ON THE SEMANTICS OF COMPLEX EVENTS IN ACTIVE DATABASE MANAGEMENT SYSTEMS

Detlef Zimmer
WestLB Systems
Völklinger Straße 4
D-40219 Düsseldorf / Germany
detlef_zimmer@westlb-systems.de

Rainer Unland
Universität-GHS Essen
Schützenbahn 70
D-45117 Essen / Germany
unlandr@informatik.uni-essen.de

Abstract

Active database management systems have been introduced for applications needing an automatic reaction in response to certain events. Events can be simple in nature or complex. Complex events rely on simpler ones and are usually specified with the help of operators of an event algebra.
There are quite a few papers dealing with extensions of existing event algebras. However, a systematic and comprehensive analysis of the semantics of complex events is still lacking. As a consequence most proposals suffer from different kinds of peculiarities. Independent aspects are not treated independently leading to shady mixtures of aspects in operators. Moreover, aspects are not always treated uniformly. Operators may have other semantics than expected.
This paper addresses these problems by an extensive and in-depth analysis of the foundations of complex events. As a result of this analysis, a (formal) meta-model for event algebras will be introduced that subdivides the semantics of complex events into elementary, independent dimensions. Each of these dimensions will be discussed in detail. The resulting language specification fulfils the criteria for a good language design (like orthogonality, symmetry, homogeneity, lean set of language constructs) to a large extend.

1 Introduction

Rules are used in *Active Database Management Systems* to trigger a timely response when situations of interest occur. They can enforce integrity constraints, compute derived data, control data access, gather statistics and much more.
In this paper the most general form of rules, the so-called *ECA-Rules* (*E*vent-*C*ondition-*A*ction-Rules) are considered. The condition of a rule is evaluated whenever its triggering event occurs. If the condition is satisfied, the specified action will be executed.
Examples of triggering events are the execution of update or retrieval operations provided by the Database Manipulation Language (DML) of the underlying database system. Such events are predefined and are called *primitive events*. To react to more sophisticated situations, *complex events* were introduced. They are constructed from simpler ones by using operators of an *event algebra*.

Example 1:

A complex event $e_c=e_1; e_3$, based on the sequence operator (;) and two primitive events e_1 and e_3, is triggered whenever e_3 occurs, provided that e_1 did already occur. Another example is a negation operator ($\neg$) that can be used to define events, denoted by $e_1; \neg e_2; e_3$, which are triggered whenever $e_1; e_3$ occurs, provided that e_2 did not occur in between e_1 and e_3.

In general, complex events are triggered by a number of primitive and/or complex 'component' events which, in addition, may have to occur in a predefined order. The events that cause a complex event to occur are *bound* to it. They provide all information about the event that is necessary for an appropriate execution of the other parts of the rule. Such information is transferred by *parameters*. Sometimes a comprehensive set of primitive and/or complex component events is available for the binding process from which the relevant subset has to be chosen.
Usually, languages for the definition of complex events rely on an event algebra. However, quite a number of existing languages do not sufficiently support essential language features like orthogonality, homogeneity, and symmetry. Moreover, the set of language constructs provided is sometimes too overloaded (overlapping), so that the same situation can be described in different ways. In most event algebras, such as those defined in HiPAC ([DaBC96]), SAMOS ([Gatz94]), Ode ([GeJS92]), Chimera ([MePC96]), and NAOS ([CoCo96]), event operators are the only way to specify the semantics of complex events. This leads to a confusion of concepts that makes the understanding of an inherently complex area even more difficult. A number of peculiarities and irregularities in existing event algebras can be attributed mainly to these confusions.
While a lot of research papers address specific aspects of complex events a systematic and comprehensive analysis of the semantics of complex events is still lacking. We have developed a formal meta-model for

event algebras based on Evolving Algebras ([Gure94]), which defines the semantics of complex events ([Zimm98], [ZiUn98], [ZiUM97]). It is based on three independent dimensions. The *event type pattern* describes the overall structure of the (complex) event to be monitored. The *event instance selection* defines in case of the occurrence of several instances of the same events which ones are to be bound to a complex event. Finally, the *event instance consumption* determines when events become invalid. Invalid events can no longer be considered for the detection of complex events.

The next section will introduce necessary definitions for the further discussion. Our meta-model is described in section 3. We first introduce the three basic dimensions that define the semantics of complex events and then discuss them in more detail. Section 4 considers related work while section 5 concludes the paper.

2 Basic definitions

This section introduces in short relevant basic definitions for the introduction of our model.

Definition 1: *primitive events, event instances and types*

Every situation that may require an (automatic) reaction by the system can form a *primitive event*. A primitive event is assumed to be instantaneous and atomic; i.e., it cannot be further dismantled and happens completely or not at all. It is bound to a specific point in time. An example is the invocation of a database operation that manipulates some data.

Similar to the distinction between types and instances of types in (object-oriented) programming languages we distinguish between event instances and event types.

An *event type* (ET) describes on an abstract level the essential factors that unambiguously identify the occurrence of an event of that type, lays down the impact of its occurrence on the occurrences of other events, and specifies the parameters that sufficiently describe the specific features of the event for further use in the reaction to that event. Each concrete occurrence of an event type is represented within the system by its specific *event instance* (EI), whose main task is to save all relevant information about the event in parameters.

Definition 2: *database, temporal, external event types*

Primitive event types are commonly classified into database, temporal, and external event types. *Database event types* correspond to database operations such as data related operations (access/read/modify/insert) or transaction related operations (begin/abort/end). *Temporal event types* specify points in time either *absolutely* (e.g., "at 5 o'clock on the 5th of November 1997"), *relatively* (e.g., "5 minutes after calling the update operation on the data item *x*") or *periodically* (e.g., "every 5 seconds").

External event types represent events that occur outside the database system (sometimes even outside the computer system) and are communicated (signaled) to the database system by special database management system operations.

While primitive events can cover a huge range of simple application demands they fall short to cover more sophisticated demands as well. This led to the introduction of complex events.

Definition 3: *complex, component, parent event types*

A *complex event type* is constructed by (recursively) combining simpler event types with the help of the operators of an underlying *event algebra*. An operator (*op*) may have one or more arguments each of which represents a primitive or complex event type. Such 'included' types are called *component event types* while the resulting complex event type is called the *parent event type*.

The construction process leads to an event type hierarchy with primitive event types constituting its leaves while the complex component event types form its interior nodes.

For the detection of complex events it is important to know which events have occurred and in what order. This information is recorded in event instance sequences.

Example 2:

Consider a complex event type E_5 that is to be signaled whenever an event of type E_1 occurs before an event of type E_2. E_5 can be described by a sequence operator (denoted by ';') that has the underlying component event types E_1 and E_2 as its parameters: E_5:=;(E_1, E_2).

Definition 4: *event instance sequence (EIS)*

An *event instance sequence* (*EIS*) is a (partially) ordered set of event instances. Its order reflects the (assigned) occurrence times of its event instances. EIS is sometimes also called *history* in literature.

Definition 5: *event context, parameters*

Events occur in a context that often is relevant to the execution of the other parts of the rule. In principle, the *event context* is defined by the database state existing at the occurrence time of the event. It can be distinguished between an action- and a data-oriented context. The *action-oriented context* provides information about the transaction (parameter *ta*), the operating system process (*proc*), the application (*appl*), and the user (*user*) being responsible for an

event. The *data-oriented context* can be further subdivided into the direct and the indirect operation context. The *direct operation context* describes the DML-operation that did trigger the event. Possible information is the identification of the operation (*op-id*), its actual parameters (*op-param*) and the identification of the object on which it was executed (*op-context*). The *indirect operation context* provides the identifications of the types (*data-type*) and the data instances (*data-id*) being accessed and reflects their state (*data-state*). The event context, as well as other relevant information, like the identification of the underlying event type and the event occurrence time, are preserved in the *parameters* of the instance representing this particular event. The type and quantity of the parameters is declared in the event type definition. In essence, a context determines which information about the context of a given event is available for further evaluation.
Note, that the context of an event depends on the class its type belongs to. Temporal events[1] usually do not have a context. External events maintain an action-oriented context while only database events provide the full size context.

Definition 6: *initiators, interiors, terminators*
The oldest and the youngest occurrence time of the component event instances representing the event instance sequence of a parent event define the time interval during which the detection of the complex event takes place. The event instance(s) with the *oldest* (*youngest*) event occurrence time represent(s) the *initiator(s)* (*terminator(s)*) (since events can occur simultaneously several event instances may have the same occurrence time (see Axiom 5)). All other events instances are called *interiors*. Although a complex event usually spans a time interval, its associated event occurrence time is a single point in time. It is equal to the occurrence time of its terminator(s) (see Axiom 4). While an initiator marks the beginning of a detection process of one or more complex events, a terminator signals the definite occurrence of at least one complex event[2].

Definition 7: *system, type and instance specific EIS*
Every event detection algorithm must take as its basis the sequence of all event instances that ever occurred in the system and are still valid. We will call this sequence the *system specific event instance sequence (EIS^S)*. From EIS^S *type specific EIS* (EIS^T) can be deduced, with each EIS^T comprising only those instances of EIS^S that are relevant to the detection of complex events of type E^T. For each event type E^T exactly one EIS^T need to be maintained. Whenever a primitive or complex event instance is detected it is inserted into EIS^S as well as into the EIS^T of each of its parent event types. Whenever a terminator of a complex event is detected the appropriate *instance specific EIS* has to be constructed from EIS^T as every occurrence of an event of type E^T is represented by its specific EIS.
EIS^S is sometimes also called the *global (event instance) history* and EIS^T the *(composite event instance) history of a type*.

Example 3:
Let us assume that EIS^1:= $ei_1^1\ ei_1^2\ ei_3^1\ ei_2^1\ ei_2^2\ ei_2^3\ ei_3^2$ represents the current system specific EIS (EIS^S). On this basis EIS^2:= $ei_1^1\ ei_1^2\ ei_2^1\ ei_2^2\ ei_2^3$ would be the type specific EIS (EIS^T) for event type E_5:=;(E_1, E_2) while EIS^3:= $ei_1^1\ ei_1^2\ \ ei_3^1\ ei_3^2$ would represent event type E_6:=;(E_1, E_3). Possible event instance sequences of type E_5 are ($ei_1^1\ ei_2^1$), ($ei_1^1\ ei_2^2$), ($ei_1^1\ ei_2^3$), ($ei_1^2\ ei_2^1$), ($ei_1^2\ ei_2^2$) or ($ei_1^2\ ei_2^3$).

Notations:
Capitals will denote *event types* and *small letters event instances*. E_i denotes an event type, E_{ij} its component event types and *EIS*(E_i) its event instance sequence. e_i^s (ei_i^s) denotes an *event* (*event instance*) of type E_i with *s* reflecting the order of the occurrence times of the events (event instances) of this type. Note, that in this paper expressions will usually be written in *prefix-notation*.

Axiom 1:
The *time domain* is equidistant and discrete and has 0 as its *origin*. Each point in time is represented by a non-negative integer.

Axiom 2:
Primitive events are detected by the system. The system generates the corresponding instances. Their event occurrence times reflect the order in which they occurred.

Axiom 3:
Event types are *independent* of each other, i.e., an event instance of an event type, which is a component event type of several parent event types, is available for all these types.

Axiom 4:
The occurrence time of the terminator of a complex event instance always represents the occurrence time of the complex event as well.

Axiom 5:
Events can occur simultaneously. Thus, different event instances may get the same event occurrence time. Some models (e.g. [Gatz94], [CKAK94]) exclude such a behavior. However, we believe that such restrictions are not adequate as a database and a temporal event may occur simultaneously or one

[1] Note, that the context of a relative temporal event corresponds to the context of the event it is based on. Thus, its instance contains the instance of the event it is based on as a separate parameter.

[2] As we will see later, a terminator may as well trigger several events of the same or of different types.

single event may trigger a number of other (complex) events, which may even be of the same type[3].

Axiom 6:
For reasons of simplicity we assume in this paper that every event has only one initiator and one terminator[4].

3 A meta-model for complex events

This semi-formal presentation of our meta-model will focus on single event types E_i and their event instance sequences EIS(E_i). The formal and complete description of our meta-model can be found in ([Zimm98]).

3.1 Subdivision of the semantics of complex events

There are in essence three questions that are to be answered for the semantics of complex events to be defined sufficiently. Each question addresses a different dimension of an event specification.
The following examples rely on the event instance sequence $EIS^1 := ei_1^1\ ei_1^2\ ei_3^1\ ei_2^1\ ei_2^2\ ei_2^3\ ei_3^2$.

Question 1: (event type pattern)
How can a small, however expressive set of operators, constructs, and descriptors look like that allows a sufficiently high number of relevant complex event types to be specified unambiguously?
This question addresses aspects like the event types whose instances are needed to trigger the underlying complex event as well as those, that are not allowed to occur, or the number and possible (partial) order of occurrences of instances, etc. For example, the event pattern $E_5 := (E_1, \neg E_3, E_2)$ requires at least one instance of E_1 to occur before an instance of E_2 occurs provided that no instance of E_3 did occur in between.

Question 2: (event instance selection)
When an event of a complex event type is detected the instance specific EIS that represents this event has to be deduced from the given type specific EIS (EIS^T). Since EIS^T may contain more than one component event instance of the same type it can be ambiguous which one(s) is (are) to be chosen as representative(s) of this type in the instance specific EIS.
Let us come back to Example 3 in which several possible instance specific EIS for E_5 were already listed. However, in principle, other combinations like ($ei_1^1\ ei_1^2\ ei_2^1\ ei_2^2$) or ($ei_1^2\ \ ei_2^1\ ei_2^2\ ei_2^3$) can also be valid. Therefore, the question is what occurrences of component event instances do exactly belong to the instance specific EIS? This question is especially relevant for the further processing of the rule since the instance selection decides what information/data will be kept for future utilization.
From now on, $EIS(E^T)$ will denote the type specific EIS of event type E^T while $EIS(ei_T^n)$ denotes the n^{th} instance specific EIS of E^T, i.e. the EIS assigned to ei_T^n.

Question 3: (event instance consumption)
When an $EIS(ei_T^n)$ is deduced from its underlying EIS^T the question arises which of the chosen component event instances can still be used to form other instance specific EIS and what component event instances are consumed by a parent event instance, i.e., what instances of EIS^T vanish after they were used in an instance specific EIS?
For example, let us assume that event instances of type $E_7 := (E_1, E_2, E_3)$ consume the chosen component instances of type E_1 and E_2 while they do preserve instances of E_3. Thus, two events e_7^1 and e_7^2 of type E_7 will be triggered by EIS^1. e_7^1 will be represented by the event instance sequence ($ei_1^1\ ei_2^1\ ei_3^2$). Since it consumes the chosen instances of type E_1 and E_2 it will modify EIS^1 to $EIS^{1'}$ ($= ei_1^2\ ei_3^1\ ei_2^2\ ei_2^3\ \ ei_3^2$). e_7^2 will rely on the event instance sequence ($ei_1^2\ ei_2^2\ \ ei_3^2$) and will transform $EIS^{1'}$ to $EIS^{1''}$ ($= ei_3^1\ ei_2^3\ ei_3^2$)[5].

Each of the above questions addresses a different dimension of an event specification. We will call these dimensions *event type pattern*, *event instance selection*, and *event instance consumption*.

3.2 Semantics of the dimensions

In this section we will informally discuss the semantics of each dimension; its syntax is listed in the appendix.

3.2.1 Event type pattern

The event type pattern of an event type E_i describes at an abstract level the event type sequence and its restrictions that will trigger an event of E_i. It must consider five aspects as will be demonstrated on the basis of Example 4.

Example 4:
Consider the event instance sequence $EIS^4 := ei_1^1\ ei_1^2\ ei_3^1\ ei_2^1\ ei_3^2$ and the event types $E_5 := (E_1, E_2)$, $E_7 := (E_1, E_2, E_3)$, and $E_{10} := (E_3, E_5)$.

---

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

[4] Note, that our meta-model considers simultaneously occurring events. Thus, there can be several initiators and terminators of a complex event (see [Zimm98] for further details).

[5] An efficient event detection algorithm would realize that the remaining event instances can never be part of any new instance of type E_7 and would remove them from the type specific EIS of E_7.

First it has to be defined what component event types make up a complex parent event type and whether there is any (partial) order imposed on them (*type and order aspect*). For example, for an event of E_7 to be detected instances of E_1, E_2 and E_3 have to occur in exactly this order.
On the basis of EIS[4] an event instance ei_7^1 will be generated as soon as ei_3^2 occurs. However, the initial time interval that is spanned by ei_7^1 is limited by ei_1^1 and ei_3^2. Since this time interval comprises several instances of event type E_1 (ei_1^1 and ei_1^2) as well as E_3 (ei_3^1 and ei_3^2) it has to be clarified as to how many of such instances are necessary at least and at most and which ones (*repetition aspect*).
The *continuity* aspect deals with the question of whether the detection process of an instance sequence of a given event type can (*loosely coupled*) or cannot (*tightly coupled*) be interrupted by the occurrence of non-relevant event instances. *Non-relevant* instances are of a type that is not expected at a given stage of a detection process. Is it, e.g., permitted that during the detection of an event of type E_7 an instance of type E_3 can occur between ei_1^1 (or ei_1^2) and ei_2^1 in EIS[4]?
Complex parent event types may contain complex component event types. This can lead to situations where the time intervals of component event instances do overlap. During the detection of the event e_5^1 (initiated by ei_1^1 or ei_1^2 and terminated by ei_2^1) the event e_3^1 occurs. Now, the event occurrence time of the instance ei_5^1 corresponds to that of ei_2^1. Thus, the event occurrence time of ei_3^1 is older than that of ei_5^1, but younger than that of the initiator of ei_5^1 (ei_1^1 or ei_1^2). Type E_{10} must specify whether it permits such concurrency (*concurrency aspect*).
Complex events consist of a number of component events. Up to now it was simply assumed that these component events are independent of each other. This assumption, however, will not always meet the intentions of a real world application. They, e.g., may want the component events to be triggered by the same transaction and/or operate on the same data. Thus, an event type has to lay down whether it requires such context conformity (*context condition aspect*).

Type and order
The skeleton of a complex event type is formed by the underlying component event types and information about the kind of their appearance (e.g., in what order). The latter usually is defined by event operators. The meta-model provides the following basic set of operators with the arity *n*:

- *sequence operator* (;): the specified instances have to occur in the order determined by this operator,
- *simultaneous operator* (==): the specified instances have to occur simultaneously; i.e., their event occurrence time must be identical,
- *conjunction operator* ($\wedge$): the specified instances have to occur; this, however, can happen in any order,
- *disjunction operator* ($\vee$): at least one of the specified instances has to occur; there are no restrictions on the order,
- *negation operator* ($\neg$): the specified instance(s) are not allowed to occur in a given interval.
  The $\neg$-operator only makes sense if applied to an interval. It is formed by the first and last parameter of the $\neg$-operator. All interior parameters represent the event types whose non-occurrence within the interval has to be guaranteed.

Repetition
The number of event instances of a component event type, that have to occur, to satisfy the parent event type pattern can be specified by a *delimiter* that is written ahead of the event type. The delimiter can be an *exact number* (with 0 requiring the non-occurrence of event instances) or a *closed* (upper bound is specified, if lower bound is not specified it is assumed to be 0) or *open* interval (only lower bound is specified). If no delimiter is specified at least one instance of the given type must occur.

Example 5:
Consider the event type E_{11}:=;((-2)E_1, (2-)E_2, (0)E_3, E_4), which defines, that first up to two instances of E_1 have to occur followed by at least two instances of E_2 followed by one instance of E_4. Moreover, no instance of E_3 is permitted to occur after the second instance of E_2 and the instance of E_4. Thus, $EIS^5 := ei_1^1\ ei_2^1\ ei_1^2\ ei_2^2\ ei_3^1\ ei_2^3\ ei_4^1$ (last five instances), $EIS^{5'} := ei_1^1\ ei_2^1\ ei_2^2\ ei_4^1$ and $EIS^{5''} := ei_1^1\ ei_1^2\ ei_2^1\ ei_2^2\ ei_1^3\ ei_4^1$ trigger an event of E_{11}, while $EIS^{5'''} := ei_1^1\ ei_2^1\ ei_1^2\ ei_2^2\ ei_2^3\ ei_3^1\ ei_4^1$ and $EIS^{5''''} := ei_1^1\ ei_1^2\ ei_2^1\ ei_1^3\ ei_4^1$ do not.

Note, that the $\neg$-operator can be simulated by the ;-operator with delimiter 0, assigned to its 'interior' parameter(s)[6].

Continuity and concurrency
The continuity and the concurrency aspects can each be specified by a *struc*ture-*mode*. The *c*ontinuity-*mode* defines whether the sequence of event instances that reflects the instance specific EIS on the level of the type specific EIS can be interrupted by event instances not relevant to the event detection (*non-continuous*); i.e. not relevant for the instance specific EIS, or not

[6] For this reason the negation operator does not belong to the meta-model. In this section it is just listed for reasons of simplicity.

(*continuous*)[7]. The *c*oncurrency-*mode* lays down whether the time intervals associated with component event instances may (*overlap*) or not (*no-overlap*). The underlined option does always represent the *default mode* that is applied if no mode is specified.

Context condition

The context condition aspect of a complex event type specifies restrictions on the context in which the events of its component types have to occur. The context is represented by the values of the parameters of its event instance. We distinguish between *absolute* and *relative* context conditions which require parameters to correspond to concrete values, respectively define whether the values of a parameters of different instances must be the same (*same*), must differ (*different*), or either (*any*). Context restrictions that are defined for a complex event type must be fulfilled by its component event types as well. However, if a component event instance is of a type that does not include information relevant for the context restriction it automatically fulfills the restriction.

Our meta-model considers three context types that can be addressed individually. The *env*ironmental *context* allows to specify whether the appropriate event instances must be triggered by the same transaction (*ta*), operating system process (*proc*), application (*appl*), or user (*user*). Therefore, an event of type $E_{7'}$:=(*same ta*) ;(E_1, E_2, E_3) is triggered whenever an event of the type ;(E_1, E_2, E_3) occurs, provided that its component events were triggered by the same transaction. Since a transaction is the most specific environment while user (group) is the most general, component events must be triggered by the same application and by the same user if they need to be triggered by the same transaction.

The second context is the *da*ta *context* which allows to specify whether the appropriate event instances do rely on the same data context. Data context can either mean the same data type, data instance or even data state (again the features are more specific from left to right). Consider the tightened event type $E_{7''}$:=(*same ta*) (*same data-id*) ;(E_1, E_2, E_3) with E_1, E_2, E_3 representing database operations. An event of $E_{7''}$ is only triggered if all relevant component event instances were caused by the same transaction with the underlying operations being executed on the same data.

The *op*eration *context* as the third context defines whether the appropriate event instances have to be caused by the execution of the same operation.

3.2.2 Event instance selection

Every time a complex event is triggered an event instance selection process has to collect the correct event instances from the then valid state of the type specific event instance sequence *EIS*(E_i) to form the instance specific EIS (*EIS*(ei_i^s)). Instance selection specifies for each component event type E_{ij} individually what instances of this type are to be chosen. While the instance pattern lays down when an event is triggered, the instance selection specifies which instances (i.e. what information) is kept for the subsequent execution of other parts (condition and action) of the underlying ECA-rule. Therefore, both dimensions are to an extent independent of each other. While at least the smallest set of event instances that fulfils all requirements of the instance pattern is to be selected it can as well be more.

First (*last*) are minimum-oriented selection strategies. They only select the absolutely necessary number of instances. *First* (*last*) picks from the set of permissible instances the ones with the oldest (youngest) timestamps. *Cumul*ative chooses (only) those instances of E_{ij} that do not contradict the order laid down by the instance pattern of E_i. A truly cumulation-oriented selection strategy is *ext-cumul* which selects all permissible instances of E_{ij}.

Example 6 :

Consider the event instance sequence EIS^6:= ei_1^1 ei_2^1 ei_1^2 ei_2^2 ei_2^3 ei_1^3 ei_3^1 with an event of type E_7:=;(E_1, (2)E_2, E_3) just being triggered by ei_3^1. $E_{7'}$:=;(*first*:E_1, *last*:(2)E_2, E_3) would result in the instance specific EIS (ei_1^1 ei_2^2 ei_2^3 ei_3^1), $E_{7''}$:=;(*last*:E_1, *first*:(2)E_2, E_3) in (ei_1^2 ei_2^2 ei_2^3 ei_3^1), and $E_{7'''}$:= ;(*ext-cumul*:E_1, *first*:(2)E_2, E_3) in (ei_1^1 ei_2^1 ei_1^2 ei_2^2 ei_1^3 ei_3^1).

Especially the instance sequence for $E_{7''}$ may look strange on a first glance since it is specified in $E_{7''}$ that the *last* instance of type E_1 is to be chosen from EIS^6 and that is ei_1^3. However, this would not lead to a legal solution since the time interval specified by the initiator ei_1^3 and the terminator ei_3^1 does not include an instance of E_2. *First* (*last*) need to be interpreted as *first* (*last*) permissible instance of the given type. Therefore, we have to backtrack. The next possibility is 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, but is the first in the interval spanned by ei_1^2 and ei_3^1. To find this solution we have implied that the underlying event pattern is to be traversed from left to right, i.e. first the correct instances for E_1 have to be identified, then the instances of E_2 and finally the instance of E_3 (which is, of course, trivial). If we traversed the event pattern in the opposite direction, we would first have to select ei_3^1, then ei_2^1 and ei_2^2, since they are the *first*

---

[7] Note, that generally the mode *continuous* cannot be modeled by the introduction of additional component event types with a delimiter 0. Consider, e.g., the event types E_7:=;(E_1, E_2, E_3), $E_{7'}$:=;(E_1, (0)E_3, E_2, (0)E_1, E_3) and $E_{7''}$:=(*continuous*) ;(E_1, E_2, E_3) and the event instance sequence EIS^9:= ei_1^1 ei_2^1 ei_1^2 ei_3^1 ei_2^2 ei_3^2. EIS^9 triggers events of the types E_7 and $E_{7'}$ but not of $E_{7''}$.

occurrences of events of E_2. Although ei_1^1 is the first occurrence of type E_1 it is also the *last* occurrence of this type in the given situation and, therefore, has to be chosen as representative of type E_1.
Thus, the result of the event instance selection depends on the order of the traversal. Therefore, to obtain a unique solution one has to define the order in which component event instances must be collected from the underlying EIS. This can be specified by the operator mode *tr-mode* whose value can be either *left-to-right* or *right-to-left*.

3.2.3 Event instance consumption[8]

Event instance consumption defines the impact of the occurrence of an event of type E_i on its type specific EIS. While component event instances that can be *shared* by events of E_i will survive, all component instances that can only be used *exclusively* by one event need to be removed from EIS(E_i). As a consequence of such removals the type specific EIS may now contain further event instances that can no longer be used for the detection of future events of E_i. This instances should be removed from EIS(E_i) as well in order to keep it as compact as possible.

Example 7:
Consider event type E_5:=;(*last*:E_1, *first*:E_2) and its event instance sequence *EIS*(E_5):= $ei_1^1\ ei_1^2\ ei_2^1\ ei_2^2$. With the occurrence of ei_2^1, an instance ei_5^1 is triggered for which the appropriate event instance sequence *EIS*(ei_5^1) has to be identified. In the given scenario ei_5^1 will consist of the instances ei_1^2 and ei_2^1. Now, let us assume that an event instance ei_2^s is consumed if it is used in an event instance sequence ei_5^s of its parent type. The occurrence of ei_5^1 may cause the deletion of (1) ei_1^2, (2) of ei_1^1 and ei_1^2, or (3) of no instance of E_1. Dependent on this result, ei_2^2 will (1, 3) or will not (2) trigger another event instance of E_5. If it is triggered, it is either caused by the event instance sequence ($ei_1^2\ ei_2^2$) (3) or ($ei_1^1\ ei_2^2$) (1). If both, ei_1^1 and ei_1^2, are consumed the remaining type specific EIS ($ei_2^1\ ei_2^2$) does only contain instances that cannot be used for the triggering of further events of E_5, since such an event has to start with an instance of type E_1. So, they should be removed from EIS(E_5) as well.

We distinguish between three different *consum*ption modes that can be specified individually for each component event type. The *shared* mode does not delete any instance of E_{ij} (3). The *exclusive* mode removes all instances of E_{ij} from *EIS*(E_i) that belong to the event specific EIS of the given event (1). The *ext-exclusive* mode deletes all instances of E_{ij} from *EIS*(E_i) that occur before the terminator of the given event (2). If E_i contains the same component event type E_{ij} several times, the strongest consumption mode will dominate the weaker ones.

3.3 Event groups

A terminator of an event type E_i, whose consumption mode is *shared*, can easily trigger a larger number of events of E_i (see Example 8). The set of events instances triggered by this terminator forms its *event group*. In this section we will discuss, which event groups can be constructed on the basis of our meta-model.

3.3.1 Event instance selection

To avoid infinite looping (e.g., by infinitely constructing parent event instances from the same component instances) different event instances of the same type must differ in that they must contain at least one different component instance.

Example 8:
Consider the event instance sequence EIS[7]:= $ei_1^1\ ei_1^2\ ei_1^3\ \ ei_2^1\ ei_1^4\ ei_2^2\ ei_3^1$ and the following variants of event type E_8:
$E_{8'}$:=;(*first:shared*:E_1, *first:exclusive:*E_2, *shared*:E_3)
$E_{8''}$:=;(*last:shared*:E_1, *first:exclusive:*E_2, *shared*:E_3)
$E_{8'''}$:=;(*cumul:shared*:E_1, *first:exclusive:*E_2, *shared*:E_3)
As is shown in Figure 1 the event groups of the terminator ei_3^1 differ, depending on the modes assigned to E_1 and E_2.

This example shows that it is already possible to produce all kinds of specific event groups. However, as it turned out, there is no easy way to generate, e.g., a complete set of all those combinations that can be achieved by exactly considering one instance of each component event type (see Figure 1, $E_{8'''}$). To support such semantics, the event instance selection modes *comb*inations and *comb*inations *min*imum were introduced. If one of these modes is used for a component type E_{ij} the different permissible instance sets of E_{ij} are alternately combined with the event instance sets of the other component event types to form the respective event group. While *comb min* ensures that only the minimum number of component event instances required by the delimiter of E_{ij} are considered (as with $E_{8'''}$), *comb* generates all permissible sets of event instances.
Note that these modes only make sense if the consumption mode of the terminator of the underlying event type E_i is *shared.*

[8] Similar terms like *event consumption* or *consumption mode* are used in other work as well (cf., e.g., [BBKZ93], [Daya95], [DiDG95], [Gatz94]), however, often in a different sense which is more closely to aspects of our dimension *event instance selection*.

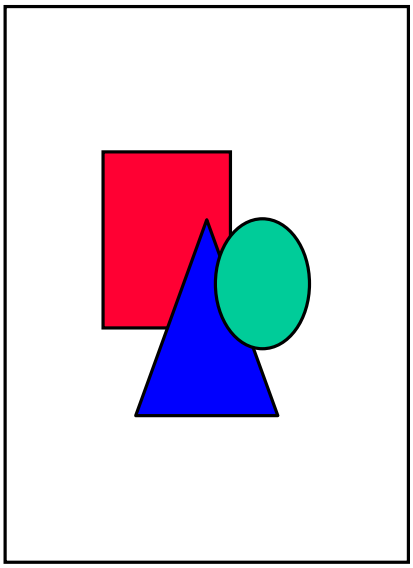

Figure 1: *Event instance selection for event groups*

Example 9:
The event type $E_{8''}$ (see Figure 1) can be defined as: ;(*comb min:shared:*E_1, *comb min:shared:*E_2, *shared:*E_3)

3.3.2 Event instance consumption

Let us come back to event type $E_{8'}$ (see Example 8). EIS^8:= ei_1^1 ei_1^2 ei_2^1 ei_1^3 ei_2^2 ei_3^1 ei_1^4 ei_2^3 ei_3^2 triggers three instances of it (see Figure 2), each of which contains ei_1^1 as a component event. If we want ei_1^1 only to be shared within the event group of $ei_{8'}^1$, apart from that, however, to be consumed by the event group (which would prevent instance sequence 3. of $E_{8'}$ in Figure 2) this cannot be specified.
The event instance consumption modes in its current form do not offer the possibility of distinguishing between the availability of event instances inside and outside of a group. Thus event instances that are shared by instances of one group cannot be protected from being shared by other groups as well.
To cope with such demands, the meta-model provides the domains *in*side and *out*side of an event group. They can be used in conjunction with the consumption mode and define the availability of event instances within a group and across all groups. The event instance consumption for the *out*side domain is applied to the union of the event instances belonging to the same event group.

Example 10:
Consider the event types E_9:=;(*first:in shared:out exclusive*:E_1, *first:exclusive*:E_2, *shared*:E_3) and $E_{9'}$:= ;(*first:in shared:out ext-exclusive*:E_1, *first:exclusive*:E_2, *shared*:E_3) and the event type $E_{8'}$ defined in Example 8. These types only differ in the consumption mode of the component event type E_1. From each type the instance sequence EIS^8 (see above) triggers three instances, where the first two instances of each type are the same (see Figure 2).

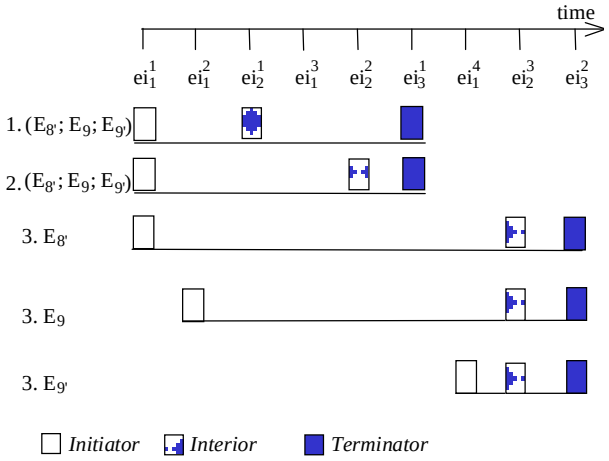

Figure 2: *Event instance consumption for event groups*

Sketch of the underlying event detection algorithm

The following is a sketch of an event detection algorithm for events of type E_i. It is executed each time a new event instance ei_n is inserted into *EIS*(E_i).

```
SketchEventDetectionAlg (ei_n:
                    newEventInstance; EIS^T:
                    currentTypeSpecificEIS);
① type(ei_n)=terminatorType;
IF TRUE
BEGIN
② instancePatternE_i(EIS^T);
   IF TRUE
   BEGIN
     ③ ei:=newInstanceE_i;
        ei.Typ:=E_i;
          ...
     ④ ei.EIS:=E_i.instanceSelection(EIS^T);
        ei.Context:= ... ;
          ...
     ⑤ EIS^T:=instanceConsumptionE_i(EIS^T);
          ...
     ⑥ RETURN(ei, EIS^T);
   END
END
```

In the first step (①), it is determined whether the new event instance is of the terminator type of E_i. Only in that case a new event of E_i may have occurred. Next, it is checked whether instances of the current type specific EIS of E_i (EIS^T) fulfil the event type pattern of E_i (②). If that is the case, a new event instance *ei* is generated and the relevant event specific parameters are set (③). Next, those event instances are selected from the current type specific EIS that are supposed to constitute the instance specific EIS of ei (④). Among others, it is now also possible to save the event specific context of ei. In the fifth step the event

instances that are to be consumed by *ei* are deleted from the event instance sequence *EIS*(E_i) (⑤). Moreover, now irrelevant other event instances may be removed from EIS^T as well. Finally, the complete description of the new event as well as the modified type specific EIS are returned (⑥).

4 Semantics of Existing Event Algebras

In the next sections we will summarize the essential features of existing event algebras[9] and transfer them into meta-model based expressions. The resulting specifications provide a clear and uniform presentation of the semantics of the underlying event algebras and, as such, may reveal design flaws. Some of them will be discussed briefly (for further examples see [Zimm98]).

This and the next chapter is intended to prove the applicability and powerfulness[10] of our approach. It is not our intention to criticize any of the work in the active DBMS area since it usually is on a very high level. We really respect and appreciate the results gained. Without them, active DBMS would still just be a dream.

We tried our best to treat each of this proposals as fair as possible. However, since our work had to rely on published papers there is a risk that some of our conclusions are debatable. Moreover, since we are dealing with ongoing research event algebras may have changed in the meantime.

Good language design

In order to discuss design flaws of languages we first have to clarify what demands are to be covered by a good language design. Following Edgar Codd it requires, among others, the language to provide a small set of language constructs (minimality) and to fulfill the requirements of symmetry and orthogonality (cf. [Codd71]).

Minimality is fulfilled if the same meaning cannot be expressed by different language constructs.

Symmetry means that the same language construct is assumed to express always the same semantics regardless of the context it is used in. Moreover, language constructs with similar meanings are assumed to always rely on the same syntax.

Orthogonality is given if the language consists of a small set of orthogonal language constructs which permits every meaningful combination of language constructs to be applicable.

[9] The event algebra of REACH ([BBKZ93]) heavily relies on Snoop. Therefore, we will not consider it in this paper.

[10] We did not intend to develop a meta-model that provides the unification of all concepts that were introduced yet in the active DBMS field. That inevitably would have led to a confusing monster that is no longer applicable. So, we did concentrate on a sufficient set of language constructs from which we assume that they will meet the requirements of general applications.

General structure of event expressions

On an abstract level a complex event type E_C can be seen as a sequence of algebra operators together with their parameters which, in effect, represent (complex) *com*ponent *e*vent *t*ypes (CPET). Therefore, E_C can be depicted by an operator tree with the inner nodes representing algebra operators (which, in turn, compute the complex component event for the next higher level event operator) and the leaves representing the primitive event types (see Figure 3). This view helps to restrict complexity since it allows us to express every complex event (on the highest level) by exactly one algebra operator together with its parameters (see Figure 3a). By replacing complex parameters by their algebra expressions (operator trees) we can gradually unfold the tree (see Figure 3d).

Example 11:

Let us consider the complex event type E_{11}:=;(E_1, E_2, E_3, E_4) (see Figure 3d). It can as well be specified as E_{11}:=;(E_7, E_4) (see Figure 3a) with E_7:=;(E_5, E_3) (see Figure 3b) and E_5:=;(E_1, E_2) (see Figure 3c).

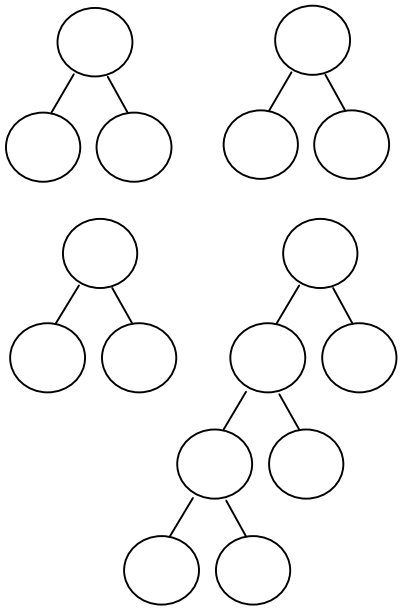

Figure 3: *Dissection of a complex event*

On a more specific level an event algebra expression is described by an algebra operator (2), possible restrictions that are to be obeyed by the given event instance sequence (1), the parameters (4), and specific information about (the treatment of) the parameters (3) (see Figure 4). While (1) can and (2) has to occur exactly once, the number of appearances of (3) and (4) reflect the number of parameters of the algebra operator.

$E_{5'}$:=(*continuos*) (*right-to-left*) ; (*first:shared*:E_1, *shared*:E_2)
1 2 3 4 3 4

Figure 4: *General structure of an event expression*

For better readability we will present the specification of the algebra operators of the different event algebras in tables whose structure reflects the general structure of a complex event as stated in Figure 4.
Each operator of the underlying event algebra is represented by an entry in the table that consists of three columns. The first column consists of two entries, separated by a dotted line. The upper entry contains the operator and its parameters in the notation of the given event algebra. The second entry presents the skeleton for the same event type, however, specified on the basis of our meta-model ((1) and (2) of Figure 4). The second column lists the parameters (*com*ponent *e*vent *t*ypes (CPET for short), (4)) for which the subsequent column may contain more specific information about/restrictions on the *com*ponent *e*vent *t*ypes (CPET modes, (3)). Since an operator may have several parameters there will an individual row for each parameter (rows are separated by a dashed line).
In the tables for Snoop, ACOOD and ADL the third column is subdivided into sub-columns each of which represents one parameter context of the specific event algebra.
Whenever for all event types the semantics of the dimension context condition is the same but different from the default mode it is specified in the head of the table. Whenever, from our point of view, the semantics of an operator is ambiguous or irregular we mark it by adding a question mark (?) and using bold letters for the mode that causes the irregularity.

Example 12:

Consider the Snoop operator ($E_1 \nabla E_2$) in its parameter context *recent* (see first row of Figure 6, columns 1 and 2 and first sub-column of column 3). The table is to be read as follows:

- first column, first row: ($E_1 \nabla E_2$) is the notation of Snoop
- first column, second row: $\vee(E_1, E_2)$ is the skeleton version of the meta-model based notation ((1) and (2))
- second column: there is an entry for each parameter (E_1 and E_2) of the underlying operator (∇, respectively $\vee$) (3).
- third column: E_1 (E_2) in the skeleton notation of the meta-model must be enriched by the additional specifications cited in this row (*exclusive:* in both cases).

Altogether this results in the following meta-model expression, that is equivalent in its semantics to the original notation of Snoop:

$\wedge$(*first: exclusive:* E_1, *first: exclusive:* E_2).

4.1 HiPAC

HiPAC ([DaMB88], [MCDA89], [DaBC96]) is an active object-oriented database management system that was developed at the Computer Corporation of America and later at the Xerox Advanced Information Technology Laboratory. It provided the basis for the two follow-on projects Sentinel and REACH.

Event type pattern
HiPAC provides the standard event operators disjunction ($\vee$) and sequence (;). In addition it introduces the unary operator $*$. Events of type $*E_i$ occur whenever one or more events of E_i were triggered by the same transaction. Thus, for a given transaction an event of $*E_i$ can occur at most once. All instances of E_i that occurred in the same transaction are accumulated to represent the instance of $*E_i$.

Event instance selection and consumption
The semantics of complex events is defined informally. Many aspects remain unclear. For example, it is not mentioned how their approach deals with event instance selection and consumption. With respect to the context HiPAC requires all component events of a complex event to be triggered by the same transaction[11].

Rules and transactionsOne of the main purposes of the HiPAC project was to study the influences of rule execution on transaction management. In order to ensure the integrity and consistency of the database every operation on it has to be performed within a transaction. Since the condition and/or action parts of a rule usually read or modify data of the underlying database the relationship between rule execution and transaction management has to be clarified.
Dayal, Hsu, and Ladin ([DaHL91]) introduce a generalized transaction model for active database management systems that is based on different types of *nested* transactions, namely concurrent subtransactions, deferred subtransactions, and decoupled transactions. *Concurrent subtransactions* are subtransactions of a nested transaction in the original sense (cf., e.g. [Moss85], [UnSc92]). If the execution of a subtransaction is explicitly delayed until the parent transaction has finished its task the subtransaction is called *deferred*. A *decoupled transaction* is executed and treated independently from the transaction it has descended from.
If the coupling mode of a rule is *immediate*, then the condition (or action in case of an EA-rule) part of the rule is evaluated as soon as the event is detected. In the *deferred* mode the condition/action is evaluated within the shelter of the responsible transaction,

[11] It is mentioned that complex events can be caused by component events triggered by different transactions belonging to the same application. But this situation is not discussed in further detail.

however, only after its last non-rule related operation was executed. If the mode is *decoupled*, then all operations are evaluated in a separate transaction.
Unfortunately, the integration of rules and transactions causes some tricky, sometimes even nasty problems, especially with respect to the deferred and decoupled mode (cf., [StRH90], [StHP89]). Since this discussion is beyond the scope of this paper we will not discuss this problems in further detail.

Meta-model based representation
Figure 5 presents the specification of HiPAC on the basis of our meta-model. As HiPAC belongs to the early work in the area of active database management systems it did concentrate on the basics; i.e., its event specification is far away from being as sophisticated and extensive as that of subsequent approaches.

HiPAC Meta-Model	*CP ET*	*CPET modes*
$E_1 \vee E_2$	E_1	probably none
$? \vee(E_1, E_2)$	E_2	probably none
$E_1 ; E_2$	E_1	probably none
$? ;(E_1, E_2)$	E_2	probably none
$*E_1 ; E_2$	E_1	*cumul: (? exclusive:)*
$? ;(E_1, E_2)$	E_2	*exclusive:*

Figure 5: *The semantics of event expressions in HiPAC*

4.2 Snoop

Snoop has been developed at University of Florida with its concepts being implemented in a prototype called Sentinel ([CKAK94], [Kris94]). Most parts of Snoop are defined formally. However, as is shown in [Zim96], the exceptions mean that its semantics is sometimes ambiguous.
(Complex) events are assumed to be strictly ordered, which means that they cannot occur simultaneously. In principle, Snoop allows the occurrence time of a complex event instance to be arbitrarily chosen from the set of occurrence times of its component event instances[12]. However, if the occurrence time of a complex event is allowed to be older than the occurrence time of its terminator, problems may arise. Consider the event type $E_4:=;(E_1, \neg E_2, E_3)$, with E_2 being a complex component event type, and the event instance sequence $EIS(E_4):= ei_1^1\ ei_3^1$. Let us assume that an instance of E_2 was initiated before ei_3^1 occurred. According to the above rule it can get an older occurrence time than that of ei_3^1. Therefore, it cannot be decided whether an event of E_4 occurred. To avoid such problems we will always assign the occurrence time of the terminator to the complex event.

[12] This may lead to problems as mentioned in footnote .

As HiPAC, Snoop requires all component events of a complex event to be triggered by the same transaction.

Event type pattern
Snoop provides the standard event operators conjunction (Δ), disjunction (∇), sequence (;), and negation (**NOT**), however, enriches this set by the additional operators **ANY**, A, P, A$*$ and P$*$. Events based on the **ANY** operator, denoted as **ANY**(m, E_1, E_2, .., E_n), with $m \le n$, occur whenever events from m out of the n distinct event types were triggered[13]. Events based on the a-periodic operator A, denoted as A(E_1, E_2, E_3), are triggered whenever an a-periodic event of E_2 occurs within the closed time interval spanned by the appropriate events of E_1 and E_3[14]. 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) regardless of how many component (and 'terminator') events did occur within the time interval.

Snoop is based on the formal semantics of *event expressions*. Each primitive event type E_i is an event expression. Event expressions combined by an arbitrary event operator form (a more complex) event expression. If E_i is an event expression, than (E_i) is an event expression, too.
Semantic is added to an *event expression E* by regarding it as a function from the underlying time domain onto Boolean. For a given point in time *t* E computes to *true*, if an event of type E occurred at *t*, and to *false* otherwise. $\sim E$ denotes the negation of the Boolean function E. It expresses the non-occurrence of an event at a given point in time.

The event operators are defined as follows:

1. $(E_1 \nabla E_2)(t):=E_1(t) \vee E_2(t)$
2. $(E_1 \Delta E_2)(t):=((\exists t_1)\ (E_1(t_1) \wedge E_2(t)) \vee (E_2(t_1) \wedge E_1(t)) \wedge t_1 \le t)$
3. **ANY**$(m, E_1, E_2, ..., E_n)(t):=(\exists t_1)(\exists t_2)\ ...\ (\exists t_{m-1})\ (E_i(t_1) \wedge E_j(t_2) \wedge ... \wedge E_k(t_{m-1}) \wedge E_p(t) \wedge (t_1 \le t_2\ ...\ \le t_{m-1} \le t) \wedge (1 \le i, j, ..., k, p \le n) \wedge (i \ne j \ne ... \ne k \ne p)$, with $m \le n$
4. $(E_1; E_2)(t):=((\exists t_1)\ (E_1(t_1) \wedge E_2(t)) \wedge t_1 \le t)$

[13] The semantics of the **ANY**-operator can be specified by the conjunction and disjunction operator. For example, the semantics of the type **ANY**(2, E_1, E_2, E_3) is equivalent to the semantics of the type $((E_1\ \Delta\ E_2)\ \nabla\ (E_1\ \Delta\ E_3)\ \nabla\ (E_2\ \Delta\ E_3))$. Therefore, we will not consider the **ANY**-operator in the specification based on our meta-model.

[14] As can be seen from the formal definition below an event of this type is supposed to appear as well if only instances of the types limiting the time interval do occur (and no instance of E_2). This is a contradiction between the formal and informal introduction of this operator in the papers. Our further treatment of this operator will rely on the formal introduction of this operator.

5. $A(E_1, E_2, E_3)(t):=(\exists t_1)\ (\forall t_2)\ (E_1(t_1) \wedge E_2(t)) \wedge (t_1 \leq t) \wedge ((t_1 \leq t_2 < t) \Rightarrow \sim E_3(t_2))$
6. $A*(E_1, E_2, E_3)\ (t):=((\exists\ t_1)\ (E_1(t_1) \wedge E_3(t)) \wedge t_1 \leq t)$
   Note, that this definition corresponds to $(E_1; E_3)$ (t), i.e., an event of the type that is based on $A*(E_1, E_2, E_3)$ can occur, even if an a-periodic event of E_2 does not occur[15].
7. $P(E_1, TI[:parameters], E_3)(t):=(\exists t_1)\ (\forall t_2)\ (E_1(t_1) \wedge (t_1 \leq t_2 < t) \Rightarrow \sim E_3(t_2)) \wedge (t:=t_1+i*TI$, for some positive integer i). TI[:parameters] is a time interval with an optional parameter list.
8. $P*(E_1, TI[:parameters], E_3)(t):=(\exists t_1)\ (E_1(t_1) \wedge E_3(t)) \wedge (t_1+TI \leq t)$
   Events of type E_1 and E_3 that occur within the time interval are accumulated.
9. $(\neg E_2)\ (E_1, E_3)(t):=(\exists t_1)\ (\forall t_2)\ E_1(t_1) \wedge \sim E_2(t) \wedge E_3(t) \wedge ((t_1 \leq t_2 < t) \Rightarrow \sim(E_2(t_2) \vee E_3(t_2)))$[16]

Event detection and resolution is done bottom-up. Starting point is always the given *system specific* sequence of *primitive* event instances, called *global event instance history H* ($H:=\{\{ ei_j^i \} \mid \forall$ i, j: ei_j^i is the *i*-th occurrence of a primitive event type $E_j\}$). In a first step, for each primitive event type E_P that occurs in H its *primitive event history* $E_P[H]$ is derived from H. The result is the set of all event instances (each of which represented as a one element set) of this type ordered in the order of their occurrence (time) (see Example 14, 1.). From these sets of primitive event histories, step by step, *composite event histories* are derived by first constructing the composite event histories that exclusively rely on primitive event histories (see Example 14, 2a.), than those, that rely on primitive event histories and the composite ones that were constructed in previous steps, and so on (see Example 14, 2b.). To construct composite histories the binary operator $\oplus$ is used. It computes the cross product of two input sets (whose elements are sets) with the resulting set being the output (see Example 13). In general, the resulting set of complex event sequences forms a superset of those sequences that can be derived from the global event instance history H. For example, in Example 14a. the last set and in Example 14b. the last four sets cannot be derived from H.

Example 13:

Consider the histories $E_1[H]=\{\{ ei_1^1, ei_3^1 \}, \{ ei_1^2, ei_3^1 \}\}$ and $E_2[H]=\{\{ ei_2^1 \}, \{ ei_2^2 \}\}$. With these histories as input parameters the operator $\oplus$ produces the result $E_1[H] \oplus E_2[H]=\{\{ ei_1^1, ei_3^1, ei_2^1 \}, \{ ei_1^1, ei_3^1, ei_2^2 \}, \{ ei_1^2, ei_3^1, ei_2^1 \}, \{ ei_1^2, ei_3^1, ei_2^2 \}\}$.

Example 14:

Consider the complex event type $E_C:=(E_1 \Delta E_2)\ ; E_3$ and the global event instance history $H=\{\{ ei_1^1 \}, \{ ei_2^1 \}, \{ ei_1^2 \}, \{ ei_3^1 \}, \{ ei_2^2 \}, \{ ei_3^2 \}\}$.

1. Deduction of the primitive event instance histories:
   a. $E_1[H]=\{\{ ei_1^1 \}, \{ ei_1^2 \}\}$,
   b. $E_2[H]=\{\{ ei_2^1 \}, \{ ei_2^2 \}\}$, and
   c. $E_3[H]=\{\{ ei_3^1 \}, \{ ei_3^2 \}\}$.
2. (Recursive) construction of the composite event instance histories:
   a. $(E_1 \Delta E_2)[H]=\{\{ ei_1^1, ei_2^1 \}, \{ ei_1^1, ei_2^2 \}, \{ ei_1^2, ei_2^2 \}, \{ ei_1^2, ei_2^1 \}\}$, and
   b. $E_C[H]=\{\{ ei_1^1, ei_2^1, ei_3^1 \}, \{ ei_1^1, ei_2^1, ei_3^2 \}, \{ ei_1^1, ei_2^2, ei_3^2 \}, \{ ei_1^2, ei_2^2, ei_3^2 \}, \{ ei_1^1, ei_2^2, ei_3^1 \}, \{ ei_1^2, ei_2^1, ei_3^1 \}, \{ ei_1^2, ei_2^2, ei_3^1 \}, \{ ei_1^2, ei_2^1, ei_3^2 \}\}$.

Event instance selection and consumption

Event instance selection and consumption are regarded as one dimension only in Snoop. Its value is regulated by the so-called *parameter context.* It determines the set of component event instances that are bound to and consumed by a parent event sequence. Snoop provides five possible modes for the parameter context from which only the first one is defined formally ([Mish91], [Anwa92]).

1. The *unrestricted* parameter context is defined by the composite event history of the underlying complex event type. Each of the permissible instance sequences of the event history triggers (and represents) an event of this type. For example, in Example 14b. the first four event sequences would trigger an event of E_C.
   We do not want to list the complete formal definition of the unrestricted parameter context. Instead we will just present the definitions of the four most relevant event operators:
   - $(E_1 \nabla E_2)[H]:=\{ei \mid ei \in E_1[H] \cup E_2[H]\}$
   - $(E_1 \Delta E_2)[H]:=\{\{ei^i, ei^j\} \mid \{ei^i, ei^j\} \in (E_1[H] \oplus E_2[H]) \cup (E_2[H] \oplus E_1[H])$, where ei^i occurs before $ei^j\}$
   - $(E_1\ ;\ E_2)[H]:=\{\{ei^i, ei^j\} \mid \{ei^i, ei^j\} \in (E_1[H] \oplus E_2[H])$, where ei^i occurs before $ei^j\}$
   - $A(E_1, E_2, E_3)[H]:=\{\{ei^i, ei^j\} \mid \{ei^i, ei^j, ei^k\} \in (E_1[H] \oplus E_2[H] \oplus E_3[H])$, where ei^i occurs before $ei^j\}$

[15] The specification of the event detection algorithm for the operator A* (cf. [CKAK93], [Kris94]) corresponds to this semantics. Thus, this semantics seams to be intended.

[16] Note, that Snoop assumes that events do not occur simultaneously. Thus, the requirement of the non-occurrence of an event of type E_2 at time *t* is not necessary, as it is already required, that an event of type E_3 occurs at this time.

Snoop	*CP*	*CPET modes*			
Meta-Model	*ET*	*recent*	*chronicle*	*continuous*	*cumulative*
$E_1 \vee E_2$	E_1	*exclusive:*	*exclusive:*	*exclusive:*	*exclusive:*
$\vee(E_1, E_2)$	E_2	*exclusive:*	*exclusive:*	*exclusive:*	*exclusive:*
$E_1 \Delta E_2$	E_1	*last:* ***shared:***	*first: exclusive:*	*comb min: shared in: exclusive out:*	*ext-cumul: exclusive:*
$\wedge(E_1, E_2)$	E_2	*last:* ***shared:***	*first: exclusive:*	*comb min: shared in: exclusive out:*	*ext-cumul: exclusive:*
$E_1 ; E_2$	E_1	*last:* ***shared:***	*first: exclusive:*	*comb min: exclusive:*	*ext-cumul: exclusive:*
$;(E_1, E_2)$	E_2	*exclusive:*	*exclusive:*	*shared in:, exclusive out:*	*exclusive:*
$\neg(E_2)(E_1, E_3)$	E_1	*last:* ***ext-exclusive:***	*first: exclusive:*	*comb min: exclusive:*	*(?* ***first:)*** ***ext-exclusive:***
$\neg(E_1, E_2, E_3)$	E_3	*exclusive:*	*exclusive:*	*shared in:, exclusive out:*	*exclusive:*
$A(E_1, E_2, E_3)$	E_1	*last:* ***shared:***	*first:* ***exclusive:***	*comb min: shared in:* ***shared*** *out:*	***first: ext-exclusive:***
$\neg(E_1, E_3, E_2)$	E_2	*exclusive:*	*exclusive:*	*shared in:, exclusive out:*	*exclusive:*
$A*(E_1, E_2, E_3)$	E_1	*last:* ***ext-exclusive:***	*first: exclusive:*	*comb min: exclusive in: exclusive out:*	*(?* ***first:)*** ***ext-exclusive:***
$\vee(\neg(\mathbf{E_1, E_2, E_3}),$	E_2	*(?cumul:) exclusive:*	*cumul: exclusive:*	*cumul: shared in: exclusive out:*	*cumul:* ***shared:***
$;(E_1, E_2, E_3))$	E_3	*exclusive:*	*exclusive:*	*shared in: exclusive out:*	*exclusive:*

Figure 6: *The semantics of event expressions in Snoop*

While the unrestricted parameter context does include all permissible instance sequences that rely on exactly one occurrence of each component event type it does not include any sequence that can be constructed by including more than one instance of at least one component event type.

2. In the *recent* context only the most recent instance from the set of instances of the initiator type is used. Instances of component event types that can no longer be part of future parent event sequences are deleted. An initiator of an event continues to initiate new event occurrences until a new initiator occurs.[17]
3. In the *chronicle* context the oldest instance of each component event type that still fulfills the parent event type pattern is selected for the instance specific EIS of its parent type. Instances of component event types can only be part of one instance specific EIS; i.e., they are consumed with their use.
4. In the *continuous* context, if a terminator event is detected, for each initiator an event instance of its parent type is generated if it fulfills the parent event type pattern. In this context, each permissible initiator instance is part of at least one instance specific EIS of its parent type.
5. In the *cumulative* context all instances of the component event types are bound to the instance sequence of their parent type. Their use implies their consumption.

A complex event type can be constructed by combining an arbitrary number of event types by event operators. Each operator may come with its own parameter context. However, in their papers the authors have not investigated this aspect of Snoop in more detail.

In contrast to our model the parameter modes of Snoop are to be specified on the level of parent event types rather than on the level of component event types. Moreover, Snoop does not explicitly differentiate between *instance selection* and *instance consumption.* Instead, both aspects are mixed in the parameter context, which means that each parameter context represents a fixed combination of an *event instance selection* value with an *event instance consumption* value. Thus, Snoop supports only a limited set of combinations.

Meta-model based representation

Figure 6 presents the specification of Snoop on the basis of our meta-model.

Inconsistencies or irregularities

Let us consider some examples describing irregularities of Snoop that were detected with the help of our specification presented in Figure 6. In these examples the Snoop event types are denoted by a parameter context followed by an event operator and its operands.

The definitions of the event operators show several inhomogenities:

1. The operator P* requires that at least one periodic event occurs within the monitoring interval defined by instances of its first and last parameter event type (E_1 and E_3). The operator A*, however, does not impose this requirement, i. e. events of the type $A*(E_1, E_2, E_3)$ are triggered even if an event of type E_2 does not occur within the interval.
2. Consider the event types $A(E_1, E_2, E_3)$ and $P(E_1$, TI [:parameters], $E_3)$. The operators A and P prohibit an event of E_3 to occur between the event of E_1 and the a-periodic, respectively periodic event. The operators A* and P* do not impose such a restriction.

The following example shows, that the formal definition of the event operator A is not consistent

[17] Note, that the semantics of the parameter context *recent* was altered. ([Mish91], [CM93]) had defined, that initiator instances can only be used once.

with the definition of its composite event instance histories.

Example 15:

Consider the global event instance history H={{ ei_1^1 }, { ei_3^1 }, { ei_2^1 }, { ei_2^2 }}. The primitive event instance histories are defined as follows: E_1 [H]={{ ei_1^1 }}, E_2 [H]={{ ei_2^1 }, { ei_2^2 }}, and E_3 [H]={{ ei_3^1 }}. The composite event history of the event type A(E_1, E_2, E_3) is defined as follows: A(E_1, E_2, E_3)[H]={{ ei_1^1 , ei_2^1 }, { ei_1^1 , ei_2^2 }}. Thus, H triggers two events of the type A(E_1, E_2, E_3) even though the event instances of E_2 occur outside the monitoring interval defined by the instances ei_1^1 and ei_3^1.

Lack of symmetry

The negation operator can be regarded as a specialization of the sequence operator. While both stipulate an order on its parameters the negation operator additionally requires the non-occurrence of a further event type E_F. Thus, in the case of the absence of instances of E_F, one would assume that both operators deliver the same result. However, in the recent mode the operators use different event instance consumption modes for their first component event type: while the sequence operator implies the shared mode, the negation operator assumes the exclusive mode. Thus, the context recent is not defined homogeneously as its semantics depends on the kind of operator it is combined with (see Example 16).

Example 16:

Consider the event types: E_6:=*recent* (E_1 ; E_3) and $E_{6'}$:=*recent* ¬(E_2) (E_1, E_3) and the event instance sequence EIS^{13}:= ei_1^1 ei_3^1 ei_3^2 . An event of type E_6 and $E_{6'}$ is triggered as soon as an event of E_3 occurs provided that an event of E_1 had occurred already. The negation operator additionally requires the non-occurrence of events of E_2 *and* E_3 between the events of E_1 and E_3 that mark the boundary of the interval. In principle, $E_{6'}$ is a specialization of E_6. Thus, one would expect that these types show the same behavior in the absence of occurrences of type E_2. This, however, is not the case. EIS^{13} causes the recognition of two events of E_6 ((ei_1^1 ei_3^1) and (ei_1^1 ei_3^2)), but only one event of $E_{6'}$ ((ei_1^1 ei_3^1)).

The instance selection modes *recent* and *chronicle* imply different kinds of instance consumption (see Example 17) which means that they are not symmetrically defined.

Example 17:

Consider the event types E_5:=*recent* (E_1 Δ E_2) and $E_{5'}$:=*chronicle* (E_1 Δ E_2) and the event instance sequence EIS^{14}:= ei_1^1 ei_1^2 ei_2^1 . EIS^{14} causes the recognition of two events of E_5 ((ei_1^2 ei_2^1) and (ei_1^1 ei_2^1)) but only one event of $E_{5'}$ (ei_1^1 ei_2^1).

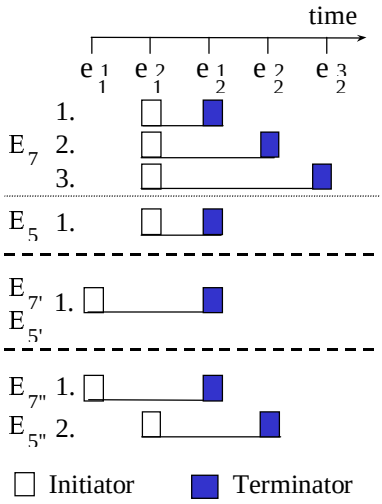

Figure 7: *The semantics of the operators A and ¬*

Lack of minimality

As can be seen from Example 18 the same semantic condition can be expressed in different ways.

Example 18:

Consider the following event types, based on the a-periodic and the negation operator with different instance selection modes:

E_7 := *recent* A(E_1, E_2, E_3)
E_5 := *recent* ¬(E_3) (E_1, E_2)
$E_{7'}$:= *cumulative* A(E_1, E_2, E_3)
$E_{5'}$:= *cumulative* ¬(E_3) (E_1, E_2)
$E_{7''}$:= *chronicle* A(E_1, E_2, E_3)
$E_{5''}$:= *chronicle* ¬(E_3) (E_1, E_2)

EIS^{15}:= ei_1^1 ei_1^2 ei_2^1 ei_2^2 ei_2^3 causes the recognition of exactly one event of E_5, $E_{7'}$ and $E_{5'}$, two events of $E_{7''}$ and $E_{5''}$, and three events of E_5 (see Figure 7).

The event types $E_{7'}$ and $E_{5'}$ ($E_{7''}$ and $E_{5''}$) have the same semantics (not only in this example, but generally), i.e. they are redundant (lack of orthogonality), while the types E_7 and E_5 do not, i.e., *cumulative/chronicle* and *recent* do not behave symmetrically.

4.3 SAMOS

The semantics of complex events in SAMOS is introduced in [Gatz94], however, informally. A more precise definition can be obtained from the labeled petri nets that form the basis for event detection. Simultaneously occurring events are explicitly excluded.

Event type pattern

SAMOS provides the binary event operators conjunction (,), disjunction (|), and sequence (;), and the unary operators negation (**NOT**), ∗, *last* and

TIMES(*n*, E). The unary operators must be used in conjunction with a monitoring time interval[18], denoted as '**IN** [*start_ point, end_point*]'. The start and the end point of the interval can be defined explicitly by absolute or relative points in time or implicitly by the occurrence of events of specified event types. If no interval is specified the system assumes the time between the definition time of the complex event type and infinity. Events which mark the monitoring interval of complex events are not bound to them.

The operators ∗ and *last* trigger only one complex event per time interval, even if the underlying pattern of the event types occurs several times. In case that several instances of the same type do occur ∗ selects the oldest instance while *last* takes the most recent one. Events based on ∗ are signaled as soon as the complex event to be monitored occurs for the first time while events based on *last* are signaled at the end of the monitoring interval.

The **TIMES** operator can be used in different variants. Events based on the variant **TIMES**(*n*, E) **IN** *I* are triggered each time *n* events of type E occur within the time interval *I*. Events based on the variant **TIMES** ([n_1-n_2], E) **IN** *I* are signaled at the end of *I* if events of type *E* occur n_1 to n_2 times in *I*. Finally, events based on the variant **TIMES**([>n_1], E) **IN** *I* are triggered at the end of *I* if events of type E occur more than n_1 times. **TIMES** uses the *cumulative* selection mode, i.e. all events of type E that caused the complex event to occur are bound to it.

SAMOS offers the possibility to specify whether component events must be triggered by the same transaction, must be caused by the same user or by database operations executed on the same data (object).

Event instance selection and consumption

In SAMOS event instance selection and consumption are pre-determined for each operator (see above). This means that a number of instance selection and consumption policies are not supported. Moreover, the operator **TIMES** influences the semantics of the sub-dimension repetition. Finally, several useful operator combinations, like ∗E_1,∗E_2, are not permissible.

Meta-model based representation

Figure 8 presents the specification of SAMOS on the basis of our meta-model.

Note, that SAMOS is the only one of the examined systems that explicitly permits complex events to be caused by component events triggered within different transactions. However, it is not explained which transaction is responsible for the execution of the rule triggered by such a complex event[19].

SAMOS (any ta) Meta-Model	*CP ET*	*CPET modes*
$E_1 \mid E_2$	E_1	exclusive:
? ∨(E_1, E_2)	E_2	*exclusive:*
E_1 , E_2	E_1	*first: exclusive:*
? ∧(E_1, E_2)	E_2	*first: exclusive:*
E_1 ; E_2	E_1	*first: exclusive:*
? ;(E_1, E_2)	E_2	*exclusive:*
E_1 ; E_2 **IN** [E_1-E_3][20]	E_1	*(? first:)* ***shared:***
? ¬(E_1, E_3, E_2)	E_2	*exclusive:*
∗E_1 ; E_2	E_1	*first:* ***ext-exclusive:***
? ;(E_1, E_2)	E_2	*exclusive:*
last E_1 ; E_2	E_1	*last:* ***ext-exclusive:***
? ;(E_1, E_2)	E_2	*exclusive:*
TIMES([n_1-n_2], E_2) **IN** [E_1-E_3]	E_1	*(? first:) exclusive:*
? ;(E_1, (n_1-n_2)E_2, E_3)	E_2	*cumul: exclusive:*
	E_3	*exclusive:*
TIMES([>n], E_2) **IN** [E_1-E_3]	E_1	*(? first:) exclusive:*
? ;(E_1, (n-∞)E_2, E_3)	E_2	*cumul: exclusive:*
	E_3	*exclusive:*
TIMES(n, E_2) **IN** [E_1-E_3]	E_1	*(? first:) shared:*
? ¬(E_1, E_3, (n)E_2)	E_2	*exclusive:*
NOT E_3 **IN** [E_1-E_2]	E_1	*(? first:) exclusive:*
? ¬(E_1, E_3, E_2)	E_2	*exclusive:*

Figure 8: *The semantics of event expressions in SAMOS*

Inconsistencies or irregularities

As is proved by [Zimm98] the operator ∗ is not orthogonal to the sequence operator as claimed in [Gatz94], page 165, since it can be modeled with the help of the sequence operator as well.

Example 19:

Consider the event types E_5:=E_1 ; E_2, $E_{5'}$:=(E_1 ; E_2) **IN** [E_1-E_3] and $E_{5''}$:=∗E_1 ; E_2. The event instance sequence EIS[16]:= ei_1^1 ei_1^2 ei_2^1 ei_2^2 ei_2^3 causes the recognition of one event of type $E_{5''}$, two events of type E_5 and three events of type $E_{5'}$ (see Figure 9).

Moreover, the semantics of the monitoring intervals is somewhat awkward as the reduction of the monitoring

[18] A monitoring interval can also be defined for the binary event operators. But for these operators it is optional.

[19] Most systems use so-called *coupling modes* to determine the transaction that is responsible for the execution of a rule and to lay down, how the further execution of the rule is to be handled. However, transaction management is a difficult task due to the *all-or-nothing* property associated with transactions. It implies that a rule that is to be executed on behalf of several initiating transactions, e.g. in its own subtransaction, must be successful in order for the triggering transactions to be allowed to commit. These dependencies between the rule to be executed and the triggering transaction have caused already a lot of problems in the case of one triggering transaction (and e.g., a decoupled execution of the rule, cf. the discussion in section 0). These problems will substantially increase in case of several triggering transactions.

[20] The type E_3 is a relative temporal event type which is based on the event type E_1.

time may cause an increase in the number of events detected ($E_{5'}$ vs. E_5).

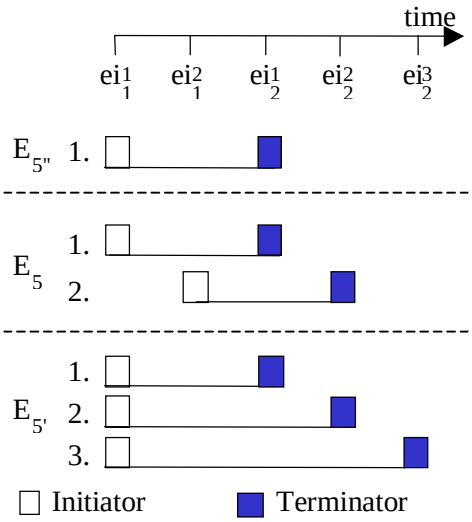

Figure 9: *The semantics of different variants of the sequence operator*

Example 20:

Consider the event type E_6:=**NOT** E_2 **IN** [E_1-E_3] and the event instance sequence EIS^{17}:= ei_1^1 ei_1^2 ei_2^1 ei_3^1 ei_3^2. EIS^{17} contains several instances of the time interval [E_1-E_3], among others I_1=[ei_1^1, ei_3^1] and I_2=[ei_1^2, ei_3^2]. As during these time intervals an instance of E_2 (ei_2^1) occurs, one may expect that EIS^{17} does not cause an event of E_6 to occur. However, the event detection algorithm of the **NOT** operator, specified by a labeled petri net (cf. [Gatz94], page 110), deletes instances of E_2, and, thus, ei_3^2 will trigger an event of E_6.

4.4 Ode

Ode ([GeJS92], [GeJa96]) defines its semantics of complex events on a formal level. Its definition is based on event histories (*histories*) which contain a finite set of events that are totally ordered along their event occurrence times. This implies that primitive events cannot occur simultaneously.

Event type pattern

An event type E is defined as a mapping from histories to histories (E: *histories*→*histories*). The resulting history E[*h*] contains those event instances of *h* that trigger an event of type E. An event type can be **NULL**, any primitive event type E_P, or a complex event type E_C. ODE provides a huge number of operators from which we only consider the following: ∧, ! (not), *relative*, *relative+*, ∨, *any*, *prior*, and *sequence*.

Let E and F be event types. The semantics of the operators is defined as follows:

1. E[*null*]=*null* for any event type E, where *null* is the empty history.
2. **NULL**[*h*]=*null*
3. $E_P[h]$ contains all event instances of type E_P being part of *h*, where E_P is a primitive event type.
4. (E∧F) [*h*]=E[*h*]∩F[*h*]
5. (!E)[*h*]=(*h*–E[*h*])
6. *relative*(E, F)[*h*] are the event instances in *h* that trigger events of F. However, each time only that part of *h* is considered that starts immediately after an instance of E.
   Let $ei^i[h]$ be the i^{th} event instance in *E[h]*. Let h_i be deduced from *h* by deleting all event instances with an event occurrence time older than or equal to the event occurrence time of $ei^i[h]$. Then *relative* is defined as follows:
   relative(E, F)[*h*]= $\bigcup_i F[h_i]$, i=1 ... |E[*h*]|[21].
7. *relative*+(E)[*h*]= $\bigcup_{i=1}^{\infty} relative^i$ (E)[*h*], where
   $relative^1$(E)=E and
   $relative^i$(E)=*relative*($relative^{i-1}$(E), E).
8. (E∨F) [*h*]=!(!E∧!F)[*h*]
9. *any* denotes the disjunction of all primitive event types.
10. *prior*(E, F)[*h*]=(*relative*(E, *any*) ∧ F)[*h*]
    The operator *prior* specifies, that an event of F has to occur after an event of E did occur. If E and/or F are complex events their event instance sequences may - in contrast to the operator *relative* - overlap.
11. *sequence*(E, F)[*h*]=
    (*relative*(E, !(*relative*(*any*, *any*)))∧*F*)[*h*]
    Sequence requires an event of type F to immediately succeed an event of type E..

Ode differentiates between *time related* and *database related* primitive event types. Database related event types are essentially either event types caused by a transaction or by an operation/method. A complex event can only occur if its database related component event instances are triggered by operations on the same data (objects) and within the same transaction.

An optional predicate can be assigned to an event type to form a so-called *logical event type*. It returns a Boolean value whose value depends on the values of the event parameters and the state of database objects. Whenever an event of its type occurs the predicate is evaluated. If it evaluates to **TRUE** the action part of the rule is executed. This means that the condition part of a rule is already part of the event specification which is why rules in Ode do not contain a separate condition part and are thus called *EA-Rules* (*E*vent-*A*ction-Rules).

[21] || denotes the cardinality

Event instance selection and consumption
[GeJS92] briefly discusses the semantics of the event instance selection. The selection consists of two steps: In the first step all possible event instance sequences that fulfil the event type pattern 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.

Meta-model based representation
Figure 10 presents the specification of Ode on the basis of our meta-model.
Note, that the semantics of the conjunction and negation operators differs substantially from their semantics in other event algebras: The conjunction operator requires both component event types to occur simultaneously. The negation operator triggers an event whenever an event of a type different from its parameter type occurs.

Ode (***same op-context***) *Meta-Model*	*CP ET*	*CPET modes*
$E_1 \mid E_2$	E_1	*exclusive:*
$\vee(E_1, E_2)$	E_2	*exclusive:*
$E_1 \wedge E_2$	E_1	*exclusive:*
$==(E_1, E_2)$	E_2	*exclusive:*
relative (E_1 , E_2)	E_1	*(? first:) shared:*
(*non-overlap*) $;(E_1, E_2)$	E_2	*exclusive:*
prior (E_1 , E_2)	E_1	*(? first:) shared:*
$;(E_1, E_2)$	E_2	*exclusive:*
sequence (E_1 , E_2)	E_1	*last: ext-exclusive:*
(*continuous*) $;(E_1,E_2)$	E_2	*exclusive:*
$! E_1$	E_2	*exclusive:*
$\vee(E_2, E_3, \ldots E_n)$	E_n	*exclusive:*

Figure 10: *The semantics of event expressions in Ode*

4.5 NAOS

The rule model of NAOS has been implemented as a prototype on top of the O_2 object-oriented database management system ([BaDK92]).

Event type pattern
NAOS ([CoCo96]) provides the binary event operators conjunction (&&), disjunction (||), strict disjunction (^), sequence (,) and strict sequence (;) and the unary operators negation (!), iteration (*n*(E)) and strict iteration (*n*.(E)). They are introduced on an informal level, i.e. their semantics is supposed to be the usual one. The occurrence time of a complex event instance is identical to the occurrence time of its terminator.

Disjunction requires that at least one of the specified parameter event types has to occur within a given monitoring interval (see above). *Strict disjunction* restricts the occurrence of parameter event type instances to *exactly one* type; no instances of the other parameter event types are allowed to occur within the monitoring interval. The *sequence* operator requires its (complex) parameter instances to occur in the specified order while the *strict sequence* operator additionally excludes an overlapping of the component instance sequences of the (complex) parameter instances. *Iteration* triggers an event each time *n* events of its component event type occur. The *strict iteration* operator additionally requires, that the component event instances have to occur in a strict sequence; i.e. without overlap.
It is stated, that the semantics of the strict sequence operator corresponds to that of the sequence operator, if one (or both) of its component event types is a primitive event type. However, this statement can obviously not be correct in case the second component event type is a complex one (since in that case the instance sequences of the component event types can overlap).
Note, that it is allowed that component event types can be of the same type. However, the semantics of such definitions remains unclear[22].

The detection of complex events depends on a so-called *unit of production* that either represents a program, a transaction, or an operation. The unit of production can be specified either explicitly or implicitly using the sequence operator and the event types denoting the beginning and the end of the production unit[23]. NAOS only considers transactions as production units, i.e., a complex event is only triggered if all its component event instances were caused by the same transaction. Each event type needs to be associated with a monitoring interval, called *validity time interval.* It defines the period during which the occurrence of the type is monitored; i.e. it can either correspond to the unit of production (implicit specification) or represent a refinement (explicit specification). An explicit specification requires either the specification of two points in time or of a period of time, however, does not allow a relative specification by the specification of two event types marking the boundary of the interval. For the

[22] Is, for example, in case of $E_C:=E_1 \| E_1$ the event pattern for E_C fulfilled as soon as the first instance of E_1 arises or must there be at least two different instances of E_1?

[23] An implicit definition of the production unit on the base of the sequence operator is, however, not possible. Consider the two event types begin and end of a transaction (BOT and EOT). The sequence operator only considers the temporal aspects of events and does not guarantee that they are triggered by the same transaction, i.e., a complex event can also be triggered by transaction events (and the events being monitored) caused by different transactions.

time being the prototype of NAOS does not consider temporal and external events, and monitoring intervals cannot be defined explicitly.

Event instance selection and consumption
For the definition of the event instance selection and consumption semantics NAOS adopted the semantics of the Snoop parameter context *continuous.*

Meta-model based representation
Figure 11 presents the specification of NAOS on the basis of our meta-model.

NAOS / *Meta-Model*	*CP* / *ET*	*CPET modes*
$E_1 \| \| E_2$	E_1	*exclusive:*
$\vee(E_1, E_2)$	E_2	*exclusive:*
$E_1 ; (E_2 \wedge E_3) ; E_4$	E_1	*exclusive:*
(*same instance* (E_1, E_4))	E_2	*exclusive:*
$\vee(\wedge(;(E_1, E_2, E_4), \neg(E_1,E_3,E_4)),$	E_3	*exclusive:*
$\wedge(;(E_1, E_3, E_4), \neg(E_1, E_2, E_4)))$	E_4	*exclusive:*
E_1 && E_2	E_1	*comb min: shared in: exclusive out:*
$\wedge(E_1, E_2)$	E_2	*comb min: shared in: exclusive out:*
$n(E_1)$ / $\wedge((n)E_1)$	E_1	*exclusive:*
n (E_1) / (*non-overlap*) $\wedge((n)E_1)$	E_1	*exclusive:*
E_1 , E_2	E_1	*comb min: exclusive:*
$;(E_1, E_2)$	E_2	*shared in: exclusive out:*
E_1 ; E_2	E_1	*comb min: exclusive:*
(*non-overlap*) $;(E_1, E_2)$	E_2	*shared in: exclusive out:*
E_1, !E_2, E_3	E_1	*comb min: exclusive:*
$\neg(E_1, E_2, E_3)$	E_3	*shared in: exclusive out:*

Figure 11: *The semantics of event expressions in NAOS*

4.6 Chimera

Chimera ([CFPT96], [MePC96], [FrMT94]) is a prototype of an active object-oriented deductive data-base management system developed at the Politecnico di Milano, Italy. The system uses *EECA-Rules* (*E*xtended-*ECA*-Rules). In comparison to ECA-Rules the condition part of these rules may additionally include *event formulas* that create bindings to the data (objects) affected by a specified set of events[24]. These event formulas are necessary as Chimera does not permit the transfer of any data from the event to the condition part. Chimera defines so-called *consumption modes*, which can be used to control the amount of data accessible by event formulas[25].
A triggered rule can only be triggered again after the evaluation of its condition part is completed. All event instances that do occur in the meantime, are irrelevant. The point in time the condition of a triggered rule is evaluated depends on the coupling mode of the condition part of the event. The evaluation is performed directly if the coupling mode *immediate* is chosen; it is suspended until the underlying transaction commits if the coupling mode *deferred* is taken. Chimera only considers database event types that, additionally, have to be triggered by the same transaction.

Event type pattern
Event operators can either work on the level of sets (types) or the level of instance. *Set-oriented* operators consider component events independently of the objects affected by the triggering DML-operations. *Instance-oriented* operators additionally require, that the component events must be caused by operations executed on the same data/object. In this paper we will only consider set-oriented operators.
The semantics of the supported algebra operators – disjunction (,), conjunction (+), negation (-), and sequence (<) – is defined formally. Similar to Snoop, Chimera introduces a function *ts*() which computes for a given event type E_i and a point in time *t*, whether an event instance of E_i has occurred up to this point in time. If the asnwer is yes, *ts*() returns the occurrence time of the most recent instance of E_i. Otherwise, the negative value of *t* is returned. The definition of *ts*() relies on the functions *type*() and *timestamp*(). They compute the type of an event instance (*type*()) and its occurrence time (*timestamp*()). Let R denote the history of event instances. For primitive event types, *ts*() is defined as follows:

$$ts(E_i, t) := \begin{cases} -t, & \text{if } \forall\, t' \,(t' \le t \wedge \neg\exists\, ei \in R: (type(ei) = E_i \wedge timestamp(ei) = t')) \\ t_E & \text{otherwise,} \\ & \text{where } t_E := max\{t' \mid t' \le t \wedge \exists\, ei \in R: (type(ei) = E_i \wedge timestamp(ei) = t')\} \end{cases}$$

The definition of *ts()* for complex event types relies on the following function:

$$occ(E_i, t) := \begin{cases} true, & \text{if } ts(E_i, t) \ge 0 \\ false, & \text{otherwise} \end{cases}$$

On this basis *ts*() is defined as follows:

$$ts(-E_i, t) := \quad -ts(E_i, t)$$

[24] An event formula can rely on either of two predicates. While the predicate *occurred* binds all objects affected by the component events to the parent event variable the predicate *hold* considers only a subset that is computed, e.g., by evaluating the *net-effect* of operations (see [CFPT96] for further details).

[25] Do not mix it up with the dimension event instance consumption of the meta-model. In Chimera the consumption mode consuming (preserving) defines that only those event instances are accessible to event formulas that were caused by the same transaction after the last complete execution of the rule.

$$ts((E_i+E_k), t) := \begin{cases} min\{ts(E_i, t), ts(E_k, t)\}, \\ \qquad if \neg occ(E_i,t) \vee \neg occ(E_k, t) \\ max\{ts(E_i, t), ts(E_k, t)\}, \\ \qquad if\ occ(E_i, t) \wedge occ(E_k, t) \end{cases}$$

$$ts((E_i, E_k), t) := \begin{cases} min\{ts(E_i, t), ts(E_k, t)\}, \\ \qquad if \neg occ(E_i,t) \wedge \neg occ(E_k,t) \\ max\{ts(E_i, t), ts(E_k, t)\}, \\ \qquad if\ occ(E_i, t) \vee occ(E_k, t) \end{cases}$$

$$ts((E_i<E_k), t) := \begin{cases} -t, \quad if\ (\neg occ(E_i, t) \vee \neg occ(E_k, t)) \vee \\ \qquad (occ(E_i, t) \wedge occ(E_k, t) \wedge \\ \qquad ts(E_i, ts(E_k, t)) < 0) \\ ts(E_k, t), if\ occ(E_i, t) \wedge occ(E_k, t) \wedge \\ \qquad ts(E_i, ts(E_k, t)) \geq 0 \end{cases}$$

Note, that the event instance selection is controlled by the event formula. It lays down what data/information is preserved for later use.

Event instance selection and consumption

Event instance consumption is determined by the coupling mode of the condition part. Whenever the condition of a rule is evaluated every component event instance occurred so far becomes irrelevant. This includes all those component event instances that do occur after the terminator of the parent event instance, however, before the evaluation of the condition part of the rule is completed.

Meta-model based representation

Figure 12 presents the specification of Chimera on the basis of our meta-model.

Chimera	*Meta-Model*
E_1 , E_2	$\vee(E_1, E_2)$
E_1 + E_2	$\wedge(E_1, E_2)$
E_1 < E_2	$;(E_1, E_2)$
-E_1	$\vee$**(E_2, E_3, ... E_n)**

Figure 12: *The semantics of the event operators defined in Chimera*

Note, that the semantics of the dimensions event instance selection and consumption is controlled by the event formula and the execution of rules.

4.7 ACOOD

ACOOD (*Ac*tive *O*bject *O*riented *D*atabase System) ([Bern91], [Erik93], [Ber94], [BeLi92], [Ekl95], [Schw95]) is an active object-oriented database management system implemented as a prototype on top of Ontos ([AnHD90], [Solo92]). ACOOD uses ECA-rules which may additionally contain a predicate that evaluates the event parameters and the database state. A complex event is only triggered if its component events fulfil this predicate.

Event type pattern

ACOOD defines the event operators conjunction (&), disjunction (|), and sequence (;) with arity *n* and the operators negation (N) and iteration (I) with arity 3. The semantics of the iteration operator corresponds to that of the Snoop operator A∗. All other operators realize the usual semantics.

Event instance selection and consumption

To control event instance selection ACOOD provides the following parameter contexts ([Erik93]):

1. *Recent*: The semantics of this context corresponds to that of the Snoop context *recent*.
2. *Chronicle*: The semantics of this context corresponds to that of the Snoop context *chronicle*.
3. *Cumulative*: Like the other two modes *cumulative* only selects one instance per component event type. However, for every possible permutation of component event instances a parent event instance is generated.

Example 21:

Consider the event type E_3:=*cumulative*(E_1 ; E_2) and the event instance sequence EIS^{11}:= $ei_1^1\ ei_1^2\ ei_1^3\ ei_2^1\ ei_2^2$. EIS^{11} causes the recognition of six events of E_3: ei_3^1 ($ei_1^1\ ei_2^1$), ei_3^2 ($ei_1^2\ ei_2^1$), ei_3^3 ($ei_1^3\ ei_2^1$), ei_3^4 ($ei_1^1\ ei_2^2$), ei_3^5 ($ei_1^2\ ei_2^2$) and ei_3^6 ($ei_1^3\ ei_2^2$).

Note, that the semantics of the ACOOD parameter context *cumulative* is different from the Snoop contexts *cumulative* and *continuous.*

In principle, whenever a component event instances triggers an event it is consumed. However, ACOOD allows the same component event type to occur several times in a parent event type pattern. In such a case the same event instance can represent the event type several times in an instance specific EIS (cf. [Erik93], page 6).

Example 22:

Consider the event type E_2:=E_1 ; E_1 ; E_1 and the event instance sequence EIS^{12}:= $ei_1^1\ ei_1^2\ ei_1^3\ ei_1^4\ ei_1^5$. EIS^{12} causes the recognition of three events e_2^1, e_2^2, and e_2^3 of E_2 represented by ei_2^1 ($ei_1^1\ ei_1^2\ ei_1^3$), ei_2^2 ($ei_1^2\ ei_1^3\ ei_1^4$) and ei_2^3 ($ei_1^3\ ei_1^4\ ei_1^5$).

ACOOD supports less configuration possibilities than Snoop.

Meta-model based representation

Figure 13 and Figure 14 present the specification of ACOOD on the basis of our meta-model.

ACOOD (? any ta) Meta-Model	*CP ET*	*CPET modes*	
		recent	*chronicle*
$E_1 \vee E_2$	E_1	*exclusive:*	*exclusive:*
$\vee(E_1, E_2)$	E_2	*exclusive:*	*exclusive:*
$E_1 \wedge E_2$	E_1	*last: ext-exclusive:*	*first: exclusive:*
$\wedge(E_1, E_2)$	E_2	*last: ext-exclusive:*	*first: exclusive:*
$E_1 ; E_2$	E_1	*last: ext-exclusive:*	*first: exclusive:*
$;(E_1, E_2)$	E_2	*exclusive:*	*exclusive:*
$N(E_1, E_2, E_3)$	E_1	*last: ext-exclusive:*	*first: exclusive:*
$\neg(E_1, E_2, E_3)$	E_3	*exclusive:*	*exclusive:*
$I(E_1, E_2, E_3)$	E_1	*last: ext-exclusive:*	*first: exclusive:*
$;(E_1, E_2, E_3)$	E_2	*cumul: exclusive:*	*cumul: exclusive:*
	E_3	*exclusive:*	*exclusive:*

Figure 13: *The semantics of event expressions in ACOOD (table 1)*

ACOOD (? any ta) Meta-Model	*CP ET*	*CPET modes*
		cumulative
$E_1 \vee E_2$	E_1	*exclusive:*
$\vee(E_1, E_2)$	E_2	*exclusive:*
$E_1 \wedge E_2$	E_1	*comb min: shared:*
$\wedge(E_1, E_2)$	E_2	*comb min: shared:*
$E_1 ; E_2$	E_1	*comb min: shared:*
$;(E_1, E_2)$	E_2	*shared in: exclusive out:*
$N(E_1, E_2, E_3)$	E_1	*comb min: shared:*
$\neg(E_1, E_2, E_3)$	E_3	*shared in: exclusive out:*
$I(E_1, E_2, E_3)$	E_1	*comb min: exclusive in: shared out:*
$;(E_1, E_2, E_3)$	E_2	*cumul: shared:*
	E_3	*shared in: exclusive out:*

Figure 14: *The semantics of event expressions in ACOOD (table 2)*

4.8 ADL

The database language ADL (*A*ctivity *D*escription *L*anguage) ([Behr94], [Behr95]), developed at the university of Oldenburg, Germany, can be used to specify active behavior. ADL uses *(EC)*A-rules* which differ to ECA-rules in that conditions can be assigned to event types. A condition can consider the values of the parameters of its component event instances. Like in Ode and ACOOD an instance of an event type can only occur if its condition is fulfilled by its parameter event instances.

Event type pattern

Primitive event instances cannot occur simultaneously. The order of potentially simultaneous event instances is determined by the system.

The occurrence time of a complex event is described by a time interval rather than a point in time. The upper (lower) bound of a time interval assigned to a complex event is defined by the youngest (oldest) point in time belonging to the time interval of one of its parameter events.

ADL supports the event operators conjunction, disjunction, two versions of the sequence (**SOFT SEQ** and **SEQ)** and the negation operator (**NOT** and **NO**), and several operators for the definition of relative time events. Most event operators correspond to operators already introduced in the previous sections. The operator **SOFT SEQ** requires the lower as well as the upper bound of the time interval of a given component event instance to be older than the appropriate values of its successors in the sequence; i.e., it requires the order to be kept, however, permits an overlap of the time intervals. The operator **SEQ** is stronger in that it excludes such an overlap of time intervals. The operator **COUNT** specifies the number of event occurrences of a given type that are required to trigger the parent event. The monitoring interval during which component events have to occur to trigger a complex event can be restricted using one of the operators **AT**, **BETWEEN** or **DURING**. The operator **AT** specifies that an event has to occur at a specified point in time. The operator **BETWEEN** uses two event types, whose events define monitoring intervals. The operator **DURING** defines a period (e.g. 3 minutes) in which all component events have to occur.

The negation operator **NO** is to be used in conjunction with **AT** and **BETWEEN**. Its semantics corresponds to the above described semantics of **AT** and **BETWEEN**. The other negation operator **NOT** is to be used in conjunction with **AT**, **DURING** or **BETWEEN**. It tightens the **NO** operator in that it additionally requires an instance of the related component event type to occur outside the specified monitoring interval (**BETWEEN**) or period (**DURING**) or at a different point in time (**AT**). The operator **EVERY** can be used to trigger periodic temporal events within a predefined period of time.

Event instance selection and consumption

In principle, the lifetime of an event instance terminates when it triggers a rule, is selected as a representative in an instance specific EIS, or when a period expires that can explicitly be assigned to the instance by using the operator **ALIVE**.

The same event type can be used several times as an operand of a conjunction operator. In this case one event of this component event type is sufficient to trigger a complex event[26].

ADL offers a huge number of modes to influence the event instances selection process:

FIFO: Its semantics corresponds to that of the parameter context *chronicle* defined in Snoop and ACOOD.

LIFO: Its semantics corresponds to that of the parameter context *recent* defined in ACOOD.

[26] Note, that this semantic is in contrast to that of ACOOD.

TAKE_ALL: Its semantics corresponds to that of the parameter context *cumulative* defined in ACOOD, i.e. it constructs all permissible permutations of event instances which select exactly one event instance from each component event type. Component event instances that are selected for an instance specific EIS remain available for further event detection.

TAKE_ALL-C(umulative): It is based on the strategy **TAKE_ALL**. The difference is, that event instances are consumed, if they cause (directly or indirectly) the occurrence of a parent event E_C whose type is not component event of a more complex parent type. In contrast to the strategies **FIFO** and **LIFO**, the appropriate event instances remain locally available until the event of type E_C is really triggered. However, with that all affected event instances are deleted.

TAKE_ALL-C-FIFO: It is based on the strategy **TAKE_ALL-C**. This strategy prevents event instances from being selected as event instances for several instance specific EIS. Whenever event instances of a type that does not have a parent type are generated only that instances survive that use older parameter event instances.

TAKE_ALL-C-LIFO: It corresponds to the strategy **TAKE_ALL-C-FIFO**. But in contrast to **TAKE_ALL-C-FIFO** the instances with younger parameter event instances are selected.

Note, that the semantics of the strategies **TAKE_ALL-C**, **TAKE_ALL-C-FIFO** and **TAKE_ALL-C-LIFO** is not defined strictly local, i.e. not restricted to one event type. Such more globally defined semantics is considered by extensions of our meta-model which are beyond the scope of this paper (see [Zimm98] for more details).

Different selection strategies may be combined on different levels in the definition of a hierarchically defined complex event type. However, this topic is not discussed in further detail.
The semantics of complex events in combination with the strategy **FIFO** is defined formally based on Evolving Algebras ([Gure94]).

Meta-model based representation
Figure 15 and Figure 16 present the specification of ADL on the basis of our meta-model.

ADL *(? any ta) Meta-Model*	*CP ET*	*CPET modes* `LIFO`	`FIFO`
E_1 **OR** E_2	E_1	*exclusive:*	*exclusive:*
? $\vee(E_1, E_2)$	E_2	*exclusive:*	*exclusive:*
E_1 **AND** E_2	E_1	*last: exclusive:*	*first: exclusive:*
? $\wedge(E_1, E_2)$	E_2	*last: exclusive:*	*first: exclusive:*
SOFT SEQ (E_1, E_2)[27]	E_1	*last: exclusive:*	*first: exclusive:*
? ;(E_1, E_2)	E_2	*exclusive:*	*exclusive:*
SEQ (E_1, E_2)	E_1	*last: exclusive:*	*first: exclusive:*
(*non-overlap*) ;(E_1,E_2)	E_2	*exclusive:*	*exclusive:*
COUNT *n* **OF** E_1 $\wedge((n)E_1)$	E_1	*exclusive:*	*exclusive:*
E_2 **AT** *time*	E_1	*exclusive:*	*exclusive:*
==(E_1, E_2)[28]	E_2	*exclusive:*	*exclusive:*
E_2 **BETWEEN** (E_1, E_3)	E_1	*last: exclusive:*	*first: exclusive:*
(*non-overlap*) ¬(E_1, E_3, E_2)	E_2	*exclusive:*	*exclusive:*
(**NO** E_2) **BETWEEN** (E_1, E_3)	E_1	*last: exclusive:*	*first: exclusive:*
(*non-overlap*) ¬(E_1, E_2, E_3)	E_3	*exclusive:*	*exclusive:*
E_2 **NOT BETWEEN** (E_1, E_3)	E_1	*last: ext-exclusive:*	*first: ext-exclusive:*
$\vee(\neg(E_3, E_1, E_2), \neg(E_4, E_1, E_2))$[29]	E_2	*exclusive:*	*exclusive:*
E_2 **DURING** *time_span*	E_1	*exclusive:*	*exclusive:*
(*same instance* (E_1)) ¬(E_1, E_3, E_2)[30]	E_2	*exclusive:*	*exclusive:*
E_2 **NOT DURING** *time_span*	E_1	*exclusive:*	*exclusive:*
(*same instance* (E_1)) ;(E_1, E_3, E_2)[31]	E_2	*exclusive:*	*exclusive:*

Figure 15: *The semantics of event expressions in ADL (table 1)*

ADL *(? any ta) Meta-Model*	*CP ET*	*CPET modes* `TAKE_ALL`
E_1 **OR** E_2	E_1	*exclusive:*
? $\vee(E_1, E_2)$	E_2	*exclusive:*
E_1 **AND** E_2	E_1	*comb min: shared:*
? $\wedge(E_1, E_2)$	E_2	*comb min: shared:*
SOFT SEQ (E_1, E_2)[32]	E_1	*comb min: shared:*
? ;(E_1, E_2)	E_2	*shared in: exclusive out:*
SEQ (E_1, E_2)	E_1	*comb min: shared:*
(*non-overlap*) ;(E_1,E_2)	E_2	*shared in: exclusive out:*
COUNT *n* **OF** E_1 $\wedge((n\text{-})E_1)$	E_1	*comb min: shared:*
E_2 **AT** *time*	E_1	*exclusive:*
==(E_1, E_2)[33]	E_2	*exclusive:*
E_2 **BETWEEN** (E_1, E_3)	E_1	*comb min: shared:*
(*non-overlap*) ¬(E_1, E_3, E_2)	E_2	*shared in: exclusive out:*

[27] Note, that in contrast to the sequence operator of the meta-model the operator **SOFT SEQ** additionally requires, that the initiator events of the component events occur in the pre-defined order.

[28] The type E_1 defines a temporal event occurring at the given *time*.

[29] The absolute temporal event type E_4 defines an event that occurs at the definition time of the ADL event type. The instance of E_4 is not consumed and remains valid until an instance of E_1 occurs.

[30] The type E_1 is a component event type of E_2, whose instances initiate the detection of instances of E_2. The relative temporal event type E_3 is based on E_1 and its instances occur *time_span* later than instances of E_1.

[31] See previous footnote.

[32] See footnote 27.

[33] The absolute temporal event type E_1 defines an event instance that occurs at time *time*.

(**NO** E_2) **BETWEEN** (E_1, E_3)	E_1	*comb min: shared:*
(*non-overlap*) ¬(E_1, E_2, E_3)	E_3	*shared in: exclusive out:*
E_2 **NOT BETWEEN** (E_1, E_3)	E_1	*comb min: shared:*
∨(¬(E_3, E_1, E_2), ¬(E_4, E_1, E_2))[34]	E_2	*shared in: exclusive out:*
E_2 **DURING** *time_span*	E_1	*exclusive:*
(*same instance* (E_1)) ¬(E_1, E_3, E_2)[35]	E_2	*exclusive:*
E_2 **NOT DURING** *time_span*	E_1	*exclusive:*
(*same instance* (E_1)) ;(E_1, E_3, E_2)[36]	E_2	*exclusive:*

Figure 16: *The semantics of event expressions in ADL (table 2)*

4.9 REFLEX

REFLEX ([NaIb94]) is an active database prototype developed at the university of Greenwich, London. It extends existing object-oriented database management systems by active facilities.

Event type pattern

REFLEX provides the event operators conjunction (**AND**), disjunction (**OR**), exclusive disjunction (**XOR**), negation (**NOT**), and the sequence operators **SUCCEEDS** and **PRECEDES**. While **PRECEDES** corresponds to the usual semantics of the sequence operator **SUCCEEDS** requires events to occur in the opposite order.

Additionally, REFLEX introduces the temporal keywords **BEFORE**, **AFTER**, **AT**, **BETWEEN**, **ON**, **WITHIN HOUR/MIN/SEC**, **EVERY HOUR/MIN/SEC**, **MIN**, **MAX**, **DATE** and **TIME** for use in time-based event specifications. The semantics of the operators and keywords is not explained in further detail. Although the general semantics of most of these constructs can be implied[37], their precise semantics remains unclear.

Meta-model based representation

A detailed definition of the semantics of the event algebra of REFLEX is not yet available. Thus, we are not able to present its specification.

5 Comparison of Existing Event Algebras

In the following we will compare the semantics of the conjunction, negation and sequence operators of the different event algebras with each operator being presented in a separate table. The disjunction operator is omitted since it has the same semantics in all event algebras.

There is an entry (row) for each possible instantiation (mode) of an operator in a given event algebra. The first column cites the event algebra and the second the instantiation of the operator in the notation of the underlying event algebra. If an operator instantiation has the same semantics in more than one event algebras each of them will be quoted in the first column with the specific notation of the operator being listed in the second column. The different algebras are separated by a dashed line. There is no entry for an event algebra in a given table if the corresponding operator is not supported. Otherwise, all possible specializations of an operator are listed. This implies that missing specializations are not supported.

The third column contains the specification of an event type on the basis of the meta-model. It is subdivided into several sub-columns.

The first sub-columns defines the skeleton of the operator/event type. Since the skeleton does only describe an operator on an more general level, it is very likely that it is the same for most of the event algebras. Therefore, it is listed in the heading of this sub-columns. Consequently the sub-columns of a given event algebra contains only the refinement of the skeleton (if necessary) or a different skeleton if the general one does not fit.

The remaining sub-columns define the specific modes of each of the component event types (parameters) of the operator (one sub-columns for each parameter).

Example 23:

Consider the first entry (row) of Figure 17. The expression "*chronicle* N(E_1, E_2, E_3)" (column 2) in Acood (column 1) has the same semantics as the expression "**NOT** E_2 **IN** [E_1-E_3]" in Samos. The skeleton of the negation operator is listed in the heading of the third column ("¬(E_1, E_2, E_3)"). To exactly meet the special semantics of the Acood/Samos representation it has to be refined by "(*any ta*)". The specific modes for the component event types are listed in sub-column 2 for E_1 ("*first: exclusive:*"), respectively sub-column 3 for E_3 ("*exclusive:*"),

Altogether this results in the following meta-model expression, that is equivalent in its semantics to the original notation of Acood/Samos:

(*any ta*) ¬(*first: exclusive:* E_1, E_2, *exclusive:* E_3)

The compact and consistent representation of the semantics of the underlying event algebras easily reveals redundancies, irregularities, and differences.

[34] See footnote 29.

[35] See footnote 30.

[36] See footnote 30.

[37] For most of the temporal keywords see ADL.

Target-Model		*Meta-Model*		
Name	*ET*	*ET*	*CPET*	
		¬(E1, E2, E3)	*E1*	*E3*
Acood	*chronicle* N(E_1, E_2, E_3)	(*? any ta*)	*first: exclusive:*	*exclusive:*
Samos	**NOT** E_2 **IN** [E_1-E_3]	(*any ta*)	*(? first:) exclusive:*	*exclusive:*
Snoop	*chronicle* ¬(E_2) (E_1, E_3)		*first: exclusive:*	*exclusive:*
Snoop	*chronicle* A(E_1, E_3, E_2)		*first:* ***exclusive:***	*exclusive:*
Snoop	*cumulative* ¬(E_2) (E_1, E_3)		*(?* ***first:) ext-exclusive:***	*exclusive:*
Snoop	*cumulative* A(E_1, E_3, E_2)		***first: ext-exclusive:***	*exclusive:*
ADL	**FIFO** E_3 **BETWEEN** (E_1, E_2)	(*? any ta*) (*non-overlap*)	*first: exclusive:*	*exclusive:*
ADL	**FIFO** (**NO** E_2) **BETWEEN** (E_1, E_3)	(*? any ta*) (*non-overlap*)	*first: exclusive:*	*exclusive:*
ADL	**FIFO** E_3 **NOT BETWEEN** (E_1, E_2)	(*? any ta*) ∨ (¬(E_2, E_1, E_3), ¬(E_4, E_1, E_3))[38]	*first: exclusive:*	*exclusive:*
ADL	AllStrategies E_3 **DURING** *time_span*	(*?any ta*) (*same instance* (E_1))[39]	*exclusive:*	*exclusive:*
Snoop	*recent* A(E_1, E_2, E_3)		*last: shared:*	*exclusive:*
Acood	*recent* N(E_1, E_2, E_3)	(*?any ta*)	*last: ext-exclusive:*	*exclusive:*
Snoop	*recent* ¬(E_2) (E_1, E_3)		*last:* ***ext-exclusive:***	*exclusive:*
ADL	**LIFO** E_3 **BETWEEN** (E_1, E_2)	(*?any ta*) (*non-overlap*)	*last: exclusive:*	*exclusive:*
ADL	**LIFO** (**NO** E_2) **BETWEEN** (E_1, E_3)	(*?any ta*) (*non-overlap*)	*last: exclusive:*	*exclusive:*
ADL	**LIFO** E_3 **NOT BETWEEN** (E_1, E_2)	(*?any ta*)∨ (¬(E_2, E_1, E_3), ¬(E_4, E_1, E_3))[40]	*last: exclusive:*	*exclusive:*
Acood	*cumulative* N(E_1, E_2, E_3)	(*? any ta*)	*comb min: shared:*	*shared in: exclusive out:*
Snoop	*continuous* A(E_1, E_3, E_2)		*comb min:* ***shared:***	*shared in: exclusive out:*
NAOS	E_1, !E_2, E_3		*comb min: exclusive:*	*shared in: exclusive out:*
Snoop	*continuous* ¬(E_2) (E_1, E_3)		*comb min: exclusive:*	*shared in: exclusive out:*
ADL	**TAKE_ALL** E_3 **BETWEEN** (E_1, E_2)	(*? any ta*) (*non-overlap*)	*comb min: shared:*	*shared in: exclusive out:*
ADL	**TAKE_ALL** (**NO** E_2) **BETWEEN** (E_1, E_3)	(*? any ta*) (*non-overlap*)	*comb min: shared:*	*shared in: exclusive out:*
ADL	**TAKE_ALL** E_3 **NOT BETWEEN** (E_1, E_2)	(*? any ta*) ∨ (¬(E_2, E_1, E_3), ¬(E_4, E_1, E_3))[41]	*comb min: shared:*	*shared in: exclusive out:*
Samos	**TIMES**(*n*, E_3) **IN** [E_1-E_2]	(*any ta*)	*(? first:) shared:*	*exclusive:*
Ode	! E_1	**∨(E_2,..,E_n)**	*exclusive:*	*exclusive:*

Figure 17: *The semantics of the negation operators*

Target-Model		*Meta-Model*		
Name	*ET*	*ET*	*CPET*	
		;(E1, E2)	*E1*	*E2*
HiPAC	E_1 ; E_2		*?*	*?*
Ode	*prior* (E_1 , E_2)	(***same op-context***)	*(? first:) shared:*	*exclusive:*
Ode	*relative* (E_1 , E_2)	(***same op-context***) (*non-overlap*)	*(? first:) shared:*	*exclusive:*
Samos	E_1 ; E_2 **IN** [E_1-E_3]	? ¬(E_1, E_3, E_2)	*(? first:)* ***shared:***	*exclusive:*
Acood	*chronicle* E_1 ; E_2	(*? any ta*)	*first: exclusive:*	*exclusive:*
ADL	**FIFO SOFT SEQ** (E_1, E_2)	(*? any ta*)	*first: exclusive:*	*exclusive:*
Samos	E_1 ; E_2	(*any ta*)	*first: exclusive:*	*exclusive:*
Snoop	*chronicle* E_1 ; E_2		*first: exclusive:*	*exclusive:*
ADL	**FIFO SEQ** (E_1, E_2)	(*? any ta*) (*non-overlap*)	*first: exclusive:*	*exclusive:*
Samos	*E_1 ; E_2	(*any ta*)	*first:* ***ext-exclusive:***	*exclusive:*
Acood	*recent* E_1 ; E_2	(*? any ta*)	*last: ext-exclusive:*	*exclusive:*
Samos	*last* E_1 ; E_2	(*any ta*)	*last:* ***ext-exclusive:***	*exclusive:*
Ode	*sequence* (E_1 , E_2)	(***same op-context***) (*continuous*)	*last: ext-exclusive:*	*exclusive:*

[38] The absolute temporal event type E_4 defines an event that occurs at the definition time of the modeled complex event type. The instance of E_4 is not consumed and remains valid until an instance of E_1 occurs.

[39] See footnote 30.

[40] See footnote 38.

[41] See footnote 38.

ADL	**LIFO SOFT SEQ** (E_1, E_2)	(? *any ta*)	*last: exclusive:*	*exclusive:*
ADL	**LIFO SEQ** (E_1, E_2)	(? *any ta*) (*non-overlap*)	*last: exclusive:*	*exclusive:*
Snoop	*recent* E_1 ; E_2		*last:* ***shared:***	*exclusive:*
HiPAC	$*E_1$; E_2		*cumul: (? exclusive:)*	*exclusive:*
Snoop	*cumulative* E_1 ; E_2		*cumul: exclusive:*	*exclusive:*
NAOS	E_1 , E_2		*comb min: exclusive:*	*shared in: exclusive out:*
Snoop	*continuous* E_1 ; E_2		*comb min: exclusive:*	*shared in: exclusive out:*
NAOS	E_1 ; E_2	(*non-overlap*)	*comb min: exclusive:*	*shared in: exclusive out:*
Acood	*cumulative* E_1 ; E_2	(? *any ta*)	*comb min: shared:*	*shared in: exclusive out:*
ADL	**TAKE_ALL SOFT SEQ** (E_1, E_2)	(? *any ta*)	*comb min: shared:*	*shared in: exclusive out:*
ADL	**TAKE_ALL SEQ** (E_1, E_2)	(? *any ta*) (*non-overlap*)	*comb min: shared:*	*shared in: exclusive out:*

Figure 18: *The semantics of the sequence operators (table 1)*

Target-Model		*Meta-Model*			
Name	*ET*	*ET*	*CPET*		
		;(E_1, E_2)	*E_1*	*E_2*	*E_3*
Samos	**TIMES**([n_1-n_2], E_2) **IN** [E_1-E_3]	(*any ta*)? ;(E_1, (n_1-n_2)E_2, E_3)	*(? first:) exclusive:*	*cumul: exclusive:*	*exclusive:*
Samos	**TIMES**([>n], E_2) **IN** [E_1-E_3]	(*any ta*)? ;(E_1, (n-∞)E_2, E_3)	*(? first:) exclusive:*	*cumul: exclusive:*	*exclusive:*
Acood	*recent* I(E_1, E_2, E_3)	(? *any ta*) ;(E_1, E_2, E_3)	*last: ext-exclusive:*	*cumul: exclusive:*	*exclusive:*
Snoop	*recent* A*(E_1, E_2, E_3)	∨(¬**(E_1, E_2, E_3)**, ;(E_1, E_2, E_3))	*last:* ***ext-exclusive:***	*(?cumul:) exclusive:*	*exclusive:*
Acood	*chronicle* I(E_1, E_2, E_3)	(? *any ta*) ;(E_1, E_2, E_3)	*first: ext-exclusive:*	*cumul: exclusive:*	*exclusive:*
Snoop	*cumulative* A*(E_1, E_2, E_3)	∨(¬**(E_1, E_2, E_3)**, ;(E_1, E_2, E_3))	*(? **first:) ext-exclusive:***	*cumul: exclusive:*	*exclusive:*
Snoop	*chronicle* A*(E_1, E_2, E_3)	∨(¬**(E_1, E_2, E_3)**, ;(E_1, E_2, E_3))	*first: exclusive:*	*cumul:* ***shared:***	*exclusive:*
Snoop	*continuous* A*(E_1, E_2, E_3)	∨(¬**(E_1, E_2, E_3)**, ;(E_1, E_2, E_3))	*comb min: exclusive in: exclusive out:*	*cumul: shared in: exclusive out:*	*shared in: exclusive out:*
ADL	AllStrategies E_3 **NOT DURING** *time_span*	(? *any ta*) (*same instance* (E_1))[42] ;(E_1, E_2, E_3)	*exclusive:*	*exclusive:*	*exclusive:*
Acood	*cumulative* I(E_1, E_2, E_3)	(? *any ta*) ;(E_1, E_2, E_3)	*comb min: exclusive in: shared out:*	*cumul: shared:*	*shared in: exclusive out:*

Figure 19: *The semantics of the sequence operators (table 2)*

Target-Model		*Meta-Model*		
Name	*ET*	*ET*	*CP*	*ET*
		∧(E_1, E_2)	*E_1*	*E_2*
Acood	*chronicle* $E_1 \wedge E_2$	(? *any ta*)	*first: exclusive:*	*first: exclusive:*
ADL	**FIFO** E_1 **AND** E_2	(? *any ta*)	*first: exclusive:*	*first: exclusive:*
Samos	E_1 , E_2	(*any ta*)	*first: exclusive:*	*first: exclusive:*
Snoop	*chronicle* $E_1 \Delta E_2$		*first: exclusive:*	*first: exclusive:*
Acood	*recent* $E_1 \wedge E_2$	(? *any ta*)	*last:* ***ext-exclusive:***	*last:* ***ext-exclusive:***
ADL	**LIFO** E_1 **AND** E_2	(? *any ta*)	*last: exclusive:*	*last: exclusive:*
Snoop	*recent* $E_1 \wedge E_2$		*last:* ***shared:***	*last:* ***shared:***
Snoop	*cumulative* $E_1 \Delta E_2$		*cumul: exclusive:*	*cumul: exclusive:*
NAOS	E_1 && E_2		*comb min: shared in: exclusive out:*	*comb min: shared in: exclusive out:*
Snoop	*continuous* $E_1 \Delta E_2$		*comb min: shared in: exclusive out:*	*comb min: shared in: exclusive out:*
Acood	*cumulative* $E_1 \wedge E_2$	(? *any ta*)	*comb min: shared:*	*comb min: shared:*
ADL	**TAKE_ALL** E_1 **AND** E_2	(? *any ta*)	*comb min: shared:*	*comb min: shared:*
Ode	$E_1 \wedge E_2$	(***same op-context***)	*exclusive:*	*exclusive:*

[42] See footnote 30.

		==(E_1,E_2)		
ADL	**LIFO COUNT** *n* **OF** E_1	(? *any ta*) ∧ ((*n*)E_1)	*exclusive:*	
ADL	**FIFO COUNT** *n* **OF** E_1	(? *any ta*) ∧ ((*n*)E_1)	*exclusive:*	
NAOS	*n*(E_1)	∧ ((*n*)E_1)	*exclusive:*	
NAOS	*n.*(E_1)	(*non-overlap*) ∧ ((*n*)E_1)	*exclusive:*	
ADL	**TAKE_ALL COUNT** *n* **OF** E_1	(? *any ta*) ∧ ((*n*)E_1)	*comb min: shared:*	

Figure 20: *The semantics of the conjunction operators*

6 Related work

A systematic and profound debate about the semantics of complex events is still in its early stage. Some papers treat specific aspects of a rule model in an isolated manner or concentrate on the specific model of a prototype (e.g., ACOOD ([Erik93]), ADL ([Behr94], [Behr95]), Chimera ([MePC96]), NAOS ([CoCo95], [CoCo96]), REACH ([BKZ93]), Reflex ([18]). Instance selection and consumption is mostly treated on very simple level if at all. In Snoop ([Mish91], [CKAK94]), a parameter context is introduced to deal with the instance selection aspect[43]. It determines the set of events that is bound to a complex event. SAMOS ([Gatz94]) copes with this aspect by introducing additional operators called *, *last* and *times*, which select the oldest (*), the most recent event (*last*) or several events (*times*) out of a set of events.

All in all the event algebras known from literature do not sufficiently consider important aspects as orthogonality, homogeneity, symmetry and a as small as possible set of language constructs.

Other papers, that do not directly rely on a specific model, discuss formal aspects on a more general level (e.g., [HuJa91], [Wido92], [Zani93], [GHJ+93], [FrMT94], [CFPT95], [CFPT96], [CoCo95]; for an overview see [PCFW95]). They do not or only insufficiently deal with complex event specification. To our best knowledge our paper is the first one that discusses the semantics of complex events on a general level and in deep detail.

7 Conclusion

This paper has informally presented the basic concepts of a meta-model for event algebras that defines the semantics of complex events. It is based on the three dimensions *event type pattern*, *event instance selection* and *event instance consumption*. The *event type pattern* describes the event type sequence that will trigger an event of this type, the *event instance selection* defines which event instances are bound to a complex event, and the *event instance consumption* determines what instances become invalid, i.e., cannot be considered for the detection of further complex events. While an event type pattern is associated with an event type, the dimensions event instance selection and event instance consumption are related to the component event types of complex event types.

The different dimensions of our meta-model can essentially be addressed independently. Only very few inevitable local dependencies do exist, which are mainly dependencies between modes that only occur in special combinations. In a detailed analysis of these dependencies we have shown, that every meaningful mode combination can be used thus guaranteeing orthogonality (cf. [Zimm98]). But also other relevant requirements of a good language design, as homogeneity, symmetry, and a small set of language constructs are fulfilled to a large extent. Moreover, in contrast to most other work, our model considers simultaneously occurring events.

The meta-model contributes to ongoing work in the area of active database systems in several ways:

- It provides to a solid and in-depth understanding of the basic properties of complex events.
- It offers a suitable basis for the development of consistent and easy understandable event algebras.
- It lays the basis for the detection of peculiarities in existing event algebras.
- It can be used to compare existing event algebras and to point out equivalencies and differences of an event operator defined in different event algebras.

To prove its applicability and powerfulness we specified relevant existing event algebras with the help of our meta-model. Then, we performed a comprehensive analysis and comparison of these algebras (cf. [Zimm98], [ZiUn98]). This analysis shows that all examined algebras suffer from partly severe errors and deficiencies.

The authors would like to thank C-LAB (Cooperative Computing & Communication Laboratory (Siemens Nixdorf Informationssysteme AG, University of Paderborn)) for its comprehensive support of our work.

[43] They were also introduced in revised versions of ACOOD ([Erik93]) and ADL ([Behr95])

REFERENCES

[AnHD90] T. Andrews, C. Harris and J. Duhl: *The ONTOS Object Database*. Technical Report, Ontologic, Inc., 1990.

[Anwa92] E. Anwar: *Supporting Complex Events and Rules in an OODBMS: A Seamless Approach*. Master thesis, University of Florida, 1992.

[BaDK92] F. Bancilhon, C. Delobel, P. Kanellakis (Eds.): *Building an Object-Oriented Database System - The story of O2*. Morgan Kaufmann, *1996*.

[BBKZ93] H. Branding, A. Buchmann, T. Kudrass, J. Zimmermann: *Rules in an Open System: The REACH Rule System*. In [PaWi93], pp. 111-126, 1993.

[Behr94] H. Behrends: *An Operational Semantics for the Activity Description Language ADL*. Technical Report TR-IS-AIS-94-04, Universität Oldenburg, June 1994.

[Behr95] H. Behrends: *A Description of Event Based Activities in Database Related Information Systems (in German)*, PhD-Thesis, Technical Report 3/1995, University of Oldenburg, Oct. 1995.

[BeLi92] M. Berndtsson, B. Lings : *On developing reactive object-oriented databases*. IEEE Data Engineering Bulletin, Special Issue on Active Databases, 15(4):31-34, Dec. 1992.

[Bern91] M. Berndtsson: *ACOOD: An Approach to an Active Object-Oriented DBMS*. Master thesis, University of Skövde, 1991.

[CFPT95] S. Comai, P. Fraternali, G. Psaila, L. Tanca: *A Uniform Model to Express the Behaviour of Rules with Different Semantics*. In Proc. 1st Workshop on Active and Real-Time Database Systems, Skövde, Sweden, Springer-Verlag, June 1995.

[CFPT96] S. Ceri, P. Fraternali, S. Paraboschi, L. Tanca: *Active Rule Management in Chimera*. In [WiCe96].

[CKAK93] S. Chakravarthy, V. Krishnaprasad, E. Anwar, S. Kim: *Anatomy of a Composite Event Detector*. Technical Report UF-CIS-TR-93-039, University of Florida, Dept. of Computer and Information Sciences, Dec. 1993.

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

[CoCo95] T. Coupaye, C. Collet: *Denotational Semantics for an Active Rule Execution Model*. In [Sell95], pp. 36-50, Sept. 1995.

[CoCo96] C. Collet, T. Coupaye: *Composite Events in NAOS*. In Proc. 7th DEXA, Zurich, Switzerland, pp. 475-481, Sept. 1996.

[Codd71] E. Codd: *ALPHA: A Data Base Sublanguage Founded on the Relational Calculus of the Data Base Relational Model*; In: Proc. ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego, CA; Nov. 1971.

[DaBC96] U. Dayal, A. Buchmann, S. Chakravarthy: *The HiPAC Project*. In [WiCe96].

[DaBM88] U. Dayal, A. Buchmann, D. McCarthy: *Rules are Objects Too: A Knowledge Model For An Active, Object-Oriented Database System*. In Proc. 2nd Int. Workshop on Object-Oriented Database Systems, Bad Münster, Germany, pp. 129-143, Sept. 1988.

[DaHL91] U. Dayal; M. Hsu, R. Ladin: *A Generalized Transaction Model for Long-Running Activities and Active Databases* (extended abstract); IEEE Data Engineering Bulletin; Vol. 14, No. 1; Special Issue on "Unconventional Transaction Management"; March 1991

[Daya95] U. Dayal: *Ten Years of Activity in Active Database Systems: What Have We Accomplished?*. In: *Proc. 1st Workshop on Active and Real-Time Database Systems*, Skövde, Sweden, Springer-Verlag, pages 3-22, June 1995.

[DiGG95] K. R. Dittrich, S. Gatziu, A. Geppert: *The Active Database Management System Manifesto: A Rulebase of ADBMS Features*. In [Sell95] pages 3-17, Sept. 1995.

[Elma92] A. Elmargarmid, (ed.): *Database Transaction Models for Advanced Applications*; Morgan Kaufmann Publishers; 1992

[Erik93] J. Erikson: *CEDE: Composite Event Detector In An Active Object-Oriented Database*. Master thesis, University of Skövde, 1993.

[FrMT94] P. Fraternali, D. Montesi, L. Tanca: *Active Database Semantics*. In Proc. of the 5th Australian Database Conf. (ADC), Christchurch, New Zealand, pp. 195-212, Jan. 1994.

[Gatz94] S. Gatziu: *Events in an Active, Object-Oriented Database System*. PhD-Thesis. Dr. Kovac, Nov. 1994.

[GeJa96] N. Gehani, H. Jagadish: *Active Database Facilities in Ode*. In [WiCe96].

[GeJS92] N. Gehani, H. Jagadish, O. Shmueli: *Composite Event Specification in Active Databases: Model and Implementation*. In Proc. 18th Very Large Data Bases, pp. 327-338, Oct. 1992.

[GHJ+93] S. Ghandeharizadeh, R. Hull, D. Jacobs, J. Castillo, M. Escobar-Molano, S. Lu, J. Luo, C. Tsang, G. Zhou: *On Implementing a Language*

for Specifying Active Database Execution Models. In Proc. 19th Very Large Data Bases, pp. 441-454, Oct. 1993.

[Gure94] Y. Gurevich: *Evolving Algebra 1993: Lipari Guide*. In Specification and Validation Methods, E. Börger (ed.). OUP, Oxford, 1994.

[HUJA91] R. Hull, D. Jacobs: *Language Constructs for Programming Active Databases*. In Proc. 17th Very Large Data Bases, pp. 455-467, Sept. 1991.

[Kris94] V. Krishnaprasad: *Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture and Implementation.* Master thesis, University of Florida, 1994.

[MCDA89] D. McCarthy, U. Dayal: *The Architecture Of An Active Database Management System.* In Proc. ACM SIGMOD Conference on Management of Data, S. 215-224, Juni 1989.

[MePC96] R. Meo, G. Psaila, S. Ceri: *Composite Events in Chimera*. In Proc. EDBT, Avignon, France, pp. 56-76, Mar. 1996.

[Mish91] D. Mishra: *Snoop: An Event Specification Language for Active Database Systems*. Master thesis, Univ. of Florida, 1991.

[Moss85] J. Moss: *Nested Transactions: An Approach to Reliable Computing*; MIT Report MIT-LCS-TR-260, Massachusetts Institute of Technology, Laboratory of Computer Science; 1981 and *Nested Transactions: An Approach to Reliable Distributed Computing*; The MIT Press; Research Reports and Notes, Information Systems Series; M. Lesk (ed.); 1985

[NaIb94] W. Naqvi, M. Ibrahim: *EECA: An Active Knowledge Model*. In Proc. 5th DEXA, Athens, Greece, pp. 380-389, Sept. 1994.

[PaWi93] N. Paton, M. Williams (eds.), *Rules in Database Systems (RIDS-93)*, 1. Int. Workshop, Edinburgh, Scotland, 1993.

[PCFW95] N. Paton, J. Campin, A. Fernandes, M. Williams: *Formal Specification Of Active Database Functionality: A Survey*. In [Sell95], pp. 21-35, Sept. 1995.

[Schw95] W. Schwinger: *Logical Events in ECA-Rules*. Master thesis, University of Skövde, 1995.

[Sell95] T. Sellis (Eds.), *Rules in Database Systems (RIDS-95)*, Second Int. Workshop, Athens, Greece, Sept. 1995.

[Solo92] S. Soloviev: *An Overview of Three Commercial Object-Oriented Database Management Systems: ONTOS, Object-Store and O2*. SIGMOD Record, 21(1):93-104, March 1992.

[StHP89] Stonebraker, M.; Hearst, M.; Potamianos, S.: *A Commentary on the POSTGRES Rules System*; SIGMOD Record, Vol. 18, Nr. 3; September 1989

[StRH90] M. Stonebraker, L. Rowe, M. Hirohama: *The Implementation of POSTGRES*; IEEE Transactions on Data and Knowledege Engineering; Vol. 2, No. 1; March 1990

[UnSc92] R. Unland, G. Schlageter: *A Transaction Manager Development Facility for Non-Standard Database Systems*; in: [Elma92]

[WiCe96] J.Widom, S. Ceri: *Active Database Systems.* Morgan Kaufmann, ISBN 1-55860-304-2, 1996.

[Wido92] J. Widom: *A Denotational Semantics for the Starburst Production Rule Language*, ACM Record, 21(3):4-9, Sept. 1992.

[Wido96] J. Widom: The Starburst Rule System. In [WiCe96].

[Zani93] C. Zaniolo: *A Unified Semantics for Active and Deductive Databases*. In [PaWi93], pp. 271-287, 1993.

[Zimm98] D. Zimmer: *A Meta-Model for the Definition of the Semantics of Complex Events in Active Database Management Systems (in German)*. PhD-Thesis, University of Paderborn, Shaker-Verlag, ISBN 3-8265-3744-0, 1998.

[ZiMU96] D. Zimmer, A. Meckenstock, R. Unland: Using Petri Nets for Rule Termination Analysis. In: Proc. Int. Workshop on Databases: Active and Real-Time (Concepts meet Practice) (DART'96), N. Soparkar, K. Ramamritham (Eds.), Rockville, Maryland, USA, pp. 29-32, Nov. 1996.

[ZiUM97] D. Zimmer, R. Unland, A. Meckenstock: *A General Model for Event Specification in Active Database Management Systems*. Proc. 5th Int. Conf. on Deductive and Object-Oriented Databases (DOOD 97), Switzerland, pp. 419-420, Dec. 1997

[ZiUn98] D. Zimmer, R. Unland: *A General Model for the Specification of Complex Events in Active Database Management Systems*. In: C-LAB Report 16/98, http::\\www.c-lab.de, 1998.

APPENDIX

Syntax of a complex event type in (extended) BNF-notation:

<ET-Def> ::= <Et-Id>:=<ET-P>
<Et-Id> ::= $E_{<pos\ int>}$
<ET-P> ::= <P-ET> | <C-ET>
<C-ET> ::= <pattern>
<pattern> ::= [<OP-mode>] <OP> (<para-list>)
<OP-mode> ::= [<struct-mode>] [(<tr-mode>)]
<struct-mode> ::= [(<cc-mode>)] [(<co-mode>)] [(<ctxt-mode>)]
<cc-mode> ::= *no-overlap* | *overlap*
<co-mode> ::= *continuous* | *non-continuous*
<tr-mode> ::= *left-to-right* | *right-to-left*
<ctxt-mode> ::= {<env-ctxt>}$^+$ {<op-ctxt>}$^+$ {<da-ctxt>}$^+$
<OP> ::= == | ; | ∧ | ∨ | ¬

<para-list> ::= <CP-ET> | <CP-ET>, <para-list>
<CP-ET> ::= [<ei-mode>] <ET-P> |
[<ei-mode>] <Et-Id>
<ei-mode> ::= [<select>:] [<consum>:] [<delimiter>]
<env-ctxt> ::= [(<relation> <environ>)]
<relation> ::= *same* | *different* | *any*
<environ> ::= *ta* | *proc* | *appl* | *user*
<op-ctxt> ::= [(<relation> <operation>)]
<operation> ::= *op-context* | *op-id* | *op-param*
<da-ctxt> ::= [(<relation> <data>)]
<data> ::= *data-type* | *data-id* | *data-state*
<select> ::= *first* | *last* | *cumul* | *ext-cumul* | *comb* [*min*]
<consum> ::= *in* <cos-mode>: *out* <cos-mode> |
<cos-mode>
<cos-mode> ::= *exclusive* | *shared* | *ext-exclusive*
<delimiter> ::= (<pos int>) | (1-∞) |
(<lower lim>-<upper lim>)
<lower lim> ::= <pos int> | 1
<upper lim> ::= <pos int> | ∞
<pos int> ::= {<pos num>}$^{+}$
<pos num> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Each (complex) event type definition (ET-Def) consists of an assignment of a unique identifier (Et-Id) and the specification of its event type pattern (ET-P). An event type is either primitive (P-ET) or complex (C-ET). The definition of a complex event type reflects the three dimensions of its semantics: The *event type pattern* is defined by an *op*erator (*op*), an *op*erator *mode* (*OP-mode*) which delivers more specific information about the pattern structure, and a list of parameters (*para-list*) which, in fact, reflect the (complex) component event types (CP-ET). If necessary a component event type is supplemented by a delimiter.

The *struct*ure-*mode* permits to define the structure of the pattern on a more specific level; i.e., whether non-relevant event instances are allowed to occur between relevant instances (*co-mode*), whether the time intervals of (component) instances are allowed to overlap (*cc-mode*), and whether the event instances have to occur in a specific context (*ctxt-mode*).

Event instance selection is specified by the event instance *select*ion mode (*select*) and the *t*raversal-*mode* (*tr-mode*).

Finally, *event instance consumption* is defined by the mode *consum*ption (*consum*).