *Chronicle*: In this context, instances of component events are taken into account in the chronological order in which they occur. When E is signaled, its parameters are computed using the oldest instance of each component event, and the parameters of these instances are deleted.
3. *Continuous*: In this context, each occurrence of an event that marks the beginning of the interval of an event expression is considered a potential candidate for stating a parameter set computation.
4. *Cumulative*: In this context, parameters of E include the parameters of all the occurrences of each component event. Whenever E is signaled, all the entries in the parameter relation associated with each component event are deleted.

Snoop uses event trees and event graphs in order to detect composite events. For each composite event an event tree is defined, and these trees are merged to form an event graph.

To illustrate parameter contexts, consider the composite events $A = any(2, E_1, E_2) ; E_3$ where E_1, E_2, E_3 are primitive events. Also consider the following sequence of event occurrences $e_{21}, e_{12}, e_{13}, e_{24}, e_{15}, e_{36}, e_{38}, e_{29}$ where e_{ij} is an occurrence of event i at time j . In the *recent* context A occurs at time 6 and includes the parameters of event instances e_{15}, e_{24} , and e_{36} . In the *chronicle* context A occurs twice. At time 6 with the parameters of event instances e_{12}, e_{21} , and e_{36} ; and at time 8 with the parameters of event instances e_{13}, e_{24} , and e_{38} . In the *continuous* context A occurs four times, all at time 6. The first occurrence of A has the parameters of event instances e_{12}, e_{21} , and e_{36} ; the second occurrence has the parameters of e_{12}, e_{24} , and e_{36} ; the third one has the parameters of e_{13}, e_{24} , and e_{36} ; and the last one includes the parameters of e_{15}, e_{24} , and e_{36} .

Figure 3 – example of snoop’s parameter contexts

Amit extends the semantics of SNOOP in several aspects:

1. Snoop does not support the operators *atleast*, *atmost*, *nth* and *unless*.
2. Snoop has limited expressive capabilities for the definition of time intervals during which event composition is relevant using the operators $A, A^*, P,$ and P^* in association with a parameter context. The lifespan element of the situation manager’s definition language covers all these possibilities and enables the definition of time intervals (e.g. the lifespan that is presented in figure 7) that cannot be expressed in snoop.
3. Snoop’s parameter contexts describe some decision possibilities for event selection (which of the candidate events that can trigger the situation, actually triggered it) and reuse (consumption). However, Snoops cannot express all possibilities of event selection and reuse policies that are expressed in Amit using a combination of a quantifier and a consumption condition. The ability to define different quantifiers and consumption conditions for each operand (in contrast to Snoop in which the parameter context is defined globally for the composed event) and the ability to evaluate event information in order to decide on the consumption policy (in contrast to snoop) enables to express Snoop’s recent, chronicle, and continuous parameter contexts in Amit along with additional reuse and consumption policies (e.g. the reuse and consumption policies of the situations that are presented in figures 14 and 15).
4. Snoop cannot use the semantic information reported with events during event composition. This information is widely used in Amit to impose event filtering (operand’s thresholds conditions), to impose reuse policies (override and consumption conditions), to partition semantically situation detection (keys), to decide on lifespan’s initiation and termination (initiators and terminators

thresholds conditions) and to impose additional conditions in the situation level (situation's where condition). Snoop, as other tools, assume that filtering (conditions) will be performed later (i.e. in the condition phase of the ECA rule). In addition to the inadequacy of this assumption, it should be noted that the only way to achieve the equivalent of simultaneous composition and content filtering in current tools is a two-phased process: phase one – composition that generates all the combinations; phase two – filtering on the results of phase one. The two-phased approach may be inefficient when the number of detected situations is much smaller relative to the number of the combinations that are produced in phase one. Furthermore, the number of combinations produced in phase one can be exponential. The ability to combine composition and filtering is a property that it improves the performance in the general case, and enables the detection of situations that are not practically feasible in other solutions, in extreme cases.

Tables 4,5, and 6 show how Snoop's operators in the recent, chronicle and continuous parameter contexts can be expressed in Amit. The cumulative parameter context can not be expressed in Amit using primitive operators. However, Amit has means to extend the language by using external functions; Snoop's cumulative parameter context functionality can be achieved by using a function that accumulates situations that are defined with a quantifier *each*. Full discussion of this extension is not in the scope of the paper.

5.1.3. General Model for Specification of the Semantics of Complex Events

Zimmer and Unland suggest a meta-model for specification of the semantics of complex events in active databases [36]. Events in this model are instantaneous, atomic occurrences and include database events, external events and temporal events.

The meta-model is based upon three independent dimensions: event instance pattern, event instance selection, and event instance consumption. These dimensions are further refined into sub-dimensions. The following paragraphs describe these dimensions in further details.

1. *Event instance pattern* of a complex event type E_i describes at an abstract level the event instance sequences that will trigger event instances of E_i . It considers five aspects.
 - 1.1. The event types whose instances must (or must not) occur in an event instance sequence and the restrictions concerning their order are defined by an event operator and its component event types. The model provides the sequence, conjunction, disjunction, negation and simultaneous operator. The simultaneous operator requires that instances of the component event types occur simultaneously.
 - 1.2. A delimiter that restricts the number of event instances of a component event type, which must occur to satisfy the event instance pattern, can be specified.
 - 1.3. Operator modes are used to define coupling and concurrency. Coupling mode defines whether event instance patterns may be interrupted by event instances not relevant to the event detection. Concurrency mode defines whether the time interval associated with the event instances, which cause a complex event to occur, may overlap.
 - 1.4. Context conditions define whether the values of a parameter of different instances must be the same or different or without any restrictions. Note that context conditions can only be imposed on context parameters (transaction, process, user, application and etc.)
2. *Event instance selection* defines which events are bounded to a complex event. This selection is performed individually for each component event and selects the first, the last or every (commutative) instance of the component that satisfies the event operator. If the *strong* keyword is specified, the first (last) instance of the component event is selected before the system checks if it satisfies the event operator. Several composite events can be triggered at once if more than one instance of an event component is selected by specifying the *combinations minimum* or *combinations* keywords. The mode *combinations minimum* defines that

only the minimum number of event instances required by the delimiter of the event component are taken into account. The mode *combination* does not impose this constraint and larger sets of event instances can be considered.

3. *Event instance consumption* determines the points in time in which events become invalid, i.e., they cannot be considered for the detection of further complex events. Three different consumption modes can be specified individually for each component event type.
 - The *shared* mode does not delete any instance of the component event.
 - The *exclusive* mode deletes all instances of the component event that were selected for the composition of the composite event.
 - The *ext_exclusive* mode deletes all instances of the component event that occurred before instances of the component events that were selected for the composition of the composite event.

The *inside* or the *outside* keyword can be specified in conjunction with the consumption mode to define the availability of event instance only inside or outside a group of composite events that are triggered together.

Amit extends the semantics of the meta-model in several aspects:

1. The meta-model does not support the operators *atleast*, *atmost*, *nth*, *unless*, *at*, *after*, and *every*.
2. The meta-model cannot express the information reported with detected composite events, thus limiting the expressiveness of nested situation.
3. The meta-model cannot express time intervals during which event composition is relevant. The terms *initiator* and *terminator* presented in the model, refer to the first and last event in an event instance sequence; these events are not used to temporal bound the event instance sequence.
4. The meta-model makes limited usage of the semantic information that is reported with events during event composition. It is limited to information reported by database events and only allows some equality conditions (parameters) on components events (operands).

5. The meta-model supports three predetermined event instance reuse policies: reuse all events, delete all events, and delete events that did not trigger a composite event.

5.1.4. Additional Research Prototypes

Research on complex events for active databases is quite comprehensive and additional research prototypes have been proposed. Most of these prototypes, including EXACT [12], REACH [37], ACOOD [3], ROCK & ROLL [13], Chimera [27], and REFLEX [28] do not offer new functionality. Other prototypes offer new functionality by introducing new operators. These include HiPAC [11] that introduces the closure operators, denoted E^* , that is signaled when E has been signaled one or more times within a transaction; SAMOS [17] that deals with the detection of complex events using colored petri-nets and introduces the history operator $TIMES(n, E)$ that is signaled after each n occurrences of E ; and NAOS [9] that introduces the strict disjunction operator that triggers a composite event if the component events occur exclusively. It also introduces some special operators for cases, in which the events are themselves composite events. Additional prototypes that are not based on event algebra but on functional programming and real time logic include PFL [33], that is based on functional programming; JEM [1][20][24] and FTL[31], that are based on temporal logic; and ADL [2].

5.2. Event Correlation

Network management tools identify network faults and send some type of alerts to an event console. These tools often flood the event console with large quantities of alerts. The system operator, who watches the event console, must look through overabundance of data before he can identify the real problem and take a corrective action.

Event correlation systems filter network-messages and correlate network data to determine if a network problem occurred. Commercial event correlation solutions include VERITAS NerveCenter [39], HP OpenView [31], SMARTS InCharge [35], and Lucent NetworkFaultManagement.

Event correlation (network management) systems are designed to handle mainly network events. Their expressive power is limited to the required functionality in

ODE	Amit
E andsign F	Operator = conjunction First operand event: "E" Second operand event: "F"
relative (E, F)	Lifespan initiator = event: "F" correlation: "ignore" Operator = nth 1 Detection mode = immediate First operand = event: "E"
relative + (E, F)	Lifespan initiator = event: "F" correlation: "add" Operator = nth 1 Detection mode = immediate First operand = event: "E"
E orsign F	Operator = nth 1 First operand = event: "E" Second operand = event: "F"
prior (E, F)	Operator = sequence First operand = event: "E1" Second operand = event: "E2"
prior (E1, E2, ... En)	Operator = sequence First operand = event: "E1" Second operand = event: "E2" ... Nth operand = event: "En"
sequence (E1, E2, ... En)	Operator = sequence First operand = event: "E1" Second operand = situation Lifespan initiator = event: "E1" Lifespan terminator = event: "E2" Operator = not Third operand = situation Lifespan initiator = event: "E2" Lifespan terminator = event: "E3" Operator = not ... Nth operand = situation Lifespan initiator = event: "En-1" Lifespan terminator = event: "En" Operator = not
first	Operator = nth1 Repeat mode = once
E F	Nested situations F where E is an operand
<n> E	Operator = nth n First operand = event: "E" quantifier: "each"
every <n> E	Operator = nth n First operand = event: "E" quantifier: "each" retain: "false"
F / E	Operator = sequence First operand = event: "E" quantifier: "first" Second operand = event: "F" quantifier: "first" Repeat mode = once

Table 3 – ODE operators expressed in Amit

Snoop	Amit
Disjunction $E1 \vee E2 \vee \dots E_n$	² Operator = nth 1 First operand event: "E1" Second operand event: "E2" ... Nth operand event: "En"
Conjunction $E1 \wedge E2 \wedge \dots E_n$	Operator = all First operand = event: "E1" quantifier: "last" Second operand = event: "E2" quantifier: "last" ... Nth operand = event: "En" quantifier: "last"
Sequence $E1 ; E2 ; \dots E_n$	Operator = sequence First operand = event: "E1" quantifier: "last" Second operand = event: "E2" quantifier: "last" ... Nth operand = event: "En" quantifier: "last"
Any(m, E1, E2, ... En)	Operator = atleast m First operand = event: "E1" quantifier: "last" Second operand = event: "E2" quantifier: "last" ... Nth operand = event: "En" quantifier: "last"
Any(m, E*)	Operator = atleast m First operand = event: "E" quantifier: "each"
A(E1, E2, E3)	Lifespan initiator = event: "E1" correlation: "add" Lifespan terminator = event: "E1" type: "discard" Lifespan terminator = event: "E3" type: "terminate" Operator = nth 1 First operand = event: "E2"
+A*(E1, E2, E3)	Lifespan initiator = event: "E1" correlation: "add" Lifespan terminator = event: "E1" type: "discard" Lifespan terminator = event: "E3" type: "terminate" Operator = nth 1 Detection mode = delayed First operand = event: "E2"
P(E1, t, E3)	Lifespan initiator = event: "E1" correlation: "add" Lifespan terminator = event: "E1" type: "discard" Lifespan terminator = event: "E3" type: "terminate" Operator = every t
P*(E1, t, E3)	Lifespan initiator = event: "E1" correlation: "add" Lifespan terminator = event: "E1" type: "discard" Lifespan terminator = event: "E3" type: "terminate" Operator = every t Detection mode = delayed

Table 4 – Snoops operators in recent parameter contexts expressed in Amit

² An Amit template can be created to express a disjunction operator explicitly. Amit template is a generic situation that is used to defined explicit situations based on parameters. A template for the disjunction operator where the specific events in the disjunction are given as parameters simplifies the definition of disjunction. Full discussion of templates in Amit is not in the scope of the paper.

Snoop	Amit
Disjunction $E1 \vee E2 \vee \dots E_n$	Operator = nth 1 First operand event: "E1" Second operand event: "E2" ... Nth operand event: "En"
Conjunction $E1 \wedge E2 \wedge \dots E_n$	Operator = all First operand = event: "E1" quantifier: "first" Second operand = event: "E2" quantifier: "first" ... Nth operand = event: "En" quantifier: "first"
Sequence $E1 ; E2 ; \dots E_n$	Operator = sequence First operand = event: "E1" quantifier: "first" Second operand = event: "E2" quantifier: "first" ... Nth operand = event: "En" quantifier: "first"
Any(m, E1, E2, ... En)	Operator = atleast m First operand = event: "E1" quantifier: "first" Second operand = event: "E2" quantifier: "first" ... Nth operand = event: "En" quantifier: "first"
Any(m, E*)	Operator = atleast m First operand = event: "E" quantifier = "each"
A(E1, E2, E3)	Lifespan initiator = event: "E1" correlation: "ignore" Lifespan terminator = event: "E3" type: "terminate" Operator = nth 1 First operand = event: "E2"
A*(E1, E2, E3)	Lifespan initiator = event: "E1" correlation: "ignore" Lifespan terminator = event: "E3" type: "terminate" Operator = nth 1 Detection mode = delayed First operand = event: "E2"
P(E1, t, E3)	Lifespan initiator = event: "E1" correlation: "ignore" Lifespan terminator = event: "E3" type: "terminate" Operator = every t
P*(E1, t, E3)	Lifespan initiator = event: "E1" correlation: "ignore" Lifespan terminator = event: "E3" type: "terminate" Operator = every t Detection mode = delayed

Table 5 – Snoops operators in chronicle parameter contexts expressed in Amit

Snoop	Amit
Disjunction $E1 \vee E2 \vee \dots E_n$	Operator = nth 1 First operand event: "E1" Second operand event: "E2" ... Nth operand event: "En"
Conjunction $E1 \wedge E2 \wedge \dots E_n$	Operator = all First operand = event: "E1" quantifier: "each" Second operand = event: "E2" quantifier: "each" ... Nth operand = event: "En" quantifier: "each"
Sequence $E1 ; E2 ; \dots E_n$	Operator = sequence First operand = event: "E1" quantifier: "each" Second operand = event: "E2" quantifier: "each" ... Nth operand = event: "En" quantifier: "each"
Any(m, E1, E2, ... En)	Operator = atleast m First operand = event: "E1" quantifier: "first" Second operand = event: "E2" quantifier: "first" ... Nth operand = event: "En" quantifier: "first"
Any(m, E*)	Operator = atleast m First operand = event: "E" quantifier = "each"
A(E1, E2, E3)	Lifespan initiator = event: "E1" correlation: "add" Lifespan terminator = event: "E3" type: "terminate" quantifier: "first" Operator = nth 1 First operand = event: "E2"
A*(E1, E2, E3)	Lifespan initiator = event: "E1" correlation: "add" Lifespan terminator = event: "E3" type: "terminate" quantifier: "first" Operator = nth 1 Detection mode = delayed First operand = event: "E2"
P(E1, t, E3)	Lifespan initiator = event: "E1" correlation: "add" Lifespan terminator = event: "E3" type: "terminate" quantifier: "first" Operator = every t
P*(E1, t, E3)	Lifespan initiator = event: "E1" correlation: "add" Lifespan terminator = event: "E3" type: "terminate" quantifier: "first" Operator = every t Detection mode = delayed

Table 6 – Snoops operators in continuous parameter contexts expressed in Amit

(domain independent) solution that supports the fundamentals of a situation definition we described earlier.

1. HP OpenView Event Correlation Services (ECS) [31] is designed to deal with the problems associated with event storms in the telecommunication environment. Events have a transit delay, which is the delay imposed by the management network, and used to reorder incorrectly ordered events. OpenView uses correlation circuits for the definition of event correlation. A correlation circuit is a set of interconnected and appropriately configured nodes, which define a logical function that represents an operator in event algebra. OpenView supports nodes that represent the conjunction, counting and unless operators. It also supports nodes for event filtering and for holding and extracting event data.
2. SMARTS InCharge [35] correlates events by employing a coding technique that matches alarms with signatures of known problems in real-time. A set of events that represent symptoms of problems is treated as a *code* that identifies the problem. A codebook is an optimal subset of events that must be monitored to distinguish the problems of interest from each other, while ensuring the desired level of noise tolerance. Consequently a codebook is a correlation matrix of problems and events. The events in the codebook are monitored and analyzed in real-time. Distinction between problems is measured by the Hamming distance between their codes, thus a decrease in the set of monitored events will cause a decrease in the tolerance for observation errors. The supported pattern on event history is a conjunction of events within a time window.
3. VERITAS NerveCenter [39] correlates network events. When a predefined network condition is detected, NerveCenter stores the event information in a finite state machine called an alarm. The alarm continues to track the status of the object being monitored. The alarm waits for subsequent events or issues polls to determine if the condition warrants further action. To correlate and filter this data, NerveCenter relies on configurable models of network and system behavior, called behavior models, for each type of managed resource. A behavior model is a group of NerveCenter objects that detect and handle a particular network or

system behavior. A typical behavior model consists of an alarm with all its supporting polls and masks, though behavior models can have multiple alarms. Any managed device can be associated with one or more behavior models.

5.3. Workflow Management

Workflow management systems (WfMS) [22] are cooperative environments in which multiple distributed processing entities cooperate in order to accomplish tasks; processing entities enact workflows by reacting to and generating new events.

Several research works [34][8][15] have proposed the use of event-condition-action rules as provided by active database management systems as one of the possible features of workflow execution; some of which use composite events to detect complex workflow situations.

Commercial WfMS [41] and standard proposals [40] do not support event composition. Although, event services (as, e.g., specified in CORBA Services [10]) support the notion of event, these services are restricted to primitive events and, typically, are hybrid in the sense that they rely on both, messages and events as coordination paradigms.

6. Conclusion

This paper has presented the “situation manager” component of Amit. Amit has been implemented in Java, and is being used as the core technology behind the e-business Management Service offering of IBM Global Services. It is also being integrated with various products and services of IBM. The situation manager was designed to achieve both: high usability level and high performance (lower bound of about 2000 events per second),

There is a substantial amount of further research that is being carried out now, and it deals with areas such as: extending Amit operators from temporal to spatio-temporal, adding uncertainty consideration, adding visualization and analysis tool around Amit, adding inference mechanism to derive rules out of model, and dealing with “deep” temporal issues.

Acknowledgement: Many of Amit ideas have been contributed by the wonderful Amit team that consists of : David Botzer, Koby Chadash, Oren Kerem, Gil Nechushtai, Royi Ronen, Ziva Sommer, Tali Yatzkar- Haham. The authors would also like to thank the reviewers for their useful comments that helped to improve the presentation of this paper.

References

1. Beck-M, Konana-P, Liu-G, Liu-Y, and Mok-A. "Active and real-time functionalities for electronic brokerage design". International Conference on Advance Issues of E-Commerce and Web-Based Information Systems, 1999.
2. 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.
3. Berndtsson-M. "ACOOD: an Approach to an Active Object Oriented DBMS.." Master's thesis, Department of Computer Science, University of Skovde, Sweden. 1991.
4. Botzer-D, Etzion-O, and Adi-A. "Semantic Event Model and its Implication on Situation Detection." Proceedings of 8th European Conference on Information Systems. July 2000; Vienna, Austria.
5. Chakravarthy-S. "Sentinel: an object-oriented DBMS with event-based rules." Proceedings of ACM SIGMOD International Conference on Management of Data. May 1997; Tucson, AZ, USA. 1997. 572-575.
6. Chakravarthy-S and Kim-SK. "Resolution of time concepts in temporal databases." Information Sciences, 80(1-2):43-89. 1994.
7. Chakravarthy-S, and Mishra-D. "Snoop: an expressive event specification language for active databases." Data and Knowledge Engineering 14.1 (1994): 1-26.
8. Cicekli-NK, and Yildirim-Y. "Formalizing Workflows Using the Event Calculus". DEXA 2000: 222-231.
9. 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.
10. "Corba". <http://www.corba.org>
11. 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.
12. Diaz-O, and Jaime-A. "EXACT: an extensible approach to active object-oriented databases." VLDB Journal. 6.4 (1997): 282-295.
13. 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.
14. Etzion-O, Gal-A, and Segev-A. "Temporal support in active databases". Workshop on Information Technologies & Systems. 1992: 245-254.
15. Etzion-O, and Kerem. "Reasoning about partially cooperative systems". Workflow Management Systems and Interoperability. Springer Verlag. 1998.
16. Fernandez-R, and Diaz-O. "Reactive Behaviour Support: Themes and Variations." Proceedings of 2nd International Workshop on Rules in Database Systems. Sept. 1995; Athens, Greece. Springer-Verlag, 1995. 69-85.
17. Gatzui-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.

18. 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.
19. Geppert-A, Gatzju-S, and Dittrich-KR. "A Designer's Benchmark for Active Database Management Systems: oo7 Meets the BEAST". Rules in Database Systems 1995: 309-326.
20. Guangtian-L, 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.
21. Hurwitz-J. "Sorting out middleware." DBMS. 11.1 (1998): 10-12.
22. Jablonski-S, and Bussler-C. "Workflow Management. Modeling Concepts, Architecture, and Implementation". International Thomson Computer Press, London 1996.
23. Jain R. "The Art of Computer Systems Performance Analysis. Techniques for Experimental Design, Measurement, Simulation, and Modeling" John Wiley & Sons. 1991.
24. Konana-P, Mok-AK, Chan Gun-L, Honguk-W, and Guangtian-L. "Implementation and performance evaluation of a real-time e-brokerage system". Real-Time Systems Symposium, 2000.
25. Kulkarni-K, Mattos-NM, and Cochrane-R. "Active Database Features in SQL3." Active Rules in Database Systems. Springer Verlag, 1999. 197-219.
26. McLeod-D, and Smith-JM. "Abstraction in databases." Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling. June 1980; Pingree Park, CO, USA. ACM Press, 1981. 19-25.
27. 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.
28. 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.
29. Paton-NW. "ADAM: An Object-Oriented Database System Implemented in Prolog." Proceedings of the 7th British National Conference on Databases. Cambridge University Press, 1989. 147-161.
30. Paton-NW, and Diaz-O. "Introduction." Active Rules in Database Systems. Springer Verlag, 1999. 3-27.
31. Sheers-KR. "HP OpenView event correlation services." Hewlett Packard Journal. 47.5 (1996): 31-42.
32. Sistla-AP, and Wolfson-O. "Temporal Triggers in Active Databases". IEEE Transactions on Knowledge and Data Engineering. 7.3 (1995): 471-486.
33. Swaup-R, Alexandra-P, and Carol-S. "PFL: An Active Functional DBPL." Active Rules in Database Systems. Springer Verlag, 1999. 297-308.
34. Tombros-D, Geppert-A, and Dittrich-KR. "Semantics of Reactive Components in Event-Driven Workflow Execution". CAiSE 1997: 409-422.
35. 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.
36. 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.
37. Zimmermann-J, and Buchmann-A. "REACH." Active Rules in Database Systems. Springer Verlag, 1999. 263-277.

38. "TIB®/Rendezvous(TM) software." TIBCO Software Inc. <http://www.rv.tibco.com>
39. "VERITAS NerveCenter™" VERITAS Software. <http://eval.veritas.com/webfiles/docs/NCOverview.pdf>
40. "White Paper - Events" Workflow Management coalition.
http://www.wfmc.org/standards/docs/Workflow_events_paper.pdf
41. "Workflow Vendors Database". Workflow And Reengineering International Association.
<http://www.waria.com/databases/wfvendors-A-L.htm>

Appendix A – Situation language DTD

<!ELEMENT amit (event | situation | lifespan | key)>

<!ELEMENT event (eventAttribute+)>

<!ATTLIST event
 name NMTOKEN #REQUIRED >

<!ELEMENT eventAttribute EMPTY>

<!ATTLIST eventAttribute
 name NMTOKEN #REQUIRED
 type (string | number | boolean) #REQUIRED >

<!ELEMENT situation (operator, situationAttribute+)>

<!ATTLIST situation
 name NMTOKEN #REQUIRED
 lifespan NMTOKEN #REQUIRED
 internal (true | false) 'false'>

<!ELEMENT operator (all | sequence | atleast | atmost | nth | not | unless | every | at | after)>

<!ELEMENT all (operandAll+ ,keyBy*) >

<!ATTLIST all
 detectionMode (immediate | deferred | delayed) 'immediate'
 where CDATA #IMPLIED
 repeatMode (once | always) 'always' >

<!ELEMENT operandAll EMPTY>

<!ATTLIST operandAll
 event NMTOKEN #REQUIRED
 as NMTOKEN #IMPLIED
 threshold CDATA #IMPLIED
 quantifier (first | last | each) 'first'
 quantifierType (absolute | relative) 'relative'
 override CDATA 'false'
 retain CDATA 'false' >

<!ELEMENT sequence (operandSequence , operandSequence+, keyBy*) >

<!ATTLIST sequence

detectionMode (immediate | deferred | delayed) 'immediate'

where CDATA #IMPLIED

repeatMode (once | always) 'always' >

<!ELEMENT operandSequence EMPTY>

<!ATTLIST operandSequence

event NMTOKEN #REQUIRED

as NMTOKEN #IMPLIED

threshold CDATA #IMPLIED

quantifier (first | last | each) 'first'

quantifierType (absolute | relative) 'relative'

override CDATA 'false'

retain CDATA 'false' >

<!ELEMENT atleast (operandAtleast+, keyBy*) >

<!ATTLIST atleast

quantity NMTOKEN #REQUIRED

detectionMode (immediate | deferred | delayed) 'immediate'

where CDATA #IMPLIED

repeatMode (once | always) 'once' >

<!ELEMENT operandAtleast EMPTY>

<!ATTLIST operandAtleast

event NMTOKEN #REQUIRED

as NMTOKEN #IMPLIED

threshold CDATA #IMPLIED

quantifier (first | last | each) 'each'

quantifierType (absolute | relative) 'relative'

override CDATA 'false'

retain CDATA 'false'

weight NMTOKEN '1'

counted (true | false) 'true' >

```
<!ELEMENT atleast (operandAtmost+, keyBy*) >
<!ATTLIST atleast
    quantity NMTOKEN #REQUIRED
    detectionMode (immediate | deferred | delayed) #FIXED "deferred"
    where CDATA #IMPLIED >
```

```
<!ELEMENT operandAtmost EMPTY>
<!ATTLIST operandAtmost
    event NMTOKEN #REQUIRED
    as NMTOKEN #IMPLIED
    threshold CDATA #IMPLIED
    quantifier (first | last | each) 'each'
    quantifierType (absolute | relative) 'relative'
    override CDATA 'false'
    weight NMTOKEN '1'
    counted (true | false) 'true' >
```

```
<!ELEMENT nth (operandNth+, keyBy*) >
<!ATTLIST nth
    quantity NMTOKEN #IMPLIED
    detectionMode (immediate | deferred | delayed) 'deferred'
    where CDATA #IMPLIED
    repeatMode (once | always) 'once' >
```

```
<!ELEMENT operandNth EMPTY>
<!ATTLIST operandNth
    event NMTOKEN #REQUIRED
    as NMTOKEN #IMPLIED
    threshold CDATA #IMPLIED
    quantifier (first | last | each) 'each'
    quantifierType (absolute | relative) 'relative'
    override CDATA 'false'
    retain CDATA 'false'
    weight NMTOKEN '1'
    counted (true | false) 'true' >
```

<!ELEMENT not (operandNot+) >

<!ELEMENT operandNot EMPTY>

<!ATTLIST operandNot
 event NMTOKEN #REQUIRED
 as NMTOKEN #IMPLIED
 threshold CDATA #IMPLIED >

<!ELEMENT unless (operandUnless, operandNot ,keyBy*) >

<!ATTLIST unless
 where CDATA #IMPLIED>

<!ELEMENT operandUnless EMPTY>

<!ATTLIST operandUnless
 event NMTOKEN #REQUIRED
 as NMTOKEN #IMPLIED
 threshold CDATA #IMPLIED
 quantifier (first | last | each) 'first'
 quantifierType (absolute | relative) 'relative'
 override CDATA 'false'>

<!ELEMENT every EMPTY >

<!ATTLIST every
 interval NMTOKEN #REQUIRED >

<!ELEMENT at EMPTY >

<!ATTLIST at
 timePattern CDATA #IMPLIED >

<!ELEMENT after (operandAfter+, keyBy*) >

<!ATTLIST after
 correlate (add | ignore | replace) 'ignore'
 interval NMTOKEN #REQUIRED >

<!ELEMENT operandAfter EMPTY>

<!ATTLIST operandAfter
 event NMTOKEN #REQUIRED
 as NMTOKEN #IMPLIED
 threshold CDATA #IMPLIED >

<!ELEMENT situationAttribute EMPTY>

<!ATTLIST situationAttribute
 name NMTOKEN #REQUIRED
 type (string | number | boolean) #REQUIRED
 expression CDATA #IMPLIED >

<!ELEMENT lifespan (initiator , terminator, keyBy*)>

<!ATTLIST lifespan
 name NMTOKEN #REQUIRED >

<!ELEMENT initiator ((startup , eventInitiator*) | eventInitiator+)>

<!ELEMENT startup EMPTY >

<!ELEMENT eventInitiator EMPTY >

<!ATTLIST eventInitiator
 event NMTOKEN #REQUIRED
 as NMTOKEN #IMPLIED
 correlate (add | ignore) 'ignore'
 where CDATA #IMPLIED>

<!ELEMENT terminator ((eventTerminator+ , expirationInterval?) | expirationInterval |
noTerminator)>

<!ELEMENT eventTerminator EMPTY>

<!ATTLIST eventTerminator
 event NMTOKEN #REQUIRED
 as NMTOKEN #IMPLIED
 quantifier (first | last | each) 'each'
 terminationType (terminate | discard) 'terminate'
 where CDATA #IMPLIED >

<!ELEMENT expirationInterval EMPTY>

<!ATTLIST expirationInterval
 timeInterval NMTOKEN #REQUIRED
 terminationType (terminate | discard) 'terminate' >

<!ELEMENT noTerminator EMPTY>


```
<!ELEMENT key (eventKey+) >
<!ATTLIST key
    name NMTOKEN #REQUIRED
    type (string | number | boolean) #IMPLIED >
```

```
<!ELEMENT eventKey EMPTY >
<!ATTLIST eventKey
    event NMTOKEN #REQUIRED
    attribute NMTOKEN #REQUIRED >
```

```
<!ELEMENT keyBy EMPTY>
<!ATTLIST keyBy
    name NMTOKEN #REQUIRED >
```