A New Perspective on Rule Support for Object-Oriented Databases*

E. Anwar **L. Maugis** **S. Chakravarthy**

Database Systems Research and Development Center
Computer and Information Sciences Department
University of Florida, Gainesville, FL 32611
sharma@snapper.cis.ufl.edu

Abstract

This paper proposes a new approach for supporting reactive capability in an object-oriented database. We introduce an *event interface*, which extends the conventional object semantics to include the role of an event generator. This interface provides a basis for the specification of events spanning sets of objects, possibly from different classes, and detection of primitive and complex events. This approach clearly separates event detection from rules. New rules can be added and use existing objects, enabling objects to react to their own changes as well as to the changes of other objects.

We use a runtime subscription mechanism, between rules and objects to selectively monitor particular objects dynamically. This elegantly supports class level as well as instance level rules. Both events and rules are treated as first class objects.

1 Introduction

The need and the relevance of reactive capability as a unifying paradigm for handling a number of database features are well-established. 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 [C$^+$89, SHP88, WF90, DB90, Int90]. Recently, there have been a number of attempts [GJS92, GJ91, DPG91, MP90, CHS93, Anw92, SKL89] at incorporating event and rule support into an object-oriented database management system (OODBMS).

Clearly, there is a paradigm shift when we move from the relational model to an OO one. This warrants reexamination of the functionality as well as the mechanism by which reactive capability is incorporated into the OO data model [BM91]. 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 events in the relational model, every method/message is a potential event,
2. The principle of encapsulation and further the distinctions between features supported (e.g., private, protected and public in C++) 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) and its effect on rule incorporation, and
4. Scope, accessibility, and visibility of object states for rules.

Furthermore, the following performance issues were considered:

1. Effect of rule specification only at class definition time and its activation and deactivation at runtime. This entails changing the class definition every time rules are added or deleted,
2. Rule management. For example, cost incurred in associating class level rules (rules that are applicable to every instance of a class) and other types of rules, and
3. Event management. For example, cost incurred for event detection (both primitive and complex) as the number of events can be very large in contrast to the relational case.

The approaches taken so far for incorporating rules into an OODBMS can be broadly classified into: i) specification of (parameterized) rules only at the class definition time (allowing binding of a rule to an instance, its activation, and deactivation at runtime) and ii) rule creation, activation, deactivation, and binding at runtime. The first approach is motivated by efficiency considerations and keeps the runtime processing (not necessarily the overhead for rule processing depending on the implementation) low and does not require any new classes for supporting rules. All specifications are pre-processed into the code of the host language. The primary drawback of this approach is that the integration is somewhat *ad hoc* and provides little or no support for the runtime specification of rules. On the other hand, the second approach tries to accomplish everything at runtime thereby incurring a reasonable amount of overhead.

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

In this approach, it is cumbersome to make a rule applicable to only a small number of instances. To the best of our understanding, Ode [GJ91, GJS92] has taken the first approach and ADAM [DPG91] the second one. It is likely that the environments used by these two systems (C++ and PROLOG, respectively) have been a factor for the approaches.

1.1 Contributions

In this paper, we take the view that the two approaches outlined above represent two end points of a spectrum; individually, neither approach fully meets the functionality and seamless [1] requirements of rule support for an OO database. Our approach 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).

Our approach synthesizes the advantages of both the approaches outlined and further extends them in several significant ways. Briefly, we support rules that are specified at class definition time (Ode style) and rules that can be constructed at runtime (ADAM style) and compile both using a uniform framework. In addition, we support primitive events and event operators for constructing complex events as first class objects. We also support rules as first class objects.

Most importantly, we introduce a monitoring viewpoint (termed *external monitoring viewpoint*) that is not present in either Ode or ADAM. This viewpoint permits: i) rule definition to be independent from the objects which they monitor, ii) rules to be triggered by events spanning sets of objects (inter-object rules), possibly from different classes, and iii) any object to *dynamically* determine which objects' state changes it should react to and associate a rule object for reacting to those changes. We present an implementation of this external monitoring viewpoint in the OO framework. We consider this generalization extremely important as the expressiveness and the extensibility of the resulting system is significantly enhanced (Ode has tried to implement the functionality of inter-object rules in a straightforward manner by making the same set of rules applicable to more than one object class [JQ92]). This feature enables the seamless integration of rules as well.

The remainder of this paper is structured as follows. Section 2 provides the motivation for our approach. In section 3 we provide the design overview and the rationale behind it. Section 4 provides implementation details. In section 5 we contrast the functionality of our system, Sentinel, with Ode and ADAM through an illustrative example. Section 6 briefly describes Ode and Adam leading to a back-of-the-envelope comparison and future research directions in section 7.

[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.

2 Motivation

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

- Although current approaches allow a rule to monitor one or more instances of the same object class, a rule is triggered by changes occurring to *only one* of the instances it monitors. To enhance expressiveness, support for rules which are triggered by changes occurring to one or more instances, possibly from different classes, is necessary,
- Some systems permit rule specification only within class definitions. This will lead 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,
- Rules and events are not always treated as first class objects, thereby resulting in a dichotomy between them and other objects. Rules and events 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 [GJ91] supports the specification and detection of complex events, the manner in which they are supported prevents expressing events spanning instances of the same as well as different classes. Furthermore, events can only be defined within a class thus perpetuating the problems outlined above.

2.1 Need for External Monitoring Viewpoint

In a number of applications, such as patient databases, portfolio management, and network 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 be a requirement that the stock object itself modifies its attributes or behavior for a new portfolio object to monitor it.

If one has to declare all the rules that are likely to be associated with an object at the class definition time, clearly, the above application requirements cannot be supported (as that information is mostly not 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, it is imperative to support the above requirement through an interface (analogous to the traditional interface for objects). We introduce an **event interface** for this purpose. It is equally important to separate the (visible) event interface from: i) its implementation and ii) the use (or invocation) of that interface by other objects. The event interface needs to be defined (or revealed) at the class definition time whereas its use needs to be supported at runtime.

Below, we give an example of how the external monitoring viewpoint also supports one or more objects to be monitored by a rule preserving both encapsulation and independent persistence.

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

rule Purchase :
when IBM→SetPrice And DowJones→SetValue
if IBM→Price < $55 & DowJones→Change < 3.4%
then Parker→PurchaseIBMStock
```

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

3 Design Rationale

Our design choices, substantiated in the remainder of this section, can be summarized as follows:

1. Augment the interface of conventional C++ objects with an **event interface** which has the ability to raise and propagate events occurring on their state, providing encapsulation,
2. Support primitive events, event operators, and rules as first class objects [2],
3. Allow rules to be triggered by events spanning several objects,
4. Allow an object to dynamically specify *which* objects to react to in response to their state changes, and
5. Provide an efficient mechanism for associating rules to all instances of a class as well as to a subset of instances, possibly from different classes.

[2]Even rules that are declared as part of the class definition are translated into instances of rule objects; of course, there existence is dependent on the existence of the object class.

To elaborate: (1) and (3) extend the expressive power of the resulting system, preserve encapsulation, support monitoring of multiple objects possibly from different classes, thereby reducing the number of rules. (2) 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. At runtime, new rules can be added and associated/applied with/to existing objects in the database, i.e., the addition of rules does not affect the definition of objects currently existing in the system. Consequently, the extensibility and modularity of the resulting system is not compromised. (4) facilitates binding of rules to event, condition and action at runtime by choosing an appropriate implementation for (1).

Also, our design allows incorporation of new features (for example, providing a new conflict resolution strategy) without modifications to application code.

3.1 External Monitoring Viewpoint

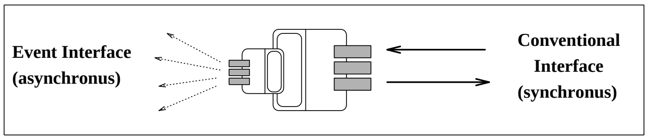

Figure 1: Behavior of a reactive class.

To treat rules independently from the objects they monitor, objects must be capable of: i) generating events when their methods are invoked and ii) propagating these events to other objects. To achieve these capabilities we extend conventional C++ objects with an **event interface**. This interface enables objects to designate some, possibly all, of their methods as primitive event generators. The implementation of the interface specification is through primitive event generators that raise an event when a method is invoked. The augmented C++ 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 generate events for the methods (defined using the event interface) when they are invoked and propagate these events to other objects asynchronously. Events are generated either *before* or *after* the execution of a method. A class that supports the external monitoring viewpoint is **termed as a reactive class** and is defined as :

Reactive class definition = Traditional class definition
+ Event interface specification

Using the event interface, events are specified as part of the class definition by the user. The event interface only specifies the events that are to be produced by that reactive object class. The semantics of the event interface is that every instance of the Reactive class will generate and signal an event for methods 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 and further only those methods that change the state that one is interested in monitoring, need to be defined in the event interface. In contrast to the conventional interface which is specified and implemented by the user, only the event interface is specified by the user; its implementation is provided by the system. The event message generated by the Reactive class consists of the following parameters :

$$\textit{Generated primitive event} = \textit{Oid} + \textit{Class} + \textit{Method} + \textit{Actual\_parameters} + \textit{Time\_stamp}$$

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 object class. An instance of a notifiable class is a consumer of an event that is of interest to that class. An association is established between an event and a notifiable object using the **subscription** mechanism.

In contrast to Ode, only primitive event specifications are part of the reactive object's class definition. The rule itself, which monitor objects, is not required to be part of the class. Rules and event operators are the consumers of primitive events generated by instances of the Reactive object class, and use these events to detect primitive and composite events.

3.2 Object Classification

In Sentinel, objects are classified into three categories : **passive**, **reactive** and **notifiable**. As with other OO databases, a designer creates a schema which defines classes for an application. However, he/she needs to also define which object classes are reactive, to produce appropriate events, and which object classes are notifiable, to make them consume and detect events.

Passive objects : These are regular C++ objects. They can perform some operations but do not generate events. An object that needs to be monitored and inform other objects of its state changes cannot be passive. No overhead is incurred in the definition and use of such objects.

Reactive objects : Objects that need to be monitored, or 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. Once a method is declared as an event generator (through the event interface), its invocation will be *propagated* to other objects. Thus, reactive objects communicate with other objects via event generators.

Notifiable objects : Notifiable objects, on the other hand, are those objects capable of being informed of the events generated by reactive objects. Therefore, notifiable objects become aware of a reactive object's state changes and can perform some operations as a result of these changes.

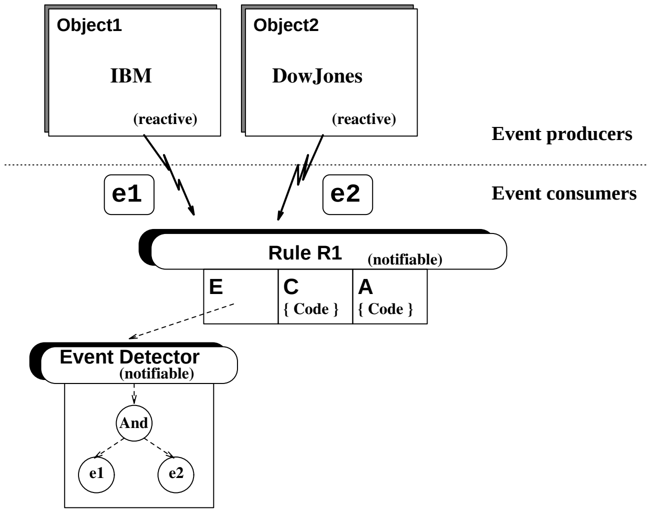

Figure 2: An Event Producer/Consumer Analogy.

Figure 2 illustrates the producer/consumer behavior of object types. Two independent objects *object1* and *object2* generate primitive events *e1* and *e2*, sending them to a rule *R1*. The rule passes the events to the event detector for storage and event detection, and if the event is detected, the rule checks the condition and takes appropriate actions.

Notifiable objects subscribe to the primitive events generated by reactive objects. After the subscription, the reactive objects propagate their generated primitive events to the notifiable objects. Lastly, the notifiable objects perform some operations as a result of these propagated events. The operations can affect passive, reactive and notifiable objects. There is a m:n relationship between notifiable and reactive objects; that is a reactive object instance can propagate events to several notifiable object instances and a notifiable object instance can receive events from several reactive object instances. Events and rules are examples of notifiable objects. Rules receive events from reactive objects, send them to their local event detector, and take appropriate actions. Event detectors receive events from reactive objects, store them along with their parameters, and use them to detect primitive and complex events.

3.3 Events

Several approaches are possible for event specification in an OO context. Currently, three approaches are used: i) events as expressions declared within class definitions, e.g., Ode [GJ91, GJS92], ii) events as rule attributes, e.g., Bauz [MP90], and iii) events as first class objects, e.g., ADAM [DPG91]. Below, we discuss the advantages and disadvantages of each approach.

Events as Expressions : This approach is motivated by runtime processing gains, since processing of event

[3]The primitive events before and after correspond to the invocation and return of methods. These two can be automatically generated; also, the designer can also explicitly generate other primitive events, within the body of the method.

specification is performed primarily at compile time and little or none at runtime. The main disadvantage is that events cannot be added, deleted or modified at runtime, thereby resulting in a dichotomy between events and other types of objects. Further, persistence of events is dependent on the existence of other objects. More importantly, events spanning distinct classes cannot be expressed. In addition, events cannot have attributes or methods of their own and hence cannot store and access the parameters computed when the event is raised. Lastly, new event types or event attributes cannot be easily incorporated, thereby compromising the extensibility.

Events as Rule Attributes: This alternative improves upon the former approach by allowing events to be added, deleted and modified dynamically. Another advantage is that event and rule association is achieved since events are part of a rule's structure. However, this approach suffers from the same disadvantages as those of the first approach.

Events as Objects: This alternative has several advantages and is superior to the former alternatives. First, this approach models the properties of events. Events have a state, structure and behavior, i.e., events exhibit the properties of objects. The state information associated with each event includes the occurrence of the event and the parameters computed when an event is raised. The structure of an event consists of the event(s) it represents while the behavior consists of specifying when to signal the event. Second, events can be created, deleted, modified, and designated as persistent as other types of objects, i.e., events are treated in a uniform manner as other objects. Furthermore, the introduction of new event types/attributes can be easily incorporated by modifying/augmenting class definitions without compromising the extensibility and modularity of the system. Moreover, events spanning distinct classes can be expressed. However, with this approach, runtime overhead is incurred when events are created, deleted and modified dynamically.

In Sentinel, we adopt the third alternative and treat events as first class objects. Furthermore, we construct complex events using a hierarchy of event operators. Event objects are consumers of events generated by reactive objects.

3.4 Rules

The OO environment offers numerous design alternatives for the incorporation of rules. Rules can be specified declaratively, embedded inside other objects as attributes or data members, or as objects. Below, we discuss the advantages and disadvantages of each alternative.

Rules as declarations only inside classes : Rules are declared by the user and then inserted by the system into each place in the code where they might be triggered. It is necessary to first determine *where* and *how* rules should be declared. Rules are associated with objects and contribute to their behavior. Thus, the natural place for declaring rules is within class definitions. We shall not discuss rule declaration syntax since it does not affect the active functionality provided. The primary advantage of this approach is that rule processing is performed primarily at compile time, and hence little or no rule processing is performed at runtime. Furthermore, the declaration of rules within class definitions offers an easy mechanism for determining the rules applicable to objects; this information is easily obtained from the class definitions themselves. In addition, the inheritance of rules is easily supported.

This approach does not treat rules as objects and their existence is dependent on the existence of other objects. Furthermore, the system is not extensible since the introduction of new rule components, e.g., rule priority levels, requires modifying class definitions containing rule declarations. The main disadvantage of this approach lies in its inefficiency in handling the addition, deletion and modification of rules. This is because changing the rules defined for objects requires the modification of class definitions and thus recompiling the system. This presents a major problem for interpretive OO environments. Furthermore, modification of a class definition may present some difficulties to already existing and stored instances of the class, thereby compromising the extensibility of the system since addition of rules should be allowed irrespective of already existing objects in the system. In addition, rules cannot be reused or shared. For example, a rule that ensures an employer's salary is always less than his/her manager's salary needs to be declared twice – once within the employee class and once within the manager class.

Rules as Data Members : By treating rules as data members we must first find a convenient type to model them. Let us assume that an appropriate type has been determined[4]. The advantage of this approach is its reusability 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 easily associated with objects since they are part of an object's structure. In addition, rules can be easily 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. Second, a rule's existence is dependent on the existence of other objects.

Rules as Objects : There are numerous advantages to treating rules as objects. First, rules can created, modified and deleted in the same manner as other objects, thus providing a uniform view of rules in an OO context. Second, rules are now 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

[4] This excludes the possibility of a class. This possibility is also examined.

semantics as other objects. Third, each rule will have an object identity, thereby allowing rules to be associated with other objects. Fourth, 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 [GJ91, GJS92] can be modeled as subclasses of the rule class. Lastly, by treating rules as first class objects an extensible system is provided. This is due to the ease of introducing new rule attributes or operations on rules; this requires the modification of the rule class definition only.

In Sentinel, we adopt the latter alternative and chose to treat rules not only as first class, but also as notifiable objects.

3.5 Rule Association

Rules in active 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 active OODBMS due to a fundamental feature of the OO paradigm, viz. *abstraction*. An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects. Rules defined on an object undoubtedly contribute to the essential characteristics, especially behavior of an object. In many applications, objects differ considerably in both structure and behavior from one another. Therefore, it is realistic to assume that different kinds of objects may have different rules applicable to them.

To accommodate rules in an OO environment, we classify rules into two main categories, namely, class level 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. To associate rules with objects, we introduce a **subscription mechanism**. This mechanism allows notifiable objects (rules in this case) to dynamically subscribe to the 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 can be implemented on varying degrees of granularity. For example, a notifiable object may subscribe to *all* the events defined in the event interface of a reactive object or subscribe to *specific* events. The latter case is more efficient since rules are checked only when specific events defined in the event interface of a reactive object are generated. This is in contrast to checking rules whenever each event defined in the event interface is generated. However, the former approach uses less storage since only one list of notifiable objects needs to be maintained per reactive object. The latter approach needs to maintain one list for each message defined in the event interface of a reactive object.

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 adopting a centralized approach where all rules defined in the system are checked when events are generated. 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 Implementation Details

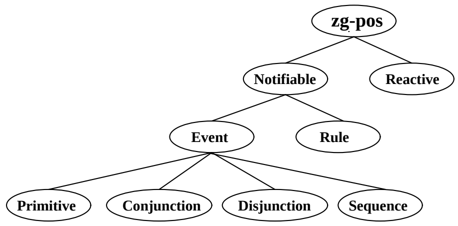

Figure 3: Sentinel Class Hierarchy for Rule Support.

The Sentinel system is being developed using Zeitgeist, a C++ OODBMS developed at Texas Instruments[PP91]. To incorporate rules in Zeitgeist we modified the class hierarchy to include the *Reactive*, *Notifiable*, *Event* and *Rule* classes. The class hierarchy introduced for rule support is illustrated in Figure 3. In Zeitgeist, persistence is provided by the *zg-pos* class for all objects that are derived from that class. Therefore, by deriving the Rule and Event classes from the *zg-pos* class, rule and event objects can be designated as persistent[5]. The Rule and Event classes are derived from the Notifiable class in order for rule and event objects to act as consumers, i.e., be capable of receiving and recording the events propagated by reactive objects.

In the following subsections we briefly outline the implementation of the Reactive, Notifiable, Event and Rule classes.

4.1 The Reactive Class

```
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 4: The Reactive Class.

The public interface of the Reactive class consists of methods by which objects acquire reactive capabilities.

[5] *Reactive* and *Notifiable* are designated for persistence, their instances can be made persistent (or transient).

Each class derived from the Reactive class inherits the private data member *consumers* and the four methods *Reactive*, *Subscribe*, *Unsubscribe* and *Notify*.

Each reactive object's definition is enlarged with the private data member *consumers*. This data member stores as its value the set of notifiable objects associated with events generated by a reactive object. When a reactive object generates events, they will be consumed by the set of notifiable objects listed in the attribute *consumers*. The Subscribe method appends a notifiable object to the consumers attribute. The Unsubscribe method reverses the effect produced by the Subscribe method. The set of objects in the consumers attribute are notified of the generated 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.

4.2 The Notifiable Class

The primary objective for defining the *Notifiable* class is for allowing objects to receive and record primitive events generated by reactive objects. Both the Event and Rule classes are subclasses of the Notifiable class; they receive and record primitive events generated by reactive objects. The *Record* method defined in the Notifiable class documents the parameters computed when an event is raised. 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 raised, and the number and actual values of the parameters sent to the reactive object.

4.3 The Event Hierarchy

Event specifications are translated into first class objects which are created, deleted, modified and designated as persistent as other types of objects. We support both primitive and complex events. Primitive events are in the form of messages sent to objects and are of two shades: begin of method (bom) and end of method (eom) events. bom and eom events are signaled before an object starts executing a method and immediately after an object returns from a method, respectively. Composite events are constructed by applying event operators to primitive events. Currently, the operators disjunction, conjunction and sequence are supported.

An event E constructed by applying the conjunction operator to two primitive or complex events E1 and E2, is signaled when both E1 and E2 occur, regardless of the order of their occurrence (or of their components). An event E constructed by applying the disjunction operator to two events E1 and E2, is used to signal an event when either E1 or E2 occur. An event E constructed by applying the sequence operator to the events E1 and E2, is signaled when event E2 occurs, provided E1 has occurred earlier. In the case where E1 and E2 are composite events, E is signaled when the last component of E2 occurs provided all the components of E1 have occurred[6].

```
class Conjunction : Event {
  Event  *EventOne, *EventTwo;
  int Raised;
  public :
    Conjunction(Event* FirstEvent, Event* SecondEvent);
    Notify(int obj, char* event, time timestamp, int argc ...);
 };
```

Figure 5: The Conjunction Subclass.

An Event superclass was defined to provide the common structure and behavior shared by all event types. By creating an event class hierarchy, primitive and complex events' structure and behavior were defined using inheritance. The primitive, conjunction, disjunction and sequence events are defined as subclasses of the Event class. Each subclass definition is augmented with the necessary attributes and operations required for modeling the event type it represents. Figure 3 illustrates the event hierarchy created.

The structure and behavior of the Conjunction subclass is shown in Figure 5. It consists of the data members *EventOne*, *EventTwo* and *Raised*. *EventOne* and *EventTwo* are pointers to event objects and represent the two events upon which the conjunction operator is applied. *Raised* indicates whether the event has been raised or not. The constructor of the class takes as its parameters the object identities of the event objects upon which the conjunction operator is to be applied. The last method *Notify* determines whether the events propagated raise the event or not and informs the rule object of the result.

4.4 The Rule Class

The primary structure defining a rule is *the event* which triggers the rule, *the condition* which is evaluated when the rule is triggered, and *the action* which is executed if the condition is satisfied. To model the structure of rules, a Rule class is defined as shown in Figure 6. Rules are notifiable objects having an event object as an attribute, and the condition and action as public member functions[7]. In addition, the rule operations create, delete, update, enable and disable are implemented as methods.

Each notifiable rule object consists of the 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. *event-id*

[6] Solutions for overcoming indefinite waiting for events are given in [CM91].

[7] In the current implementation, each rule defined has its own condition and action implemented as methods defined in the Rule class.

```
class Rule : Notifiable {          /* Rule class made notifiable */
     char* name;                   /* Rule name */
     Event* event-id;              /* Event*/
     PMF *condition, *action;      /* PMF is a pointer to a member function */
     Coupling mode;                /* Coupling mode */
     int enabled;                  /* Rule enabled or not */

     public:
       virtual int Enable();
       virtual int Disable();
       virtual Update(Event* eventid);
       virtual int Condition();
       virtual int Action();
       Rule(Event* eventid, PMF condition, PMF action, Coupling mode);
       ~Rule();
   };
```

Figure 6: The Rule Class.

```
class Employee : Reactive {       /* make Employee class reactive */
    float salary;
    event begin Change_Salary(float x);              /* event interface */

    public:
       event end Get_Salary();                       /* event interface */
       event begin && end Get_Age();                 /* event interface */
       char* Get_Name();
};
```

Figure 7: A Reactive Subclass.

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 *enabled* denotes whether the rule is enabled or not.

4.5 Usage of Reactive Class

Primitive events are generated by an object when it invokes a method. In the interest of reducing the amount of overhead, we recommend[8] the user to specify which member functions should generate events upon their invocation, i.e., which methods are to be treated as primitive event generators. Using this information, event generation (and hence rule checking) is limited to only those methods designated as potential primitive events. When the event should be raised is specified by the *before* or *after* prefixes. Potential 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 message events (bom) will be generated when an employee object receives the private Change-Salary and the public Get-Age messages. End of message events (eom) will be generated as a result of executing the methods Get-Salary and Get-Age. Notice that a method may generate both bom and eom events; this is the case for the member function Get-Age. The method Get-Name does not generate any events, and hence its invocation

[8] An alternative is to assume that *all* member functions are potential events which generates twice the number of member functions defined on that class.

```
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 8: A Class Level Rule.

does not cause any rule evaluation.

After specifying the event interface, the application/user needs to create the appropriate event and rule objects which are informed of the generated primitive events. This is the mechanism by which primitive and complex events are detected and their parameters recorded.

4.6 Event Creation

Events are created, modified, deleted and designated as persistent in the same manner as other objects. Creation of primitive event objects requires indicating the *method* which raises the event and *when* the event should be raised. For instance, a primitive event object that detects the *end* of the execution of the method *Set-Sal* by an employee object can be created by:

Event* sal = new Primitive("end Emp::Set-Sal(float x)");

The parameter of the Primitive constructor is the signature of the event which uniquely identifies *the method* that raises the event in addition to specifying *when* the event is raised. Therefore, the event is raised *after* the execution of the method *Set-Sal* defined in the *Emp* class.

Composite events are instances of one of the Event subclasses representing complex events. A complex event raised *after* depositing money into a bank account *followed* by an *attempt* to withdraw money is created by:

Event* DEP = new Primitive("end ACN::Dep(int x)");
Event* WTD = new Primitive("before ACN::Wtd(int x)");
Event* DepWit = new Sequence(DEP, WTD);

This event is raised when the method *Dep* is executed followed by an attempt to execute the method *Wtd*.

4.7 Creating Class and Instance Rules

Rules can be classified into class level and instance level rules depending on their applicability. Class level rules are applicable to all instances of a class whereas instance level rules are applicable to particular instances, possibly from different classes. Since class level rules model the behavior of a particular 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 *cou-*

```
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 9: An Instance Level Rule.

pling mode. Class level rules are declared in the rule section of a class as shown in Figure 8. In the above example 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 and instances derived from the person class. The rule is executed using the coupling mode specified.

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.

The instance level rule is then created as illustrated in Figure 9. This rule has as its event a complex event that is raised 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. 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.

Once the rule *IncomeLevel* subscribes to the objects Fred and Mike, all primitive events generated by Fred and Mike are propagated to the rule object. Therefore, the IncomeLevel rule object is monitoring the Employee object Fred and the Manager object Mike simultaneously.

5 Examples

In this section we provide an example which highlights the features of our approach and compares it with Ode and ADAM.

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 \$55 and the percentage change in the DowJones Industrial average is less than 3.4%. 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 using the conjunction operator. Although Ode supports complex events, it cannot express this event since it spans two classes. Since ADAM does not support complex events, it also cannot express this event. Therefore, we consider the Sentinel approach only.

```
/* 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 10: Instance level rule spanning two classes.

Features / System	Monitoring Viewpoint	Event Scope	Rule Scope	Complex Events	Events as Objects	Rules as Objects	Coupling Modes	Rules on Rules	Environ-ment
Ode	Internal	Intra-obj	Class, Instance	Relative, Prior, Sequence	No	No	I, Df, Det	No	C++
ADAM	Internal	Intra-obj	Class	No	Yes	Yes	I	Yes	PROLOG
Sentinel	Internal, External	Intra-, Inter-obj	Class, Instance	Conjunction, Disjunction, Sequence	Yes	Yes	I, Df	Yes	C++, Zeitgeist

Figure 11: Comparison of Active OODBMS Features.

First, the event interfaces of the *Stock*, *Portfolio* and *FinancialInfo* classes need to specified when defining the classes. Invocations of the method *SetPrice* in the Stock class and the method *SetValue* in the FinancialInfo class need to generate events and thus are part of the event interface of their 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 and is created as shown in Figure 10. 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 \$55 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.

6 Related Work

Although a number of efforts have addressed incorporating active capability in the context of an OODBMS [MP90, C^{+}89, SKL89], only Ode [GJ91, GJS92] and ADAM [DPG91] are pertinent to our work.

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* class definitions. Events in Ode are implicit and are considered as the disjunction of all nonconstant public methods. Constraints are applicable to *all* instances of the class in which they are declared while triggers are applicable only to those instances specified *explicitly* by the user at runtime. More recently Ode [GJS92] has proposed a language for specifying composite events which is similar to Snoop [CM91]. Detection of events is accomplished by using a finite automata.

ADAM [DPG91] is an active OODB implemented in PROLOG. Both events and rules are treated as first class 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. Rule operations are implemented as class methods. Events are classified into DB events, clock events and application events. Events are generated either *before* or *after* the execution of a method. A comparison of Ode, ADAM and Sentinel is shown in Figure 11.

7 Summary and Future Research

This paper describes the *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 processing mechanisms can be easily changed/replaced without changing the 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 supported the specification and detection of simple as well as complex events, and compile time as well as runtime rules. We have significantly reduced rule checking overhead by introducing the demand-based subscription mechanism.

Our design easily supports customizing the behavior of event and rule objects. For example, various conflict resolution strategies can be implemented by defining methods within the rule class. Also, methods defined on the rule class can be designated as event generators to define rules on the rule object class itself. Although the before and after events are supported as part of the event interface, users' can use the notify mechanism to generate (or signal) events at arbitrary points in their methods.

7.1 Future Research

We are currently investigating:

- Transformation of higher-level user specification of an active database to Sentinel,
- Support for all events and parameter contexts in Snoop [CM91].
- Performance evaluation of our design choices,
- Communication among applications and cooperative transactions using the active database paradigm.

References

[Anw92] E. Anwar. Supporting complex events and rules in an oodbms: A seamless approach. Master's thesis, DBSRDC, University of Florida, Nov. 1992.

[BM91] C. Beeri and T. Millo. A model for active object-oriented databases. In *Proc. of VLDB*, Barcelona , Sept. 1991.

[C+89] S. Chakravarthy et al. HiPAC: A Research Project in Active, Time-Constrained Database Management, Final Report. Technical Report XAIT-89-02, Cambridge, Aug. 1989.

[CHS93] S. Chakravarthy, E. Hanson, and S.Y.W. Su. Active Database Research at the University of Florida. *IEEE Quarterly Bulletin on Data Engineering*, Jan. 1993.

[CM91] S. Chakravathy and D. Mishra. An event specification language (snoop) for active databases and its detection. Technical Report UF-CIS TR-91-23, DBSRDC, University of Florida, Sep. 1991.

[DB90] M. Darnovsky and J. Bowman. *TRANSACT-SQL USER'S GUIDE, Release 4.2*. Document 3231-2.1, Sybase Inc., May 1990.

[DPG91] O. Diaz, N. Paton, and P. Gray. Rule Management in Object-Oriented Databases: A Unified Approach. In *Proc. of VLDB*, Barcelona, Sept. 1991.

[GJ91] N. H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proc. of VLDB*, Barcelona, Sep. 1991.

[GJS92] N.H.Gehani, H.V.Jagadish, and O.Shmueli. Event Specification in an Object-Oriented Database. In *Proc. of ACM-SIGMOD*, San Diego, June 1992.

[Int90] InterBase Software Corporation, Bedford, MA. *InterBase DDL Reference Manual, InterBase Version 3.0*, 1990.

[JQ92] H. V. Jagadish and X. Qian. Integrity Maintenance in an Object-Oriented Database. In *Proc. of VLDB*, Vancouver, Canada, Aug. 1992.

[Mau92] L. Maugis. Adequacy of active oodbms to flight data processing servers. Master's thesis, National School of Civil Aviation / University of Florida, Aug. 1992.

[MP90] C. B. Medeiros and P. Pfeffer. A Mechanism for Managing Rules in an Object-oriented Database. Technical report, GIP Altair, Sept. 1990.

[PP91] Edward Perez and Robert W. Peterson. Zeitgeist Persistent C++ User Manual. Information Technologies Laboratory Technical Report 90-07-02, 1991.

[SHP88] M. Stonebraker, M. Hanson, and S. Potamianos. The POSTGRES rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, Jul. 1988.

[SKL89] S. Y. W. Su, V. Krishnamurthy, and H. Lam. "An Object-Oriented Semantic Association Model (OSAM*)". *Theoretical Issues and Applications in Industrial Engineering and Manufacturing*, pages 242–251, 1989.

[WF90] J. Widom and S. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In *Proc. of ACM-SIGMOD*, pages 259–270, May 1990.