A General Model for Event Specification in Active Database Management Systems

Detlef Zimmer
Axel Meckenstock

C–LAB*
Fürstenallee 11
D-33102 Paderborn
{det|axel}@c-lab.de

Rainer Unland

Universität -GH- Essen
Schützenbahn 70
D-45117 Essen
unlandr@informatik.uni-essen.de

Abstract

Active database systems have been developed for applications that need an automatic reaction in response to certain conditions being satisfied or certain events occurring. Events can be simple in nature or complex. Complex events can be built from simpler ones with the help of event operators of an event algebra. While numerous papers propose extensions of the set of event operators only very few address the foundations of the semantics of complex events. For this reason most proposals mix different concepts (aspects) of complex events and offer event operators as the only means to control their semantics. This leads to peculiarities as aspects are not handled uniformly by operators and have other semantics than expected or operators of different algebras which, on the first glance, look the same may have different semantics. We have developed a formal meta model for complex events. It splits up the semantics of complex events into elementary, essentially independent dimensions. The resulting elementary building blocks can be used to define flexible and extensible event algebras. Moreover, our meta model helps to detect and eliminate peculiarities like the ones discussed above.

1 Introduction

Rules are used in **Active Database Systems** to monitor situations of interest and to trigger a timely response when these situations occur. Rules are useful for a number of database tasks: They can enforce integrity constraints, compute derived data, control data access, gather statistics and more. In this paper **ECA-Rules** (**E**vent-**C**ondition-**A**ction-Rules) are considered. The condition of a rule is evaluated when its triggering event occurs, and if it is satisfied its action will be executed.

Examples for triggering events, denoted as e_i, are the execution of update or retrieval operations provided by the DML of the underlying database system. Such events are pre-defined and called **primitive events**. To react on more sophisticated situations **complex events** have been introduced. They are defined from simpler ones by using operators of an **event algebra**.
For instance, events based on the sequence operator, denoted like $e_1;e_3$, are triggered whenever e_3 occurs provided that e_1 has already occurred. Another example are negation operators which can be used to define events, denoted like $e_1;\neg e_2;e_3$, which are triggered whenever $e_1;e_3$ occurs provided that e_2 did not occur between the occurrence of e_1 and e_3.

In general, complex events are triggered by a set of primitive events which often have to occur additionally in a predefined order. The events which caused a complex event to occur are bound to it and are used for the definition of its parameters. The parameters of an event are used to transfer information about the event to the other rule parts. Sometimes there may be different events to be chosen for the binding to a complex event.

In this paper we present a formal meta model which defines the semantics of complex events. It is based on the three basically independent dimensions **event condition**, **event in-**

*Cooperative Computing & Communication Laboratory (Siemens Nixdorf Informationssysteme AG, Universität Paderborn)

المنارة للاستشارات
www.manaraa.com

stance selection and **event instance consumption**, which can be further split in (sub)dimensions. The **event condition** is responsible for the specification of the point in time events occur, the **event instance selection** defines which events are bound to a complex event and the **event instance consumption** determines when events get invalid, i.e. they cannot be considered for the detection of complex events any longer.

In most event algebras event operators are the only way to specify these different dimensions thus leading to a mixture of concepts which makes the understanding of an inherent complex area even more difficult. We will show that currently in existing event algebras there are a number of peculiarities and irregularities which mainly can be put back to these mixtures.
Let us consider, e.g., event e_1 to be triggered before e_3 is triggered twice. In Snoop [CKAK94], as we will show later, this sequence cause e_1;e_3 to be triggered twice while e_1;$\neg e_2$;e_3 is triggered only once.

Literature does only provide relatively few attempts for the subdivision of the semantics of complex events. In Snoop [CKAK94] parameter contexts were introduced. They are responsible for the determination of the set of events that are bound to complex events. SAMOS [Gat94] copes with this semantics by the introduction of two additional operators called '*' and 'last', which select the oldest ('*') or most recent event ('last') out of a set of events.

In the following section we introduce some basic definitions which are used for the definition of our meta model described in section 3. In section 4 our meta model is used to define the semantics of Snoop [CKAK94] and shows how it can be used to detect irregularities. Section 5 compares our model with the models presented in literature, and section 6 concludes the paper.

2 Basic Definitions

This section introduces the basic definitions as far as they are needed for the further discussion of our concepts. A complete description of our formalized model can be found in [Zim96]. Where useful, we will present definitions and algorithms in a C++-like notation.

Definition 1:
An **event** is an indicator for the occurrence of a situation which may require an (automatic) reaction from the system. It is defined to be an instantaneous, atomic (happens completely or not at all) occurrence at one point in time.

The call of a database operation like the manipulation or retrieval of some data is an example for an event.

Definition 2:
For this paper we assume an equi-distant discrete **time domain** TD having 0 as the **origin** and consisting of points in time represented by non-negative integers.

In general events should be permitted to occur simultaneously. Some models (see [Gat94, CKAK94]) exclude such a behaviour. However we believe that such restrictions are not adequate as one single event may trigger a number of other (complex) events, which may even be of the same type[1].

Definition 3:
In the system a concrete event is represented by an **event instance** (EI) which contains the necessary information about the specific event. An **event type** (ET) describes the common essentials of a sufficiently similar set of event instances on a more abstract level. It specifies the occasion at which its events occur, defines the parameters of the event instances and lays down the impact their occurrences have on occurrences of other events.
Formally, event types are objects of the class EVENT_TYPE, which will be introduced later.

An update operation on some data x, e.g., may be invoked several times. In this case the event type specifies the pattern of the update operation. The actual invocation of the update operation is an event and this event is represented internally, that is in the system, by an event instance.

An event instance contains information like the identification of its event type (*type*), the event occurrence time (*time*), the actual set of event instances that represent those events that caused this event to occur and that are bound to it (*event_seq*) and other type specific para-

[1] In [CKAK94] a so-called parameter context *continuous* was introduced by which a single event can trigger multiple complex events of one type.

www.manaraa.com

meters (*type_spec_data*).
Formally, event instances are objects of the class EVENT (or of a subclass of the class EVENT):

```
class EVENT
{
   EVENT_TYPE          *type;
   TD                  time;
   EVENT_SEQUENCE      *event_seq;
   void                *type_spec_data;
}
```

Definition 4:
There are a number of elementary event types, called **primitive event types** (PET), which are pre-defined. Primitive event types are commonly classified into either *database* or *temporal* or *external event types*. *Database event types* correspond to database operations like, e.g., data manipulation or transaction operations while *temporal event types* specify points in time either *absolutely* (e.g. "at 5 o'clock at the 5th of november 1996") or *relatively*[2] (e.g. "5 minutes after calling the update operation on the data x"). *External event types* represent events that occur outside of the database or even the computer system and are communicated (signalled) to the database system by special database operations.

For the detection of complex events it is important to know which events have occurred and in what order they were triggered. To describe this information we introduce event instance sequences.

Definition 5:
An **event instance sequence** (EIS) is a partially ordered set of event instances that have occurred in the system. The order of the event instances corresponds to the order of their event occurrence times. It can be differentiated between instance oriented and type oriented *event instance sequences*. Instance oriented *event instance sequences* are used in event instances (see above) to represent the events which caused this event to occur. The type oriented *event instance sequences* are maintained for each event type and contain only those event instances of the system that are relevant for the detection of complex events of this type. Formally, event instance sequences are defined by the following class:

```
class EVENT_SEQUENCE
{
   list of EVENT     event_list;
                     insert( EVENT );
                     delete( EVENT );
   TDxTD             interval();
}
```

The methods *insert()* and *delete()* insert and delete the given event instance. The method *interval()* computes the time interval spanned by the event times of the event instances belonging to the sequence.
The event instance sequence that contains only event instances of primitive event types that have occurred until a specified point in time k is called **primitive event instance history** and is denoted as EIH_k^{prim}.
The **event instance history** EIH_k is the unification of all event instances of all event types defined in the system, that have occurred between the origin 0 and a given point in time k ($k \in TD$).

Applications sometimes need to react on more complex situations than expressible by primitive events. For this reason complex events are introduced, which consist of a combination of primitive events:

Definition 6:
A **complex event type** (CET) can be constructed by combining simpler event types with the help of the operators of an **event algebra**. The definition of a complex event type consists of an operator (op) which combines a number of event types. These types are called **component event types** (*component_types*) while the constructed complex event type is called **parent event type** (*parent_types*). Complex component event types (CPET) can be used to construct even more complex event types. The recursive construction of (complex) event types leads to an event type hierarchy where the primitive event types constitute its leaves. Formally, event types are defined by the following class:

```
class EVENT_TYPE
{
   list of EVENT_TYPE     parent_types;
   OPERATOR               op;
   list of EVENT_TYPE     component_types;
```

[2] As the complex event type is the essential part of the relative temporal event types (RTET), its semantics is similar to the semantics of complex types without offset. Thus we will not consider RTETs in this paper in more detail.

www.manaraa.com

```
    EVENT_SEQUENCE          seq;
    EVENT_SEQUENCE          detect();
}
```

The detection of a complex event is performed by the method *detect*(). Whenever an event is detected an object of the class EVENT is instantiated and its member variables are set. For PET the member variables *component_types*, *op* and *seq* are empty.

We assume that event types are independent of each other, i.e. events respectively event instances of a component event type that is used by several (parent) event types are available for all these types.

For the following discussions we use capitals to denote event types and small letters to denote events respectively event instances. We will use E_i to denote an event type, E_{ij} to denote its component event types and EIS^{E_i} to denote its event instance sequence. We will use e_i^s to denote the events of E_i, ei_i^s to denote the event instance representing e_i^s and $EIS^{ei_i^s}$ to denote the event instance sequence of ei_i^s. The upper index s reflects the order in which the events respectively the event instances of E_i occur. The event instances of an event instance sequence are denoted in the order of their timestamps.

Consider a complex event that is to be signalled whenever an event of an event type E_1 does occur before an event of another event type E_2 occurs. Such complex events are represented by an event type E_3 which is based on the event types E_1 and E_2 combined by a **sequence operator**. The sequence operator is denoted as ';' and the event type E_3 as $E_3 :=\ ;\ (E_1,\ E_2)$.

Note:
Time plays a subtle role on several levels of event detection and treatment. Since, in general, it will take some time to detect a (simple) event the "logical" event detection time will differ from the real physical event occurrence time, that is it will be later. It must be guaranteed that this delay in the detection of events does not cause a change in the order in which events occur on the logical level in comparison to the real order. Especially complex events will span a time interval since they consist of several (simple) events. Therefore, a selection strategy is necessary which determines the point in time out of the time interval that is assigned to the complex event as its event occurrence time. Typical selection strategies are the begin or the end of the time interval. We assume that the method *time selector ()* assigns a correct (logical) event occurrence time to an event instance.

Definition 7 :
The event occurrence times of the event instances belonging to $EIS^{ei_i^s}$ of a complex event instance ei_i^s define a time interval that reflects the period during which the detection of e_i^s took place. The event instances whose event occurrence times are equal to the left side of the time interval $e_i^s \rightarrow event\_seq.interval()$ represent the **initiator events** (**initiators**) of e_i^s, the event instances whose event occurrence times are equal to the right side of this time interval represent the **terminator events** (**terminators**) of e_i^s. The events which are neither initiators nor terminators are the **mediators** of e_i^s. While initiators mark the beginning of the detection process of complex events terminators mark the occurrence of complex events. For reasons of simplicity we assume that every event has only one initiator and one terminator.

Definition 8 :
The **Global Event Detection Algorithm** is executed whenever an event instance ei_{prim} is inserted into EIH^{prim}. It consists of the following steps:

```
1. for all E in ei_prim->type->parent_types do
     E->seq.insert(ei_prim);
2. for every E->seq.insert(e)
     event_instance_list := E.detect()
     if event_instance_list not nil
       for all e in event_instance_list do
         EIH.insert(e);
         for all E in e->type->parent_types do
           E->seq.insert(e);
```

We assume that primitive events are detected by the system, that the system generates and inserts the corresponding instances into the event instance history EIH^{prim} and that the order of the event occurrence times of these instances reflect the order in which the events they represent have occurred.
In the first step ei_{prim} is distributed to the appropriate event instance sequences of its parent types. In the second step for every event type E_i and for every insertion of an event instance into its EIS^{E_i} the event detection algorithm, specified by the *detect()* method of E_i

www.manaraa.com

(for more details see section 3.5), is executed. The instances ei_i^s of the detected events e_i^s are inserted into the EIS^{E_j} of their parent types E_j. An execution cycle of the *detect()* methods terminates if no new event is detected or if the given event type does not have a parent type. The global event detection algorithm terminates when the execution cycles of all *detect()* methods have terminated.

The *insert()* and *detect()* methods can be executed concurrently, i.e. events can concurrently be inserted into different type oriented event instance sequences, and the *detect()* method can concurrently be executed with other *detect()* methods. However, it has to be guaranteed that event instances are inserted into the event instance sequences in the order of their event occurrence times.

The *detect()* method will be considered in further detail in the following section.

3 The Meta Model

This section presents the different dimensions of complex events. A detailed formal definition can be found in [Zim96]. We focus on single event types E_i and their event instance sequences EIS^{E_i}. The interactions between the different event types and their event instance sequences are handled by the global event detection algorithm.

Consider the event instance sequence $EIS^1 := ei_1^1\ ei_1^2\ ei_3^1\ ei_2^1\ ei_2^2\ ei_2^3\ ei_3^2$. There are several questions that have to be answered to define the semantics of complex events:

1. What concrete sequences of events trigger a complex event?
   For the definition of such sequences a number of aspects have to be considered: the type of the instances belonging to a sequence, the number and the order of their occurrences, the non-existence of instances of a type, etc. The event instance sequence EIS^1, e.g., triggers an event e_4^1 of the event type $E_4 :=\ ;(E_1, E_2, E_3)$. But it does not trigger an event of a type which requires that no instance of E_3 is allowed to occur between the instances of E_1 and E_2.

2. What parts of the event instance history can be used in the condition- and action part of a rule?
   The event instances which can be accessed in these rule parts are defined by the event instance sequence *event_seq* of the complex event instance representing the event that had triggered the rule. In general there are several possibilities for its definition. Consider the event e_4^1 of E_4 triggered by EIS^1. The event instance sequence of ei_4^1 can be defined as $(ei_1^1\ ei_2^1\ ei_3^2)$ or $(ei_1^1\ ei_2^3\ ei_3^2)$ or $(ei_1^1\ ei_1^2\ ei_2^1\ ei_2^2\ ei_2^3\ ei_3^2)$ or by any other valid combination.

3. What instances are exclusively consumed by a complex event instance, that is what instances disappear after they were used by this complex event instance, and what instances are only used but not consumed by a complex event instance?
   Event instances of the type E_4 may, e.g., consume the necessary instances of type E_1 and E_2 while they preserve instances of E_3. Thus, by EIS^1 two events e_4^1 and e_4^2 of E_4 may be triggered, where ei_4^1 contains the event instance sequence $(ei_1^1\ ei_2^1\ ei_3^2)$ and ei_4^2 $(ei_1^2\ ei_2^2\ ei_3^2)$.

Each question addresses a different dimension of an event specification. We will call these dimensions *event condition*, *event instance selection*, and *event instance consumption*. In the following we will concentrate on the semantics of each dimension; the complete syntax is listed in the appendix.

3.1 Event condition

The event condition part of a type E_i (should not be confused with the condition part of a rule) specifies the pattern, i.e. the sequence of event types, that triggers events of E_i. It has to consider four aspects. We will introduce them with the help of the following example.

Consider the event instance sequence $EIS^2 := ei_1^1\ ei_1^2\ ei_3^1\ ei_2^1\ ei_3^2$ and the event types $E_4 :=\ ;\ (E_1, E_2, E_3)$, $E_5 :=\ ;\ (E_1, E_2)$ and $E_6 :=\ ;\ (E_3, E_5)$.

The event type E_4 specifies that event instances of the types E_1, E_2 and E_3 must occur in the

المنارة للاستشارات
www.manaraa.com

same order as it is specified by the sequence operator in the definition of E_4 (*pattern*). An event e_4^1 of type E_4 will be triggered as soon as ei_3^2 occurs. However, the initial time interval that belongs to ei_4^1 starts with ei_1^1 and terminates by ei_3^2. Within this time interval there are two instances of event type E_1 (ei_1^1 and ei_1^2) and two instances of event type E_3 (ei_3^1 and ei_3^2). It has to be clarified how many of these instances are at least and at most necessary to trigger an event of type E_4 (*frequency*). In the event type E_5 it is specified that first an instance of type E_1 has to occur and then an instance of type E_2. However, it remains unclear whether these event instances are to be tightly coupled (without interruption by any other event instance) or loosely coupled (arbitrary other event instances can occur between the instances of E_1 and E_2) (*coupling*). During the detection of an event e_5^1 of E_5 (initiated by e_1^1 or e_1^2 and terminated by e_2^1) an event e_3^1 of E_3 occurs. Now the event occurrence time of the instance ei_5^1 may be taken from ei_2^1 and thus the event occurrence time of ei_3^1 may be older than that of ei_5^1. The type E_6 must specify whether it allows such concurrency (*concurrency*).

3.1.1 Pattern

Event operators help to define the shape of the pattern that uniquely identifies events of that event type. Our model provides the following basic, however complete set of operators:
The $==$-operator (**simultaneous operator**) requires that instances occur simultaneously, the ;-operator (**sequence operator**) requires that instances occur in a specified order, the $\wedge$-operator (**conjunction operator**) requires that a number of instances occur in any order, the $\vee$-operator (**disjunction operator**) requires that at least one of the specified instances occurs and the $\neg$-operator (**negation operator**) requires that the given instance(s) should not occur in a given period respectively interval.

Periods are specified in form of event instances which mark the beginning and the end of the time interval being considered for evaluation. The negation operator makes only sense in conjunction with a period during which the non-occurrence of event instances is monitored. Thus the negation operator needs at least three operands. The first one specifies the beginning and the last one the ending of the period during which the non-occurrence of the instances of the 'inner' types are to be monitored.

3.1.2 Frequency

For every component event type E_{ij} a **delimiter** may be specified which restricts the number of event instances of E_{ij} which must occur for the event condition to evaluate to true. If no delimiter is given one or more instances of a type must occur. As a delimiter an upper and/or a lower bound may be given. Thus the number of event instances required for an event condition to evaluate to true may be restricted to a range or even to a concrete number.

3.1.3 Coupling and Concurrency

Operator modes are, among others, used to define the coupling mode and the concurrency feature. The mode (*coup mode*) defines whether event patterns have to occur continuously (**continuous**) or may be interrupted by event instances not being relevant for the event detection (**non-continuous**). The mode (*cc mode*) considers the time intervals associated with event instances ($e \rightarrow event\_seq.interval()$) and is used to define whether the time intervals associated to the event instances which cause a complex event to occur may or may not overlap (**overlapping** vs. **non-overlapping**).

3.2 Event Instance Selection

The event instance selection is responsible for the construction of the instance oriented event instance sequences, i.e. it determines what event instances are taken from the type oriented event instance sequence EIS^{E_i} to form the event instance sequence $EIS^{ei_i^s}$. This selection is performed individually for each component event type E_{ij} and it is quite natural to take at least all those instances which caused the complex event to occur.

Consider the event instance sequence $EIS^3 := ei_1^1\ ei_2^1\ ei_1^2\ ei_2^2\ ei_2^3\ ei_1^3\ ei_3^1$, the event type $E_4 :=\ ;(E_1, E_2, E_3)$ and the event e_4^1 of E_4 which is

www.manaraa.com

triggered by EIS^3. Let us assume that the instances ei_1^2 and ei_3^1 have already been selected for the types E_1 and E_3. Thus, we will focus on the selection of instances of the type E_2. Consider the four event instance sets: (1) (ei_1^2 ei_2^2 ei_3^1), (2) (ei_1^2 ei_2^3 ei_3^1), (3) (ei_1^2 ei_2^2 ei_2^3 ei_3^1) and (4) (ei_2^1 ei_1^2 ei_2^2 ei_2^3 ei_3^1). While the sets (1) and (2) contain only one instance of E_2 the sets (3) and (4) contain several instances. The set (1) contains the oldest instance of E_2 which together with ei_1^2 and ei_3^1 fulfils the event condition of E_4 while the set (2) contains its most recent instance. Note, that the selection of event instances depends on instances that were already selected: if we had chosen ei_1^1 instead of ei_1^2 the instance ei_2^1 would be the oldest instance of E_2 which could be selected. The set (3) contains only instances which occurred between the selected instances of E_1 and E_3 and thus takes the event condition into account while the set (4) contains every instance of E_2.

The selection strategies can be classified according to the aspect whether only the minimum number of event instances required by the delimiter of E_{ij} is selected (minimum instance set oriented) or more (cumulative oriented) (would be to select in the most extreme case the whole set of event instances of E_{ij} belonging to EIS^{E_i}).

First (**last**) are minimum set instance oriented selection strategies. They always select the set of instances with the oldest (youngest) timestamps. A cumulative oriented selection strategy is **cumulative** which selects the complete instance set of E_{ij}. The other cumulative oriented selection strategy **restricted cumulative** chooses only that instances of E_{ij} that are considered by the event condition of E_i (**restricted cumulative**).

To get a unique solution one has to define in what order event instances have to be collected from the given event instance sequence. Let us consider the event type $E_4 :=$; (**last** :E_1, **first** :E_2, E_3) and the event sequence EIS^3. Since ei_3^1 had triggered the recognition of an event e_4^1 of E_4 it is the terminator instance of the time interval assigned to ei_4^1. However, the initiator instance of ei_4^1 is not yet clear. Of course, it must be an instance of E_1. However, which one is to be chosen: ei_1^1 or ei_1^2 or ei_1^3. If the event type sequence were traversed from right to left, that is first the correct instance of event type E_3 is identified (which, of course is trivial) then the instance of E_2 and finally the instance of E_1, we would have to choose ei_3^1, ei_2^1, and ei_1^1. If we traversed the event type sequence in opposite direction, we would first have to choose ei_1^3 since it represents the last occurrence of an event of E_1 in EIS^3. However, this would not lead to a legal solution because the time interval specified by initiator instance ei_1^3 and terminator instance ei_3^1 does not include an instance of E_2. Therefore, we have to backtrack. This means, that first or last have to be interpreted as first or last legal instance of the given type. The next possibility would be ei_1^2. The interval spanned by ei_1^2 and ei_3^1 contains ei_2^2 which is not the first instance of E_2, however, the first in the interval spanned by ei_1^2 and ei_3^1. Therefore, ei_1^2, ei_2^2, and ei_3^1 would be the correct solution if left to right traversal were chosen. With right to left traversal ei_1^1, ei_2^1, and ei_3^1 would be the correct solution. To summarize, it first has to be specified in what order the time interval which belongs to a concrete event instance of an event type has to be traversed, either from left-to-right or from right-to-left. The corresponding event instance sequence is then traversed in this order and the first legal solution that is identified is the correct one.

This semantics can be controlled by the **operator mode** *trav mode* which can be either **left-to-right** (default mode) or **right-to-left**.

3.3 Event Instance Consumption

Event instance consumption defines the impact of the occurrences of events of a complex event type E_i on the availability of the events of its component event types for the subsequent detection of events of E_i. The set of events which are considered by the detection of events of the type E_i are defined by the event instance sequence EIS^{E_i}. Thus, event instance consumption defines the set of event instances which are deleted from EIS^{E_i} whenever an event e_i^s occurs.

Consider the event type $E_3 :=$; (**last** :E_1, **first** :E_2) and its event instance sequence $EIS^{E_3} :=$ ei_1^1 ei_1^2 ei_2^1 ei_2^2. With the occurrence of e_2^1 an instance ei_3^1 is generated whose event instance sequence $EIS^{ei_3^1}$ contains the instances ei_1^2 and

www.manaraa.com

ei_2^1. Now let us assume that an event instance ei_2^s of the type E_2 is consumed if it is used in an event instance sequence of a ei_3^s. The occurrence of ei_3^1 may cause the deletion of (1) ei_1^2 or of (2) ei_1^1 and ei_1^2 or (3) no deletion. Dependent on this result ei_2^2 will (1+3) or will not (2) trigger another event e_3^2 of E_3 and if it is triggered ei_3^2 may get the event instance sequence (ei_1^2 ei_2^2) (3) or (ei_1^1 ei_2^2) (1).

We distinguish three different consumption modes which can be specified individually for each component event type. The **shared** mode does not harm any instance of E_{ij} (3). The **exclusive parameter** mode removes all instances of E_{ij} from EIS^{E_i} that belong to the event instance sequence of an instance ei_i^s (1). The **exclusive** mode deletes all instances of E_{ij} from EIS^{E_i} that occur before the terminator instance of ei_i^s (2).

If the same component event type E_{ij} is used several times in the definition of E_i the strongest consumption mode will dominate the weaker ones.

3.4 Event Groups

A terminator instance can trigger the recognition of several events of its parent type if it is used in the *shared* mode. All such events (event instances) of the same event type E_i which are triggered by the same terminator form an **event group**.

To avoid infinite looping (e.g. by infinitely constructing event instances from the same basic event instances) different event instances of the same type must differ in that they are not allowed to exclusively use the same (basic) event instances. Consider the event instance sequence $EIS^4 := ei_1^1\ ei_1^2\ ei_1^3\ ei_2^1\ ei_1^4\ ei_2^2\ ei_3^1$ and the event type $E_4 :=$; (.... E_1, ... E_2, **shared** :E_3). The occurrence of ei_3^1 will trigger a number of events of E_4, depending on the parameters defined for E_1 and E_2. Let us assume that E_2 has the parameters **first : exclusive parameter**, E_1 is either **first : shared** (complex event type E_5), **last : shared** (E_6), or **restricted cumulative : shared** (E_7). Figure 1 presents the semantics of the event types.

This example shows that the event instance selection modes introduced above are not suitable for the definition of event types that behave like the event type E_8 (see figure 1). The instances of the event group of E_8 consider all those event instance sets which contain exactly one instance for each component event type.

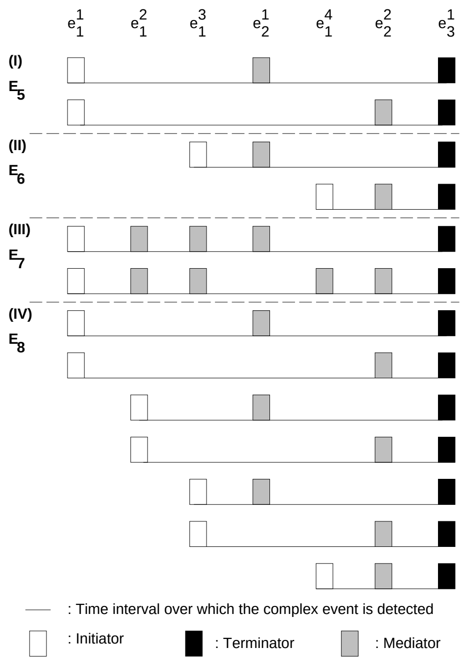


Figure 1: Event Instance Selection for Event Groups

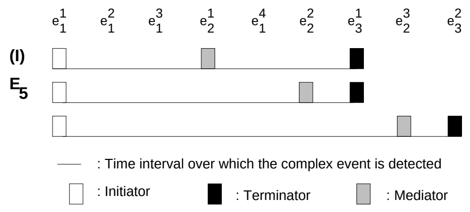
To support this semantics the event instance selection modes **combinations** and **combinations minimum** are introduced. If one of these modes is used for a component type E_{ij} the different instance sets of E_{ij} are alternately used and combined with the event instance sets of the other component event types to form the event instance sequences *event_seq* of the event instances belonging to a group. While the mode **combinations minimum** defines that only the minimum number of event instances required by the delimiter of E_{ij} are taken into account **combinations** does not impose this constraint, i.e. it can also consider larger sets of event instances.
Note, that these modes must only be used for event types E_i whose events share their terminator events.

The event type E_8 whose behaviour is shown in figure 1 can be defined as $E_8 :=$; (**combinations minimum : shared** : E_1, **combinations minimum : shared** : E_2, **shared** : E_3).

Now consider the event type E_5 defined above and the event instance sequence EIS^4 ex-

المنارة للاستشارات
www.manaraa.com

tended by ei_2^3 ei_3^2. Figure 2 shows the semantics of E_5. Every pair of events of E_2 and E_3 will subsequently trigger an event of E_5. The event instance consumption modes do not offer the possibility to distinguish between the availability of event instances inside and outside a group. Thus event instances that are shared by instances cannot be protected from being shared by other groups, too.



e_1^1 e_1^2 e_1^3 e_2^1 e_1^4 e_2^2 e_3^1 e_2^3 e_3^2
(I)
E_5
: Time interval over which the complex event is detected
: Initiator
: Terminator
: Mediator

Figure 2: Event Instance Consumption for Event Groups

To cope with this semantics we introduce the domains inside a group (**inside**) and outside a group (**outside**). They can be used in conjunction with the consumption modes and define the availability of event instances only inside and, additionally about outside a group (as in figure 2).
The event instance consumption for the **outside** domain is applied to the union of the event instance sequences *event_seq* of the event instances belonging to the same group.

3.5 Event Detection

The detection of the events of a event type E_i is performed by its *detect()* method. It is called whenever event instances are inserted into EIS^{E_i}. The skeleton of *detect()* shows the interactions between the dimensions introduced above:

```
detect()
{
  EVENT_SEQUENCE  event_instance_list = {};

  while (event_condition())
  {
    e := new EVENT;
    e.event_seq := event_instance_selection();
    e.time := time_selector();
         :
    event_instance_consumption(e);
    event_instance_list += e;
  }
  return (event_instance_list);
}
```

In the first step the event condition of the event type E_i is evaluated. If the condition is fulfilled a new event instance ei_i^s is generated and - among others - its event instance selection sequence *event_seq* is computed and the event instances that were consumed by ei_i^s are finally deleted from the event instance sequence EIS^{E_i}. These steps are repeated until the event condition is no longer fulfilled.

4 Specification of Snoop

In this chapter the semantics of the event definition language Snoop [CKAK94] is specified with our meta model.

Snoop has been developed at the University of Florida. The concepts of Snoop have been implemented in a prototype called Sentinel [CKAK94, Kri94]. The definition of the semantics of Snoop is only partially formal. Thus sometimes - as it is shown in [Zim96] - its semantics is ambiguous.

In addition to the standard event operators conjunction ($\triangle$), disjunction (∇), sequence (;) and negation (NOT) Snoop defines the operators: ANY, A, P, A^* and P^*. Events based on the ANY operator, denoted as ANY(m, E_1, E_2, .., E_n), where m $\leq$ n, occur whenever m events out of the n distinct event types occur. Events based on the a-periodic operator A, denoted as A(E_1, E_2, E_3), occur whenever the a-periodic event (E_2) occurs during the closed time interval specified by E_1 and E_3. The periodic operator P is used to define periodically occurring temporal events. A^* and P^* are cumulative versions of the operators A and P, i.e. events based on them are only triggered once at the end of the time interval (E_3).

Based on the initiator and terminator events Snoop defines four **parameter contexts**:

In the **recent** context only the most recent instance of the set of instances that may have started the detection of a complex event is used. When a complex event occurs the instances of the component event types which cannot be initiators of future events are deleted. An initiator of an event continues to initiate new event occurrences until a new initiator occurs.
In the **chronicle** context the oldest instance of each component event type is bound to the in-

www.manaraa.com

stances of its parent type. The instances of the component event types can only be used once.
In the **continuous** context, if a terminator event is detected, for each initiator an event instance of its parent type is generated. In this context, an initiator is used at least once for detecting complex events.
In the **cumulative** context all instances of the component event types are bound to the instance of the parent type. The instances of the component event types can only be used once for an event detection.

Every Snoop operator can be combined with one of these parameter contexts to form an event type. A complex event can be constructed by applying several operators which may have assigned different parameter contexts. This aspect has not been investigated by Snoop in further detail.
In contrast to our model the parameter modes of Snoop can be specified on the level of complex event types rather than on the component event type level. The parameter contexts define a fixed combination of the values of the dimensions *event instance selection* and *event instance consumption* introduced in our model. Thus the configuration possibilities offered by Snoop are limited to these combinations.

Table 3 presents the specification of Snoop on the basis of our meta model. The first column of the table presents in each block the definition of an event type. The first definition is based on Snoop while the second one is based on our meta model. The second column lists the component event types that are necessary for the definition of the complex event types on the basis of our meta model. The other columns show how the different parameter contexts of Snoop can be modelled in our meta model. For every component event type the values of the dimensions *event instance selection* and *event instance consumption* are listed. We have used the following abbreviations for these values: *sh* for **sh**ared, *ex* for **ex**clusive, *in* for **in**side, *out* for **out**side, *param* for **param**eter, *cum* for **cum**ulative, *comb* for **comb**inations, *min* for **min**imum and *rest* for **rest**ricted. Whenever the semantics of Snoop, from our point of view, was ambiguous or irregular we marked it by adding the '?' and '?!' symbols.

Let us consider some examples which describe some irregularities of Snoop which were detected with the help of our specification presented in table 3. In these examples the Snoop event types are denoted by a parameter context followed by an event operator and its operands.

Consider the event types E_4 := **recent** E_1;E_3 and E_5 := **recent** NOT(E_2)[E_1,E_3] and the event instance history $EIH^1 := ei_1^1\ ei_3^1\ ei_3^2$. The history EIH^1 causes the recognition of two events e_4^1 and e_4^2 of E_4 represented by ei_4^1 ($EIS^{ei_4^1} := ei_1^1 ei_3^1$) and ei_4^2 ($EIS^{ei_4^2} := ei_1^1 ei_3^2$) but of only one event e_5^1 of E_5 represented by ei_5^1 ($EIS^{ei_5^1} := ei_1^1 ei_3^1$).
An event type based on the negation operator is an extension of the event type based on the sequence operator defining the time interval during which the non-existence of events is monitored. Thus in the case of the non-occurrence of the specified events one would assume that the behaviour of these event types is the same. But, unfortunately, they use different event instance consumption modes for their first component event type: While the event type based on the sequence operator uses the shared mode the event type based on the negation operator uses the exclusive mode.

The *recent* and the *chronicle* parameter contexts use different event instance consumption modes. While in the *recent* parameter context the event instances are often[3] shared they are used exclusively in the *chronicle* parameter context:
Consider the event types E_3 := **recent** ($E_1 \triangle E_2$) and E_4 := **chronicle** ($E_1 \triangle E_2$) and the event instance history $EIH^2 := ei_1^1\ ei_2^1\ ei_2^2$. The history EIH^2 causes the recognition of two events e_3^1 and e_3^2 of E_3 represented by ei_3^1 ($EIS^{ei_3^1} := ei_1^1 ei_2^1$) and ei_3^2 ($EIS^{ei_3^2} := ei_1^1 ei_2^2$) but of only one event e_4^1 of E_4 represented by ei_4^1 ($EIS^{ei_4^1} := ei_1^1 ei_2^1$).

The event instance consumption mode of the first component event type of events based on the a-periodic operator used in the *chronicle* context is **exclusive parameter**. Thus, the characteristics of the a-periodic operator get lost, as it is signalled only once (for every event instance of the first component event type).
Consider the event type E_4 := **chronicle** A(E_1, E_2, E_3) and the event instance history $EIH^3 := ei_1^1\ ei_2^1\ ei_2^2\ ei_2^3\ ei_3^1$. The history EIH^3 causes

[3] Only terminator events are used exclusively.

المنارة للاستشارات
www.manaraa.com

event type	comp. event type parameter context	recent	chronicle	continous	cumulative
$E_1 \triangle E_2$ =	E_1	last : sh :	first : ex param :	comb min : sh in : ex out :	cum : ex param :
$\wedge(E_1, E_2)$	E_2	last : sh :	first : ex param :	comb min : sh in : ex out :	cum : ex param :
ANY $(3, E_1, E_2, E_3)$	E_1		first : ex param :	comb min : sh in : ex out :	cum : ex param :
= $\wedge(E_1, E_2, E_3)$	E_2	?	first : ex param :	comb min : sh in : ex out :	cum : ex param :
	E_3		first : ex param :	comb min : sh in : ex out :	cum : ex param :
$E_1 \triangledown E_2$ =	E_1	ex param :	ex param :	ex param :	ex param :
$\vee(E_1, E_2)$	E_2	ex param :	ex param :	ex param :	ex param :
$E_1 ; E_2$ =	E_1	last : sh :	first : ex param :	comb min : ex param in : ex out :	cum : ex param :
$;(E_1, E_2)$	E_2	ex param :	ex param :	sh in : ex param out :	ex param :
NOT $(E_2)[E_1, E_3]$	E_1	last : **ex** ?!	first : ex param :	comb min : ex param in : ex out :	? ex param :
= $\neg(E_1, E_2, E_3)$	E_3	ex param :	ex param :	ex param out :	ex param :
A (E_1, E_3, E_2) =	E_1	last : sh :	first : ex param :	comb min : ex param in : **sh out :**	? first : ?! ex :
$\neg(E_1, E_2, E_3)$	E_3	ex param :	?! ex param :	?! sh in : ex param out :	?! ex param :
A* (E_1, E_2, E_3) =	E_1	last : ex :	first : ex param :	comb min : ex param in : ex out :	? first : ex :
$\vee(\neg(E_1, E_2, E_3)$,	E_2	? ex :	rest cum : sh :	rest cum : sh in : ex out :	rest cum : ex :
$;(E_1, E_2, E_3))$	E_3	ex param :	?! ex param :	sh in : ex param out :	ex param :

? The semantics are not quite clear ?! The semantics seem to be irregular

Figure 3: The semantics of the complex events defined in Snoop

the recognition of only one event e_4^1 of E_4 represented by ei_4^1 ($EIS^{ei_4^1} := ei_1^1 ei_2^1$) and not - as one may expect - three events.

Let us consider event types based on the aperiodic operator A used in the *continuous* parameter context. The event instance consumption mode of their first component event type is **exclusive parameter inside : shared outside**, i.e. its instances can only be used once inside a group but they are shared between different groups. Thus instances of the second component event type occurring after an instance of the first type are not only associated with this instance but also with every older instance of the first type:
Consider the event type E_4 := **continuous** A(E_1, E_2, E_3) and the event instance history $EIH^4 := ei_1^1\ ei_2^1\ ei_1^2\ ei_2^2\ ei_2^3$. The history EIH^4 causes the recognition of the events e_4^1, e_4^2, e_4^3, e_4^4 and e_4^5 of E_4 represented by ei_4^1 ($EIS^{ei_4^1} := ei_1^1 ei_2^1$), ei_4^2 ($EIS^{ei_4^2} := ei_1^1 ei_2^2$), ei_4^3 ($EIS^{ei_4^3} := ei_1^2 ei_2^2$), ei_4^4 ($EIS^{ei_4^4} := ei_1^1\ ei_2^3$) and ei_4^5 ($EIS^{ei_4^5} := ei_1^2 ei_2^3$).

5 Related Work

A systematic and profound debate about event specification has not yet started. Instead, first, however, often fragmentary attempts can be found in papers describing specific rule models or prototype systems.

In ADL (**A**ctivity **D**escription **L**anguage) [Beh94] the event instance selection policy is fixed to the **last** mode[4]. The event instance consumption strategy is fixed too and corresponds to the **exclusive parameter** mode.

In Ode [GJS92] the semantics of the event instance selection is discussed shortly. The selection of event instances consists of two steps: In the first step the alternative event instance sequences which fulfil the event condition are computed. In the second step the event instance selection is performed through queries on the event instance sequence set computed in the first step. A detailed description of possible selection strategies as well as their realization is postponed to a future paper.
For every event condition a fixed event instance

[4]It is mentioned that in principal different event instance selection policies may be used. But this topic is not examined in further detail.

www.manaraa.com

consumption policy is chosen. For instance, event types based on the sequence operator have the following semantics: E_3 := ; (**first : exclusive parameter :** E_1, **shared :** E_2).

For the operators of the event algebra of SAMOS [Gat94] the event instance selection and event instance consumption policies are fixed. But the user has the possibility to partly influence these aspects by using different combinations of operators. To see this let us have a look at a sequence of events. We will examine three different kinds of SAMOS operators which can be combined with the SAMOS sequence operator: The *-operator, the last-operator and the TIMES-operator. Consider the four SAMOS event types $E_3 := (E_1;\ E_2)$, $E_4 := (*E_1;\ E_2)$, $E_5 := (last E_1;\ E_2)$, $E_6 :=$ (TIMES([>0],E_1); E_2). Their semantics seems[5] to be equivalent to the semantics of the following event types of our model:
E_3 := ;(**first : exclusive parameter :** E_1, **exclusive parameter :** E_2),
E_4 := ;(**first : exclusive parameter :** E_1, **exclusive parameter :** E_2),
E_5 := ;(**last : exclusive parameter :** E_1, **exclusive parameter :** E_2),
E_6 := ;(**cumulative : exclusive parameter :** E_1, **exclusive parameter :** E_2).
The event instance consumption mode *shared* is not available in SAMOS. To summarize, SAMOS offers some fixed combinations of the event instance selection and the event instance consumption modes. Thus the configuration possibilities are restricted.
Note that the semantics of the SAMOS event type $E_3 := (last E_1;\ E_2)$ is different to the semantics of the Snoop event type E_4 := **recent** E_1;E_2.

To our very best knowledge [CFPT95] is the only work, that presents a formal model of an active database system which is composed of a set of dimensions. This work might be seen as an extension of [PDW+93].
According to complex events only one dimension, called **event instance consumption**, has been presented. But this dimension is used in a different way than we have used it: In their model event instance consumption determines when events become invalid for the triggering of rules while we consider event instance consumption as the aspect, which defines when events become invalid for the detection of other events.

[5]Because of the informal definition of the semantics of the composition of operators it is not really clear, which semantics is used.

6 Conclusions

In this paper we have presented a formal meta model which defines the semantics of complex events. It is based on the three dimensions **event condition**, **event instance selection** and **event instance consumption**. The **event condition** is responsible for the specification of the point in time events occur, the **event instance selection** defines which events are bound to a complex event and the **event instance consumption** determines when events become invalid, that is can not be considered for the detection of complex events any longer.
In principle, while an event condition is associated to an event type itself the dimensions event instance selection and event instance consumption are related to the component event types of complex event types.
The three dimensions are independent with respect to their usage, i.e. they can be combined without any restrictions. Especially the event instance selection and the event instance consumption policies can be chosen separately for each component event type of a complex event type. Moreover our model considers simultaneously occurring events.

The model presented in this paper contributes to the ongoing work in the area of active database systems in several ways:

1. We developed a flexible meta model which can be used to specify the semantics of complex events defined in other rule models.
2. The presented model can be used to compare the semantics of complex events defined in existing rule models.
3. Our model can be used as a base for the definition of new rule models. A higher level interface can be built on top of the model to enhance the applicability.
4. The semantics of our model is defined formally. Thus different interpretations of

www.manaraa.com

the semantics of a given complex event are not possible.

5. The meta model helps to gain a better understanding of the basics complex events rely on and their interrelationships.

7 Appendix

Formally the syntax of a complex event type is defined as follows:

```
< CET >         ::= [< time select > :]
                    < event cond >
< event cond >  ::= [< op mode >] < OP >
                    ( < CPET_list > )
< CPET_list >   ::= < CPET > |
                    < CPET > , < CPET_list >
< CPET >        ::= [< opd mode >] < ET >
< ET >          ::= < PET > | < CET >
< OP >          ::= == | ; | ∧ | ∨ | ¬
< op mode >     ::= [< cc mode > :] [< coup mode > :]
                    [< trav mode > :]
< cc mode >     ::= non-overlapping | overlapping
                    | [default : overlapping]
< coup mode >   ::= continuous | non-continuous
                    | [default : non-continuous]
< trav mode >   ::= left-to-right | right-to-left
                    | [default : left-to-right]
< opd mode >    ::= [< par select > :]
                    [< cosu > :] [< delimiter > :]
< par select >  ::= first | last | cumulative
                    | restricted cumulative
                    | combinations [minimum]
                    | [default : last]
< cosu >        ::= inside < cosu mode > :
                    outside < cosu mode >
                    | < cosu mode >
< cosu mode >   ::= exclusive | shared
                    | exclusive parameter |
                    [default:exclusive parameter]
< delimiter >   ::= (< integer >) | (< range >)
< range >       ::= < integer > - < integer > |
                    < integer > - | - < integer >
< time select > ::= begin | end | [default : end]
```

The definition of the syntax reflects the three dimensions of the semantics of a complex event: The **event condition** is defined by an operator (*op*), an operator mode (*op mode*) and a list of component event types (*CPET_list*) if necessary supplemented by a delimiter. The **event instance selection** is defined by the modes *par select* and *trav mode* while the **event instance consumption** is defined by the mode *cosu*.

References

[Beh94] H. Behrends. An Operational Semantics for the Activity Descpription Language ADL. Technical Report TR-IS-AIS-94-04, Universität Oldenburg, June 1994.

[CFPT95] S. Comai, P. Fraternali, G. Psaila, and L. Tanca. A Uniform Model to Express the Behaviour of Rules with Different Semantics. In M. Berndtsson and J. Hansson, editors, *First Int'l Workshop on Active and Real-Time Database Systems (ARTDB-95)*, 1995.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proc. 20th Very Large Data Bases*, pages 606–617, October 1994.

[Gat94] Stella Gatziu. *Events in an Active, Object-Oriented Database System.* Phd-Thesis. Dr. Kovac, November 1994.

[GJS92] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model and Implementation. In *Proc. 18th Very Large Data Bases*, pages 327–338, October 1992.

[Kri94] V. Krishnaprasad. Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture and Implementation. Master's thesis, University of Florida, 1994.

[PDW+93] N.W. Paton, O. Diaz, M.H. Williams, J. Campin, A. Dinn, and A. Jaime. Dimensions of Active Behaviour. In N. W. Paton and M H. Williams, editors, *Rules in Database Systems, Edinburgh 1993*, pages 40–57, 1993.

[Zim96] D. Zimmer. A Formal Metamodel for the Definition of the Semantics of Complex Events. C-LAB Report 29, C-LAB, Fürstenallee 11, 33102 Paderborn, Germany, http://www.c-lab.de, December 1996.

www.manaraa.com