Contents

Design of $\mathcal{SENTINEL}$: An Object-Oriented DBMS With Event-Based Rules

S. Chakravarthy **E. Anwar** **L. Maugis**
Database Systems Research and Development Center
Computer and Information Sciences Department
University of Florida, Gainesville, FL 32611

D. Mishra
Intergraph Corporation
Mapping Science Division
Huntsville, AL 35894

Abstract

This paper describes the design of Event-condition-action (ECA) rules for supporting *active* capability in an object-oriented DBMS and sketches its incorporation in a C++ environment. We summarize Snoop – an expressive event specification language which supports temporal, explicit, and composite events in addition to the traditional database events. A small set of event operators are introduced for building composite events.

We introduce an *event interface* to extend the conventional object semantics to include the specification of primitive events. This interface is used as a basis for the specification of events spanning objects, possibly from different classes, and detection of primitive and composite events. New rules can be introduced on existing objects, enabling objects to react to their own changes as well as to the changes of other objects. We introduce a runtime subscription mechanism between rules and objects to monitor objects selectively. This mechanism elegantly supports class level as well as instance level rules. Finally, using an illustrative example we compare the functionality of our approach with related work.

Index terms: Active functionality, Event-condition-action rules, object-oriented paradigm, Event hierarchy, Reactive and notifiable objects, Class and instance level rules

1 Introduction

Active or reactive database management systems (DBMSs) provide an event-based rule capability that can be used to support a number of database functionality (e.g., integrity enforcement, view materialization, applicability of compiled query plans when access methods change) in a uniform way. Rules used for supporting active capability consists of: an event expression, one or more conditions, an action, and a set of attributes. An event expression specifies the sequence of events whose occurrence triggers the evaluation of the condition. A condition is a side effect-free boolean computation on the database state and an action is an arbitrary sequence of operations. Attributes, typically, specify coupling mode and precedence relationship among rules. A rule with these components is termed an ECA or event-condition-action rule in the literature [5].

Most of the earlier research on active databases and commercial implementations have concentrated on the support for active capability in the context of relational database systems [5, 4, 16, 17]. Recently, there have been a number of attempts [12, 9, 15, 1, 8, 10] at incorporating event and rule support into an object-oriented database management system (OODBMS).

1.1 Paradigm Differences

Clearly, there is a paradigm shift when we move from the relational model to an object-oriented one. This warrants re-examination of the functionality as well as the mechanism by which active capability is incorporated into the object-oriented data model [2]. Furthermore, the differences between the two data models profoundly influence how the concepts and techniques are carried over from one model to the other. Below, we enumerate some of the differences between the data models that led to the design choices presented in this paper:

1. In contrast to a fixed number of pre-defined primitive system events in the relational model (e.g., update, insert, delete), every method/message is a potential event in an OODBMS. These methods are defined in application classes, making detection potentially more difficult.

2. The principle of encapsulation and further the distinctions between features supported (e.g., private, protected, and public in C++, class and instance attributes in Common Lisp with Flavors) need to be accounted for; this is orthogonal to both the access control issue and global nature of rules in the relational database context,

3. The principle of inheritance (both single and multiple/mixin) and its effect on rule incorporation, and

4. Scope, accessibility, and visibility of object states for rules.

The above differences imply that some of the techniques used in the relational context may not be appropriate for the OO paradigm. For example, large number of methods (both user and system defined) entail that event detection cannot be hard-wired into a few system defined functions. The user should be able to selectively define events and the system should be responsible for detecting them efficiently. As another example, the design should be uniformly applicable to all features supported by the paradigm (e.g., inheritance and various attribute categories in a class) and should be preferably independent of the implementation. Unlike a relational system where all the shared data is considered global (subject to access privileges), the scoping rules of an OO model need to be preserved. Finally, addition of attributes (or new abstractions) to support rules need to conform to sound software engineering principles.

1.2 Contributions

As there are several commercially available OODBMSs, techniques for incorporating active capability into OODBMSs is gaining attention. However, earlier approaches do not consider the data model differences completely and do not fully meet the functionality as well as seamlessness[1] requirements of rule support for an object-oriented database. Our approach, described in this paper, clearly separates the modeling issues from the implementation choices. As a result, the functionality of our system is not dictated by the environment although the implementation choices are to a large extent influenced by the environment chosen (C++ in our case).

In this paper, we focus on the design of event-condition-action rules (or ECA rules) for an OODBMS. Our approach extends earlier work [12, 9] in several ways. The design is applicable to any OO data model (i.e., Common Lisp with Flavors, Smalltalk, C++, CLOS), as we have clearly separated the modeling issues from implementation and environment choices. For concreteness and for comparison with related work, we discuss implementation choices for C++ and sketch our implementation strategy for the Zeitgeist *passive* OODBMS.

[1]By seamless approach we mean that the concepts proposed blend homogeneously into the paradigm into which they are introduced without circumventing the tenets of the paradigm.

Most importantly, we introduce a monitoring viewpoint (termed *external monitoring viewpoint*) which preserves the orthogonality of event, rule, and object existence and furthermore permits: i) rule definition to be independent of the objects which they monitor, ii) rules to be triggered by events spanning several objects (inter-object rules), possibly from different classes, and iii) an object to *dynamically* determine which objects' state changes it should react to and associate a rule for reacting to those changes.

The remainder of this paper is structured as follows. Section 2 briefly describes related work. Section 3 summarizes the motivation for our approach and includes a new perspective on monitoring. In section 4 we provide the design overview and the rationale behind it. Section 5 provides an implementation sketch of the design in the C++ environment. In section 6 we present the functionality of our system, **Sentinel** using an illustrative example. Section 7 contains conclusions.

2 Related Work

Although a number of efforts have addressed incorporating active capability in the context of an OODBMS [15, 8, 11], mostly Ode [12] and ADAM [9] are pertinent to the material presented in this paper. Below, we briefly summarize them.

Ode provides active behavior by incorporating rules in the form of constraints and triggers. Both constraints and triggers consist of a condition and an action, and are defined *within* a class definition. Events in Ode are implicit and are considered as the disjunction of all non-constant public methods. Constraints are used to maintain the notion of object consistency and hence are applicable to *all* instances of the class in which they are declared. Triggers, on the other hand, are used for monitoring database conditions other than those representing consistency violations and are applicable only to those instances specified *explicitly* by the user at runtime. Triggers in Ode are parameterized. The activation of all types of triggers occurs explicitly by the user. Triggers are checked at the end of each method and are appended to a *to-be-executed* list, if they evaluate to true. Trigger bodies are executed in separate transactions after the commit (not necessarily immediately after) of the transaction firing them. More recently Ode [13] has proposed a language for specifying composite events. They specify composite events using a set of operators similar to the one presented in [6, 7]; events are declared within a class definition. Basic (primitive) events are defined and composite events are constructed by applying operators to basic events. The basic events supported are *object state events, method execution events, timed events* and *transaction events*. The event operators supported are *relative, prior, sequence, choose, every, fa* and *faAbs*. Detection of events is accomplished by using a finite automata.

ADAM [9] is an active OODB implemented in PROLOG. It focuses on providing a uniform approach to the treatment of rules in an object-oriented environment. Rules and events in ADAM cannot be specified as part of an object class specification. Both events and rules are treated as first class objects which are created, deleted and modified in the same fashion as other objects. An object's definition is enlarged to indicate which rules to check when the object raises an event. Thus each class structure is augmented with a *class-rules* attribute; this attribute has as its value the set of rules that are to be checked when the class raises an event. A *Rule-class* is defined where each rule is an instance of that class. The structure of the Rule-class consists of the attributes *event, active-class, is-it-enabled, disabled-for, condition* and *action*. The event attribute indicates the event which triggers the rule, the active-class attribute indicates the class name on which the rule is applicable, the is-it-enabled attribute specifies whether the rule is enabled or not, the disabled-for attribute has as its value the set of instances for which the rule is disabled while the condition and action attributes specify the rule's condition and action respectively. Rule operations are implemented as class methods. Only primitive events are supported.

3 Motivation

The design of ECA rules in Sentinel was primarily motivated by the following limitations of the extant systems:

- Support for an expressive event specification language to complement the rule expressiveness in an OO framework,
- Current approaches allow a rule to monitor one or more instances of the same object class; however, a rule is triggered by changes occurring to *only one* of the instances. To enhance expressiveness, support for rules which are triggered by changes occurring to one or more instances, possibly from different classes, is necessary,
- Permitting rule specification only within a class definition leads to difficulties when rules are added, deleted, or modified, since instances of these changed classes may be previously stored in the database. This compromises the extensibility of the system since the addition of rules is not divorced from the behavior of pre-existing objects and methods in the system. Similarly, supporting rules only at runtime restricts the modeling capability of the system,
- Rules and event detectors/recorders[2] are not always treated as first class objects, thereby resulting in a dichotomy between them and other objects. Rules and event recorders/detectors cannot be added, deleted, and modified in the same manner as other objects. Furthermore, they are not subject to the same transaction semantics. Finally, their persistence is dependent on the existence of other objects, and
- Specification of events and the mechanism by which they are detected. Although Ode [12] supports the specification and detection of composite events, the manner in which they are supported prevents expressing events spanning instances of the same as well as different classes.

In addition to the above limitations, the viewpoint currently used for monitoring objects is restrictive as explained below.

3.1 A New Monitoring Viewpoint

In a number of applications, such as patient databases and portfolio management, monitored and monitoring objects are often defined not only independently but at different points in time. For example, when a patient class is defined (and instances are created), it is not known who may be interested in monitoring that patient; depending upon the diagnosis, additional groups or physicians may have to track the patient's progress. Similarly, stock objects may have to accommodate a varying number of objects (e.g., portfolio) that may be interested in their state (e.g., price) for buying and selling purposes. That is, there is a need to monitor pre-defined objects, preferably, without having to change their class definitions for that purpose. For example, there should not

[2] As will be elaborated later, an event detector object is created for the detection of a composite event; for primitive events an event recorder object is created.

be a requirement for the stock object to modify its attributes or behavior so that a new portfolio object can monitor it.

Clearly, if one has to declare all the rules that are likely to be associated with an object at the class definition time, the above application requirements cannot be supported (as the specifics may not be available at the object definition time). Even if one were to allow rule definition at runtime on a class, if the rule firing is restricted to *only* events of the same class, then also the above application requirements cannot be adequately supported.

Consistent with the notion of encapsulation, we introduce an **event interface** for this purpose (analogous to the traditional interface for objects). We also separate the (visible) event interface from: i) its implementation and ii) the use (or invocation) of that interface by other objects. The event interface is defined (or revealed) at class definition time whereas its use is supported at runtime. Below, we give an example of how the external monitoring viewpoint supports a rule that monitors one or more existing objects and preserves encapsulation.

```
Stock IBM;
Portfolio Parker;
FinancialInfo DowJones;

RULE Purchase :
WHEN IBM-stock-price-changes And DowJones-average-changes       /* Event */
IF IBM-stock-Price < $80 and DowJones-average-Change < 3.4% /* Condition */
THEN PurchaseIBMStock for Parker                                 /* Action */
```

In the above example, three objects are created, namely, IBM, Parker, and DowJones belonging, respectively, to classes Stock, Portfolio and FinancialInfo. A rule, *Purchase*, is defined independently of these three objects and monitors two objects – the IBM object and the DowJones object. The rule is triggered when events spanning these two objects are generated, specifically, when the IBM object invokes the method that changes its price *and* the DowJones object invokes the method that changes its value. The condition then checks the IBM stock price and the percentage change in the DowJones value. If the condition is satisfied, the Parker object purchases IBM stock. We discuss how the above rule is specified and executed in Sentinel in a later section.

4 Design of Events and Rules for Sentinel

The following design choices were made for incorporating active capability into Sentinel:

1. Support primitive (e.g., database, temporal, and external) events as well as composite events using a small set of event operators. They were classified into a hierarchy to facilitate its incorporation into OODBMSs,

2. Augment the specification of an object class with an **event interface**,

3. Support primitive event recorders, composite event detectors, and rules as first class objects,

4. Allow rules to be triggered by events spanning several objects,

5. Allow an object to dynamically specify *which* objects to react to in response to their state changes, and

6. Provide an uniform mechanism for associating rules to all instances of a class as well as to a subset of instances, possibly from different classes.

The above design augments an OO model to include active capability. Event interface extends the expressive power of the resulting system, preserves encapsulation, supports monitoring of multiple objects possibly from different classes, and reduces the number of rules to be specified (items 1 and 3). It supports an incremental design capability for user applications. At design time, while defining a class, the user is not required to explicitly list all the rules applicable to that class (item 3). At runtime, new rules can be added and associated (applied) with (to) existing objects in the database. Consequently, to a large extent, the extensibility and modularity of the resulting system is not compromised. Finally, the above design facilitates creation of rules with event, condition, and action components both at compile time and at runtime by choosing an appropriate implementation. As rules are objects in their own right, it is possible to introduce new features (for example, providing a new conflict resolution strategy) without modifications to application code. This can be done by adding methods only to the rule class.

4.1 The Event Interface

To treat rules independently from the objects they monitor, objects must be capable of: i) detecting primitive event occurrences when their methods are invoked and ii) signal these event occurrences to other objects. We extend an object with an **event interface** to enable objects to designate some, possibly all, of their methods as primitive event generators. The implementation of the interface specification is based on the OO environment into which it is incorporated and is discussed in the next section. The augmented object is depicted in Figure 1. Traditionally, objects receive messages defined using the conventional interface, perform some operations and then return results. Now, in addition, they detect event occurrences for the methods (defined using the event interface) when they are invoked and signal these event occurrences to other objects *asynchronously*. Events are generated either *before* or *after* the execution of a method. An object definition that includes an event interface is termed a **reactive** object. That is,
Reactive object definition = Traditional object definition + Event interface

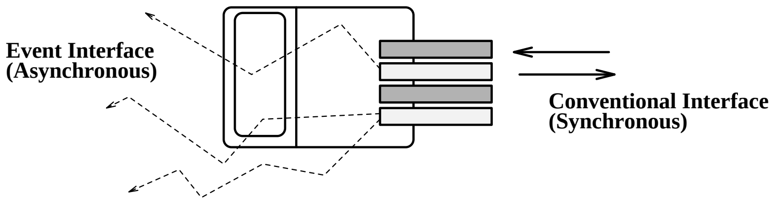

Figure 1: Semantics of the Event Interface

Using the event interface, primitive events are specified as part of the object class definition. The event interface only specifies the primitive events that are to be detected by that reactive object class. The semantics of the event interface is that the reactive object will generate and signal an event occurrence for each method (when the method is invoked for any instance of that class) specified in the event interface. Although every method of a class corresponds to two[3] potential primitive events, the designer may want to specify a meaningful subset as part of the event interface specification. Hence, only objects that are likely to be monitored need to be made instances of the reactive class. Similarly only those methods that are likely to be used as events need to be defined in the event interface. In contrast to the conventional interface which is specified and implemented

[3]Generation of primitive events before and after corresponds to the invocation and return of methods.

by the user, only the event interface is specified by the user; its implementation is provided by the system. An event produced (i.e., detected and signaled) by a Reactive object consists of the following parameters:

Generated primitive event = Oid + Class + Method + Actual_parameters + Time_stamp

Since instances of the Reactive class are producers of events, they need to know the consumers of those events. This leads us to the introduction of the Notifiable objects. A notifiable object is a consumer of an event that is of interest to that object. An association is established between an event and a notifiable object using the **subscription** mechanism.

4.2 Object Classification

In order to support active capability, we introduce two additional object types: **reactive** and **notifiable**. These two object types augment the **passive** objects supported in an OO system. An active database designer creates a schema which defines object classes for an application. In addition, the designer categorizes object classes to be reactive – to produce appropriate events, and notifiable, to consume events generated by a reactive class.

Passive objects : These are conventional objects. They do not generate events. An object that needs to be monitored (by informing other objects of its state changes) cannot be passive.

Reactive objects : Objects that need to be monitored (i.e., on which rules will be defined) need to be made reactive. The event interface of objects enables them to declare any, possibly all, of their methods as event generators. If a method is declared as an event generator, the object will detect and signal other objects when the method is invoked. Thus, reactive objects communicate with other objects to signal the occurrence of primitive events.

Notifiable objects : Notifiable objects, on the other hand, are those objects capable of being informed of the events produced by reactive objects. Therefore, notifiable objects become aware of a reactive object's state changes and take appropriate measures (by evaluating conditions and executing actions).

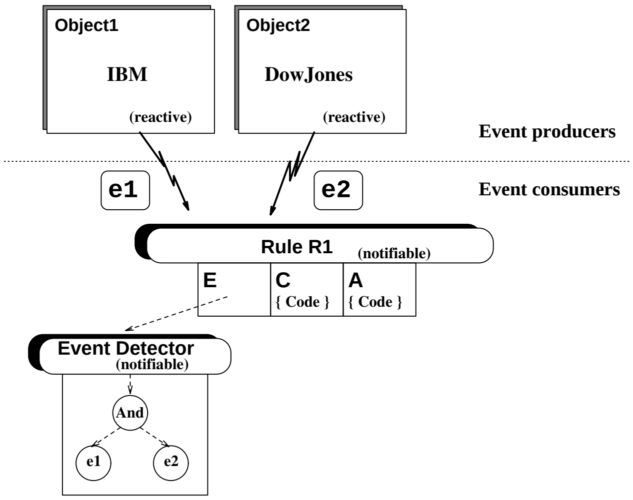

Figure 2: An Event Producer/Consumer Analogy.

Figure 2 illustrates the producer/consumer behavior of objects. Two independent objects `object1` and `object2` detect (at different points in time) distinct primitive events `e1` and `e2` and signal them to a rule `R1`. The rule passes the events to the composite event detector for storage and detection, and if the event is detected, the rule is fired (i.e., checks the condition and possibly executes the action).

Notifiable objects subscribe to reactive object instances. After the subscription, the reactive objects signal the occurrence of primitive events to the notifiable objects. Lastly, the rule objects execute the condition and action parts as a result of these signaled events. There is a many-to-many relationship between notifiable and reactive objects; that is, signaling of an event occurrence by a reactive object instance can be relevant to several notifiable object instances and a notifiable object instance can receive events from several reactive object instances. Event detectors/recorders and rules are examples of notifiable objects. Rules receive event occurrences from reactive objects, send them to their local event detector/recorder, and take appropriate actions. Event detectors/recorders receive events from reactive objects, store them along with their parameters, and use them to detect composite events.

4.3 Event Specification Language

Before we introduce the design of the event specification language, we summarize the notion of an event, an event modifier, and an event expression. We assume a discrete and finite representation of time in the form of a time line which has an origin (assumed to be at "0"). The granularity of the time line, however, is determined by the system/application. Further, we adopt two *indefinite* time constants: *Now* and ∞ (infinity). Now is a moving time point that keeps track of the *current time*. ∞ is used to represent an indefinite time interval, such as (*Now* to ∞) or an indefinite time point which is greater than the current time.

We make a distinction between an event expression and an event. Informally, an *event* is an atomic (happens completely or not at all) occurrence whereas an event expression is an expression that defines a time interval. Typically, most of the operations that are of interest to the application take a finite amount of time for their execution. For example, a database operation, function/method, or a transaction can be viewed as an event expression. Our interest in event expressions comes from the fact that one needs to choose a point on the time line, within the closed interval defined by an event expression, to denote the occurrence of that event. We introduce the notion of an event modifier to transform an event expression to one or more events. An event corresponds to a point of interest on the time line for that event expression.

We define two event modifiers *begin-of* and *end-of* to transform an arbitrary interval into two logical events. These two modifiers make the corresponding events instantaneous and precisely map the point of the occurrence of an event onto a point on the time line. For example, the *end-of* event for a function $\mathcal{F}$ (i.e., end-of-$\mathcal{F}$) can be mapped to the point after the last statement in the function $\mathcal{F}$ before control is returned to the calling function. The *begin-of* event can be similarly defined. Events are associated with database operations (i.e., event expressions) of interest and their parameters are viewed as the parameters of the event corresponding to the operation.

4.3.1 Event Classification

Events can be broadly classified into: i) primitive events – events that are pre-defined in the system (using functions/methods and event modifiers); for each of them a mechanism for its efficient detection is likely to be embedded in the system (corresponds to the implementation of the event interface) and ii) composite or composite events – events that are formed by applying a set of operators to primitive and composite events. This classification is shown in Figure 3.

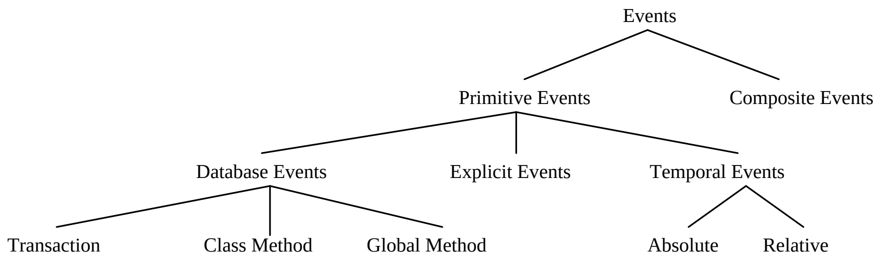

Figure 3: Event Classification

Primitive events are further classified into database events, temporal events, and explicit (or external) events. Database events are related to database operations such as transaction, a method, or a function. Temporal events are related to time and are of two types: absolute and relative. Absolute events map to discrete points along the time line (e.g., at 5 p.m.) whereas relative events are defined with respect to an explicit reference point (e.g., 5 seconds after an event e occurs, where e is either a primitive or a composite event). Explicit events are those events that are detected and signaled along with their parameters by application programs (i.e., outside of the DBMS) and are only managed by the system. Prior to their usage, explicit events and their formal parameters need to be registered with the system. Each event (primitive or otherwise) has a well-defined set of parameters that are instantiated for each occurrence of that event.

Database events correspond to the operations (transformed into events using the modifiers) used for manipulating data. For the object-oriented model, they correspond to the begin-of and end-of variants of methods on objects and transaction operations. A *temporal event* is an instance of the temporal event class. A temporal event is defined by specifying a point on the time line either using an absolute value or using an offset from a well-defined point. We use a string having the format (hh:mm:ss)mm/dd/yy to define a point on the time line. The same format is used for representing both absolute and relative temporal events.

An absolute temporal event is specified with an absolute value of time and is represented as: $< timestring >$. For example, 2 p.m. on March 15th, 1994 is specified as $< (14:00:00)03/15/1994 >$. We use an intuitive notation for examples to improve readability. For a relative event both the reference point and the offset are explicitly specified. The reference point may be any event including an absolute temporal event. The syntax for a relative event is $event + [timestring]$. In the representation of an offset, an empty field in the time string is substituted with the minimum value for that field. A relative event subsumes an absolute event; hence a relative event is a primitive event. However, the absolute version is retained for practical reasons. Note also that an absolute event is a definite event (A *definite event* is an event that is guaranteed to occur). A relative event E + [t] is a definite event provided E is definite.

Explicit/External events: Explicit (also called external or abstract) events are defined by users or application programs and registered with the system. The parameters are also explicitly specified and supplied at the time an external event is signaled to the DBMS. For consistency, we stipulate that parameters of explicit events must include event-type and the time of occurrence. Explicit events are assumed to be detected outside of the system but are signaled to the system along with their parameters. Thus these events are accepted and processed (but are not detected) by the system.

4.3.2 Event Operators

Various categories of primitive events discussed so far are useful for modeling a number of applications requiring active functionality. However, primitive events are not sufficient to model the event component of the following rules:

1. Sample IBM stock price every 30 minutes from 8 a.m. to 5 p.m. each working day and compute the maximum price of the stock over a day,
2. When 4 withdrawals are made on an account in a day, flag it and do not allow any more withdrawals,

Some of the above (e.g., no. 2) can be modeled by multiple ECA rules or chaining rules and specifying control information (and using explicit events, if need be) as part of the rule. Specifying control information as part of a rule can be avoided to a large part by providing a language for constructing event expressions to suit the application requirements. This will not only help the rule designer concentrate on the semantics of his/her application, but also enable the system to optimize event expressions and detect them efficiently.

We define a composite event as an event obtained by the application of an event modifier to a composite event expression. By default, the end-of event modifier is assumed to be applied. We define a composite event expression, recursively, as an event expression formed by using a set of primitive events, event operators, and composite events constructed so far. Figure 4 shows the event operators supported in Sentinel. Below, we briefly describe each of these operators and its semantics.

Disjunction ($\vee$): Disjunction of two events E1 and E2, denoted $E1 \vee E2$ occurs when E1 or E2 occurs [4]. This operator is useful when the occurrence of one or more (i.e., exclusive-Or under the assumption that simultaneous events cannot occur; otherwise it is an inclusive-Or) events out of a set of events may cause a rule to fire. For example, a constraint that needs to be checked either for an insertion or for the deletion of an object can be specified using this operator. A composite event end-of-insert $\vee$ end-of-delete specifies the required semantics. Although an event modifier can be used on this expression as a whole, both expressions have identical semantics.

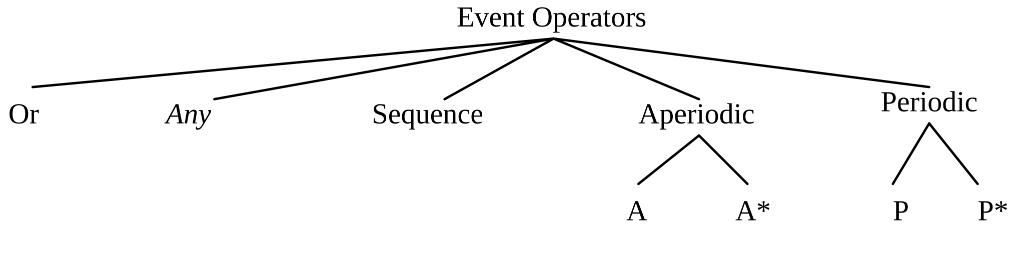

Figure 4: Event Operators

Sequence (;): Sequence of two events E1 and E2, denoted *E1;E2*, occurs when E2 occurs provided E1 has already occurred. This implies that the time of occurrence of E1 is guaranteed to be less than

[4]In this paper, we have assumed that no two events can occur simultaneously. However this assumption is not realistic in a multiprocessor (or a distributed) environment. Moreover, explicit events can occur simultaneously with non-explicit events. In case of concurrent occurrences, it is necessary to gather the parameters of all the events and then signal the composite event only once. How long the system should wait for detecting and computing the parameters of a disjunction is not clear. It may be meaningful to include a time window for dealing with simultaneous event occurrences. However, this discussion is beyond the scope of this paper.

the time of occurrence of E2. This constructor is useful when a predefined order has to be imposed over the occurrences of events. For example, in a banking application, special consideration might be given for the sequence of events `deposit; withdraw`. This will distinguish a periodic withdrawal from the one that relates to a specific deposit followed by a withdrawal.

It is possible that after the occurrence of E1, E2 does not occur at all. To avoid this situation, it is desirable that every expression having ';' include definite events such as end-of-transaction or an absolute temporal event. The event end-of-transaction is a natural choice for a definite event (as it always occurs, when there is no starvation, irrespective of whether a transaction aborts or commits) in database applications.

Conjunction (Any, All): The conjunction event, denoted Any(I, E1, E2, ... En) where I <= n, occurs when any I events out of the n events (corresponding to n *distinct* events specified) occur, ignoring the order of their occurrence. Conjunction of events is useful in situations where firing of a rule depends upon the occurrence of a number of events and not necessarily their order. It is possible to simulate the 'Any' operator with a combination of ';' and 'Or'. For example; Any(2,E1, E2) can be written equivalently as $(E1; E2) \vee (E2; E1)$. However this rewriting results in long event expressions that may be difficult to comprehend. All is a shorthand for Any(m, E1, ..., En) where m >= n. Again, it is desirable that this event be used along with a definite event (as in Any(2, E1, E2) $\vee$ end-of-transaction). Note that multiple occurrences of the same event does not count in the above. A variant of the above can be expressed as Any(n, E*), where an upper bound n is specified as a pre-determined constant. Of course, E in the above connotes *distinct occurrences* of the same event E as opposed to the event types in the previous case. Using this variant, 4 occurrences of a withdrawal event can be expressed as Any(4, end-of-withdrawal*). This variant of Any can also be used to specify interesting events, such as Any(2, E1, Any(2, E2*), Any(2, E3*)).

Aperiodic Event Operators (A, A*): The Aperiodic operator A allows one to express the occurrence of an aperiodic event bounded by two arbitrary events (for providing an interval). There are two variants of this event specification. The non-cumulative variant of an aperiodic event is expressed as A(E1, E2, E3), where E1, E2 and E3 are arbitrary events. The event A is signaled each time E2 occurs during the closed interval defined by the occurrences of E1 and E3. **A** can occur zero or more times (zero times either when E2 does not occur in the interval or when no interval exists for the definitions of E1 and E3). This operator is useful when a rule has to be associated for each occurrence of E2 within a prescribed interval expressed by events. For example, the event corresponding to example 2 (of Section 4.3.2), can be expressed as A(8 a.m., Any(4, withdraw-on-an-account*) 5 p.m.). There are situations when a given event is signaled more than once during a given interval (e.g., within a transaction), but rather than firing the rule every time the event occurs, the rule has to be fired only once. To accommodate this, we provide an operator A^*(E1, E2, E3) that occurs only once when E3 occurs and accumulates the parameters for each occurrence of E2. This constructor is especially useful for integrity checking because the constraints need to be checked only at the end of the transaction.

Periodic Event Operator (P, P*): We define a periodic event as an event E that repeats itself within a constant and finite amount of time. Only a time specification is meaningful for E. The notation used for expressing a periodic event is P(E1, [t], E3) where E1 and E3 are events and t is the time specification. It is important to note that t is a *constant* in the above and cannot contain wild card specification in *all* fields because this will result in continuous occurrences of P. Note that the event of interest in P is the middle event which is always a time specification and hence does not include any state change operation. To make the event more practical and meaningful for real-life applications, it may be useful to allow a parameter along with the frequency specification.

To accommodate this, we define a cumulative variant of P (denoted P*) which includes a parameter for each occurrence of the periodic event. In the absence of this parameter, the cumulative operator just accumulates time of occurrences of the periodic event as the parameter object.

Illustrative Examples: Using the operators introduced so far, we illustrate how the event component of the following examples can be modeled. We use a simplified syntax to make the rules readable.

1. Sample IBM stock every 30 minutes from 8 a.m. to 5 p.m. each working day and compute the maximum value over a day

```
On          P*(8 a.m., [30 mins]: IBM-stock-price, 5 p.m.)
Condition   true
Action      compute maximum using the parameters collected
```

2. When 4 withdrawals are made on an account in a day, do not allow further withdrawals

```
On          A(8 a.m., Any(4, withdraw-on-an-account*), 5 p.m.)
Condition   true
Action      block further withdrawals ~\\[-8mm]
```

4.3.3 Parameter Computation of Composite Events

The detection of an event corresponding to an event expression may require detection of one or more constituent events as well as one or more occurrences of constituent events. Situations requiring multiple event occurrences to be included in the parameter set give rise to alternate ways of computing parameters as the events (especially composite) are likely to occur over an interval. Furthermore, the parameters may have to be computed in a way that suits the the semantics of the application which may vary from one class of application to another. The notion of parameter contexts is primarily introduced to both disambiguate the parameter computation and at the same time accommodate a wide range of application requirements.

We explain the parameter contexts introduced below by using the following event expressions:

A = Any(2,E1, E2) ; E3   B = $E1 \vee E2 \vee E3$ and   C = E1 ; Any(2,E2, E3)

where E1, E2, and E3 are primitive events. Figure 5 shows the occurrences of different instances of event E1, E2 and E3 for the composite events A and C. Furthermore, we adopt the notation $E(E_1, E_2, \ldots, E_n)$ to represent an event expression E, where $E_{i=1..n}$ are its parameter event set.

The parameter contexts proposed below are motivated by a careful analysis of several classes of applications. We have identified four parameter contexts that cover a wide range of applications. Briefly, we indicate the characteristics of the applications that motivated our choices of parameter contexts:

1. Applications where the events are happening at a fast rate and multiple occurrences of an event only refine the previous data value. This is typical of sensor applications (e.g., hospital monitoring, global position tracking),

2. Applications where there is a correspondence between the occurrence of events and this correspondence needs to be maintained. Applications that exhibit causal dependency (e.g., between aborts, rollbacks, and other operations; between bug reports and releases) come under this category,

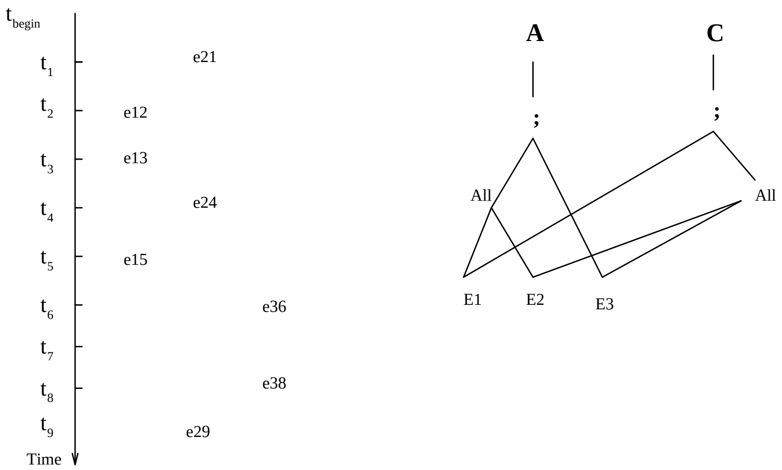

Figure 5: Event Occurrences

3. Trend analysis and forecasting applications (e.g., securities trading, stock market, after-the-fact diagnosis) where composite event detection along a moving time window needs to be supported. Monitoring the occurrence of 20% change in DowJones average in any half-hour period is an example of trend analysis, and

4. Applications where it is not clear as to how parameters need to be restricted.

We introduce the following contexts for the classes of applications described above:

Recent: In this context, only the most recent occurrence of each E_i that has started the parameter computation is taken into account for computing the parameters of E. When E occurs, the composite event is signaled and all the occurrences that are used in the parameter relation are deleted. For example, in the *recent* context, parameters of event A will include parameters of event instances e15, e24 and e36 (A is detected when e36 occurs). In this context, *not all occurrences* of a primitive event will be used in computing the parameters of a composite event. In the recent context, the parameters of C will be <e13, e24, e36> and *not* <e15, e24, e36>. e13 is the most recent occurrence of E1 that started the parameter computation and not e15 (or e12).

Chronicle: In this context, instances of component events are taken into account in the chronological order of their occurrence for computing the parameters. When a composite event E is signaled, its parameters are computed by using the oldest instance of each component event and then they are deleted. For example, parameters of event A in the chronicle context will be computed by using event instances e12, e21 and e36. When the next E3 type event occurs at e38, then the A will be signaled with the parameters <e13, e24, e38>. The buffer is flushed once the event is signaled. When the rule is deactivated the entire set of partial buffers are flushed. In this context, the same primitive event occurrence is used *at most once* for computing the parameters of the composite event.

Continuous: In this case, each occurrence of an event that marks the beginning of the interval of an event expression is considered a potential candidate for starting a parameter set computation. This is especially useful for tracking trends of interest on a sliding time window. For example, an event specification for computing 20% change in DowJones in any 2 hour interval can be elegantly modeled using this parameter context. In Figure 5, each of the occurrences e12, e13, and e15 (as well as e21 and e24) would start the detection of the event A. The first occurrence of A will have as parameters <e21, e12, e36>. The parameters of the second occurrence of A will consist of <e12, e24, e36>. The parameters of the third occurrence of A are <e13, e24, e36>. In this context, an occurrence of a primitive event will be used *at least once* for computing the parameters of the composite event.

There is a subtle difference between the chronicle and the continuous contexts. In the former, pairing is with respect to the event that detects the composite event (end of the interval) whereas in the latter pairing is with respect to the event that begins the interval for the composite event. For clarification, consider the event expression C and the event occurrences shown in Figure 5. The event C is initiated by one of the events e12 or e13 or e14 *but not* e21 and e24 as is the case for the composite event A. Parameters computed for the event C in the chronicle context are <e12, e24, e36> whereas the parameters computed for the event C in the continuous context are <e12, e24, e36>, <e13, e24, e36>. In the recent context, the parameters computed for C are <e13, e24, e36>.

Cumulative: In this context, parameters of E include the parameters of all the occurrences of each E_i. Whenever E is signaled, all the entries in the parameter relations associated with each E_i are deleted. For example, parameters of event A will include all the instances of each event up to e36 when it occurs. The entire set of events and their parameters shown in Figure 5 (except e38 and e29) is the parameter relation. For the event C, when e36 occurs, all event parameters up to that point (does not include e21) is computed as parameters. Unlike the continuous context, an event occurrence does not participate in two parameter computations in the cumulative context.

Event expressions can broadly be classified into two categories. The first category consists of expressions consisting of primitive events and *only* the operators P, A and $\vee$. A context specification to these expressions does not affect the semantics of the corresponding composite events. The second category consists of event expressions that involve operators Any, Sequence, and cumulative versions of P and A. These composite events are characterized by a time interval during which the occurrences of primitive events need to be monitored. Context specification does affect the semantics of these operators.

4.4 Event Speicification in Sentinel

In Sentinel, the primitive events and event operators are modeled as an object hierarchy. The root of the hierarchy is the event class and primitive events and event operators form subclasses of the event class. A recorder for each primitive event specified in the event interface forms an instance of the primitive event class. The state information associated with each event includes the event type, time of occurrence of the event, and the parameters computed when an event occurs. The structure of a composite event consists of the participating event(s), while the behavior consists of specifying when to signal the event. A composite event is created as an object of the event class using the instances of the operator object classes as required. Composite event detectors (as well as primitive event recorders) can be created, deleted, modified, and designated as persistent. Furthermore, the introduction of new event types/attributes can be easily incorporated by modifying/augmenting the event class definition (e.g., we have an event hierarchy for supporting composite events) without compromising the extensibility and modularity of the system. Moreover, events spanning distinct classes can be expressed. By making an event an object, creation and deletion of events are done at runtime. Event recorders/detectors are final consumers of primitive events produced by reactive objects (elaborated further in Section 5.1).

The above approach overcomes the disadvantages of treating events either *as expressions* of a rule or as *rule attributes*. The main disadvantage of these approaches is that events cannot be added, deleted or modified at runtime, thereby resulting in a dichotomy between events and other types of objects. Furthermore, persistence of events (as well as rules) is dependent on the existence of other objects. More importantly, events spanning distinct classes cannot be expressed in this approach. In addition, events cannot have attributes or methods of their own and hence cannot store and access the parameters computed when the event is signaled. Lastly, new event types or event attributes cannot be easily incorporated, thereby compromising extensibility.

In addition to database events, a subset of composite events of the event specification language described above is supported by defining subclasses of the event class. The *conjunction* composite event detector E1andE2 is constructed by applying the conjunction operator to two events E1 and E2; The event All(E1, E2) is detected and signaled when both E1 and E2 occur, regardless of the order of their occurrence. An event detector E1orE2, constructed by applying the disjunction operator to two events E1 and E2, is used to detect and signal the event E1 $\vee$ E2 when either E1 or E2 occurs. Analogously, an event detector E1seqE2, constructed by applying the sequence operator to the events E1 and E2, detects and signals the event E1;E2 when E2 occurs, provided E1 has occurred earlier. In the above, E1 and E2 may potentially be composite events. For the BNF of event expressions and the detection of composite events refer to [6].

4.5 Rule Specification in Sentinel

Several alternatives for the treatment of rules in an OODBMS exist: i) as declarations within a class, ii) as data members, and iii) as objects. We chose to treat rules as objects as they are separate entities that exist independently of other objects in the system. Rules can be designated as transient or as persistent objects. In addition, they are also subject to the same transaction semantics as other objects. Each rule can be associated with and monitor multiple objects. The structure and behavior of rules can be tailored to model the requirements of various applications. For example, it is possible to create subclasses of the rule class and define special attributes or operations on those subclasses. As an example, hard and soft constraints of Ode [12] can be modeled as subclasses of the rule class. Extensions are incorporated by adding new attributes and methods on rules, and requires the modification of the rule class definition only.

One alternative to the above is to treat rules as declarations only inside a class. The declaration of rules within class definitions is a straightforward mechanism for associating rules to objects. In this approach, processing the declaration of a rule is performed primarily at compile time by a preprocessor. Rules need to be defined at class definition time making it difficult to add rules later. Furthermore, only rules defined with parameters can be bound to object instances. The main disadvantage of this approach lies in its difficulty in handling the addition, deletion and modification of rules, as it requires the modification of user's class definitions and thus recompilation. This can be a problem in environments that do not support late binding and incremental compilation. Furthermore, modification of a class definition may present difficulties to existing instances of that class, thereby compromising the extensibility of the system; in the least, addition of rules should not interfere with instances for which the rule is not applicable. Finally, in this approach rules applicable to more than one object class need to be duplicated in both the classes [14].

Yet another alternative is to treat rules as data members (attributes of the object class in which the rule is defined). By treating rules as data members some of the above problems are solved. The advantage of this approach is its re-usability and extensibility; once a type has been defined it can be used throughout an application as well as in other applications. Furthermore, the introduction of new rule components only requires redefining the type definition. Moreover, rules are associated with objects since they are part of an object's structure. In addition, rules can be added, deleted and modified dynamically. However, the main disadvantage is that it does not support inheritance. This is because the *value* of a data member cannot be inherited. Also, a rule's existence is dependent on the existence of other objects.

4.6 Association of Rules with Objects

Typically rules in relational databases have been treated as global constraints which must be satisfied by all relations in the database. This global treatment of rules is no longer meaningful in the context of an OODBMS. This is because objects differ considerably in both structure and behavior from one another and therefore it is realistic to assume that different kinds of objects may have different rules applicable to them.

In order to accommodate rules in an OO environment, we classify rules into two main categories, namely, class level rules and instance level rules. Class level rules are applicable to all instances of a class while instance level rules are applicable to specific instances, possibly from different classes. Rules, regardless of their classification, are treated as first class notifiable objects. In order to associate rules with objects, we introduce a **subscription** mechanism. This allows notifiable objects (rules in this case) to dynamically subscribe to events generated by reactive objects. After the subscription takes place, a notifiable object will be informed or notified of the events generated by reactive objects and react to those events.

The subscription mechanism introduced in this paper has three main advantages. First, runtime rule checking overhead is reduced since only those rules which have subscribed to a reactive object are checked when the reactive object generates events. This is in contrast to the approach in Ode [12] where each method is a potential event and all rules are checked at the end of each non-public member function. Second, a rule can now be applied to different types of objects in an efficient manner; the rule is defined only once and then subscribes to the events generated by different types of objects. This is more efficient than defining the same rule multiple times and applying each rule to one type of object. Lastly and more importantly, rules triggered by events spanning distinct classes can be expressed. This is accomplished by a rule subscribing to the events generated by instances of different classes.

4.7 Summary

The above design is generic in that it can be incorporated into any given OO environment. Some of the implementation choices are likely to be influenced by the features of the host environment (e.g., late binding/linking available in Common Lisp with Flavors). For example, event detection [8] can be incorporated using the Whopper mechanism provided in Common Lisp with Flavors. As another example, a notion similar to the class variable can be used to support rules applicable to an entire class. In the next section, we sketch how the above design is incorporated into Zeitgeist which uses C++.

5 Incorporating Sentinel Design into Zeitgeist

The Sentinel system is being implemented using Zeitgeist OODBMS developed at Texas Instruments, Dallas [3]. Zeitgeist is *passive*; that is, it does not provide any form of ECA rule support. Zeitgeist is an open, modular, extensible architecture for object-oriented database systems, implemented in C++ on Sun4 Unix platforms. In order to incorporate rules in Zeitgeist we modified the class hierarchy to include newly defined classes, namely, the `Reactive`, `Notifiable`, `Event` and `Rule` classes. The class hierarchy introduced for rule support is illustrated in Figure 6[5].

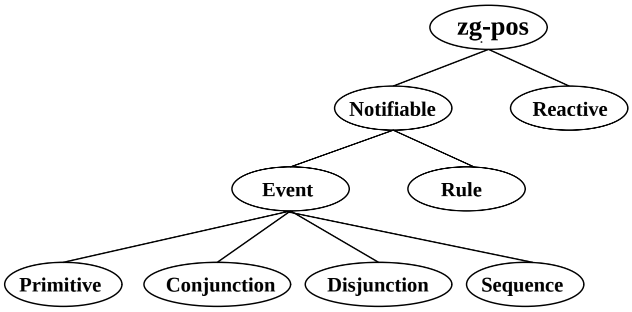

Figure 6: Sentinel Class Hierarchy for Rule Support.

In the following subsections we sketch the implementation of Reactive, Notifiable, Event and Rule classes and indicate the implementation status at the end. Where appropriate, we compare our implementation choices with Ode and ADAM.

5.1 Event Specification, Detection, and Signaling

Specification: Database events are specified using the event interface as shown in Figure 7. Through the event interface the user identifies only those member functions that generate events[6]. Using this information, event detection (and hence rule checking) is limited to only those methods designated as potential primitive events. The point at which the event should be detected is specified by the *begin* or *end* prefixes. Primitive events are specified using the event interface in the public, private and protected sections of a subclass of the Reactive class as shown in Figure 7.

In the employee class definition shown in Figure 7 begin-of-method events will be generated when an employee object receives the private Change_Salary and the public Get_Age messages. End-of-method events will be generated as a result of executing the methods Get_Salary and Get_Age. A method may generate both events (e.g., Get_Age). The method Get_Name does not generate an event, and hence its invocation cannot cause any rule execution. Composite events are specified using the operators described earlier. They are automatically translated into composite event detector objects by the preprocessor.

Detection: As database events correspond to the execution of methods, event signaling requires trapping the invocation of (or return from) a method designated as an event generator. This can be accomplished in at least three ways: i) modifying member function definitions to insert code which signals the event at the appropriate place in the body of the method, ii) creating a wrap-around function for each member function specified as an event without modifying the original function, and iii) intercepting all method invocations of a reactive object (by creating a new class definition that traps the method invocation designated as an event generator and in turn invokes the original method).

The last alternative is preferable as it does not alter the original method name. The new class method has the same name as the method to be detected and invokes that method. The effect of this is to create a wrap-around method for compile time environments. The difference

[5] In Zeitgeist persistence is provided by the zg-pos class. Since the Event and Rule classes are subclasses of the zg-pos class, event and rule objects can be designated as persistent.

[6] To avoid generating twice the number of member functions defined on that class as events.

```
class Employee : Reactive {                    /* make Employee class reactive */
   float salary;
   event begin Change_Salary(float x);                     /* event interface */
   public:
      event end Set_Salary();                              /* event interface */
      event begin && end Get_Age();                        /* event interface */
      char* Get_Name(); };

/* method Set_Salary */
Employee::Set_Salary (float value) {
   salary = value;
   return; };
```

Figure 7: Event Interface Specification for a Reactive Subclass

between the last two alternatives is that the second alternative requires changing the name of the member function (designated as an event generator) whereas the last one does not require that the member function name be changed. In an environment such as Common Lisp with Flavors, the last alternative can be implemented using the system provided feature such as Whopper without changing the user defined method in any way.

Currently, the second approach is used in our implementation. This approach does not require modification of user defined methods. A wrap-around member function is introduced for each event specification. For the event end set_salary in Figure 7, the wrap-around function created is shown in Figure 8. Rather than insert code that signals the event before each return statement in the method Set_Salary, a new method is created. This newly created method invokes the method Set_Salary and then signals the event. In this way the method Set_Salary is not modified and hence does not require recompilation. This approach: i) does not need a full C++ parser, ii) provides a way of differentiating methods that generate events from those that do not generate events, and iii) from a software engineering viewpoint, there is some correspondence between specification and usage which is useful for developing, testing, debugging, and maintenance tools.

```
/* created wrap-around method */
   end_Set_Salary (float value) {
     /* Notify(...)           if it were a begin event */
     Set_Salary(value);
     Notify(...) };
```

Figure 8: An example of a Sentinel wrap-around method

Signaling: Primitive event recorder objects (notifiable objects) receive and record the occurrence of events signaled by a reactive object. Reactive objects notify these recorders when methods are invoked and each recorder checks whether an event received corresponds to the one expected. Reactive objects signal rules, which in turn signal a composite event detector, which finally notifies primitive event recorders. They are created by specifying the *method* which generates the event and *when* the event should be generated. A primitive event object will only receive events from the rule it belongs to. Similarly, a rule will only receive events from the objects it has subscribed to. For instance, a primitive event recorder that detects the `end` of the execution of the method `Set_Salary` by an instance of Employee can be created by:

```
Event* empsal = new Primitive (``end Employee::Set_Salary(float x)'' );
```

The parameter of the Primitive constructor is the signature of the event which uniquely identifies *the method* that generates the event in addition to specifying *when* the event should be generated. Therefore, the event is signaled `after` the execution of the method `Set_Salary` defined in the `Employee` class. Each execution of the method (by any instance) generates only one event which is sent to the all the notifiable objects subscribed to that event occurrence. When specifying a primitive event recorder, there is no need to specify the instance being monitored, as an event recorder is part of a rule which monitors a particular object. Event detection will be performed only on the instances monitored by the rule.

Composite events are instances of one of the Event subclasses (see Figure 6) representing composite events. Composite event detectors are objects which are created, modified, deleted and designated as persistent in the same fashion as other objects. For instance, a composite event detector for an event which is detected and signaled *after* depositing money into a bank account *followed* by an *attempt* to withdraw money is created by:

```
Event* deposit = new Primitive (``end Account::Deposit(float x)'' );
Event* withdraw = new Primitive (``before Account::Withdraw(float x)'' );
Event* Deposit_followedby_Withdraw = new Sequence (deposit, withdraw)
```

The last event is signaled when the method *Deposit* is executed followed by an attempt to execute the method *Withdraw*. Currently, the operators disjunction, conjunction and sequence are supported.

5.2 Reactive and Notifiable Objects

The public interface of the Reactive class (shown in Figure 9) consists of methods using which objects acquire reactive capabilities. Each class derived from the Reactive class inherits the private data member *consumers* and the four methods *Reactive*, *Subscribe*, *Unsubscribe* and *Notify*.

```
class Reactive {                 /* notifiable objects that consume events */
 list-of-notifiable-subscribers *consumers;
  public :
     Subscribe (Notifiable *obj);
     Unsubscribe (Notifiable *obj);
     Reactive() {consumers = Null; };
     Notify (int *obj, char *event-name, time timestamp, int argc ...);
};
```

Figure 9: The Reactive Class.

The *consumers* data member stores as its value the set of notifiable objects associated with events generated by a reactive object. When a reactive object generates an event, it will be sent to the set of notifiable objects stored in the attribute *consumers*. The *Subscribe* method manages the set of notifiable objects associated with each reactive object. The parameter of the Subscribe method, *obj*, is the identity of a notifiable object wishing to be notified of generated primitive events. The Subscribe method appends the notifiable object to the consumers attribute. The Unsubscribe method reverses the effect produced by the Subscribe method. It takes as its parameter the identity of a notifiable object and removes it from the set of notifiable objects associated with a reactive object. The set of objects in the consumers attribute are notified of the primitive events via the Notify method. The Notify method informs the consumers of : i) the identity of the reactive object generating the primitive event, ii) a unique string identifier that indicates the event generated along with whether it was generated `before` or `after` execution of the method, iii) a time-stamp indicating the time when the event was generated, and iv) the number and actual values of the parameters of the message invoked by the reactive object.

The *Notifiable* class allows objects to receive and record primitive events generated by reactive objects. Both the Event and the Rule class are subclasses of the Notifiable class; they receive and record primitive events generated by reactive objects. The *Record* method of the Notifiable class documents the parameters computed when an event is signaled. It takes as its parameters the identity of the reactive object which generated a primitive event, the primitive event generated, the time-stamp indicating when the event was signaled, and the number and actual values of the parameters sent to the reactive object. It records these parameters if the event is signaled.

5.3 Rule Class and Rule Creation

A rule is defined by *the event* whose occurrence triggers the rule, *the condition* which is evaluated when the rule is triggered, and *the action* which is executed if the condition evaluates to true. Two alternatives for implementing rules in an OODBMS exist. First, a class can be created for each rule and the rule be an instance of that class. The condition and action parts of the rule will be implemented as methods of the newly created class. The second alternative is to create only one rule class and instantiate each rule as an instance of that rule class. The condition and action part of each rule can be implemented as methods in the rule class. The advantage of the first approach is that the addition, deletion, and modification of rules can be easily supported since it does not affect the definition of other classes; only the class definition of the rule is modified. The disadvantage is that the number of classes defined may be potentially very large since a class is created for each rule. The advantage of the second approach is that only one class is defined and all rules are instances of that class. However, each time a rule is added the definition of the rule class is modified, specifically each time a rule is added a new condition and a new action member function are added to the rule class.

In Sentinel, we have chosen the second approach for implementing rules. Rules are notifiable objects having an event object as an attribute, and the condition and action as public member functions. In addition, the rule operations create, delete, update, enable and disable are implemented as methods. Each notifiable rule object consists of data members *name*, *event-id*, *condition*, *action*, *mode* and *enabled*. The rule attribute *name* takes as its value the name of the rule and can be used by the user to access the attributes and methods of the rule. The rule attribute *event-id* denotes the identity of the event object associated with the rule. The data members *condition* and *action* are pointers to the condition and action member functions, respectively. The attribute *mode* denotes the coupling mode while the last attribute *enabled* denotes whether the rule is enabled or not. When a rule is enabled it receives and records signaled primitive events. The condition method is executed when the corresponding event occurs, and if satisfied, the action method is executed.

In Sentinel, rules are classified into class level and instance level rules. Class level rules are applicable to all instances of a class whereas instance level rules are applicable to specific instances and can possibly span several classes. Since class level rules model the behavior of a particular

```
class Person : Reactive {                    /* make person class reactive */
  public :
    event begin Marry (Person* spouse);          /* event interface */

                  /* class level rule specification */
  Rules :
    R : Marriage;
    E : Event* marry = new Primitive ("begin Person::Marry (Person* spouse)");
    C : if sex == spouse.sex
    A : abort;
    M : Immediate                         /* coupling mode */
};
```

Figure 10: A Class Level Rule.

class, they are declared within the class definition itself. On the other hand, instance level rules are declared in the application code. Rules, regardless of where they are declared, are translated into notifiable rule objects. The declaration of a class level rule entails specifying a *rule name*, an *event*, a *condition*, an *action*, and a *coupling mode*. Class level rules are declared in the rule section of a class as shown in Figure 10. The rule name is *Marriage*, the event is a person object receiving the message *Marry*, the condition checks whether the person objects getting married are of the same sex, and the action aborts the triggering transaction. Notice that the method Marry is declared as a primitive event generator inside the person class definition. This rule, when enabled, is applicable to all person objects. The rule is executed using the coupling mode specified. Class level rule specification can be viewed as a rule interface for a given object class providing information about the rules declared for that class.

Instance level rules, on the other hand, are applicable to only those instances explicitly specified by the user. Let us assume that a specific employee, *Fred*, and his manager *Mike*, should always have the same yearly income. Therefore, whenever Fred or Mike update their income this rule should be checked. Notice that this rule is applicable to instances from different classes, specifically, the employee and the manager classes. Creation of an instance level rule is illustrated in Figure 11. This rule has as its event a composite event that is signaled when an employee object executes the method Change_Income *or* a manager object executes the method Change_Income. Both these methods must be declared as primitive event generators in their respective class definitions. The condition part of the rule checks whether the incomes are equal and the action sets the incomes to the same amount. In order for the *IncomeLevel* rule object to be notified of the events generated by the employee object Fred and the manager object Mike, the rule must subscribe to those objects.

5.4 Implementation Status

Currently the above design has been incorporated into Zeitgeist. A preprocessor (not a full C++ parser) transforms the event interface and class level rules, respectively, into event generators (using wrap-around methods) and rule objects. A Wall street application for managing portfolios has been implemented using the active capability supported by this design. Composite events are also supported using the event operators.

```
Employee Fred;
Manager Mike;
Event* emp = new Primitive ("end Employee::Change-Income(float amount)");
Event* mang = new Primitive ("end Manager::Change-Income(float amount)");
Event* equal = new Disjunction (emp, mang);
Rule IncomeLevel (equal, CheckEqual(), MakeEqual());          /* Rule creation */
Fred.Subscribe (IncomeLevel);          /* Rule subscribes to events generated by Fred */
Mike.Subscribe (IncomeLevel);          /* Rule subscribes to events generated by Mike */
```

Figure 11: An Instance Level Rule.

```
                         /* code in application program */
Stock IBM;
Portfolio Parker;
FinancialInfo DowJones;

Event* stockprice = new Primitive ("end Stock::SetPrice(float amount)");
Event* newvalue = new Primitive ("end FinancialInfo::SetValue(float amount)");
Event* pricevalue = new Conjunction (stockprice, newvalue);
                         /* Rule creation */
Rule* Purchase(pricevalue, PurchaseCondition(), PurchaseAction(), Immediate);
IBM.Subscribe(Purchase);          /* Rule subscribes to events generated by IBM object */
DowJones.Subscribe(Purchase);  /* Rule subscribes to events generated by DowJones object */
```

Figure 12: Instance level rule (with a composite event) spanning two classes.

6 Functionality of Sentinel

In this section we provide an example that highlights the features of the design and implementation presented in this paper and compare our approach with Ode and ADAM. This comparison will highlight rule specification and expressiveness of our approach.

Consider a rule that requires the monitoring of events spanning several objects from different classes and further the rule can be meaningfully specified only at runtime. A portfolio *Parker* is interested in purchasing IBM stock if its price is less than $80 and the percentage change in the DowJones Industrial average is less than 3.4%. Therefore, the rule needs to monitor IBM price changes *and* DowJones value changes and is triggered when both these changes occur. Hence, this rule can be modeled by constructing a composite event created by using the conjunction operator. Although Ode supports composite events, this event cannot be expressed in Ode. This is because the event spans two classes. ADAM cannot express this event since it does not support composite events. Therefore, we consider the Sentinel approach only.

First, the event interfaces of the *Stock*, *Portfolio* and *FinancialInfo* classes need to be specified when defining the classes. Invocation of the method *SetPrice* in the Stock class the method *SetValue* in the FinancialInfo class need to generate events and thus are part of the event interface of their respective classes. No methods in the Portfolio class are declared as event generators. The next step entails creating the event object which monitors these two events. This event object is an instance of the Conjunction class. The rule object *Purchase* is then created. Its event is the *pricevalue* event object and its condition is the method *PurchaseCondition* which checks whether the IBM price is less than \$80 and whether the DowJones percentage change is less than 3.4%. If the condition evaluates to true, the method *PurchaseAction* will be invoked and it will purchase IBM stock for the Parker portfolio. After creating the rule, it subscribes to the events generated by the IBM stock instance and the DowJones FinancialInfo instance as shown in Figure 12.

7 Conclusions

In this paper we have presented a design for seamless integration of ECA rules having expressive events into an object-oriented environment. The functionality supported in Sentinel subsumes the functionality of earlier work on adding active functionality in the OO environment as illustrated in Section 6. The design expressly takes into account the characteristics of the OO environment and the differences between the relational and OO models. We have introduced a new monitoring viewpoint, termed *external monitoring viewpoint* that separates object and rule definitions from the event specification and detection process. This results in a modular and extensible system. Event detection and rule execution mechanisms can be easily changed/replaced without changing object definitions. Furthermore, this monitoring viewpoint allows objects to monitor and react to their own state changes as well as the state changes of other objects. We have outlined the implementation of compile time as well as runtime rules. We have introduced a demand-based subscription mechanism. We have introduced an event hierarchy and a set of operators (modeled as detector objects of an event class) for supporting composite events. The complete language presented in this paper is currently being implemented.

Acknowledgements

This work was supported in part by the NSF Grant (IRI-9011216), the Office of Naval Technology and the Navy Command, Control and Ocean Surveillance Center RDT&E Division, and Sofréavia Services, France.

References

[1] E. Anwar, L. Maugis, and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. In *Proceedings, International Conference on Management of Data*, pages 99–108, Washington, D.C., May 1993.

[2] C. Beeri and T. Milo. A model for active object-oriented databases. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 337–349, Barcelona (Catalonia, Spain), Sept. 1991.

[3] J. A. Blakeley, C. W. Thompson, and A. M. Alashqur. Zeitgest query language (zql). Technical Report TR-90-03-01, Texas Instruments, March 1990.

[4] S. Chakravarthy. Active Database Management Systems: Requirements, State-Of-The-Art, and an Evaluation. In H. Kangassalo, editor, *Entity-Relationship Approach: The Core of Conceptual Modeling*, pages 461–473. Elsevier Science Publishers, North-Holland, 1991.

[5] S. Chakravarthy et al. HiPAC: A Research Project in Active, Time-Constrained Database Management, Final Report. Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA, Aug. 1989.

[6] S. Chakravarthy and D. Mishra. An event specification language (snoop) for active databases and its detection. Technical Report UF-CIS TR-91-23, University of Florida, E470-CSE, Gainesville, FL 32611, Sep. 1991.

[7] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data and Knowledge Engineering*, 1994. (To appear).

[8] S. Chakravarthy and S. Nesson. Making an Object-Oriented DBMS Active: Design, Implementation and Evaluation of a Prototype. In *Proc. of Int'l Conf. on Extended Database Technology (EDBT), Venice, Italy*, pages 482–490, Apr. 1990.

[9] O. Diaz, N. Paton, and P. Gray. Rule Management in Object-Oriented Databases: A Unified Approach. In *Proceedings 17th International Conference on Very Large Data Bases*, Barcelona (Catalonia, Spain), Sept. 1991.

[10] S. Gatziu and K. R. Dittrich. Events in an object-oriented database system. In *Proc. of the 1st International Conference on Rules in Database Systems*, September 1993.

[11] S. Gatziu and K. R. Dittrich. Detecting composite events in active databases using petri nets. In *Proc. of the 4th International Workshop on Research Issues in data Engineering: Active Database Systems*, pages 2–9, February 1994.

[12] N. H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proceedings 17th International Conference on Very Large Data Bases*, pages 327–336, Barcelona (Catalonia, Spain), Sep. 1991.

[13] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event Specification in an Object-Oriented Database. In *Proceedings, International Conference on Management of Data*, pages 81–90, San Diego, CA, June 1992.

[14] H. V. Jagadish and X. Qian. Integrity Maintenance in an Object-Oriented Database. In *Proceedings International Conference on Very Large Data Bases*, Vancouver, BC, Canada, Aug. 1992.

[15] C. B. Medeiros and P. Pfeffer. A Mechanism for Managing Rules in an Object-oriented Database. Technical report, GIP Altair, Sept. 1990.

[16] M. Stonebraker. The integration of rule systems and database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):415–423, October 1992.

[17] J. Widom and S. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In *Proc. of ACM-SIGMOD*, pages 259–270, May 1990.