Events in an Active Object-Oriented Database System[1]

Stella Gatziu
Klaus R. Dittrich
Institut für Informatik, Universität Zürich
Zürich, Switzerland

Abstract

In this paper we investigate the definition, detection, and management of events in the active object-oriented database system SAMOS. First, we present various event specification facilities based on simple but nevertheless powerful constructs which support the modelling of time aspects as well. Second we show how events can be detected in an efficient way. Finally, we deal with the internal representation of events using the benefits of the underlying data model.

1 Introduction

Most new developments in database technology aim at representing more real-world semantics in the database which would otherwise be hidden in applications. *Active* database systems (aDBS) are able to recognize specific situations (in the database and beyond) and to react to them without immediate, explicit user or application requests. In addition to the functionalities supported by a passive DBS, an aDBS registers *events, conditions, actions* and the association between them by means of *ECA-rules* (Event-Condition-Action) introduced in [3]. An event indicates a point in time when the DBMS has to react, and a condition relates to a database state; it has to be evaluated when the occurrence of the corresponding event is *signalled*. If the condition holds, the associated action has to be executed. An action can be any executable program including database operations.

Events are one of the most essential issue in an aDBS, and thus, their definition, their detection and their (internal) representation address a main research area in active database systems. Looking at existing work, many questions still remain open or are not yet answered satisfactorily:

- An event can always be regarded as a specific point in time [6]. Because only some points in time for which a reaction is required are of interest, we have to specify them in some way, e.g., as an explicit time definition (at 18:00) or as the beginning or the end of a database operation. Events may be defined in even more complex ways, e.g., as the point in time when the last of a set of events has occurred (*complex events*). Thus, an aDBS has to provide various constructs for the specification of interesting events. Most of the existing systems are restricted to “simple” event definitions; only recent work in ODE [11] and Sentinel [2] deals with advanced event specification. Nevertheless, no simple but powerful constructs exist for the specification of events. An additional open question is how time aspects can be integrated into event definitions.
- An aDBS has to detect the occurrence of all defined events in order to be able to react to them. The so-called *event detector* is responsible for that task, and therefore, one of the main system components of an aDBS. The support of various constructs for modelling a wide range of real-world situations as events requires a mechanism for the detection of *all* these events in an efficient way.
- Defined events have to be represented in the system such that information about them can be retrieved and accessed in an efficient way. The event management may benefit from the underlying data model in that concepts already offered by the data model are used, i.e., data and events are represented using the same constructs. In the recent past, several proposals take into account object-oriented features [1, 4, 5, 10]. Additionally, the user should not be concerned with the way events are represented in the system.

1. to appear, Proc. of the 1st Intl. Workshop on Rules in Database Systems, Edinburg, August 93

In this paper we elaborate on these aspects in the context of SAMOS (Swiss Active Mechanism-Based Object-Oriented Database System). The combination of active and object-oriented characteristics within one, coherent system is the overall goal of SAMOS. It addresses the three principal problems of an aDBS, namely rule specification, rule execution, and rule management. Rule specification is concerned with the nature of events, conditions and actions and their relationships to the data model. Rule execution refers to the processing of rules, which has to be carried out in the context of the general transaction model supported by the database system (some aspects are presented in [7]). Rule management incorporates tasks for the internal representation of rules and events, for the event detection and the selection of all rules that have to be executed. A short general description of SAMOS can be found in [8].

In this paper, we focus on rule specification and rule management. In particular, we introduce powerful yet rather simple constructs for the specification of events based on an event algebra. Furthermore, we integrate time specification facilities into event definitions. We show that even complex defined events can be detected in a relatively easy way using Petri nets. Lastly, we use the benefits of the object-oriented environment for the internal representation of events, e.g., by exploiting inheritance hierarchies.

The remainder of this paper is organized as follows. Section 2 presents the way a rule or an event is defined, and section 3 discusses the various kinds of events supported by SAMOS. In section 4 we deal generally with implementation aspects and in particular with those aspects concerning events, e.g., the event detectors.

2 Rule Definition Language

In analogy to the data definition language (DDL) which supports the modelling of data structures (and behavior), SAMOS supports a *rule definition language* as a means to specify ECA-rules. The syntax[2] of the definition of a rule is:

```
DEFINE RULE rule_name
  ON event_clause IF condition_clause DO action_clause
```

In this rule definition, events are defined as a part of a rule. However, events will often be used in multiple rules. As a matter of reusability and convenience, it is often undesirable to describe the same event (especially a complex one) in each rule. Therefore, events may be named and defined separately (outside the rule definition):

```
DEFINE EVENT event_name event_clause
```

Corresponding rules can then refer to the event via its name:

```
DEFINE RULE rule_name
  ON event_name IF condition_clause DO action_clause
```

The event specification part of the rule definition language includes all constructs that can be used for the specification of events. Actually, we use the notion of “event” as the point in time when the system has to react to an occurrence within the database or its environment. Those occurrences are described within the event definition. In order to distinguish between event occurrence and event definition, we use the notion of *event pattern* to denote the definition of an event. Then, an *event* corresponds to an actual *occurrence* of an event pattern, and more than one event can relate to one event pattern. In the sequel, we will simply talk about events whenever the distinction is clear from the context.

Rule and event definitions have to be “understood” by SAMOS; they are thus analyzed and transformed into an internal form by means of a rule compiler. Compiled rules and events are made a persistent part of the database and build the *rule-* and the *eventbase*, respectively. Of course, like the database the rule- and the eventbase may change over time. In an object-oriented environment, the rule- and the eventbase are represented in an object-oriented way. Thus, events and rules are treated as objects and every user-defined event can be represented as an instance of a system-defined class *event_pattern*.

One important aspect of our approach is that the user is not concerned with the way events are represented by the system, e.g., whether they are objects or class attributes, whereas he has to care for these issues in Sentinel [1] and in ODE [10]. The user only defines event patterns using a high level

2. We use the following conventions: “|” denotes alternatives and an expression enclosed in square brackets is optional. Terminal symbols are printed in capital letter.

specification language and the system translates the event definitions and stores them into event objects. In the sequel, we present in turn the way the user defines event patterns, and the way the system deals with those information.

3 Event Specification

A specific contribution of SAMOS is its extensive collection of event specification features. Events can be subdivided into two categories: *primitive* events which correspond to elementary occurrences, and *composite* events (read: events that are described as *compositions* of others) that are described by an *event algebra*. An expression of the event algebra is composed out of other composite or primitive events based on *event constructors*. In this section, we introduce the event specification part of the rule language and present the choices offered to users for the definition of primitive and composite event patterns.

3.1 Primitive Events

A primitive event describes a point in time specified by an occurrence in the database (*method* events), in the DBMS (*transaction* events), or in its environment (*time* and *abstract* events).

3.1.1 Time Events

First of all, an event can be specified as an explicit point in time. These events are called *time* events. They can be defined as absolute (e.g., `93.06.28,22:00`)[3] or as periodically reappearing points in time. The syntax of the latter is:

```
EVERY frequency(YEAR|MONTH|WEEKDAY|HOUR|MINUTE) time WITHIN interval
```

The use of the repeating factor *frequency* is demonstrated by the example of the time event `EVERY 5 DAY 20:00` which is signalled only every fifth day. The optional part *interval* represents the time interval in which the periodical time event should occur. For example, the time event `EVERY 5 DAY 20:00 WITHIN [05.01-05.31]` is signalled every fifth day in May.

In some cases, a time event cannot be defined as an absolute point in time but only in relation to an *implicit* one that represents the point in time when a system activity is performed. To that end, we provide *relative time events (t+x)*, where *t* is an implicit time point and *x* is a relative time (e.g., `1 min`). The possibilities for the definition of time intervals and of implicit time points are presented in subsection 3.4.

3.1.2 Method Events

In an object-oriented environment, users manipulate and access objects by sending messages to them. Thus, each message sent to an object requiring the execution of a method gives rise for two events (*method* events): the points in time when the object begins, and when it has finished the execution of the method. Method events are defined according to the syntax

```
(BEFORE|AFTER)"."(class_name|object_name|"*")"."method_name
```

A method event can be related either

- to one class, if a class name is given; it is signalled before or after the execution of the appropriate method on any arbitrary object of this class
- to a particular object, if an object name is given (we assume that unique names are assigned to objects); it is signalled before or after the execution of the appropriate method on this object
- to multiple classes, in case of a “*”; it is signalled when the method is executed on any arbitrary object of any class that has a method with the specified method name (this situation may occur due to the inheritance of methods).

3. Within the definition of absolute time points, default values are supported if some parts of a concrete time specification are missing. For example, without giving the year, the actual year is assumed.

3.1.3 Transaction Events

Transaction events are defined by the start or the termination time of (user-defined) transactions. Assuming that transaction programs are named, a transaction event can be restricted to transactions executing this particular transaction program. Transaction events are defined according to the syntax:

```
(BOT|EOT|ABORT)[transaction_name]
```

3.1.4 Abstract Events

Up to now, we have introduced several kinds of events which are conveying specific semantics known by SAMOS such that their occurrence can be detected by the system itself. However, users and applications may need other events according to their specific semantics as well (*abstract* events). They are defined and named by users:

```
DEFINE EVENT event_name
```

and can be used in rules like any other event. They are not detected by SAMOS, but users/applications have to notify the system about their occurrence by issuing the explicit operation:

```
RAISE EVENT event_name
```

3.2 Composite Events

Primitive events as presented above describe only elementary occurrences. In order to model even complex occurrences as events, we support composite events built from others by means of six event constructors. The *disjunction* of events `(E1|E2)` occurs when either `E1` or `E2` occurs. The *conjunction* of events `(E1,E2)` occurs when both `E1` and `E2` have occurred, regardless of order. A *sequence* of events `(E1;E2)` occurs when first `E1` and afterwards `E2` occurs. `E1` and `E2` can be any primitive event discussed above as well as any composite event.

Composite events defined with the following three constructors are signalled depending on how many events of a specific event pattern have occurred during a predefined time interval. A composite event with the "*"-constructor `(*E IN I)` will be signalled only once (after the first occurrence of `E`), even if more events of `E` occur during the time interval `I`. A *history* event `(TIMES(n,E) IN I)` is signalled *each time* `n` events of `E` have occurred during the time interval `I`. A *negative* event `(NOT E IN I)` occurs if `E` *did not* occur in the time interval `I`. Only negative events always require the definition of a time interval which muss not have a non-finite end. For history events and events defined with the "*"-constructor where no time interval is specified, we assume the time between the event definition and infinity.

3.3 Event Parameters

Event patterns can be parameterized such that information can be passed to the condition or action parts, if necessary. The actual parameters are bound to the formal ones of an event pattern during the signalling of the event. The set of formal parameters is fixed (except for abstract events), i.e., the user cannot define event parameters themselves. We distinguish between *environment* parameters and parameters depending on the kind of the event. Environment parameters are:

- *occ_point*(`E`): the point in time of the occurrence of an event `E`,
- *occ_tid*(`E`): the identifier of the transaction in which event `E` has occurred (*occurring transaction),*
- *user_id*(`E`): the identifier of the user who has started the occurring transaction of `E`.

Occ_point is defined for composite events only, while for primitive events, all environment parameter apply.

Concerning parameters depending on the event kind, method events have in addition the parameters of the method and the object identifier of the object executing the method. A disjunction has the parameters of the event that causes the signalling of the composite one. Events with the "*"-operator have the parameters of the first occurring event. A sequence or a conjunction has the union of the parameters of their components, while history events have the union of the parameters of the *n* occurring events. Note that time and negative events have no parameters.

3.3.1 Influence of Event Parameters on the Definition of Composite Events

The definition of composite events can be extended with the keyword *same* to denote that the component events have the same value of a particular parameter. In case of negation and disjunction, none or only one component event can occur and hence, a comparison of parameters would be meaningless. For all other compositions, we can define the *same* keyword in relation to the environment parameter *occ_tid* and *user_tid*. For example, the sequence `(E1;E2):same transaction` is signalled when `E1` and afterwards `E2` have occurred within the same transaction (the occurring transaction of `E1`). The history event `TIMES(5,E):same user` occurs if `5` events of the event pattern `E` have occurred in transactions started by the same user.

Concerning method events (in any composition), we may require that the appropriate methods are executed on the same object. For example, assume the sequence `(E1;E2):same object` where `E1` and `E2` are method events related to a class `X`. Then the sequence is signalled only if the corresponding methods are executed on the same object of class `X`.

For composite events defined with history and "*"-constructors, it makes sense also to monitor the repeated occurrence of events of a specific event pattern, in case of events having the same parameter. For example, the event definition `*E:same parameter` means that all occurrences of `E` having all of their parameters apart from *occ_point* identical to the parameters of the first occurrence of `E` are ignored, but no `E`-occurrences with different parameters.

Checking of parameters may also take place in the condition of a rule. Such a rule definition is however semantically different from a definition of a rule with a composite event with the *same* keyword. For example, assume the order of event occurrences $E1^1(x)$ $E2^1(y)$ $E1^2(z)$ $E2^2(x)$ where the parameter in brackets refers to the *user_id* and the index relates to the first or second occurrence of an event of a specific event pattern, and consider the following two rule definitions:

```
DEFINE RULE R1 ON (E1;E2):same user IF ... DO ...

DEFINE RULE R2 ON (E1;E2) IF <user_id is the same> DO ...
```

Rule `R1` is executed after $E1^1$ and $E2^2$. Rule `R2` is executed twice: first after $E1^1$ and $E2^1$ and second after $E1^2$ and $E2^2$. In both cases, however the condition would not hold. For that matter, the two rule definitions model a different real-world semantics.

3.4 Monitoring Intervals

Sometimes it is required that a (primitive or composite) event `E` is signalled only in case it has occurred during a specific time interval `I`. Especially, negative events require a time interval, while for events defined by means of the history and the "*"-constructor a time interval is usually at least desirable. Therefore, *monitoring intervals* are introduced for those time intervals during which the event has to occur in order to be considered relevant. Before we present the monitoring intervals, we show how a time interval in general is defined in SAMOS.

3.4.1 Defining Time Intervals

Generally, a time interval is specified by two points in time, a *start_time* and an *end_time*. Furthermore, time intervals can be defined to reappear periodically, e.g., `EVERY MONTH [15.,18:00-15.,24:00]`. They can also be computed from other time intervals: we provide operators *overlap* and *extend* to represent the intersection and the union of intervals, respectively.

S*tart_time* and *end_time* can be defined explicitly as absolute points in time, e.g., `93.10.15, 12:15`. The keywords "now" and "infinite" in definitions like `[now-93.06.30]` or `[93.02.01-infinite]` express the actual time (i.e., the point in time when the interval is defined) and the infinity, respectively. Note that for negative events the definition of the *end_point* as infinite is not allowed.

Still, the desired points in time may not be known at definition of the time interval. In this case, they can often be defined as the point in time denoting the occurrence of some other event (i.e., according to its *occ_point*) or the completion of the execution of a rule[4]. In these cases, we refer to *implicit* points in time. They can be used for the definition of time intervals and for the definition of relative events (see subsection 3.1) as well. For example, the time event `occ_point(E1)+00:10` is signalled `10` minutes after the occurrence of `E1`.

3.4.2 Defining Monitoring Intervals

A monitoring interval can now be defined during event definitions, e.g.,

4. This corresponds to the `occ_point` of an abstract event which models the point in time when the last statement of the rule's action is successfully executed.

```
DEFINE EVENT E1 AFTER.*.UpdateSalary IN [93.05.01-93.08.30]
DEFINE RULE R1
  ON E1 IF ...
```

or

```
DEFINE RULE R2
  ON BOT(Program_test) IN EVERY WEEK [Sa-So] IF ...
```

In the first case, the monitoring interval has to be considered for the execution of all rules that refer to the method event `E1` and for all composite events that contain this event. In the second case, the monitoring interval concerns only rule `R2` and has no influence on composite events.

3.4.3 *Examples*

- `E` muss not occur again during two days in August after its first occurrence:
  `NOT E IN overlap([93.08.01-93.08.31],[occ_point(E)+2DAYS)])`
- `E2` has to occur within one hour after the occurrence of `E1`:
  `(E1 ; E2 IN [occ_point(E1)+01:00])`

3.4.4 *Monitoring Intervals and (De)Activate Operations*

As discussed above, monitoring intervals enable the timely restriction of event signalling with regard to rule execution, i.e., a rule is not always executed even if the appropriate event has occurred. An additional mechanism to make rules eligible or ineligible to be executed are the operations *activate* and *deactivate* on rules. They can be called in programs and inside the action part of another rule. In SAMOS, we provide both monitoring intervals and the operations *(de)activate* in order to achieve the dynamical change of the behavior of a rule inside the action of another rule. Now we give an example to show how these two mechanisms work together:

```
DEFINE EVENT E1 Program_test IN [93.05.01-93.05.31]
DEFINE RULE R1
  ON E1
  ...

DEFINE RULE R2
  ON E2 IF ... DO deactivate R1
```

We assume that `R1` is in an active state. Then it can be executed when the event `E1` is signalled in May and the rule `R2` has not yet been executed.

4 Implementation Issues

In addition to the usual functionalities of (passive) DBMS, an active DBMS has to perform tasks like the analyzation and management of rules and the efficient detection of events. Thus, the implementation of an aDBS requires an answer to the following questions:

a. what happens after the definition of a rule or an event pattern?
b. what happens after a primitive event occurs?

Figure 1 illustrates how SAMOS answers these two questions. The left side of the figure shows what kind of structures are created for handling the information gained from the rule or event definition. The right side of the figure shows how these information is used in order to react to the occurrence of a primitive event. We discuss details in the sequel.

After a rule (or an event pattern) is defined, the *analyzer* sends the information about the correct rule and event definitions to the *rule* and *event manager*, respectively. They have to store these information such that it can be retrieved and accessed efficiently. Furthermore, for each defined event the *event detector* has to be "programmed" (see section 4.1 for details).

After the event detector signals the occurrence of a primitive event, whether this event is used in an event composition, the component responsible for the detection of composite events has to be informed. Furthermore, it has to be checked whether at least one rule is defined for the appropriate event pattern; because events occur frequently this search is required frequently as well, thus it has be done efficiently. If such a rule exists, this event is inserted into the *event register* which includes all "events of interest", i.e., all events whose patterns are used in a rule. Based on the (updated) information in the event register, the *rule execution component* has to determine the appropriate rule(s) to be executed. Afterwards, in cooperation with the transaction manager of the underlying DBMS, the rule execution begins (condition evaluation and action execution).

Therefore, in order to implement an aDBS, the architecture of a (passive) DBMS has to be augmented by new components like an analyzer, a rule and an event manager, an event detector for primitive and composite events and a rule execution component. The prototype implementation of SAMOS is based on the commercial ooDBS ObjectStore. Since ObjectStore is a "black box" for our implementation, the new components are located on top of it. In this paper, we concentrate on the detection and the internal representation of events.

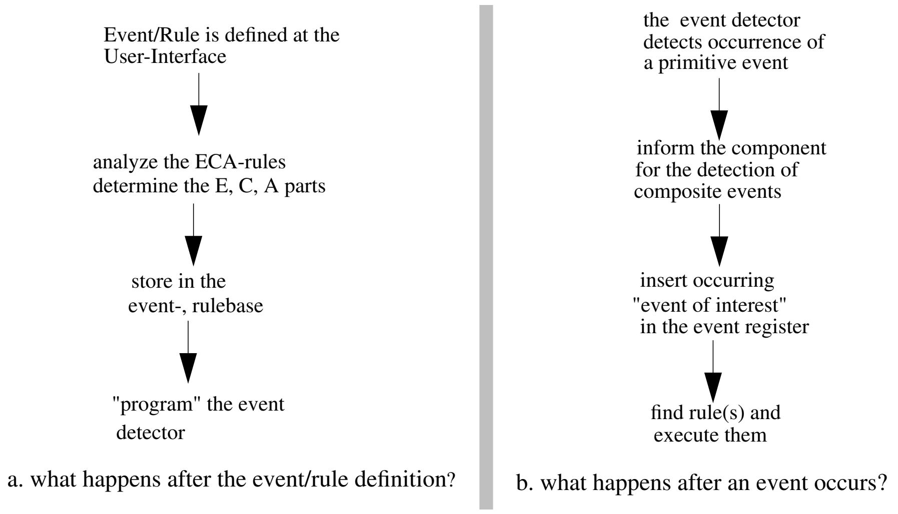

Figure 1. Control flow of tasks in SAMOS

4.1 Event Detection

The system (e.g., the event register) "knows" about the occurrence of an event only when it has received the message *raise event*. This message is sent from the event detector except for abstract events where it is sent directly by users, applications or action parts of a rule. Consequently, all other primitive events and all composite ones have to be detected by the system.

4.1.1 Detecting Primitive Events

Each kind of the primitive events requires a different way to detect it. For method events, one possible approach is to replace the original method (internally) by a new one that contains a call of the operation *raise event* and a call of the original method (the order depends on the kind of the method event relating to BEFORE or AFTER keywords). The new method is called a *method wrapper*. This approach is feasible in Smalltalk-based DBMS (e.g., Gemstone) where the method wrapper can be compiled and then replace the original method *at runtime* [13].

Nevertheless, ObjectStore is based on C++ and, therefore, does not support modification and recompilation of methods at runtime. Hence, we choose the following approach to signal method events without user intervention: the body of these methods for which a method event is defined is modified by inserting a call of the operation *raise event*; i.e., the method implementation has to be parsed, modified and then recompiled.

In ObjectStore, all transactions in progress are instances of the class *os_transaction*. The operations *abort*, *commit* and *begin* on transactions are implemented as methods of this class. Thus, the detection of the appropriate transaction events can be done by overriding those methods in order to include a call of the *raise event* operation for the appropriate transaction events.

Time events are detected by monitoring the system clock. Concretely, we make use of the UNIX system command *cron* which executes commands (in our case calls of the operation *raise event*) at specified dates and times. The commands have to be inserted into a *crontab* file. This applies to the absolute and to the periodical time events except those defined with a frequency and a time interval. To detect the latter ones, we have to insert the next one into the *crontab* file every time a periodical event is signalled.

4.1.2 Detecting Composite Events Using Petri Nets

In SAMOS, we use *Petri nets* (PN) [15] to model and to detect composite events. Incidentally, Petri nets have been used for active databases for the modelling of the active behavior during the database design process [14]. In the context of SAMOS, we define a PN as a tuple (N, M_0) where N represents the (static) structure of the PN and M_0 is the *initial marking* of the PN.

N is a 5-tuple (P, T, A, P_s, P_e) where P is a finite set of *places*, T is a finite set of *transitions*, and $A \subset (P \times T \cup T \times P)$ is a finite set of *arcs* denoting the connection between places and transitions. For each transition t we define its *input places* ${}^{\bullet}t = \{p \mid (p,t) \in A\}$ and its *output places* $t^{\bullet} = \{p \mid (t,p) \in A\}$. $P_s \subset P$ is the set of all places that model event patterns and that are marked as soon as the corresponding event has occurred. These places are termed *input places* of the PN, since all other places can only be marked upon the firing of transitions. $P_e \subset P$ is the finite set of all places that model composite event patterns. These places are termed *output places* of the PN, since an event of the composite event pattern has occurred when one of them is marked.

In an actual state, a PN has a non-negative number of *tokens* assigned to places. The number of tokens per place is defined by a *marking* M, i.e., a mapping $M: P \to \mathcal{N} \cup \{0\}$. M_0 is the *initial marking* of the PN; it defines the number of tokens for auxiliary places. Every time an input place of the PN is marked it has to be checked whether the corresponding transition(s) can fire. Let M be a marking. Then, a transition t can *fire* if (and only if) $\forall p \in {}^{\bullet}t: M(p) \geq 1$, i.e., all of its input places are marked. The firing of a transition leads to the marking of all its output places. Thus, when t fires, the subsequent marking M' satisfies the following equation:

$$M'(p) = \begin{cases} M(p) + 1 & \forall p \in t^{\bullet} \\ M(p) - 1 & \forall p \in {}^{\bullet}t \\ M(p) & \text{otherwise} \end{cases} \qquad \text{(Equation 1)}$$

Event patterns can have parameters, and thus, the information about actual parameters has to be treated as a property of places. To that end, *individual tokens* are used to represent actual parameters and can be assigned to places. After the firing of a transition t all output places have to be marked, i.e. the value of the tokens of all input places of t flow into the (individual) tokens of the output places of t. This means that the information about the actual parameter of an event flows through the PN. The way this flow happens is based on the principles introduced in subsection 3.3. The firing rule of the PN taking into account individual tokens is different from the firing rule illustrated in Equation 1. These aspects can be found in [9].

A Petri net pattern is defined for each constructor of the event algebra. As an example, figure 2 shows the Petri net pattern for a sequence. Note that in this case $E1 \in P_s$ and $E2 \in P_s$ holds, while $(E1;E2) \in P_e$. The firing of $t3$ leads to the marking of the place $(E1;E2)$ which corresponds to the signalling of the sequence event. The place H is an auxiliary place and serves to ignore all occurrences of $E2$ before the occurrence of $E1$.

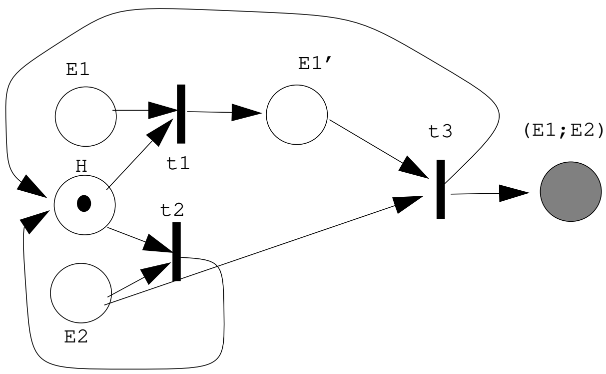

Figure 2. The Petri net pattern for a sequence

Every time the user defines a composite event pattern using one of the event constructors, the system has to create an appropriate Petri net. For that matter, it retrieves the corresponding Petri net pattern and instantiates a new PN that uses the structure of the pattern. The names of the places of the new PN denote the components of the composite event. If these components in turn are composite events the retrieval and instantiation process continues recursively.

As soon as the event detector signals the occurrence of a primitive event, the corresponding input place of the Petri net is marked. Then, the Petri net possibly fires transitions and an output place of the PN can be marked. In this case, the composite event (signalled by marking this output place) is inserted into the event register, together with the information (actual parameters) included in the respective (individual) token. The following example shows how a specific Petri net for a sequence works. Figure 3 shows which places are marked and which transitions fire when an event occurs.

	E1	E2	H	E1'	(E1;E2)
E2 occurs :	–	*	*	–	–
t2 fires	–	–	*	–	–
E1 occurs :	*	–	*	–	–
t1 fires	–	–	–	*	–
E1 occurs :	*	–	–	*	–
E2 occurs :	*	*	–	*	–
t3 fires	*	–	*	–	*
t1 fires	–	–	–	*	–
E2 occurs :	–	*	–	*	–
t3 fires	–	–	*	–	*

Figure 3. An Example

Instead of modelling all composite events by independent PNs, we combine them into one *combined* PN whenever possible. Such a combination is applied in the following cases:

1. An event participates in *more than one* composition, e.g., E1 in E=(E1;E2) and in EE=(E1,E3). In this case, the PNs for the two composite events are combined into one. While in the independent existence of Petri nets more places (one for each composition) for the event E1 exist, only one place is required in the combined Petri net. Thus, every time E1 occurs, only one place has to be marked. Figure 4 illustrates how E and EE are combined into one Petri net. The place E1 is duplicated into E1' and E1''. The Petri net parts for the two composite events E and EE use the input places E1' and E1'', respectively.

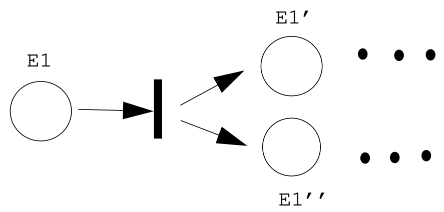

Figure 4. Building a Petri net for two composite events with the same component event

2. A composite event takes part in *further* composite events, e.g., E=(E1,E2) and EE=(E|E3). Then, the output place modelling the composite event E is an input place for the Petri net modelling the composite event EE. Figure 5 illustrates the Petri net for E and EE, it shows also how a conjunction and a disjunction are modelled as Petri nets.

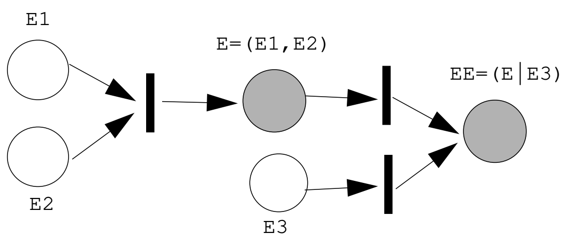

Figure 5. Building a Petri net for two composite events where one of them is a component event of the other one

Finally, the entire event detector for composite events is a *combination Petri net* (cPN) which consists of all independent and all combined Petri nets.

The use of Petri nets for the detection of composite events has several advantages:

- Composite events can be detected step by step, after each occurrence of a primitive event, and, hence, do not require the inspection of a large set of (primitive) events stored in the event register whenever new events are inserted there.
- Petri nets allow the modelling of the semantics of all event constructors in an abstract way and, at the same time, the implementation of the event detector for all defined event patterns as well.
- Petri nets allow for modelling the semantics of *all* constructors used for the specification of events, though some slight extensions are needed:
  - I. Because an event `E` with the monitoring interval `[s-e]` can be transformed into the sequence `(TS;(NOT TE;E))`, where `TS` and `TE` are time events denoting the *start_point* `s` and *end_point* `e` of the monitoring interval, respectively, events with monitoring intervals can also be modelled as Petri nets.
  - II. Composite events with the *same* keyword can be modelled as Petri nets as well. To that end we have to compare the parameters, i.e, the value of the corresponding individual tokens. This corresponds to a new firing rule in Equation 1.

4.1.3 Internal Representation of Petri Nets

The system maintains a data structure for the internal representation of Petri nets including a *connection matrix* (places **x** transitions) and a *place vector*. The matrix is used for the representation of the connections between places and transitions. For a connection between an input place p_i and a transition t_j, the `(i,j)` element of the connection matrix has the value 1. For a connection between a transition t_l and an output place p_k, the `(k,l)`-element of the matrix has the value -1. All other elements of the matrix have the value 0. Each element of the place vector relates to a place of the connection matrix and contains the number of tokens and possibly the value of individual tokens. For the sake of simplicity, the algorithm described below (which is actually implemented in our current prototype) does not consider individual tokens.

Based on the data structure

```
N : ARRAY[1..P,1..T]
m : ARRAY[1..P]
```

the algorithm implementing the firing rules of Petri nets in SAMOS after the occurrence of the event `E` is as following:

```
firing:=FALSE
mplace:=find(E)  /* find the element of the vector m representing the place
                    modelling this event E */
m[mplace]:=m[mplace]+1  /* increase the number of tokens */
REPEAT
FOR j:=1 TO T DO
      IF N[mplace,j] < 0 THEN     /* find all transitions which have as input
                                     place the mplace */
          IF  N[ ,j] + m > 0 THEN  /* try to fire each transition with mplace as
                                      input place by adding the j-th column of
                                      N (= N[ ,j])  to the vector m. If all input
                                      places are marked, the vector-addition
                                      results in a vector with positive
                                      elements */
              m:=m + N[ ,j]
              firing:=TRUE
          FOR k:=1 to S DO
              IF N[k,j] > 0  /* find the output place of the fired transition */
              THEN mplace:=k
              ELSE firing:=FALSE
UNTIL NOT firing
```

4.2 Event Manager

The event manager has to represent the information gained from the analysis of event definitions, i.e, it is responsible for managing the eventbase which consists of all defined events patterns. The structure of the eventbase is described by the *event schema*. In accordance to the idea “to stay in the same world and exploit its advantages”, we built the event schema by means of object-oriented features.

Event patterns are represented as objects and therefore for such an object, all object-oriented characteristics are available for those meta data. For example, event patterns have an identity and can be manipulated and accessed like any other object by means of methods. Especially, taking inheritance into account, the event schema is described by the class hierarchy illustrated in figure 6, where subclasses of a "general" class *event_pattern* exist for groups of event patterns with the same kind (e.g., for method events). Thus, each defined event pattern is internally represented as an instance of the appropriate subclass. We point out some classes to give further explanations. Note that in this paper we restrict ourselves only to the structure and do not present the behavior of the event hierarchy.

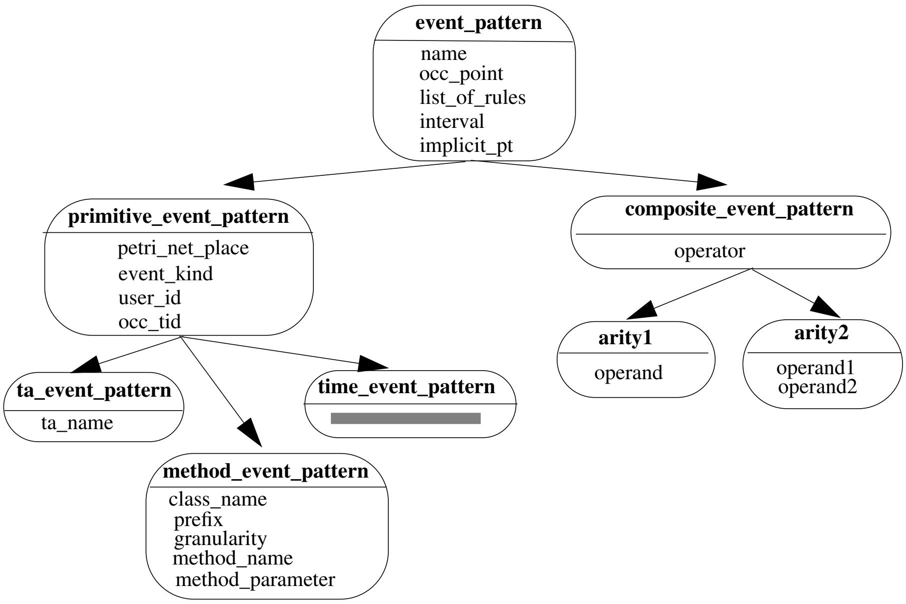

Figure 6. Event-Hierachy

The *event_pattern* superclass provides the common structure shared by all kinds of events and comprises attributes like *name* as the event name (user or system given), the *occ_point*[5] and the *list_of_rules* as the list of rules which refer to the appropriate event and have to be executed after this event has occurred. The list of rules is split into activated and deactivated rules (according to *activate* and *deactivate* operations). To denote the connection between an event pattern and a monitoring interval, we make use of the attribute *interval* which represents a reference to an object of the class *interval*[6], which has all defined time intervals as instances. Points in time can be defined implicitly in relation to the occurrence of an event. Therefore, the attribute *implicit_pt* is a reference to an object of the class *interval* which is defined implicitly in relation to this event pattern.

The subclass *primitive_event_pattern* has additional attributes *event_kind* ("method", "time", "transaction" or "abstract")[7] and *petri_net_place*. The latter contains the index of the field of the place vector which contains the tokens of the place modelling this primitive event. In this way, after the primitive event occurred, we find the place of the Petri net that has to be marked on the basis of the value of the attribute *petri_net_place* (i.e., the implementation of the `find` operation of the firing algorithm is easy).

The subclass *method_event_pattern* has as attribute a list of the name of the classes for which this method is applicable (a method event can be related to one or multiple classes) and a *prefix* ("before" or "after") depending on the kind of the method event relating to BEFORE or AFTER keywords. Because a method event relates to either a class or to an object, the attribute *granularity* determines which case applies and has as its value either "all" or the object name.

Note that for abstract events we do not need to create a separate subclass of the class *event_pattern* because they are represented as instances of the class *primitive_event_pattern* itself.

The subclass *composite_event_pattern* has an attribute *operator* to represent the event constructor. The component events are represented as attributes of the subclasses *arity1* and *arity2* which contains references to the appropriate events. The parameters of composite events are not explicitly

5. Because events have different parameters according to their kind, we introduce those in the appropriate subclass only.
6. The structure of the class *interval* and the subclass *time events* is rather complex and is not described in this paper.
7. The kind of a primitive event can be determined after analyzing the event definition. For the sake of simplicity, however, it can also be given as part of the event definition. We have chosen the second approach.

represented in the event structure because they are calculated during the firing of the Petri net based on the parameter of the components events.

Finally, figure 7 shows the evaluation of tasks in SAMOS. This figure applies figure 1 by using the knowledge introduced in the previous subsections.

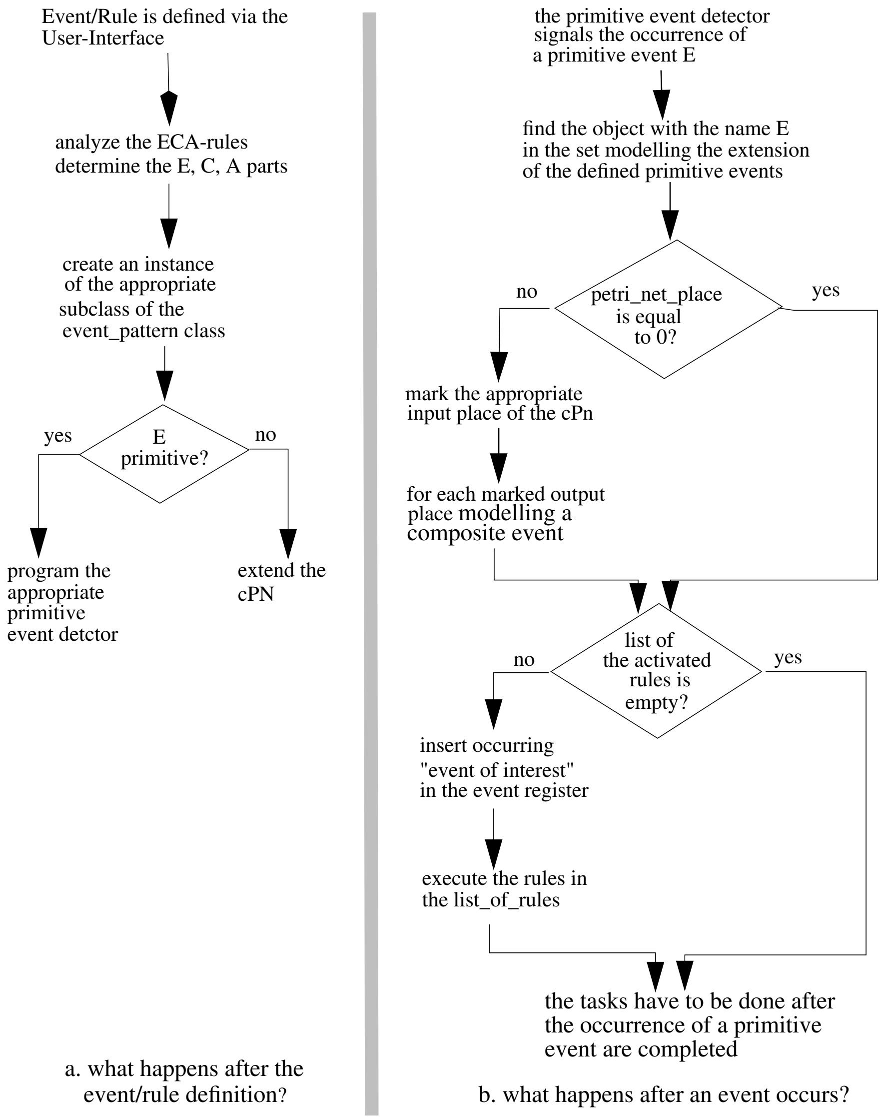

Figure 7. Control flow of tasks in SAMOS

5 Conclusions

In this paper, we have investigated how events are defined, detected and managed in the context of the active object-oriented database system SAMOS we are currently implementing.

Rules and events are defined using a rule definition language. For the sake of ease of use, the event specification part of the language is simple, but nevertheless powerful. We achieved this goal by supporting only *six* event constructors for the definition of composite events. These constructs are *orthogonal*:

- to the modelling of time using monitoring intervals, in order to support the timely restriction of (composite or primitive) events which are signalled only within the specified interval and, in order to provide the signalling of an event with regard to a performed system activity (defined by implicit points in time).

- to the support of a fixed set of event parameters which, e.g., can be used in the definition of composite events on the basis of the keyword *same*.

In these aspects, our work differs from work done on other event languages (in ODE [11, 12] and Sentinel [2]).

Furthermore, we presented how SAMOS detects the occurrences of events (primitive or composite). The detection of composite events based on Petri nets shows that even complex events can be detected in a relatively simple way. By using Petri nets, we are able to model and to detect the exact semantics of all the event constructors.

Finally, we have shown how the system represents and manages information about events so that it can be retrieved and accessed efficiently. For that matter, we specified an event schema that describes the structure of the eventbase using the benefits of the underlying object-oriented environment. In our approach, it is also important that the user is not required to know the internal representation and the handling of events.

Our current research focuses on the refinement of the event detector for time events and on so-called *value events* (as introduced in [7]). Our future research concerns the investigation of the strengths and possible problems of the rule language in concrete applications environments, e.g., in stock applications. In the longer term we plan to provide (design) tools for active database systems. In detail a graphic editor, a debugger and tools analyzing interrelationships among various rules (e.g., to detect cycles) can help the user to overcome the complexity of applying an active database system.

Acknowledgment

The work of S. Gatziu is supported by the UBILAB, the Informatics Laboratorium Union Bank of Switzerland (UBS). Furthermore, we would like to thank all our colleagues, especially Andreas Geppert, who read earlier drafts of this paper.

References

1. Anwar E, Maugis L, Chakravarthy S. A New Perspective on Rule Support for Object-Oriented Databases. Proc. ACM SIGMOD, Washington, May 93
2. Chakravarthy S, Mishra D. An Event Specification Language (Snoop) For Active Databases and its Detection. Techn. Report UF-CIS-TR-91-23, University of Florida, September 91
3. Dayal U, et al. The HiPAC Project: Combining Active Databases and Timing Constraints. ACM Sigmod Record, 17(1), March 88
4. Dayal U, Buchmann A, McCarthy D. Rules Are Objects Too: A Knowledge Model For An Active, Object-Oriented Database System. In: K. R. Dittrich (ed.). Proc. 2nd Intl. Workshop on Object-Oriented Database Systems. (Lecture notes in computer science no. 334)
5. Diaz O, Paton N, Gray P. Rule Management in Object-Oriented Databases: A Uniform Approach. Proc. 17th Intl. Conf. on Very Large Data Bases, Barcelona, September 91
6. Dittrich K.R, Gatziu S. Time Issues in Active Database Systems. Proc. Intl. Workshop on an Infrastucture for Temporal Databases, Arlington, Texas, June 93
7. Gatziu S, Geppert A, Dittrich K.R. Integrating Active Concepts into an Object-Oriented Database System. Proc. of the 3. Intl. Workshop on Database Programming Languages, Nafplion, August 91
8. Gatziu S, Dittrich K.R. SAMOS: an Active Object-Oriented Database System. IEEE Bulletin of the TC on Data Engineering, 15(1-4), Special Issue on Active Databases, December 92
9. Gatziu S, Dittrich K.R. Detecting Composite Events in Active Database Systems Using Petri Nets. Technical Report, Institut für Informatik, Universität Zürich, July 93
10. Gehani N.H., Jagadish H.V. Ode as an Active Database: Constraints and Triggers. Proc. 17th Intl. Conf. on Very Large Data Bases, Barcelona, September 91
11. Gehani N.H., Jagadish H.V., Shmueli O. Event Specification in an Active Object-Oriented Database. Proc. ACM SIGMOD, June 92
12. Gehani N.H., Jagadish H.V., Shmueli O. Composite Event Specification in Active Databases: Model & Implementation. Proc. 18th Intl. Conf. on Very Large Data Bases, Vancouver, August 92
13. Kotz-Dittrich A. Adding Active Functionality on an Object-Oriented Database System: A Laye-

red Approach. Proc. GI Conf. Datenbanksysteme in Büro, Technik und Wissenschaft, Braunschweig, März 93

14. Navathe S.B., Tanaka A., Chakravarthy S. Active Database Modelling and Design Tools: Issues, Approach, and Architecture. IEEE Bulletin of the TC on Data Engineering, 15(1-4), Special Issue on Active Databases, December 92

15. Reisig W. A Primer in Petri Net Design. Springer Verlag, October 91