ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Dr. Sharma Chakravarthy for giving me an opportunity to work on this challenging topic and for providing continuous guidance, advice, and support throughout the course of this research work. I thank Dr. Eric Hanson and Dr. Li-Min Fu for serving on my supervisory committee and for their careful perusal of this thesis. I would like to thank Sharon Grant for maintaining a well administered research environment. I also thank many of my friends for making my stay in Gainesville memorable.

On a more personal note, I would like to thank my father Dr. Sami Anwar, my mother Mrs. Afaf Anwar, my great bother Tarek, my two wonderful sisters Hanan and Nahla, and my cute little nephew Tamer. Without their love, support and constant encouragement this work would not have been possible.

This work was (in part) suported by the National Science Foundation Research Initiation Grant IRI-9011216 and by the Office of Naval Technology and the Navy Command, Control and Ocean Surveillance Center RDT&E Division.

TABLE OF CONTENTS

ACKNOWLEDGMENT . . . i
0.1 INTRODUCTION . . . 1
0.2 PREVIOUS RELATED WORK . . . 2
0.2.1 High Performance Active Databases (HiPAC) . . . 2
0.2.2 Ode . . . 3
0.2.3 ADAM . . . 7
0.2.4 Object Integrity using Rules . . . 9
0.3 MOTIVATION AND ISSUES . . . 11
0.4 DESIGN OF ECA RULES FOR SENTINEL . . . 15
0.4.1 Design Goals . . . 16
0.4.2 Need for a Reactive Asynchronous Object Interface . . . 18
0.4.3 Alternatives to Incorporating Rules in an OODBMS . . . 20
0.4.4 Events . . . 21
0.4.5 Events as Objects . . . 24
0.4.6 Rule Specification . . . 31
0.5 Implementation Details . . . 37
0.5.1 The Reactive Class . . . 38
0.5.2 The Notifiable Class . . . 40
0.5.3 The Rule Class . . . 40
0.5.4 The Event Class . . . 43
0.5.5 Event Detection Implementation . . . 44
0.5.6 Rule Implementation . . . 44
0.6 Examples . . . 47
0.6.1 Example One . . . 47
0.6.2 Example Two . . . 54
0.6.3 Example Three . . . 58
0.6.4 Example Four . . . 60
0.6.5 Example Five . . . 63
0.6.6 Example Six . . . 64
0.7 CONCLUSION AND FUTURE RESEARCH . . . 67
0.7.1 Contributions and Conclusions . . . 67
0.7.2 Future Research . . . 69

REFERENCES . . . 70

0.1 INTRODUCTION

During the past years data base management systems (DBMS) have undergone dramatic changes as a result of the increasing requirements of modern day applications. Conventional record-oriented data base systems are subject to the limitations of a finite set of data types and the need to normalize data. These limitations have led to the evolution of a new paradigm, namely object-oriented data bases (OODBMS), which offer increased modeling power, flexible abstract data-typing facilities and the ability to encapsulate data and operations via the message metaphor. Despite the ability to model complex objects and relationships, these OODBMSs lack some of the requirements of a large class of new applications, specifically those requiring monitoring of situations and responding to them automatically, possibly subject to timing constraints.

Active data bases have been proposed to meet some of the requirements of non-traditional applications. Active OODBMSs extend the normal functionality of OODBMSs with support for monitoring user-defined situations and reacting to them without user or application intervention. These DBMSs continuously monitor situations to initiate appropriate actions in response to data base updates, occurrence of particular states or transition between states, possibly within a response time window.

The key distinction between an active and a passive object, as conveyed in the database literature, lies in an active object's ability to initiate asynchronous actions, as a separate thread of execution, without necessarily receiving messages. Typically, "passive" objects respond to messages through a *synchronous* interface; they *receive a message* and based on its interpretation *then perform* some operations and return a result. Extant active OODBMSs define active objects as objects which monitor their *own* state by executing operations *asynchronously* in response to changes occurring internally to their *own* state.

Rules, in the context of an active DBMS, consist primarily of three components: an event, a condition, and an action. An event is an indicator of a happening (either simple or complex). Events are recognized by the system or signalled by the user. For example, database events such as insert, delete, and update are detected by the OODBMS. The condition specifies an optional predicate over the database state which is evaluated when its corresponding event occurs. The conditions to be monitored may be arbitrarily complex and may be defined not only on single data values or individual database states, but also on sets of data objects, transitions between states of materialized/derived objects, trends and historical data. Actions are the operations to be performed when an event occurs and its associated condition evaluates to true. Actions may be programs which may in turn cause the occurrence of other events. Once rules are specified declaratively to the system, it is the system's responsibility to monitor the situations (event-condition pairs) and execute the corresponding action when the condition is satisfied without any user or application intervention. The advantage of using rules as a means of providing active behavior, is the freedom from explicitly hard-wiring code which checks the situations being monitored into each program that updates the database.

The mechanism by which rules are integrated into an OODBMS has a profound impact on the active functionality provided. Considerable differences exist between the approaches taken for rule integration by current active OODBMSs. In this paper, we describe a mechanism for integrating rules into an OODBMS that subsumes and enhances upon the active functionality provided by these existing systems. Our proposed approach improves upon the notion of an active object, the types of events

that can be expressed and detected, the flexibility provided and the performance of the system.

This paper is organized as follows. Section 2 begins by defining the problem statement and presenting the contributions of this paper. A detailed survey of the existing active OODBMSs then follows in Section 3. Special emphasis is given to determining the active functionality provided, the types of events supported, how rules are specified and the extensibility of the system. Section 4 proceeds by presenting our proposed approach while specifically addressing the issues involved in integrating rules in an OODBMS as well as specifying and detecting events. Section 5 presents a comparison between our approach and existing active OODBMSs based on several given examples while Section 6 discusses the implementation issues involved. The final section, Section 7, presents the conclusion and highlights future areas of work.

0.2 PREVIOUS RELATED WORK

In this section we provide an overview of the current active OODBMSs found in the literature. Later in this paper, we provide a back-of-the-envelope comparison of Sentinel with other work discussed here.

0.2.1 High Performance Active Databases (HiPAC)

HiPAC ([C+89],[DBM88]) was a research endeavor on active and time constrained data management. Event-condition-action rules form the basis for active behavior in HiPAC. Rules are treated as first class objects and each rule is an instance of the system defined rule class. Rules in HiPAC are defined by specifying a rule identifier, an event, a condition, an action, timing constraints, contingency plans and rule attributes.

Since rules are first class objects in HiPAC, they are treated in a uniform manner as other objects in the system. The rule operations supported in HiPAC are create, delete, enable, disable, and fire. The first four of these operations are performed at the request of applications, while the fifth operation is invoked automatically by the system in response to event signals. In addition, since rule operations are performed in transactions, they are subject to concurrency control. Create, delete, enable and disable are considered write operations, while fire is considered a read operation. Therefore, concurrent firing of rules is allowed, whereas the other operations require exclusive access to the rule object. HiPAC does not incorporate any form of conflict resolution when multiple rules are triggered. In fact, all triggered rules are executed concurrently, subject to serializability criteria.

Comments on HiPAC

- The primary advantage of HiPAC is that it treats rules in a uniform manner as other objects. Rules are conceived as first class objects which are created, deleted and modified in the same fashion as other objects.
- Rule operations are implemented as methods thus ensuring that a rule's state is accessed only through its interface.
- HiPAC fully addresses rule execution semantics and provides a flexible approach for rule execution by introducing the notion of coupling modes.
- No conflict resolution is performed when multiple rules are triggered; all multiple triggered rules are executed concurrently.

```
class supplier {
   Name sname;
   Name state;

   public :
      const Name Get-Name();
      void Change-Name(Name new);

   constraint :
      state == Name("NY") || state == Name("") : printf("Invalid supplier location\n");
   triggers :
      T1() : quantity < MINIMUN ==> Reorder();
};
```

Figure 0.1.

- Although HiPAC addresses complex events, some events cannot be expressed and detected, namely, access, conjunction and periodic events.

0.2.2 Ode

Ode ([GJ91b],[GJ91a]) is a database system and environment based on the object paradigm. The database is defined, queried and manipulated using the database programming language O++, which is an upward-compatible extension of the object oriented programming language C++ [GJ91b]. Ode provides active behavior by the incorporation of *constraints* and *triggers*. Each constraint and trigger consists of a condition and an action. Constraints and triggers are defined declaratively within a class definition as shown in Figure 3.1. Constraints are used to maintain the notion of object consistency and hence are applicable to *all* instances of the class (along with its subclasses) in which they are declared. Triggers, on the other hand, are used for purposes other than object consistency and are applicable only to those instances, of the class in which they are declared (along with its subclasses), specified *explicitly* by the user. A trigger is activated on an object by using the call:

$$object_id- > T_i(arguments)$$

This call associates trigger T_i with the object whose identity is $object_id$.

It is important to realize that there is no notion of ECA rules in Ode. An ECA rule is viewed as a unit consisting of an event E_i, a condition C_i and an action A_i, where the signalling of event E_i causes the evaluation of condition C_i and if C_i is satisfied, the execution of A_i. The approach taken in Ode differs in that a condition C_i is paired with an action A_i only, forming a constraint/trigger. Therefore, constraints and triggers are fired as a result of the invocation of *any* non-constant member function. Thus events in Ode are considered as the disjunction of all non-constant member functions.

Events are generated as a result of the of the invocation of non-constant public member functions. Private and protected member functions do not generate events. All events signalled by an object of class A cause the evaluation of all constraints and triggers declared within class A. Event detection occurs via a method based mechanism : constraints and triggers are precompiled into each place in the code where they might be activated, specifically, at the end of each non-constant public member function and before the commit of every transaction.

Constraints

Constraints consist of a predicate (condition) and an optional handler. Constraints implement the notion of integrity constraint maintenance and are of two types: soft and hard.

Soft constraints, in contrast to hard constraints, allow temporal inconsistencies to exist within a transaction. All soft constraints are precompiled into one public method called *soft-constraints*. All hard constraints are precompiled into one public method called *hard-constraints*. The soft-constraints method is invoked at the end (before commit) of a transaction. This is basically the deferred coupling mode, hence, soft constraints will be evaluated only once, regardless of how many times a relevant object has been updated. The hard-constraints method is invoked at the end of each non-constant public method, i.e. it uses the immediate coupling mode. This means each triggering of a hard constraint will cause it to fire.

The evaluation and execution of the condition and handler occur within the triggering transaction. If the condition evaluates to false, the handler is executed. The handler is basically code which attempts to rectify the inconsistent state. If after execution of the handler the inconsistent state still exists, the transaction will abort. This means the condition is evaluated twice, once before and once after the execution of the handler. If no handler is specified, the transaction is immediately aborted.

Triggers

Triggers in Ode are used for monitoring database conditions other than those representing consistency violations. Triggers in Ode are parameterized and can be activated multiple times with different parameters([GJ91b], [GJ91a]). Triggers consist of a name, parameters and a trigger body. Triggers are declared within a class definition and can be of two forms :

trigger-condition ==> trigger-action
or
within expression ? trigger-condition ==> trigger-action [: timeout-action]

The second form is used for specifying timed triggers. In contrast to constraints, a trigger-action is executed if the trigger-condition evaluates to true. There are two types of triggers: *once-only* and *perpetual* triggers. Once-only triggers are deactivated immediately after being fired, while perpetual triggers are automatically reactivated after being fired. The activation of all types of triggers occurs explicitly by the user. Each trigger is precompiled into its own public method (the name of the public method is the name of the trigger). The invocation of a trigger method by an instance causes its activation on that instance. Furthermore, the precompiler generates a public method, *triggers*, which contains all the trigger conditions. The public method triggers is precompiled at the end of each non-constant public method. If a trigger condition evaluates to true, its respective trigger body is appended to a *to-be-executed* list. Trigger bodies are executed in separate transactions after the

commit (not necessarily immediately after) of the transaction firing them. An active trigger's body will be executed only once, regardless of the number of times the relevant object is updated within a transaction. Since trigger execution is in the detached mode, the trigger condition is evaluated twice, once at the end of the public member function and once before execution of the trigger's action.

More recently Ode([GJS92c],[GJS92a],[GJS92b]) has proposed a language for specifying composite events. They have also adopted a declarative approach for specifying events; events are declared within class definitions. Basic (primitive) events are defined and composite event are constructed by applying operators to basic events. The basic events that are supported are *object state events* (creation, deletion, access, update, read), *method execution events* (before or after the execution of a method), *timed events* and *transaction events*. The event operators supported are *relative, prior, sequence, choose, every, fa* and *faAbs*.

Basic events can be qualified with a *mask* thus producing logical events. A mask is an optional predicate that allows users to specify more *specific* events. For instance, assume that the execution of the withdraw method constitutes a basic event. This is specified as :

after withdraw(Item I, int q)

In order to specify that the withdrawal of a *large* sum of money constitutes a basic event, a mask can be used. The mask hides the occurrences of the withdrawal of *small* sums of money. For instance :

after withdraw(Item I, int q) && q > 100

is an event that is raised only if the amount withdrawn is greater than 100.

As previously mentioned, the event operators supported are relative, prior, sequence, choose, every, fa and faAbs. Although the relative and prior operators are sequence operators, their semantics differ when applied to composite events. For example, the event relative(E, F), where E and F are composite events, is raised when the *last* logical event of E occurs prior to the *first* logical event of F. On the other hand, the event prior(E, F) is raised when the *last* logical event of E occurs prior to the *last* logical event of F irrespective of the occurrences of the remaining logical events of E and F. The sequence operator denoted as sequence(E_1,...,E_n), is raised when E_k occurs *immediately after* E_{k-1}. A modifier + has also been introduced which denotes infinite repetition. This modifier can be applied to the relative, prior and sequence operators. For example, relative+(E) implies the infinite disjunction :

relative(E) | relative(E, E) | relative(E, E, E) | ...

Limited repetition is also supported by introducing an integer constant with the operators relative, prior and sequence. For instance, the fifth occurrence of the execution of the method deposit is specified as :

relative 5 (after deposit)

The *choose* operator is used for selecting particular occurrences of events. To raise an event after the fifth transaction commits is specified as :

choose 5 (after tcommit)

```
class stockRoom {
   Item items[max];
   int n;

   public :
      stockRoom();

   trigger :
      T1() : perpetual sequence(after deposit; after withdraw) ==> printLog();
};
```

Figure 0.2.

The *every* operator is used for specifying events that occur periodically. For example, to raise an event periodically after five transactions commit is specified as :

every 5 (after tcommit)

The last two operators *fa* and *faAbs* are used for monitoring the occurrence of an event during an interval marked by the occurrences of other events. The event denoted as fa(E, F, G) is defined as the first occurrence of event F relative to the event E, and with no occurrences of event G (relative to E) taking place prior to F([GJS92c]). The operator *faAbs* denoted as faAbs(E, F,G) is similar to the fa operator however the event G is defined relative to the beginning of the the history of events rather than relative to event E.

To illustrate how complex events are declared in Ode consider the code in Figure 3.2. The perpetual trigger T1 is triggered when the withdraw method is executed immediately after the execution of the deposit method.

Detection of composite events is accomplished by using finite automata. Each event expression maps an event history to another event history that contains only the events at which the event expression is satisfied and the trigger should fire [GJS92a]. Each event expression has an automaton associated with it that reaches the acceptance state when the event is raised. Input to the automaton is the event history, the sequence of logical events, of the object with which the automaton is associated.

Events in Ode are treated as expressions declared within class definitions at compile time. This approach has several disadvantages. First, the treatment of events as expressions results in a dichotomy between events and other objects. Second, events cannot be created, deleted and modified dynamically. In addition, the introduction of new event types, attributes and operations require major modifications thus compromising the system's extensibility. The major disadvantage of this approach is the inability to express complex events that are raised by occurrences of events in different classes. To clarify, Ode has adopted a local view of complex events; a complex event defined in class A can only be raised by events occurring in that same class A. Therefore, complex events cannot be raised by events spanning over several different classes. For example, assume that a bank and a real estate class are defined. In addition, assume a person is interested in purchasing real estate if the interest rates at the bank drop and the price of real estate decreases. This complex event is raised

when a conjunction of events take place specifically, an event in the bank class and an event in the real estate class. Complex events of this nature cannot be expressed in Ode due to their local view of events.

Comments on Ode

- By not grouping an event, condition and action together, excessive checking occurs. This is because each event causes the evaluation of each hard constraint and trigger predicate, regardless of whether the event triggers them or not.
- Constraints and triggers are precompiled into methods. This approach severely compromises the ensurance of inheritance, since a subclass may override the methods defined in its superclasses.
- In our opinion, the demarcation between constraints and triggers is arbitrary and does not contribute in any way to the processing of rules. Furthermore, triggers may be used to specify constraints.
- There is no conflict resolution. Multiple triggered constraints and triggers are fired in random order.
- The need for once-only and perpetual triggers can be eliminated if a mechanism for deactivation is provided. Again the reasons for not supplying deactivation is not clear.
- Although complex events are supported, there is a dichotomy between events and other objects. This is because events are not treated as objects which are created, deleted and modified as other objects in the system.
- Events are declared within class definitions and thus cannot be designated as persistent or transient. Furthermore, adopting a declarative approach for event specification prevents events from being created, modified and deleted dynamically.
- We feel that the introduction of the *mask* is unnecessary since the condition can determine whether to trigger the rule or not. In addition, optimization techniques should handle its efficient evaluation.
- A local view of events is adopted; an event defined in class A can only be raised by events occurring in that same class A. Thus events that are raised by occurrences spanning over several classes cannot be expressed. Therefore, intra-object events can be expressed whereas inter-object events cannot be expressed.

0.2.3 ADAM

ADAM [DPG91] is an active OODB implemented in PROLOG. It focuses on providing a uniform approach to the treatment of rules in an object oriented environment. In addition, it adopts the ECA format for rules, defines and treats rules as first class objects. Moreover, rule operations are implemented as class methods. Events in ADAM are also treated as objects which are created, modified and deleted in the same fashion as other objects.

Rules are incorporated in ADAM by using an object based mechanism. Basically, 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. In order for ADAM to support the inheritance of rules, each class definition is enlarged with an *activated-by* attribute. This attribute is defined as follows :

```
attribute(att-tuple(activated-by,global,set,optional,rule-class,
     [activated-by wof class ::
         class-rules of class
           union
        activated-by of is-a of class]))
```

This attribute forces any update to the *class-rules* attribute of any class to be reflected in the *activated-by* attribute of all its subclasses. This process is performed automatically by the system [DPG91].

The Event Object

Events in ADAM are classified into DB events, clock events and application events. An *event-class* is defined which has three subclasses: *db-event, clock-event and application-event.* Each event is an instance of one of these three subclasses. Events in ADAM are basically generated either *before* or *after* the execution of a method. When an event is raised, all the methods' arguments are passed by the system to the condition and action part of the rule. Thus, the condition and action code may refer to the method's input or output parameters during evaluation. Events are created, modified, and deleted in the same manner as other objects in the system. In order to create an event, the user must specify the *name of the method* generating the event and *when* the event should be raised. For example, an event object is created as follows :

```
new([OID,[
      active_method([put-age]),
      when([before])
]]) => db_event.
```

This event is raised *before* the method *put-age* is executed. Notice that the active-method attribute does not indicate the class in which the method put-age is defined; only the method name is specified.

The Rule Object

ADAM defines a class named the *Rule-class* where each rule is an instance of that class.[1] The structure of the Rule-class consists of the attributes *event, active-class, is-it-enabled, disabled-for, condition and action.* The Rule-class also has methods which define rule operations. A rule is created by sending the following message :

```
new([OID,[
event([3@db-event]),
active-class([student]),
is-it-enabled([true]),
disabled-for([1@student,23@student]),
condition([(
```

[1]Rules could also be instances of a class derived from the Rule-class.

```
            current-arguments([StudentAge]),
            StudentAge > 90
    )]),

    action([(
      current-object(TheStudent)
      current-arguments([StudentAge]),
      get-cname(StudentName) => TheStudent,
      writeln(['The student ', StudentName,
               'with age ', StudentAge,
               'exceeds the expected age']),
      fail
    )])
    ]) => integrity-rule
```

Let us assume that the object identifier of the event instance created previously is 3@db-event. The *event* attribute indicates the event which triggers the rule. In this example, the event attribute has as its value 3@db-event hence, the rule is triggered *before* the execution of the method *put-age*. This method may be potentially defined in several different classes. In order to disambiguate which put-age method triggers the rule, we refer to the *active-class* attribute. The *active-class* attribute has as its value *student*. Therefore, the rule is triggered *before a student object* executes the method *put-age*. The *is-it-enabled* attribute specifies whether the rule is enabled or not while the *disabled-for* attribute has as its value the set of student objects for which the rule is disabled. The *condition* attribute specifies the condition to be checked when the event is raised and the *action* attribute specifies the action to be performed if the condition is satisfied. Both the condition and action refer to the arguments of the method put-age.

ADAM allows a rule's constituents to be modified dynamically. For example, it is possible to specify the condition and action parts of the rule at run-time. Furthermore, the condition and action parts are defined dynamically rather than at compile time. The dynamic characteristics provided by ADAM are influenced by the interpretive environment in which ADAM is implemented and thus it is difficult to accomplish all of this in a language such as C++.

Comments on ADAM

- ADAM provides a uniform treatment of rules in an object oriented context as other objects. Rules are treated as objects which are created, deleted and modified in the same manner as other objects.

- Rule operations are implemented as methods thus ensuring that a rule's state is accessed only through its interface.

- Inheritance of rules from superclasses to subclasses is supported. However, the method by which inheritance is supported is specific to the PROLOG language and hence cannot be easily applied to other object oriented programming languages.

- ADAM only supports the immediate coupling mode and does not support the other coupling modes proposed in HiPAC.

- Although ADAM does not support complex events, the system is extensible enough to support them. This is due to their treatment of events as objects.
- ADAM does not efficiently allow a rule to be applicable to only one instance of a class. This is accomplished by disabling the rule for all other instances.

0.2.4 Object Integrity using Rules

Object Integrity using Rules (henceforth OIR) [MP90] lays special emphasis on the application of production rules in object-oriented databases to enforce integrity constraint maintenance. Although this effort was designed and implemented for the O_2 system, it is general enough to be applied to any object-oriented database system. Constraints are typically classified as either static or dynamic. Static constraints determine the consistency of a database state while dynamic constraints monitor the correctness of state transitions.

OIR supports the maintenance of static as well as some types of dynamic constraints, specifically *two-state predicate constraints*. Two-state predicate constraints are dynamic constraints that can be expressed in terms of the initial and final states of a transaction.

Constraints in OIR are perceived and implemented as objects which can be created, deleted and modified in the same fashion as other objects. A powerful feature provided is the non-restrictiveness of constraint applicability; constraints can be defined on one class (*intra-class*), several classes (*inter-class*) and on object behavior (only if they can be defined as pre- or post conditions to methods). Moreover, constraints have a scope. They can be *global* in nature and hence hold for all applications that run on a database, or *local* and apply to particular applications only. Rules in OIR are treated as objects that are instances of O_2 Rule class (or a subclass of the Rule class). Rule objects are tuples of the form :

$$< \mathbf{Name, E, Q, A, P, S, AP} >$$

where N(ame) is a string that identifies the rule; E(vent) is an expression describing *one* event that triggers the rule; Q(uery) is the condition to be tested when the event E is raised. The condition is an O_2 query. A(action) is the action to be performed if the query Q is satisfied. The action is a sequence of CO_2 operations. P(riority) is a priority level for rule execution. This is used to determine which rule to fire when multiple rules are triggered; S(tatus) indicates whether the rule is enabled or disabled; AP(plicability) indicates when to check the rule, e.g. pre- or post-method execution. OIR supports the rule operations Add, Delete, Enable, Disable, Fire and Change-priority. Each operation is implemented as a method of the system defined Rule class.

Events in O_2 are associated with either message sending or the passing of time. Message-related events are of the form [Receiver, Method]. Events of this type are raised when the Method is sent to the Receiver. The Receiver can either specify a particular instance or a class name. Time related events are expressed as TIME(value).

Constraints are specified declaratively as production rules to the system and later transformed into O_2 rule objects. Static constraints consist of two components: a predicate and an action. Static constraints are defined using the following syntax :

$$< \mathbf{P} -> \mathbf{A} >$$

P is a first order logic predicate and A is a designer-defined action.

On the other hand, dynamic constraints are expressed in terms of temporal logic. Temporal logic augments first order logic with the modal operators *always*, *sometime*, and *next*. OIR supports the modalities *always* and *sometime* only. Dynamic constraints are specified declaratively as follows :

sometime P_i **before** Transaction
sometime P_o **after** Transaction
always P_i **before** Transaction
always P_o **after** Transaction

Each dynamic constraint is transformed into a set of static constraints to be checked before and after a transaction. For example, a dynamic constraint is transformed into :

$$< \ P_i(State_i) \ -> A_i > \text{ and } < \ P_o(State_o) \ -> A_o >$$

where i and o indicate input and output states, P_i and P_o are first order logic predicates and $State_i$ and $State_o$ are input and output states of Transaction.

After the user declares the constraints as production rules they are transformed into an initial set of O_2 rules. This transformation consists of three phases. The first phase transforms production rules of the form $<$ P $->$ A $>$ into pairs of the form $<$ Q, A $>$. Q is a query that represents the predicate P. The second phase consists of determining *all* events that may potentially violate the constraint. These events are determined by performing a phase analysis on the query Q. Basically, the query Q is treated as a path expression that is examined to extract all references to objects or class names. This static analysis is not sufficient since the predicates on classes and objects interact. Therefore, the third phase examines the database schema in order to identify all methods which, sent to the potential source of violation, may indeed cause violation [MP90]. The output of this last phase are pairs of the form [Receiver, Method]. Therefore, predicate rules are transformed into a set of rule objects R_1, R_2,...,R_n,where each R_i has the same $<$ Q,A $>$ and different events.

Comments on OIR

- Provide a uniform treatment of rules in an object oriented environment; rules are objects which are created, deleted and modified in the same fashion as other objects.
- Rule operations are implemented as methods thus ensuring that a rule's state is accessed only through its interface.
- Support the inheritance of rules from superclasses to subclasses.
- Constraints can be defined on one class, several classes and on object behavior.
- A large amount of processing time is required to transform production rules into rule objects. This is because all events that may violate a constraint have to be determined by the system. This is in contrast to requiring the user to explicitly specify the events which triggers the rule. However, this is a compile time and not a run-time overhead.
- The condition and action parts of rules are redundant. This is because potentially many objects may have the same condition and action.

- They only support the immediate coupling mode and do not support the other coupling modes proposed in HiPAC.
- They do not support complex events.

0.3 MOTIVATION AND ISSUES

In strong contrast to the relational model, numerous design issues and difficulties arise when one attempts to incorporate active capability in the object oriented model. In this section we shall examine these various issues in order to

Local vs. Global Rules

Rules in an active relational database have been treated as global constraints which must be satisfied by all relations in the database. This global treatment of rules can no longer be valid 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 and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer [Boo91]. 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. Hence, rules in an active OODBMS need to be viewed as local constraints defined on sets of objects. This local view does not prohibit users from defining global rules[2] as well; global rules may be easily defined for the system by utilizing inheritance. To illustrate the applicability of local rules, consider a stock market application. A consumer may specify a rule that states if IBM stock price falls below a certain threshold, then purchase as much IBM stock as possible. This rule is applicable only to IBM stock objects and not other stock objects such as DEC, Apple, etc. The above notion of local and global rules is applicable to instances of a particular application. To elaborate, local rules are applicable to a subset of the instances defined in an application while global rules are applicable to all instances within an application. It is possible to extend this notion of local and global rules to the application level. Local rules can be applicable only to the application in which they are defined and not applicable to all other applications. Similarly, global rules can be applicable to all applications that are supported on the database.

Rule Specification

Rule specification plays an important role in any active DBMS. Although there is a consensus amongst the database community about the constituents of a rule, no general rule specification format has been agreed upon. In fact, several active OODBMSs have followed different approaches to rule specification. In the relational model, rules have been specified partially by a query and partially by a transaction. The object-oriented model provides numerous alternatives to the designer for specifying rules. To elaborate, rules can be specified in a declarative manner, embedded inside other objects as attributes or data members, or as objects. Rule specification has a profound impact on the active functionality of a system. Therefore, a large portion of research time was devoted to the careful analysis of the advantages and disadvantages of each alternative, aiming at determining the approach that achieves maximum functionality.

[2] This *global* view of rules is confined to the application in which the rules are defined.

Rule Evaluation

Another aspect regarding rules is when and how rules are checked and executed. Rules are triggered from within transactions, consequently, when rules are checked and executed is related to the triggering transaction. HiPAC ([C+89],[DBM88]) proposes three different possibilities for rule checking and execution, viz, *immediate, deferred and detached.* The immediate coupling mode states that the rule should be checked immediately after it is triggered and from within the triggering transaction. The second mode, deferred, specifies that a rule should be checked at the end (before commit) of the triggering transaction. The last mode, detached, states that a rule should be checked from a separate transaction. How rules are checked imposes a significant overhead on the performance of the system. To elaborate, a rule may be checked whenever *any* event occurs or whenever a *particular* event occurs. Rule checking overhead is significantly reduced by checking rules only when particular events take place. Furthermore, multiple rules may be triggered simultaneously. In this situation, two possibilities are applicable, namely execution of *all* or a *subset* of the triggered rules. Another issue regarding multiple triggered rules is rule execution priority; the triggered rules may be executed serially (in some random order or according to a prespecified priority) or concurrently.

Compile time vs. Run-time Rules

Although there is a tendency among most active OODBMS to treat rules as entities known at compile time, we recognize the need for run-time rules as well. Compile time rules connote persistent, immutable and tested rules. Many applications may indeed require additional flexibility not provided by compile time rules. Run-time rules permit users to dynamically create rules and test their correctness and performance before actually designating them as persistent. In addition, run time rules may be utilized as a means for testing hypothetical scenarios without resorting to adding them to application programs and recompiling the system. This is especially true as interpretive object oriented environments are becoming popular.

Events

An important rule component is the event which represents the occurrence of a particular database state or situation. Considerable amount of work has been conducted on event specification in the relational model. Unfortunately, this work cannot be directly adopted to the object oriented context for several reasons. The main reason is that events in a relational context primarily deal with the operations that are performed on tables, specifically, insert, delete, and modify. On the other hand, in an OO context, the constantly changing entity is the object. An object changes state via the message passing protocol; the object receives messages and based upon the semantics of the message, performs some operations. Therefore, in an OO context events are primarily related to objects and the messages they receive. Secondly, when an event is signalled, it is necessary to pass some parameters to the condition and action. This is required since the condition and action may potentially reference the entity that has been updated. To illustrate, a condition may check to see if the changed value of a data item falls below a certain threshold. The parameters collected and passed on to the condition and action differ considerably in the relational versus the object oriented model. In the relational model, the parameters collected are usually the name of the relation and the tuples being accessed, inserted, modified or deleted. For example, after the insertion of a tuple into a table, the parameters collected are the relation name and data items being inserted. In an OO

model, these parameters are not applicable and therefore it is necessary to determine the parameters that need to be collected when events are signalled.

When examining the types of events supported in existing active OODBMSs, it is noticed that the events supported are relatively simple in nature. Although these events serve the requirements of most applications, it is recognized that a large class of applications require the ability to express and monitor events which may be arbitrarily complex in nature. To clarify, consider the communication links between two computers existing on separate sites. These computers are connected by three communication links that require constant monitoring. Upon detecting the failure of all three links, the machines need to be shut-down and the links re-established. This example illustrates the need for supporting the specification and detection of complex events, i.e. the monitoring of *communication* $link_1$ and *communication* $link_2$ and *communication* $link_3$.

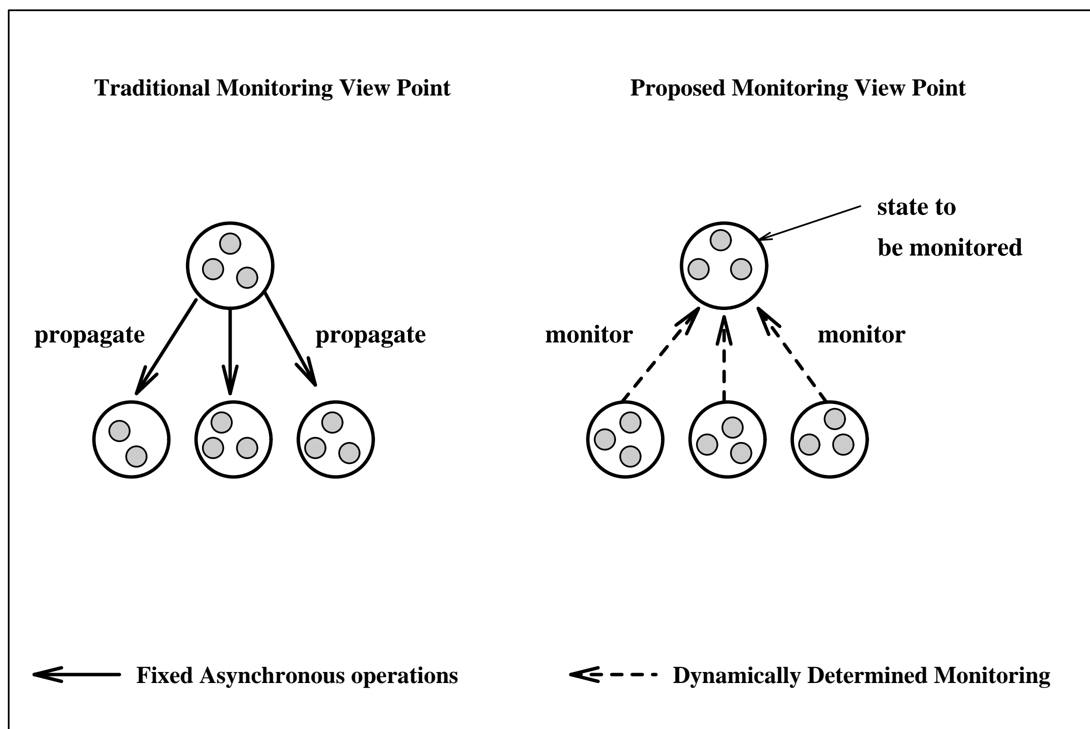


Figure 0.3. Monitoring View Point

<u>Active Functionality</u>

As mentioned previously, the active functionality provided by current active OODBMSs consists of allowing objects to perform some *asynchronous* operations as a result of changes to their *own* state. Therefore, an active object is viewed as an object capable of responding to situations occurring *internally* to its state. Furthermore, these asynchronous operations are determined at compile time and cannot changed at run-time. Numerous existing applications require objects to monitor and *react* to changes occurring to their own state as well as changes occurring to the state of other objects. Thus an active object needs to monitor and react to changes occurring internally to itself as well as externally to other unrelated objects. Moreover, what

an object monitors and reacts to needs to be determined dynamically. To illustrate consider a stock market application. Although the prices of most products fluctuate as a result of supply and demand, some stock object prices are sensitive to external changes. For example, the price of oil may increase dramatically due to the eruption of war in the Gulf. Hence, an oil object needs to monitor and react to world events, i.e. situations occurring externally to its own state. As another example, consider a hospital application. A doctor object may be required to monitor the temperature of a particular patient and react to those temperature changes by requesting some medical examinations. Furthermore, the doctor object can potentially monitor *any* patient object. Therefore, the patient object monitored by the doctor object needs to be determined dynamically. Moreover, the doctor object can also stop monitoring a patient when the patient's health improves. Hence, the doctor object should be able to stop monitoring and reacting to a patient object dynamically.

The active functionality provided by current active OODBMSs is limited and does not accommodate the above monitoring view point. Furthermore, the asynchronous operations performed by an object, as a result of changes to its own state, cannot be changed at run-time. In order to enhance the active functionality currently provided, we propose a *reactive asynchronous interface to objects*. This reactive interface allows an object to send messages out *asynchronously* regarding changes occurring internally to its *own* state. Therefore, this allows objects to be aware of changes occurring to *other* objects and *reacting* to those changes by executing some asynchronous operations. Hence, the thrust of our approach lies in the monitoring view point. To elaborate, in contradistinction to the traditional *internal* monitoring view point adopted by current active OODBMSs, we extend the monitoring view point to an *external* one also. Hence, an active object can monitor its own state as well as the state of other objects and react to those state changes accordingly. This monitoring view point enhances the traditional internal monitoring view point, where an active object can only monitor its own state. Moreover, we have introduced a subscription and notification mechanism. This mechanism allows an object to dynamically determine which objects to monitor and react to. Hence, the subscription and notification mechanisms enable an object to dynamically subscribe to the events of other objects that change their state. To recount, the essential characteristic of a reactive object lies in its ability to *react* to changes occurring internally to its state as well as communicate to any other object regarding the change in its state. Figure 0.3 depicts the traditional and proposed monitoring view points.

The task of incorporating rules into an OODBMS involves other numerous issues not directly addressed here. The set of rules defined in the system may potentially be very large, hence, efficient rule management mechanisms need to be employed. Furthermore, each database operation may possibly trigger events and cause rule evaluation. Rule evaluation undoubtedly imposes an overhead on the system and if not performed efficiently may severely decrease system performance. This necessitates the introduction of rule optimization strategies as well as techniques for reducing rule checking. Although these issues are beyond the scope of this paper, they are mentioned in some discussions to assay the performance of our approach with respect to these issues as well as demonstrate the extensibility of our design.

0.4 DESIGN OF ECA RULES FOR SENTINEL

While examining existing active OODBMSs two main limitations become apparent. Firstly, these systems allow active objects to monitor their own state only. An active object is defined as an object capable of initiating asynchronous operations in response to changes occurring to its own state as well as to the state of other objects.

This implies that an active object should be able to take actions by monitoring its own state as well as the state of other objects. Although existing active OODBMSs agree with this definition of an active object, they do not support it in its entirety. These systems adopt an internal monitoring view point only, thus allowing active objects to monitor their internal state only. Furthermore, the asynchronous operations initiated by an active object, in response to changes to its own state, are pre-defined at compile time and cannot be changed at run-time. The second limitation is concerned with the types of events which can be expressed and detected. These active OODBMSs support the specification and detection of events which are simple in nature only. Although Ode([GJS92c],[GJS92a],[GJS92b]) supports the specification and detection of complex events, the method by which they are incorporated results in a dichotomy between events and other types of objects. Furthermore, they have adopted a local view of events, thus events that are raised by occurrences spanning over several different classes cannot be expressed.

This paper provides a seamless way of integrating rules in an object-oriented framework. We concentrate on improving the active functionality provided by current active OODBMSs with an emphasis on the modularity and the extensibility of the resulting system. This is accomplished by extending the monitoring view point of objects; we view active objects as objects capable of monitoring their own state as well as the state of other objects. By introducing *a reactive asynchronous object interface*, objects are now able to propagate changes occurring on their state to other objects. This propagation enables objects to be *informed of* and *react to* changes occurring to other objects. We also allow objects to dynamically determine which objects to monitor and how to react to the state changes of those objects. This is accomplished by the subscription and notification mechanisms. The novel aspect of our work lies not only in introducing a reactive asynchronous interface to objects and a subscription and notification mechanism, but in addressing complex events. Our proposed approach supports the specification and detection of simple as well as complex events. While supporting complex events special emphasis was given to treating events uniformly as other objects in the system. In addition, we improve upon the types of events that can be expressed by allowing complex events to be raised by events spanning over several different classes. This section is organized as follows. The first subsection presents our design goals. The next subsection discusses the need for a reactive asynchronous interface to objects. The remaining sections provide a detailed explanation of the integration of rules, complex event specification, and detection in an OO context.

0.4.1 Design Goals

An active OODBMS needs to address several issues including how to specify and integrate rules; how to specify and detect events; how and when rules are checked and executed; how multiple triggered rules are executed. These issues have received significant attention in several active OO systems, specifically, ([GJ91b], [DPG91]). The approach taken in ODE is a method-based mechanism; rules in the form of constraints and triggers are precompiled into every method where they might be activated. ADAM uses an object-based mechanism; the object's definition is enlarged to indicate which rules to check when events are signalled. ODE's approach has several drawbacks including :

1. Excessive rule checking is incurred; constraints and triggers are checked at the end of each non-constant public member function.

2. Constraints and triggers cannot be modified dynamically. This is because constraints and triggers are specified inside class definitions and then preprocessed into member functions.

3. Rules on constraints and triggers cannot be specified. For example, it is not possible to specify a rule that states if trigger T_i fires then perform some action A_i.

The approach taken in ADAM has the following drawbacks:

1. They provide an inefficient mechanism for specifying rules applicable to only one instance of a class. This is accomplished by disabling the rule for all other instances.

2. A rule is associated to only one particular class. This makes it impossible for a rule to be triggered by events occurring in different classes.

3. They do not provide a mechanism for the specification and detection of complex events.

4. They support the immediate coupling mode and do not support the other coupling modes proposed in HiPAC.

The approach taken in this paper attempts to avoid the drawbacks of Ode and ADAM and attempts to combine and extend the strengths of each. During our design phase, we found it necessary to fulfill the following goals:

- To provide a seamless incorporation of rules in the C++ programming language.
- To enhance the monitoring view point currently provided by active OODBMSs, specifically, enabling objects to monitor their own state as well as the state of other objects.
- To provide an efficient mechanism for associating rules to all instances of a class as well as to a subset of those instances.
- Support the inheritance of rules from a class to its subclasses.
- To reduce the amount of rule checking necessary upon the signalling of events.
- To enable the specification and detection of primitive as well as complex events.
- Support the immediate, deferred and detached coupling modes proposed in HiPAC.
- Support various contexts defined in [?] for complex event detection and parameter computation.

In order to achieve the above goals several design decisions were made. Firstly, we found it necessary to introduce a reactive asynchronous interface to objects. In addition, rules and events were found necessary to be treated as objects. Lastly, a new mechanism, the subscription and notification mechanism, was incorporated in

order to improve the active functionality provided and reduce the amount of rule checking required.

The object-oriented environment offers numerous design alternatives for the incorporation of rules. In the following subsections we describe the advantages and disadvantages of each alternative, aiming at shedding light upon the rationale behind our design choices. We begin by explaining the need for adopting an external monitoring view point, i.e. the need for introducing a reactive asynchronous object interface.

0.4.2 Need for a Reactive Asynchronous Object Interface

Currently, existing active OODBMSs allow active objects to monitor their internal state only. When examining the needs of numerous applications we realize that objects require the ability to monitor their own state as well as the state of other objects. Therefore, in contrast to adopting an internal monitoring view point, an external one is required. To illustrate the need for this different monitoring view point consider a stock market application. Assume that the prices of the stock items IBM, DEC and Apple are continuously fluctuating. Furthermore, assume that a person object, Mike, is interested in acquiring IBM stock if its price is less than $85 per share. This example illustrates the need for the Mike object to monitor changes occurring to the IBM object and react to those changes by purchasing IBM stock if the price is appropriate. Although this behavior can be achieved by other active systems, the objects which are affected by changes to the IBM object need to known at compile time. Thus the Mike object needs to be known at compile time in order to model this behavior. More importantly, Mike will always be affected by IBM price changes during the duration of the program. Therefore, in order to allow this behavior to be determined dynamically, a mechanism must be devised whereby the Mike and IBM objects can communicate. This communication can be established by introducing a *reactive asynchronous object interface*; this interface equips objects with the capability of *propagating* changes occurring on their state to other objects. Consequently, objects are informed of changes occurring to other objects and can react to those changes by performing some operations. As another example consider a hospital application with a patient object Rebecca and a doctor object Dr_Toskes. Assume that Dr_Toskes needs to perform an ice-bath procedure on Rebecca if her temperature rises above a certain level. Therefore, the Dr_Toskes object needs to be informed of changes occurring to the Rebecca object and then react to those changes accordingly. Hence, the Rebecca object needs to *propagate* changes occurring to its temperature to other objects, thereby allowing other objects to react to changes occurring to its own state.

The above examples illustrate the need for a reactive asynchronous interface to objects. Objects require a mechanism by which they can *communicate* with one another. In order to achieve this, we classify objects as being either *passive*, *reactive*, or *notifiable*. Passive objects are those objects which perform operations synchronously; a passive object receives a message and then performs some operations. We define reactive objects as objects capable of *propagating* changes occurring on their state to other objects. Notifiable objects are those objects being informed or *notified* of the changes occurring to the state of a reactive object. Thus, notifiable objects perform operations as a result of changes occurring externally to their state, i.e. changes to other objects. Figure 4.1 illustrates the behavior of a reactive object.

We chose the C++ object-oriented programming language for modeling an active OODBMS. The choice of this object-oriented programming language does not compromise the generality of our approach; our approach is not language specific

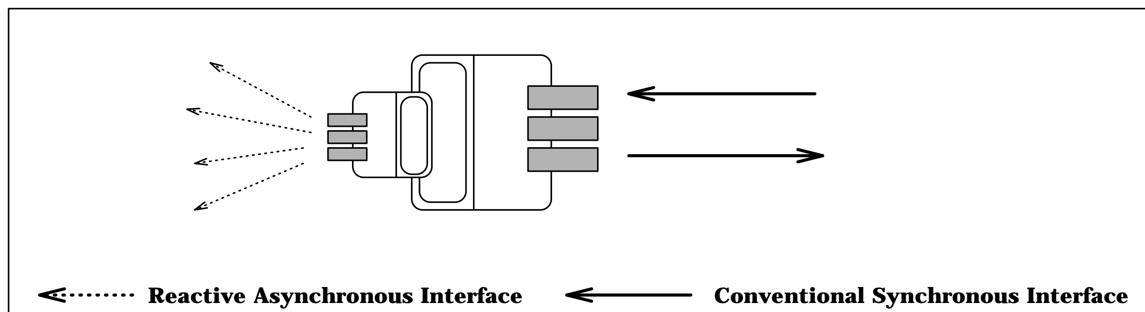


Figure 0.4. Reactive Object's Behavior.

and thus can be implemented using other object-oriented programming languages. Two classes were defined, the *Reactive* and the *Notifiable* classes, for modeling the reactive asynchronous object interface. Any instance of a class derived from the *Reactive* class has the capability of propagating events occurring on its state to other objects. Similarly, any instance of a class derived from the *Notifiable* class has the capability of being notified of events occurring to other objects. Therefore, in the stock market example, the class Stock is derived from the Reactive class, thereby allowing the IBM stock object to propagate changes occurring to its state to other objects. Similarly, in the hospital example, the class Patient is derived from the Reactive class, thereby allowing the Rebecca object to propagate changes occurring to its state to other objects. Furthermore, a Rule class was defined and derived from the Notifiable class, thus rule objects are objects capable of being notified of events occurring to other objects. Hence, the stock market example can be modeled by creating a Purchase rule object which is applicable to the Mike person object. In this example the event is an update of IBM stock price, the condition is the price is less than $5000, and the action is Mike purchasing IBM stock. The Purchase rule object needs to be *notified* of updates occurring to the IBM stock object and then react by sending an asynchronous purchase message to Mike (if the price is less than $5000). Consequently, the IBM stock object needs to *propagate* changes occurring to its price to the Purchase object. In this way the IBM and Mike objects are communicating via the Purchase rule object; the IBM object notifies the Purchase rule object of its price changes and then the Purchase rule object checks if the price is less than $5000. If the price is found appropriate, the Purchase rule object sends out an asynchronous purchase message to Mike. Similarly, in the hospital example a rule object Temperature is created which is applicable to the Dr_Toskes object. This rule has an event of update to Rebecca's temperature, condition of the temperature exceeding a certain level, and an action of sending an ice-bath message to Dr_Toskes. The Temperature rule object needs to be informed of changes occurring to the Rebecca object. In order for this information to reach the Temperature object, the Rebecca object needs to *propagate* changes occurring to its state to the Temperature object. Once this propagation takes place, the Temperature object checks the temperature level; if the temperature if found to exceed a certain level, the Temperature object sends out an asynchronous ice-bath message to Dr_Toskes. Therefore, the Dr_Toskes object is reacting to changes occurring to the Rebecca object via the Temperature rule object. The class hierarchy created is illustrated in Figure 0.5. In later subsections we describe a new technique, *the subscription and notification mechanism*, that enables notifiable objects to *subscribe* to the events propagated by reactive objects.

Once a notifiable object X subscribes to a reactive object Y, object X will be notified of the events propagated by object Y. Therefore, in the stock market example, the Purchase rule object must *subscribe* to the IBM object in order to be informed of changes occurring to the IBM object. Similarly, the Temperature rule object must *subscribe* to the Rebecca object.

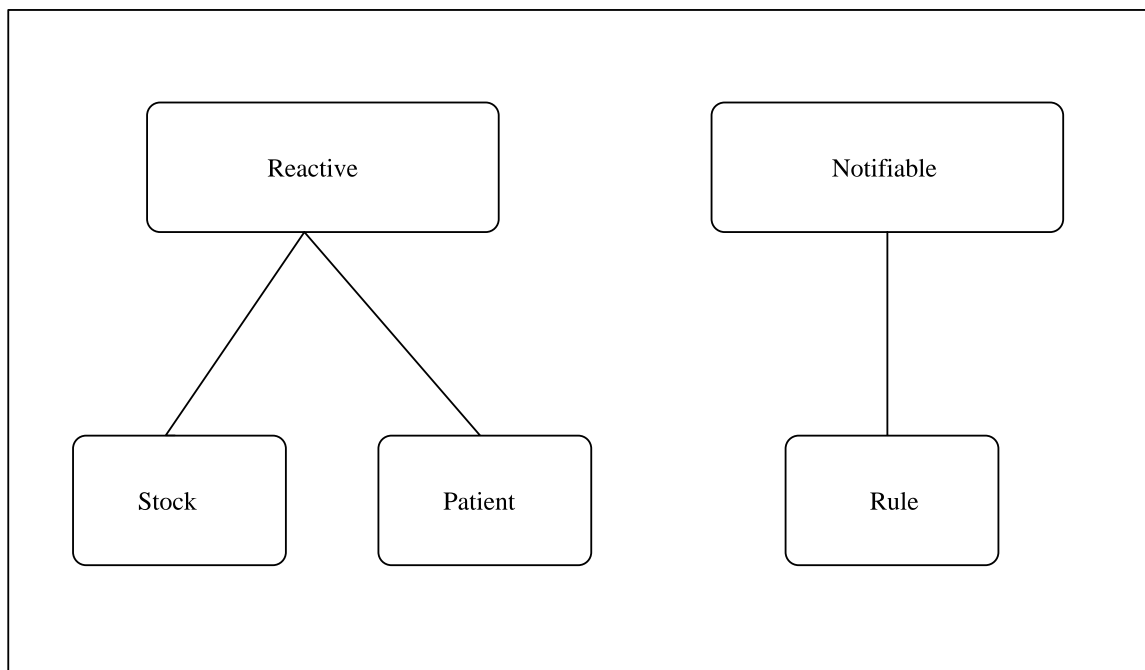


Figure 0.5. Reactive and Notifiable Class Hierarchy.

0.4.3 Alternatives to Incorporating Rules in an OODBMS

The object-oriented 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. Undoubtedly, the mechanism by which rules are specified in an OODBMS has a profound impact on the active functionality provided. We chose to treat rules as first class objects. This subsection discusses the advantages and disadvantages of each alternative aiming at providing insight to our design decision.

Rules as Declarations

The first design alternative for specifying rules is the declarative approach. 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 its simplicity; the user is only required to declare rules and not be concerned with issues regarding event detection, rule checking etc. 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. However, by following this approach rules are not treated in a uniform manner as other objects and their existence is dependent upon the existence of other objects. Furthermore, the system is not extendible since the introduction of new rule components, e.g. rule priority levels, requires modifying all 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 object-oriented environments. In addition, some rule declarations will be redundant. For example, assume a rule is defined which states that an employer's salary must always be less than the manager's salary. In order to model this rule it is necessary to declare it twice, specifically, once within the employee class and once within the manager class.

Rules as Data Members

Each data member has a type associated with it. Therefore, 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[3]. 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. The disadvantage of this approach is twofold. The main disadvantage is it does not support inheritance. This is because the *value* of a data member cannot be inherited. Secondly, a rule's existence is dependent upon the existence of other objects.

Rules as Objects

In our approach rules are treated as first class objects. This design decision is based on the numerous advantages gained by 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 object-oriented context. Secondly, rules are now separate entities that exist independently of other objects in the system. Moreover, the user has the flexibility of determining the longevity of rules, i.e. rules can be designated as transient or as persistent objects. In addition, they are also subject to the same transaction semantics as other objects. Third, each rule will have an object identity, thereby allowing rules to be associated to 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. Lastly, by treating rules as first class objects an extendible 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 the following subsections we define an event in the context of an object-oriented environment, describe the different types of events, and discuss our design decisions pertaining to the specification and incorporation of events.

---

[3]This excludes the possibility of a class. This possibility is examined in the next subsection.

0.4.4 Events

In this subsection we provide a brief description of the events and event operators proposed in Snoop[?] for the object oriented environment. An event is defined as something that happens at a point in time. In an object-oriented context, the events of interest are concerned with changes to an object's state. An object's state changes as the result of an update operation. Update operations occur through the invocation of private, protected and public member functions. Therefore, we view each message sent to an object as a potential event. Considering messages sent to objects as events per se is ambiguous; it is not clear whether the event is raised before or after the execution of the method. To resolve this ambiguity, the *before message* and *after message* clauses are introduced. The *before message* indicates the signalling of the event before the message is executed; similarly, the *after message* indicates the signalling of the event after the execution of the message.

Events are categorized as being either primitive or complex. Primitive events are further classified into begin of message (BOM), end of message (EOM), and temporal events. Complex events are derived by applying event operators to primitive events. The event operators are disjunction, conjunction, sequence, aperiodic and periodic.

Primitive Events

The term primitive event connotes that they are the simplest form of events detected by the system. Primitive events are the building blocks from which composite events are formed and detected. In the following sections we give an overview of the different types of primitive events.

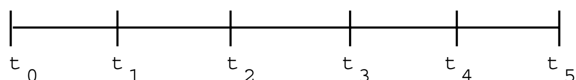
BOM and EOM

BOM and EOM are events that are raised immediately before and after the invocation of a private, protected and public member function respectively. The parameters collected after the detection of these events include the signature of the message, identity of object receiving the message, a time stamp and the parameters of the message itself. The signature of a message is a string which uniquely identifies a method. The time stamp represents the time of occurrence of the event.

The term *begin* refers to the point before the receipt of the message and *end* refers to the point after executing all operations within the method including the return statement. It is worth accentuating that messages sent to objects are considered as primitive events regardless of the type of operations performed by the method.

Temporal events

Temporal events are events associated with time. Time can be visualized as an infinite line divided into segments of equal length.



Temporal events may be specified in two ways, either as an absolute point on the time line or relative to the occurrence of another event E. These two methods are termed as absolute and relative temporal events.

Absolute events

As mentioned previously, absolute events refer to a particular point t_p on the time line. Hence, absolute events comprise of a time string denoting a point in time. The

time string has the form of (hh:mm:ss:)mm/dd/yy. For example, an absolute event may be (12:30:00)04/25/92. The parameter collected is a time string, representing the time of occurrence of the absolute event.

Relative events

Relative events consist of an event E and a time string. The time string denotes a time lapse after the occurrence of the event E. To illustrate, assume a person must dial a telephone number within one minute after hearing the dial tone. The event E is the sound of the dial tone and the time string is [(00:01:00)//]. The parameter collected is the time of occurrence of the relative temporal event.

Composite Events

Composite events provide a powerful mechanism for expressing events. Many applications exist which are not well served by primitive events alone. For example, an application may require that event E be expressed as the conjunction of events E1 and E2. A composite event is derived by applying event operators to primitive events. The operators are disjunction, conjunction, sequence, aperiodic and periodic.

Conjunction

The conjunction of two events, All(E1,E2), is signalled when both E1 and E2 occur, regardless of the order of execution. This operator expresses events which fire when a set of events occurs. The parameters collected are the outerunion of E1's and E2's parameters. To illustrate the usefulness of this operator, consider the communication links between two computers on separate sites. Assuming the computers have three independent communication links, it is necessary to re-establish the links if all three links fail.

Disjunction

The disjunction of two events, E1 V E2, is used to signal an event when either E1 or E2 occur. This operator expresses events which fire due to the occurrence of one or more events. For example, in a process control environment, it is of interest to open all valves if the pressure or temperature increases. The parameters collected are the outerunion of E1's and E2's parameters.

Sequence

The sequence operator expresses events which are fired by a sequential occurrence of a set of events. This composite event, denoted by E1;E2, is signalled when the last event in the sequence (E2) occurs, provided all its successors have occurred. The time of occurrence of a successor event is greater than or equal to the occurrence time of its predecessors. This operator is used for expressing events which occur in a predefined order. For example, in a windows environment, the event *press button* followed by *release button* causes a new window to pop up on the screen.The parameters collected are the outerunion of all the parameters in the sequence.

Aperiodic

Aperiodic events can be expressed using the two operators A and A*.

Operator A

An aperiodic event using the A operator is denoted by A(E1,E2,E1'). This event is signalled whenever the event E2 occurs during the interval defined by the occurrences

of E1 and E1'. This operator is useful for monitoring events within a predefined time interval. For example, during the flight of an aircraft, it might be required to monitor the height of the plane starting from take off till landing. The event take off and landing are the events E1 and E1' respectively. The event E2 is the monitoring of the height of the aircraft. The parameters collected are the parameters of the events E1, E2 and E1'.

Operator A*

An aperiodic event using the A* operator is denoted by A*(E1,E2,E1'). The event E2 may occur repeatedly during the time interval defined by E1 and E1'. Each time E2 occurs, the parameters are collected. However, E2 is signalled only at the time E1' is signalled. Hence, E2 is signalled *only one time* during this time interval, regardless of the number of times it occurs. This operator can be used for integrity checking, where integrity checks occur at the end of a transaction.

Periodic

A periodic event is an event which occurs repeatedly after a finite and constant amount of time. Periodic events, denoted by P(E1,t,E1'), consist of an event E1, a time interval t and a terminating event E1'. To illustrate the usefulness of the operator, consider a computer disk storage device. It may be required to perform a backup of the disk every consecutive week for the year of 1992. This can be specified by P((00:00:00)01/01/1992, (00:00:00)00/14/00, ((23:59:59)12/31/1992). The parameters collected for this event are the outerunion of the parameters of E1 and E1'.

0.4.5 Events as Objects

The fact that events are constituents of rules may lead us to treat events as rule attributes.[4] This approach works well for systems supporting events of primitive nature only. However, as a system progresses towards supporting both primitive and complex events a more elaborate approach is needed. In order to determine how to specify events we must first examine their properties. The first property noticed is that each event has a state. The essential state associated with each event is whether the event has occurred or not, and the parameters collected when an event is raised. Furthermore, events also have a structure. For example, a complex event CE may consist of the conjunction of two events E1 and E2. In addition, events also have some behavior. For example, the complex event CE is signalled when both events E1 and E2 are signalled.

From the above discussion, one observes that events have a state, structure and some behavior, i.e. events exhibit the properties of objects. Therefore, by treating events as objects we are able to fulfill all the requirements of supporting primitive and complex events. More importantly, there is no dichotomy between events and other types of objects; events are objects which are created, deleted and modified as other objects. A system defined *Event* class was created. This definition allows subclasses to be created having different structures and behavior. Each such subclass defines the attributes and necessary operations for modeling exactly one of the complex event types. By treating events as objects, richer and more complex event definitions can be created and easily incorporated. To illustrate the extensibility and flexibility of this approach consider the following scenario. It may be of interest to a particular

[4]This excludes the possibility of a class.

system to determine the number of times each event occurs. This requirement is easily provided by augmenting the class definition with an *occurrence-times* attribute. In the next subsection we describe the event hierarchy created when modeling the different types of events discussed previously.

The Event Hierarchy

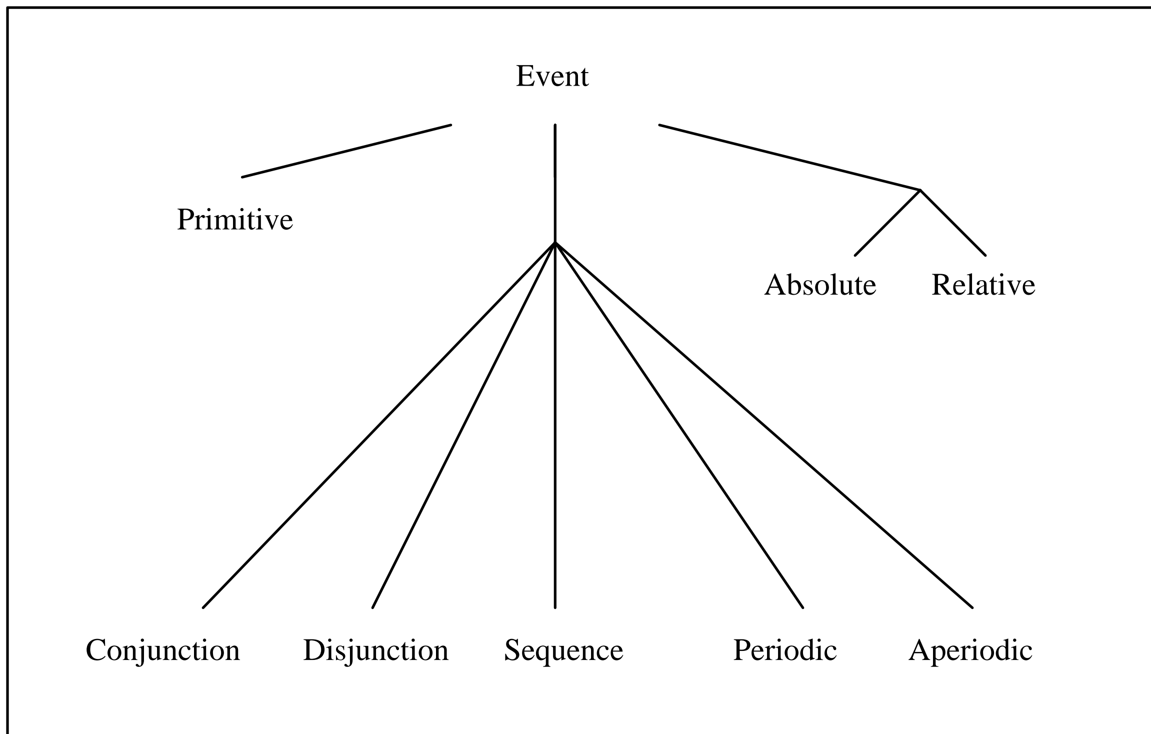


Figure 0.6. Event Hierarchy

The definition of events involves the description of their structure and behavior. The simplest type of event supported is the primitive event which is in the form of messages sent to objects. All other complex events are built from primitive events. By defining the Event class, primitive and complex events' structure and behavior can be defined by using inheritance. The primitive, conjunction, disjunction, sequence, aperiodic, periodic, absolute and relative 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 4.3 illustrates the event hierarchy created.

In order to illustrate the structure and behavior of an event let us consider the Primitive subclass definition shown in Figure 4.4.

The private data member *event-name* uniquely identifies the method which generates the event. The other data member, *occurred*, indicates whether the event has occurred or not. Additional attributes may be included to represent time of occurrence, number of occurrences, etc. Events are created, modified and deleted in the same manner as other objects. For instance, an event object can be created as illustrated in Figure 4.5.

```
class Primitive : Event
{
  int occurred;
  char *event-name;

  public :
     Primitive (char *name) {event-name = name; };
};
```

Figure 0.7.

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

Figure 0.8.

The event-name string 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.

Primitive event specification

In an object-oriented framework, changes to an object's state occurs via the message passing protocol. Whenever an object receives a message, this message may potentially change the object's state. One of our main objectives is to reduce the amount of rule checking required upon the signalling of events. This can be accomplished by requiring the user to specify which member functions are to be treated as *primitive event generators*. Basically, a primitive event generator is a method which raises an event and causes rule evaluation upon its invocation. Therefore, rule checking is reduced since rules are checked only when a message designated as a primitive event generator is invoked. This is in contrast to checking rules after the invocation of every method. The invocation of all other methods do not impose any overhead on rule checking.

In order for the user to specify a method as a primitive event generator, the method must be defined in a class derived from the reactive class. Event generators are specified in the public, private and protected sections of a reactive class as follows:

In the employee class definition shown in Figure 4.6, BOM events will be generated when an employee object receives the private Change-Salary and the public Get-Age messages. EOM events 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 does not cause any rule evaluation.

After specifying the event generators, the user is responsible for creating the appropriate event and rule objects which are informed of the generated primitive

```
class Employee : Reactive
{
  int age;
  float salary;
  char *name;

  event begin Change-Salary(float x);

  public:
    event end Get-Salary();
    event begin && end Get-Age();
    char* Get-Name();
};
```

Figure 0.9.

events. For example, in the previous example the user can potentially create four Primitive event objects. This is illustrated in the Figure 4.7.

```
Event* change = new Primitive ("begin Employee::Change-Salary (float x)" );

Event* getsal = new Primitive ("end Employee::Get-Salary ()" );

Event* end-getage = new Primitive ("end Employee::Get-Age ()" );

Event* begin-getage = new Primitive ("begin Employee::Get-Age ()" );
```

Figure 0.10.

Temporal events are also a form of primitive events. Temporal events are generated by treating the clock as a reactive instance of the class definition given in Figure 4.8.

Inheritance of Primitive events

Inheritance is a mechanism whereby relationships among classes are established. Inheritance forms a hierarchy among classes where subclasses typically augment and redefine the structure and behavior of its superclasses. Each reactive class definition explicitly specifies methods that can generate primitive events. The generation of these primitive events describes a part of the class behavior and consequently is inherited to all subclasses. A subclass may invoke the methods defined in its superclasses; these invocations generate primitive events only if the method is declared as an event generator in a superclass. A subclass may also define new methods. For these newly defined methods to generate events, they must be declared as event generators within the subclass definition. Furthermore, a subclass may redefine the

```
class Time : Reactive
{
 int hh, mm, ss;
 int month, day, year;

 public :
   event end Update-Time();
};
```

Figure 0.11.

methods defined in its superclasses. In order for these redefined methods to generate primitive events, they must be designated as primitive event generators in the subclass definition. If the redefined methods are not specified as event generators their subsequent invocations do not generate primitive events. To illustrate these concepts consider the example given in Figure 4.9.

The above example defines two classes, viz, the Rectangle and Square classes. The Rectangle class is derived from the Reactive class, consequently all instances of the Rectangle class can propagate changes occurring on their state to other objects. The Square class is a subclass of the Rectangle class and hence inherits the methods defined in the Rectangle class. Instances of the Square class can also propagate changes occurring on their state to other objects since the Square class is a subclass (although not a direct subclass) of the Reactive class. The methods Draw and Area defined in the Rectangle class are designated as primitive event generators. Their subsequent invocations will generate the EOM *end Rectangle::Draw()* and BOM *begin Rectangle::Area()* primitive events respectively. The invocation of the method Rotate defined in the Rectangle class does not generate any events. Similarly, the method Magnify defined in the Square class is designated as primitive event generator. Hence, its subsequent invocations will generate the BOM *begin Square::Magnify(int times)* primitive event. The methods Area and Reduce defined in the Square class do not raise events upon their invocation.

As shown above, two instances have been created, namely, a door rectangle object and a box square object. Whenever the door object receives the message Draw an EOM *end Rectangle::Draw()* is generated. Furthermore, whenever the door object receives the message Area a BOM *begin Rectangle::Area()* event is generated. Similarly, whenever the box object receives the message Magnify a BOM *begin Square::Magnify(int times)* primitive event is generated. Since the Square class inherits the method Draw, the box object generates an EOM *end Rectangle::Draw()* primitive event when it receives the message Draw. It is important to notice that the name of the event generated contains the class name Rectangle and not Square. This is because the Draw method is designated as a primitive event generator in the Rectangle class only. In order for the box object to generate the EOM *end Square::Draw()* primitive event upon execution of the Draw method, the Draw method must be redeclared in the Square class and designated as a primitive event generator. Although,

```
class Rectangle : Reactive
{
  public :
     event end Draw();
     event begin Area();
     Rotate (float angle);
};
```

```
class Square : Rectangle
{
  public :
     Area();
     event begin Magnify (int times);
     Reduce (int times);
};
```

```
Rectangle door;
Square box;

door.Draw();                  /* generates EOM primitive event */
box.Draw();                   /* generates EOM primitive event */
box.Area();                   /* does not generate a primitive event */
door.Area();                  /* generates BOM primitive event */
box.Rectangle::Area();        /* generates BOM primitive event */
box.Magnify (2);              /* generates BOM primitive event */
box.Reduce(4);                /* does not generate a primitive event */
```

Figure 0.12.

the Square class inherits the primitive event generator Area, no events are generated when the box object executes the method Area. This is because this method is redeclared in the Square class and *not* designated as a primitive event generator. The only way for the box object to generate an event upon executing the method Area is by explicitly invoking the method Area defined in the Rectangle class, i.e box.Rectangle::Area().

Complex Event Specification

An event expression in Snoop[?] can be denoted as E(E_1,E_2, ... ,E_n) where each E_i is an event constituent of the complex event. Each E_i may be primitive or complex event expressions. In order to visualize a complex event object, it is helpful to consider a tree T=(V,E) consisting of a set of nodes V and a set of edges E. Each node represents one of the event object constituents E_i of the complex event. All the leaf nodes represent primitive event objects.

In order to illustrate the structure of a complex event object assume that a graduate student graduates if he/she completes 39 credit hours of course work or completes 33 credit hours of course work and 6 hours worth of research. The complex event object which detects student graduation has the form depicted in Figure 4.10.

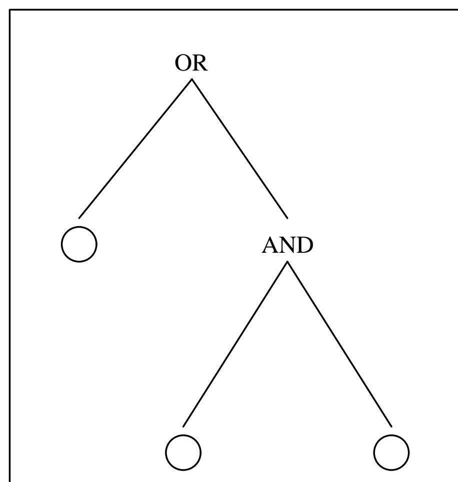


Figure 0.13.

```
Event pe1 = new Primitive ( "end Student::Finished-39 ()" );
Event pe2 = new Primitive ( "end Student::Finished-33 ()" );
Event pe3 = new Primitive ( " end Student::Research-6 ()" );
Event ce1 = new And (pe2, pe3);
Event ce2 = new Or (pe1, ce1);
```

Figure 0.14.

This complex event can be created as illustrated in Figure 4.11. This code first creates three primitive event objects : pe1, pe2 and pe3. These primitive event objects are raised when a student object executes the methods Finished-39, Finished-33 and Research-6 respectively. These three methods must be declared as primitive event generators inside the student class definition. Two complex event objects, ce1 and ce2, are then created. The object ce1 is an instance of the And class and is created by *anding* the events pe2 and pe3. The object ce2 is an instance of the Or class and is created by *oring* the events pe1 and ce1. Therefore, the complex event ce1 is raised when both pe2 and pe3 occur (irrespective of the order of occurrence) whereas ce2 is raised when either event pe1 or event ce1 occur.

Complex Event Detection

The monitoring of a complex event begins after the creation of the event object. When a primitive event is signalled, its parameters are recorded in the tree node representing that primitive event. Thus each tree node acts as an *event recorder*, i.e. it records the parameters collected during the signalling of the event which it represents. The detection of a complex event occurs by detecting each complex event constituent E_i and then traversing the tree in an upward fashion until the root node is reached. Once the root node is reached, the complex event is signalled. After the complex event is signalled, all the tree nodes are cleared of the parameters they recorded and the monitoring is resumed again.

In the above discussion, it is implicitly assumed that each event E_i is signalled only once during the monitoring of the complex event. This may not always be the case since each E_i may be signalled multiple times before the complex event is signalled. Since *each* signalling of an event E_i produces a different set of parameters, we have to define which set of parameters are to be used in the condition and action evaluation. The solution to this problem is in supporting the different contexts, namely the *recent*, *chronicle* and *cumulative* contexts proposed in [CM91]

In recent context, only one set of parameters for each event E_i is recorded at all times. The parameters recorded are the most recently generated parameters, i.e. the parameters of the last occurrence. This method is simple to implement since the system does not have to store all parameters generated. If an event constituent E_i is raised multiple times, the system discards the previous stored parameters and stores the most recently generated parameters. In this paper we have delimited the discussion to recent context.

The chronicle context computes the parameters of a complex event by pairing the parameters generated for each component E_i in chronological order. Therefore, the system must maintain a queue of all the set of parameters collected for each event component E_i. The last context, the cumulative context, is similar to the chronicle context. A history of all parameters generated is maintained; however, the parameters used in condition and action evaluation include all the parameters collected for each event component E_i.

0.4.6 Rule Specification

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. Rules are instances of a system defined Rule class. The Rule class is derived from the system defined Notifiable class, thereby enabling rule objects to be notified of the primitive events generated by reactive objects.

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. 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 to notifiable rule objects.

There are mainly two differences between class level and instance level rules. First, class level rules are applicable to all instances of a class, throughout program execution (when enabled). Instance level rules, however, are applied to a varying subset of instances. Secondly and more importantly, a class level rule can only be applied to one type of object (e.g. to only person objects). Instance level rules are more powerful since they can be potentially applied to different types of objects. Instance level rules can thus monitor situations spanning over different classes. This is accomplished by the rule subscribing to the different types of objects to be monitored.

```
class Person : Reactive
{
 ...

 public :
    event begin Marry (Person* spouse);

 Rules :
    R : Marriage;
    E : Event* marry = new Primitive ( "begin Person::Marry (Person* spouse)" );
    C : if sex == spouse.sex
    A : abort
};
```

Figure 0.15.

The declaration of a class level rule entails specifying a *rule name*, an *event*, a *condition* and an *action*. Class level rules are declared in the rule section of a class as shown if Figure 4.12. 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. Furthermore, it is checked *immediately* from within the triggering transaction whenever the event is raised, i.e. this is the immediate coupling mode.

Instance level rules, on the other hand, are applicable to only those instances explicitly specified by the user. Instance level rules are declared in application code. Let assume that a specific employee, *Fred*, should have a yearly income of less than $30,000. Furthermore, assume that this rule should be checked at the end (before commit) of the triggering transaction, i.e. in the deferred coupling mode.

In order to model the above rule we must first be able to detect the end of a transaction. This can be accomplished by creating a *dummy* class whose sole purpose

is in signalling the end of a transaction. This class can be defined as shown in Figure 4.13.

```
class Transaction : Reactive
{
  public :
     event end End-Transaction();

};
```

Figure 0.16.

After the user defines the class in Figure 4.13, the user must explicitly invoke the method *End-Transaction* before the commit of a transaction. This invocation signals the end of a transaction. For example, the user can create a Transaction instance, *EndTran*, and invoke the message given in Figure 4.14 before the commit of a transaction :

```
Transaction EndTran;

EndTran.End-Transaction();
```

Figure 0.17.

The instance level rule is then created as illustrated in Figure 4.15.

```
Employee Fred;

Event* change-inc = new Primitive ("end Employee::Change-Income(int amount)");

Event* end-of-tran = new Primitive ("end Transaction::End-Transaction()");

Event* level = new And (change-inc, end-of-tran);

Rule IncomeLevel (level, if amount > 30000, Change-Income(30000));
```

Figure 0.18.

This rule has as its event a complex event object that is raised when an employee object executes the method Change-Income *and* a transaction object executes the

method End-Transaction. Both these methods must be declared as primitive event generators in their respective class definitions. The condition part of the rule checks whether the income is greater than $30,000 and the action sets the salary to $30,000. In order for the *IncomeLevel* rule object to be notified of the events generated by the employee object Fred and the Transaction object EndTran, the rule must subscribe to those objects. This is accomplished as shown in Figure 4.16.

```
EndTran.Subscribe(IncomeLevel);

Fred.Subscribe(IncomeLevel);
```

Figure 0.19.

Once the rule *IncomeLevel* subscribes to the objects Fred and EndTran, all primitive events generated by Fred and EndTran are propagated to the rule object. Therefore, the IncomeLevel rule object is monitoring the objects Fred and EndTran simultaneously. The advantage of the subscription and notification mechanism is in enabling rule objects to monitor situations spanning over different classes. In this particular example, the rule object IncomeLevel is monitoring both the Person and Transaction classes.

By defining a rule class, we can model the applications better. For example, single thread execution systems require some form of conflict resolution policy when multiple rules are triggered simultaneously. A conflict resolution mechanism can be incorporated by adding a *priority* data member in the rule class. Therefore, the highest priority triggered rule will be fired. Furthermore, the immediate, deferred and detached coupling modes, proposed in HiPAC, can be incorporated by adding appropriate data members within the rule class definition.

A notifiable rule object subscribes to a set of reactive objects. All the primitive events generated by those reactive objects are propagated to the rule object via the notification mechanism. The notifiable rule object records only those primitive events of interest and discards the rest.

The benefits incurred by defining the above rule class are enumerated below :

1. Rules are created dynamically enabling the modification of their attributes during run-time.

2. Rules are treated uniformly as other objects in an OODBMS.

3. A rules existence is independent of the existence of other objects.

4. Rules may be specified as persistent or transient.

5. Rules have an identity, therefore, any updates to their state is reflected in all objects referring to them.

6. Rules may be designated to be reactive. This provides the capability of specifying rules on rules.

7. As rules are instances of a rule class, rules may be arranged in a hierarchical structure. This provides a natural classification technique for rules.

The Subscription and Notification Mechanism

The subscription and notification mechanism was designed to reduce the amount of rule checking required upon the signalling of events. Furthermore, this mechanism allows rules to monitor different types of objects simultaneously. This technique was implemented by defining the *notifiable* and *reactive* classes. They are discussed in the following subsections.

The Notifiable Class

The primary objective for defining the notifiable class is allowing objects to receive and record primitive events generated by reactive objects. The rule class is a subclass of the notifiable class, thus rule objects receive and record primitive events generated by reactive objects. The notifiable class is defined as illustrated in Figure 4.17.

```
class Notifiable
{
 ...

 public :
    Record (int* obj, char *event-name, int argc ...);
};
```

Figure 0.20.

The method Record performs the necessary operations for documenting propagated primitive events. It takes as its parameters the identity of the reactive object which generated a primitive event, the primitive event generated, and the number and actual values of the parameters sent to the reactive object.

The Reactive Class

In order for a class to provide reactive capabilities it requires a facility for specifying which of its methods generate primitive events, a mechanism for propagating generated primitive events along with their parameters to notifiable objects, and a method for notifiable objects to request the acquisition of information regarding generated primitive events. The requesting mechanism is termed as the subscription mechanism and the propagation of generated primitive events is termed as the notification mechanism.

Due to the fact that potentially many classes may require reactive capabilities, i.e. the subscription and notification mechanism, a class was defined whose sole objective is the provision of these reactive capabilities. This class is named the reactive class and is defined as illustrated in Figure 4.18.

The public interface of the reactive class constitutes the mechanisms by which the subscription and notification processes are provided to subclasses. Each subclass inherits the private data member *head* and the six methods shown above. All instances

```
class Reactive
{
 list-of-subscribers *head;

 public :
    Subscribe (Notifiable *obj);
    Unsubscribe (Notifiable *obj);
    Reactive() {head = Null; };
    Notify (int *obj, char *event-name, int argc ...);
};
```

Figure 0.21.

of classes derived from the reactive class are reactive objects. We will proceed by describing the operations performed by each method.

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 adds the notifiable object to the set of notifiable objects currently associated with a reactive object. The unsubscribe method performs the opposite operation provided 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 list of subscribers are the recipients of all information regarding generated primitive events and are informed of these occurrences via the notify method. The notify method takes as its parameters the identity of a reactive object, a unique identifier string event-name and an integer argc. The declaration of the notify method ends with an ellipsis thus it may additionally take an unspecified number and type of paramters. The *obj* parameter specifies the reactive object which generated a primitive event and the event-name parameter represents the generated primitive event, i.e. the message sent to the reactive object. The number of parameters sent to a reactive object are specified by argc while the actual paramters are specified using the ellipsis feature.

The advantages gained by defining the above reactive class include :

- Providing an efficient mechanism whereby other classes may share the structure and behavior of the reactive class. This is accomplished by inheritance.

- Prevention of reactive objects from accessing and potentially mishandling the set of notifiable objects associated with them. This is accomplished by defining the *list-of-subscribers* data member in the private section of the reactive class.

- The abstraction of the essential characteristics of the reactive class relative to the perspective of reactive objects.

- Providing an extensible reactive model. The modification of the reactive class definition tailors the system according to new requirements.

0.5 Implementation Details

Our Sentinel project was developed using an OODBMS, Zeitgeist, that was developed at Texas Instruments. Zeitgeist is an open, modular, extensible architecture for object oriented database systems. Zeitgeist is implemented using C++ on Sun-4 Unix platforms. This OODB architecture consists of several modules namely, the transactional store, the persistent object store, the object communications, the object translation, the object manager, the type manager, the extended transactions, the change manager, the object query and the user interfaces module. We shall begin by describing each module and then explain how we extended Zeitgeist into an active OODBMS.

The Transactional Store provides atomic commit, object identifier-base retrieval and crash recovery. Its interface hides the details of platform, storage organization and replication.

The Persistent Object Store adds object oriented concepts such as object identity, knowledge of inter-object references (used to cluster and prefetch), and typing and inheritance to the capabilities of the Transactional Store.

The Object Communications module's function lies in reliably moving objects between the Persistent Object Store and the Object Manager, and between different Object Managers. This movement is necessary in order to deliver messages remotely to objects rather than bringing an object to the message and then delivering the message. This module provides remote procedure calls and handles the details of message delivery and retry. It uses the Object Translation module to convert objects between internal and external formats for shipment and delivery.

The Object Translation module's function is to map objects between their internal in memory form and their external representations. This module is compiler and platform dependent since it deals with the run-time representation of objects.

The Object Manager provides the user with the programming language interface to the database. This module allows applications to perform several operations including to save, retrieve, and send messages to objects held in the Persistent Object store. Furthermore, it allows applications to start, commit and abort transactions. Both eager and lazy fetching policies are supported by this module.

The Type Manager serves as the central repository of type information. Applications use this module to share, reuse, evolve type definitions and to generate type declarations. The Object Translation module uses this module to find types, formats and extents during translation. The Object Query uses it to determine if an object is of a type that allows it to be lazily fetched. The Object Query module uses it to learn additional semantics associated with language constructs and to get handles to methods in order to execute them during a query.

The Extended Transactions enhances the transaction facilities provided by the Object Manager. This module provides application specific concurrency control, nested transactions and cooperative work.

The Change Manager is responsible for keeping track of object versions, the parts of an object, and dependencies among objects. The user has the choice of keeping no version history, linear versions, branching versions and the option of saving whole objects or reconstructing objects from forward or backward deltas.

The Object Query module provides a set oriented interface for transient and persistent objects. It supports object identity and queries over complex objects, methods and inheritance structure.

The User Interfaces module provides instance and type inspectors in addition to reflecting the types of objects stored.

In order to incorporate rules in Zeitgeist we modified the class hierarchy to include the newly defined Reactive, Notifiable, Event and Rule classes. The class hierarchy created is illustrated in Figure 6.1. The purpose of the *zg-pos* class is to allow objects to become persistent. Therefore, by deriving the Rule class from the zg-pos class, rule objects can be designated as persistent. Notice that the Rule class is not a direct subclass of the zg-pos class; the Rule class is derived from the Notifiable class which is a direct subclass of the zg-pos class. The Rule class is derived from the Notifiable class in order for rule objects to be capable of receiving and recording the events propagated by reactive objects. Similarly, the Event class (and its subclasses) is derived from the zg-pos class. Hence, event objects can also be designated as persistent. The Reactive class is derived from the superclass Zeitgeist. In the following subsections we describe the Reactive, Notifiable, Event and Rule classes.

0.5.1 The Reactive Class

In order for a class to provide reactive capabilities it requires: a facility for specifying which of its methods generate primitive events, a mechanism for propagating generated primitive events along with their parameters to notifiable objects and a method for notifiable objects to request the acquisition of information regarding generated primitive events. The requesting mechanism is termed as *the subscription mechanism* and the propagation of generated primitive events is termed as *the notification mechanism*.

Due to the fact that potentially many classes may require reactive capabilities, i.e. the subscription and notification mechanism, a class was defined whose sole objective is the provision of these reactive capabilities. This class is named the reactive class and is defined as follows :

The public interface of the reactive class constitutes the mechanisms by which the subscription and notification processes are provided to subclasses. Each subclass inherits the private data member *head* and the six methods shown above. All instances of classes derived from the reactive class are reactive objects. We will proceed by describing the operations performed by each method.

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 adds the notifiable object to the set of notifiable objects currently associated with a reactive object. The unsubscribe method performs the opposite operation provided 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 list of subscribers are the recipients of all information regarding generated primitive events and are informed of these occurrences via the notify method. The notify method takes as its parameters the identity of a reactive object, a unique identifier string event-name and an integer argc. The declaration of the notify method ends with an ellipsis thus it can additionally take an unspecified number and type of parameters. The *obj* parameter specifies the reactive object which generated a primitive event and the event-name parameter represents the generated primitive event, i.e. the message sent to the reactive object. The number of parameters sent to a reactive object are specified by argc while the actual parameters are specified using the ellipsis feature.

The advantages gained by defining the above reactive class include :

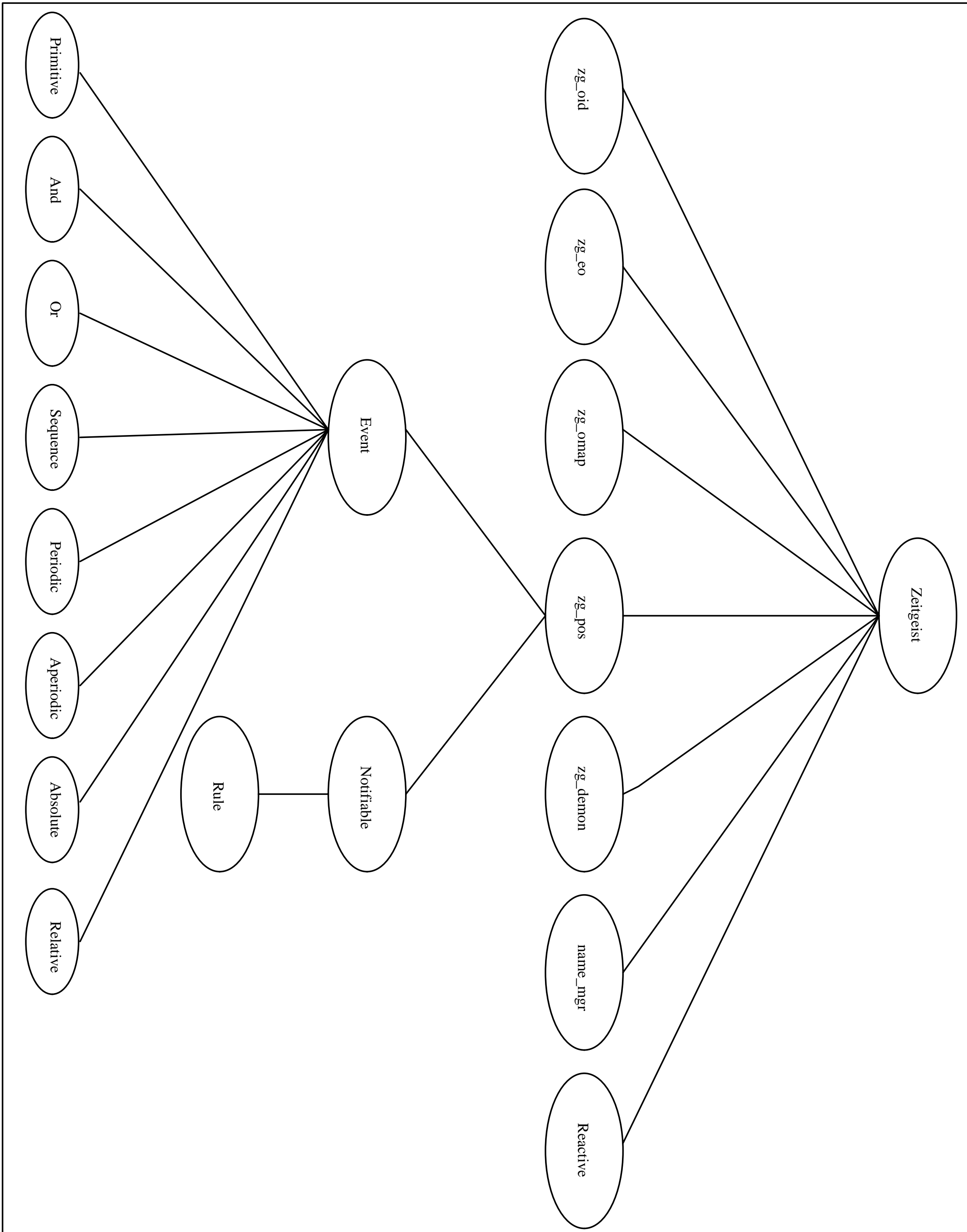


Figure 0.22.

```
class Reactive
{
 list-of-subscribers *head;

 public :
     Subscribe (Notifiable *obj);
     Unsubscribe (Notifiable *obj);
     Reactive() {head = Null; };
     Notify (int *obj, char *event-name, int argc ...);
};
```

Figure 0.23.

- Providing an efficient mechanism whereby other classes may share the structure and behavior of the reactive class. This is accomplished by inheritance.
- Prevention of reactive objects from accessing and potentially mishandling the set of notifiable objects associated with them. This is accomplished by defining the *list-of-subscribers* data member in the private section of the reactive class.
- The abstraction of the essential characteristics of the reactive class relative to the perspective of reactive objects.
- Providing an extensible reactive model.The modification of the reactive class definition tailors the system according to new requirements.

0.5.2 The Notifiable Class

The primary objective for defining the notifiable class is allowing objects to receive and record primitive events generated by reactive objects. The rule class is a subclass of the notifiable class, thus rule objects receive and record primitive events generated by reactive objects. The notifiable class is defined as given in Figure 6.3.

The method Record performs the necessary operations for determining whether the rule should be fired or not. This method takes as its parameters the identity of the reactive object which generated a primitive event, the primitive event generated, and the number and actual values of the parameters sent to the reactive object. This method then passes these parameters on to the event part of the rule. The event objects then check whether the generated event triggers the rule or not. If the generated event triggers the rule, these parameters are then stored in the event object. Otherwise, the generated events are discarded.

0.5.3 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. In order to model the structure of rules, a rule class is defined. Rules are notifiable objects having an event object as an attribute,

```
class Notifiable
{
  ...

  public :
    Record (int* obj, char *event-name, int argc ...);
};
```

Figure 0.24.

and the condition and action as public member functions[5]. In addition, the rule operations create, delete, update, enable and disable are implemented as methods. The definition of the class rule is as illustrated in Figure 6.4.

Each notifiable rule object consists of data members *name*, *event-id*, *condition*, *action* and *enabled*. The rule attribute *name* takes as its value the name of the rule while 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 last attribute *enabled* denotes whether the rule is enabled or not. When a rule is enabled it receives and records propagated primitive events. The condition method is executed when the corresponding event occurs, and if satisfied, the action method is executed.

There are numerous advantages gained by defining a rule class. First, rule operations are implemented as class methods, thus ensuring that a rule's state is accessible only through its interface. In addition, the introduction of new rule attributes or behavior can be easily incorporated by extending the rule structure and/or interface. Thus by defining a rule class, we can model applications better. For example, single thread execution systems require some form of conflict resolution policy when multiple rules are triggered simultaneously. A conflict resolution mechanism can be incorporated by adding a *priority* data member in the rule class. Therefore, the highest priority triggered rule will be fired[6]. Another advantage is inheritance and overriding. Overriding is a mechanism for redefining the methods of superclasses in subclasses. This mechanism is useful for redefining the operations on rules. For example, integrity constraint rules are rules which must be enabled at all times. In order to guarantee that integrity constraint rules are never disabled, a new subclass may be defined where the disable method is redefined to have no effect. To elaborate, the user may define a subclass, *Integrity-Constraints*, from the rule class as shown in Figure 6.5. The *disable* method is redefined to print an error message. Hence, the user should instantiate all rules representing integrity constraints from this class.

The benefits incurred by defining the above rule class are enumerated below :

[5]Each rule defined has its own condition and action implemented as methods defined in the Rule class.

[6]We propose to fire multiple triggered rules concurrently.

```
class Rule : Notifiable
{
  char* name;
  Event* event-id;
  PMF *condition;
  PMF *action;
  int enabled;

  public:
    virtual int Enable();
    virtual int Disable();
    virtual Update(Event* eventid);
    virtual int Condition();
    virtual int Action();
    Rule(Event* eventid);
    ~Rule();
};
```

Figure 0.25.

```
class Integrity-Constraints : Rule
{
  public:
    virtual int Disable()
    {
        printf("\nCannot Disable This Rule\n");
     }
};
```

Figure 0.26.

1. Rules are created dynamically enabling the modification of their attributes during run-time.
2. Rules are treated uniformly as other objects in an OODBMS.
3. A rules existence is independent of the existence of other objects.
4. Rules may be specified as persistent or transient.
5. Rules have an identity, therefore, any updates to its state is reflected in all objects referring to it.
6. Rules may be designated to be reactive. This provides the capability of specifying rules on rules.
7. As rules are instances of a rule class, rules may be arranged in a hierarchical structure. This provides a natural classification technique for rules.

0.5.4 The Event Class

The definition of events involves the description of their structure and behavior. The structure and behavior of events depend on the event type. The Event class was defined and several classes were derived from it. Each subclass defined the necessary attributes and operations for modeling the event type it represents. This class hierarchy was previously illustrated in Figure 4.3. To illustrate the structure of an event type let us consider the And class. Event objects of this class are composed by applying the conjunction operator to two events. The And class is defined as shown in Figure 6.6.

```
class And : Event
{
 Event* left;
 Event* right;

 public :
    And (Event* l, Event* r);
};
```

Figure 0.27.

The above class definition has the data members *left* and *right*, and the constructor *And*. Both the *left* and *right* data members have as their value an event object identifier representing the two events which constitute the complex event object. The constructor creates an And event object by taking the identifiers of two event objects as its parameters. The other subclasses are defined in a similar fashion.

0.5.5 Event Detection Implementation

In order to incorporate rules in Zeitgeist, an event detection mechanism had to be designed. The system is required to detect the execution of member functions. This can be basically accomplished in two ways. The first approach requires transforming member function definitions to include code which signals the event. This is the approach followed by Ode. Therefore, this approach requires the modification of member functions. Although a parser can be easily used to insert the code which signals the event in the appropriate places, there are two main disadvantages to this approach. First, after the methods are modified they need to be recompiled. Hence, this approach does not work well for large numbers of compiled methods and existing library routines. The detection of BOM events is not difficult, since it requires inserting code at the beginning of methods only. However, the detection of EOM events requires inserting code immediately before each return statement and at the end of each method. Therefore the second disadvantage is the significant amount of preprocessing time required.

The second approach does not require modifying user defined methods. In fact, only class definitions are modified and need to be recompiled. This is accomplished by creating wrap-around member functions. To elaborate, let us assume that the execution of the method *Set-Salary* constitutes a primitive EOM event. Rather than inserting 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. To illustrate the wrap-around concept assume that the employee class is defined as shown in Figure 6.7.

The method Set-Salary is declared as an event generator hence, its execution should raise an event. A new member function is generated and declared inside the employee class definition. This member function invokes the method Set-Salary and then signals the event. The new employee class definition is shown in Figure 6.8.

The Notify method informs subscribers of the occurrence of the event. Hence, in order for events to be generated the newly defined method must be invoked. This is a disadvantage since the user must *explicitly* generate events by invoking the newly defined methods. We opted for this approach since it severely decreased our compilation time. Our proposed design can be implemented by using a parser and following the first approach.

0.5.6 Rule Implementation

The declaration of each rule object is stripped of the condition and action. The condition and action code is then inserted

We support class level and instance level rules. Class level rules are declared within class definitions whereas instance level rules are declared in application code. When rules are declared, regardless of their type, the user is under the impression that the condition and action are rule attributes. Actually, this is not the case. The condition and action parts are transformed into member functions of the Rule class. This is necessary since conditions and actions are executed during run-time and the only way to implement them are as methods. As an example consider the instance level rule declaration shown in Figure 6.9.

The above code is stripped of the condition and action part and transformed into the code given in Figure 6.10.

The condition and action part are then transformed into member functions are inserted into the Rule class. The name of the condition method is generated by

```
class Employee : Reactive
{
  int age;
  float salary;

  public :
     event end Set-Salary (float value);
     Set-Age (int value);
};
```

```
Employee::Set-Salary (float value)
{
    salary = value;
    return;
};
```

Figure 0.28.

```
class Employee : Reactive
{
  int age;
  float salary;

  public :
      Set-Salary (float value);
      Set-Age (float value);
      end-Set-Salary (float value)
      {
        Set-Salary(value);
        Notify(...)
      }
};
```

Figure 0.29.

```
Rule* Withdraw (wevent, "if amount > balance", "abort");
```

Figure 0.30.

```
Rule* Withdraw (wevent, &Rule::ConditionWithdraw, &Rule::ActionWithdraw);
```

Figure 0.31.

augmenting the word Condition with the rule name. In this particular example the method name is *ConditionWithdraw*. Similarly, the name of the action method is generated by augmenting the word Action with the rule name. In this particular example the method name is *ActionWithdraw*. The new Rule class definition is as shown in Figure 6.11.

```
class Rule : Notifiable
{
  ...
  int ConditionWithdraw()
{
    if amount > balance
      return(1);
    else
      return(0);
  }

  int ActionWithdraw() {abort};
};
```

Figure 0.32.

0.6 Examples

This section provides a comparison between our proposed approach and the active OODBMSs Ode and ADAM. This comparison will be in the form of illustrating how various rules are specified and implemented in each system. The examples given here accentuate how our proposed approach subsumes and enhances the active functionality provided by Ode and ADAM.

0.6.1 Example One

Let us define a rule which is applicable to all instances of a supplier class. The rule states that if a supplier's location is specified, then the location is required to be New York City. When the rule is violated an error message should be produced stating an invalid location specification.

Ode

This rule translates to a hard constraint in Ode and is specified declaratively within the supplier class definition as shown in Figure 0.33.

```
class supplier {
  Name sname;
  Name state;

  public :
    const Name Get-Name();
    void Change-Name(Name new);
    const Name Get-State();
    void Change-State(Name new);

  constraint :
    state == Name("NY") || state == Name("") : printf("Invalid supplier location\n");
};
```

Figure 0.33. Hard constraints in Ode

This hard constraint is checked after a supplier object is accessed via the invocation of the non-constant public member functions *Change-State* and *Change-Name*. If the constraint evaluates to true, the statement associated with the constraint, the handler, is executed. After execution of the handler, the constraint is checked once again. If the constraint is still not satisfied, as it will not be in this particular example, the triggering transaction is aborted.

The supplier class definition is preprocessed and stripped of the constraint section. This section is then inserted into a newly generated member function called *hard-constraints*. The hard-constraints member function is then called from within each non-constant public member function, specifically, before each return statement. The Change-State and Change-Name member functions are modified to reflect this invocation. The preprocessed supplier class definition and the two member functions are illustrated in Figure 0.34, Figure 0.35 and Figure 0.36 respectively.

Although this rule is easily specified in Ode, unnecessary checking of the rule is incurred as a result of invoking non-constant public member functions other than Change-State; each invocation of the member function Change-Name causes the checking of the supplier rule although the data member *state* is not accessed from within it. Another comment regarding Ode is the excessive preprocessing time required. This is due to inserting the invocation of the hard-constraints member function before *each* return statement found in *each* non-constant public member function.

ADAM

In ADAM, the supplier rule is translated to an event object and a rule object. The member function which raises the event is then identified in addition to determining whether the event is raised *before* or *after* the execution of that member function. In this example the event is raised after the execution of the member function Change-State. Hence, the event object is created by sending the message shown in Figure 0.37.

```
/* Preprocessed class definition */

class supplier {
  Name sname;
  Name state;

  public :
    const Name Get-State();
    void Change-State(Name new);
    const Name Get-Name();
    void Change-Name(Name new);
    void hard-constraints();      /* Newly generated member function */
};
```

Figure 0.34.

```
/* Newly generated hard-constraints member function */

void supplier::hard-constraints()
{
  if (state == Name("NY")  ||  state == Name("") )
    return;
  else
    printf("Invalid supplier location\n");
};
```

Figure 0.35.

```
/* Preprocessed non-constant member functions */

void supplier::Change-State(Name new)
{
  state = new;
  hard-constraints();       /* Inserted code */
  return;
};

void supplier::Change-Name(Name new)
{
  sname = new;
  hard-constraints();       /* Inserted code */
  return;
};
```

Figure 0.36.

```
new ( [OID, [
                active-method ([Change-State]),
                when ([after])
]]) => db-event
```

Figure 0.37.

This event is raised *after* the method *Change-State* is executed. Assuming that the event identifier of the event shown above is 3@db-event, the rule object is then specified as illustrated in Figure 0.38.

```
new ([OID, [
    event ([3@db-event]),
    active-class ([supplier]),
    is-it-enabled ([true]),
    disabled-for ([]),
    condition ([(
            current-arguments ([state]),
            state != Name("NY") or state != Name("")
    )]),
    action ([(
          current-object(Thesupplier),
          current-arguments ([state]),
          writeln ([’Invalid location specified’]),
           fail
    )])
]) => integrity-rule
```

Figure 0.38.

This rule specifies *3@db-event* as its event attribute, which is the object identifier of the event created previously. The class on which this rule is applicable is specified by the *active-class* attribute. In this particular example, the active-class attribute assumes the value of supplier. In order to make the rule applicable to all instances of the supplier class the *disabled-for* attribute is left empty. The condition section of the rule specifies the condition to be evaluated upon occurrence of the event is raised, viz, checking whether the supplier state is not New York City. The action section of the rule defines the action to be performed when the condition is satisfied, namely, print an error message and then abort the triggering transaction.

Sentinel

In our approach we first identify the type of event which triggers the rule. In this particular example the event is a primitive EOM event since the event is raised after the execution of the method Change-State. The next step entails identifying the rule type. This is determined by whether the rule is applicable to all instances of a particular class or to a subset of those instances. This rule is class level rule since it is applicable to all instances of the supplier class. Therefore, the rule is specified declaratively within the supplier class definition. The supplier class definition declares the event generator and the class level rule as shown in Figure 0.39.

```
class supplier : Reactive {
  Name sname;
  Name state;

  public :
    const Name Get-Name();
    void Change-Name(Name new);
    const Name Get-State();
    event end void Change-State(Name new);        /* Specification of an event generator */

  R: Location
  E: new Primitive ("end supplier::Change-State(Name new)");
  C: InValid-State();
  A: abort;
};
```

Figure 0.39.

As in Ode, the class definition is preprocessed and stripped of the code which specifies the event generators and the class level rules. The code defining the class level rules is inserted at the beginning of the main program where the rules and their respective events are created dynamically. Each event defined by the user is an instance of an existing system defined class. Similarly, each rule defined by the user is an instance of the system defined Rule class. Each rule processed by the system appends the Rule class definition with two methods representing the rule's condition and action. The name of the condition method is generated by appending the word Condition with the user provided rule name. Therefore, in this example the name of the method representing the condition is *ConditionLocation*. Similarly, the name of the action method is generated by appending the word Action with the user provided rule name. Therefore, in this example the name of the method representing the action is *ActionLocation*. The next step is to generate the wrap-around member functions which are used for notifying rule objects of generated events. These wrap-around member functions are declared and defined within the supplier class definition. A special wrap-around member function is created for the constructor whose function is to make the class level rules subscribe to each supplier object created. The resulting preprocessed supplier class has the form shown in Figure 0.40.

```
class supplier : Reactive{
   Name sname;
   Name state;

   public :
      void supplier();
      void subscribe-supplier()                  /* Newly generated wrap-around member function */
      {
          Subscribe(&Location);
          supplier();
      }
      const Name Get-Name();
      void Change-Name(Name new);
      const Name Get-State();
      void Change-State(Name new);
      void end-Change-State(Name new)       /* Newly generated wrap-around member function */
      {
           Change-State(new);
           Notify(this,"end Change-State(Name new)",1, &new);
      }
};
```

Figure 0.40.

The modified Rule class definition is as shown in Figure 0.41 while the condition and action member functions are defined as illustrated in Figure 0.42.

```
class Rule : Notifiable
{
   char* name;
   Event *event-id;
   PMF *condition;                          /*  PMF is a pointer to a member function   */
   PMF *action;
   int enabled;
   public :
      virtual int Enable();
      virtual int Disable();
      virtual int ConditionLocation(int* obj);
      virtual int ActionLocation(int* obj);
};
```

Figure 0.41.

```
int Rule::ConditionLocation(int* obj)
{
   if( ((supplier*)obj)->InValid-State());
      return(1);
   else
      return(0);
};

int Rule::ActionLocation(int* obj)
{
   Abort;
    return(1);
};
```

Figure 0.42.

The code shown in Figure 0.43 creates the event and rule objects and is placed at the beginning of the main program.

```
main(argc, argv)
int argc, char** argv;
{
   Rule Location (new Primitive ("end Change-State(Name new)", &Rule::ConditionLocation, &Rule::ActionLocation);
   ...
}
```

Figure 0.43.

0.6.2 Example Two

The second example is related to employees and their respective managers. The rule states that an employee's salary must always be less than the manager's salary.

Ode

This rule translates into two complementary hard constraints in Ode. The first hard constraint is declared within the employee class and is violated if the salary is greater than the manager's salary. The second constraint is declared inside the manager class and is violated if the salary is less than all the employees' salaries. The code in Figure 0.44 shows how these constraints are specified.

```
class manager;
class employee
{
   manager *mgr;
   float sal;
   public :

      float salary();
      constraint :
         sal < mgr->salary();
};

class manager : public employee
{
   employee *emp<MAX>;
   int sal_greater_than_all_employees();
   public :

      constraint :
         sal_greater_than_all_employees();
};
```

Figure 0.44.

ADAM

In ADAM, two events must be detected and they are the execution of the method *Set-Salary* by an employee object and the execution of the method *Set-Salary* by a manager object. Since the method which raises the event in both cases has the same name, only one event object needs to be created. The event object is created as shown in Figure 0.45.

This rule is applicable to both the employee and manager classes. Inheritance of rules is supported in ADAM, i.e. rules attached to a superclass are inherited by all

```
new ( [OID, [
                active-method ([Set-Salary]),
                when ([after])
]]) => db-event
```

Figure 0.45.

subclasses. Although the manager class is a subclass of the employee class, inheritance cannot be utilized in this particular example. This is because the condition to be evaluated when an employee object executes the method Set-Salary is different from the condition to be evaluated when a manager object executes the method Set-Salary. Therefore, it is necessary to create two different rule objects. The first rule object should have the active-class attribute as *employee* while the second rule object should have the value of the active-class attribute as *manager*. Both rule objects are applicable to all instances of their respective active-classes hence, the disabled for attribute is left empty. Furthermore, both rules have the same event attribute value which is assumed to be *2@db-event*. The rule objects are created by the code fragment given in Figure 0.46 and Figure 0.47.

```
                    /* Rule object for employee class */

new ([OID, [
     event ([2@db-event]),
     active-class ([employee]),
     is-it-enabled ([true]),
     disabled-for ([]),
     condition ([(
               current-arguments ([sal]),
               sal > mgr->salary(),
     )]),
     action ([(
            current-object(Theemployee),
            current-arguments ([sal]),
            writeln (['Invalid Salary']),
             fail
     )])
]) => integrity-rule
```

Figure 0.46.

```
/* Rule object for manager class */

new ([OID, [
        event ([2@db-event]),
        active-class ([manager]),
        is-it-enabled ([true]),
        disabled-for ([]),
        condition ([(
                    current-arguments ([sal]),
                    sal < sal_greater_than_all_employees(),
         )]),
        action ([(
                 current-object(Themanager),
                 current-arguments ([sal]),
                 writeln (['Invalid Salary']),
                  fail
         )])
]) => integrity-rule
```

Figure 0.47.

Sentinel

This example illustrates how our approach provides an elegant means for monitoring events spanning over several different classes. The rule is triggered when either an employee or manager object executes the method Set-Salary. This rule can be easily modeled by creating a complex event object which consists of applying the *conjunction operator* to two primitive EOM events.

The next step involves specifying the event generators of the classes employee and manager. In the employee class the method generating an event is *Set-Salary*. Hence, execution of this method by an employee object generates the primitive EOM event *end employee::Set-Salary(float amount)*. Although, this method is inherited by the manager class, the user must redefine the member function Set-Salary inside the manager class. This redefinition is necessary in order for a manager object to generate the primitive EOM event *end manager::Set-Salary(float amount)* upon execution of Set-Salary. If the method is not redefined in the manager class, the Set-Salary method will *always* generate the primitive EOM event *end employee::Set-Salary(float amount)*, regardless of whether the object executing Set-Salary is of type employee or manager. Therefore, the employee and manager classes are defined by the user as shown in Figure 0.48.

In this particular example, the rule should be applied to all employee and manager instances, i.e. the rule is a class level rule. However, this rule does not need to be defined in both the employee and manager classes. It is sufficient to define it in the employee class and then it will be inherited by the manager class.

```
class manager;
class employee : Reactive
{
   manager *mgr;
   float sal;
   public :
      float salary();
      event end Set-Salary(float amount);
      int CheckSal();

   R:ValidSalary
   E: new Or (new Primitive ("end employee::Set-Salary(float amount)"),
                    new Primitive ("end manager::Set-Salary(float amount)"))
   C: if (strcmp(event-name, "end employee::Set-Salary(float amount)") ==0)
           CheckSal();
        else if( strcmp(event-name, "end manager::Set-Salary(float amount)") == 0)
           sal_greater_than_all_employees();
   A: abort;
};


class manager : public employee
{
   employee *emp<MAX>;
   int sal_greater_than_all_employees();
   public :
         sal_greater_than_all_employees();
         event end Set-Salary(float amount);
};
```

Figure 0.48.

The condition part of the *ValidSalary* rule first determines the type of the object generating the event. This is checked dynamically by examining the event name. If the event name is *end employee::Set-Salary(float amount)*, then the object is an instance of the class employee. However, if the event name is not *end employee::Set-Salary(float amount)*, i.e. *end manager::Set-Salary(float amount)*, then the object is an instance of the manager class. If an employee object generates the event, then the employee's salary is compared to the manager's salary. On the other hand, if a manager object generates the event, then the manager's salary is compared to the salaries of all employees. The third parameter of the rule, the action, aborts the triggering transaction when the condition is satisfied.

This example illustrates how our approach provides a more succinct solution to implementing this rule when compared to Ode and ADAM. In Ode it was necessary to define two complementary constraints, although both constraints are used for the same purpose, viz, checking that an employee's salary is always less than the manager's salary. In ADAM although only one event object was created, it was also necessary to create two rule objects. We defined only one rule and capitalized on the facility provided by any object-oriented programming language, namely, inheritance.

0.6.3 Example Three

Let us now consider rules which are applicable to only a subset of the instances of a particular class. Assume it is of interest to always monitor the inventory level of wine bottles in a storehouse. It is of no interest to monitor other inventory items. If the number of wine bottles falls below a certain threshold, then it is necessary to place a reorder request.

Ode

This rule is applicable only to the inventory item wine. Hence, a trigger is used to model this rule. Triggers in Ode have to be explicitly activated on an object by the user. The trigger is declared inside the class *inventitem* as shown in Figure 0.49.

```
class inventitem
{
   inventitem (Name iname, double weight, int qty, double price);
   void deposit(int amount);
   void withdraw(int amount);
   int reorder-level();
   void place-order();

   trigger :

      order() : perpetual after withdraw && qty < reorder-level() ==> place-order();
};
```

Figure 0.49.

The name of the trigger is *order* and it does not take any parameters. The mask checks to see whether the quantity at hand is less than a certain threshold. If this mask evaluates to true the action of reordering of the item is performed. The next step is to activate the trigger on the wine inventory item. This is accomplished by the code in Figure 0.50. The trigger is activated in the application code.

```
inventitem*  Wine(RedWine, 12, 350, 37.50);

Wine->order();          /* Activation of the trigger order on the inventitem Wine */
```

Figure 0.50.

As with hard constraints, triggers are checked before each return statement in each non-constant public member function. Therefore, this trigger will be checked in all the member functions defined in the class inventitem. After the trigger is fired, the user must explicitly reactivate it again.

ADAM

In ADAM the event object must be created first. Since the rule should be checked only when the inventory is decreased, the active method is *withdraw*. The event is created as illustrated in Figure 0.51.

```
new ( [OID, [
                active-method ([withdraw]),
                when ([after])
]]) => db-event
```

Figure 0.51.

The main disadvantage of ADAM lies in the difficulty of attaching a rule to exactly one instance only. This is accomplished by disabling the rule for all other instances of the class. Hence, the *disabled-for* rule attribute will be extremely large. This has a profound affect on rule checking since each object executing the active method is compared to all objects in the disabled-for attribute. A more serious problem is the ability to keep track of all newly created instances and include them in the disabled-for attribute list. Let us assume that a wine, cigarette, coffee and tea inventitem instances have been created with respective identities of 1@inventitem, 2@inventitem, 3@inventitem and 4@inventitem. Furthermore, assume the identity of the event object created previously is 1@db-event. The rule object is then created by the code fragment give in Figure 0.52.

Sentinel

The rule is triggered by the method *withdraw* and hence this method is specified as an event generator in the inventitem class definition. Since this rule is applicable

```
new ([OID, [
        event ([1@db-event]),
        active-class ([inventitem]),
        is-it-enabled ([true]),
        disabled-for ([2@inventitem, 3@inventitem, 4@inventitem]),
        condition ([(
                    current-arguments ([qty]),
                    qty >= reorder-level()
         )]),
        action ([(
                 current-object(Theinventitem),
                 current-arguments ([qty]),
                 place-order()
         )])
]) => integrity-rule
```

Figure 0.52.

only to the *wine* instance, i.e. it is an instance level rule, it is not defined within the inventitem class. The rule is defined in the main program and then it subscribes to the events generated by the wine instance. The code shown in Figure 0.53 creates the event and rule objects in addition to performing the subscription.

```
inventitem Wine (RedWine, 12, 350, 37.50);

Event* level = new Primitive("end inventitem::withdraw(int amount)");

Rule QtyLevel (level, CheckLevel(), place-order());

Wine.Subscribe(QtyLevel);
```

Figure 0.53.

The code defining the rule is translated into :

0.6.4 Example Four

Let us consider a banking system. The possible operations performed on a bank account are deposit and withdraw. A withdrawal can be performed if the amount to

```
class Rule : Notifiable
{
    char* name;
    Event *event-id;
    PMF *condition;
    PMF *action;
    int enabled;
    public :
        virtual int Enable();
        virtual int Disable();
        virtual int ConditionQtyLevel(int* obj);
        virtual int ActionQtyLevel(int* obj);
};
```

Figure 0.54.

```
int Rule::ConditionQtyLevel(int* obj)
{
    if( ((inventitem*)obj)->CheckLevel() )
        return(1);
    else
        return(0);
};


int Rule::ActionQtyLevel(int* obj)
{
    ((inventitem*)obj)->place-order();
};
```

```
inventitem Wine (RedWine, 12, 350, 37.50);
Event* level = new Primitive ("end inventitem::withdraw(int amount)");
RULE    QtyLevel (level, &Rule::ConditionQtyLevel, &Rule::ActionQtyLevel);
Wine.Subscribe(QtyLevel);
```

Figure 0.55.

be withdrawn does not exceed the current bank account balance. Deposit operations may be performed without any restrictions.

Ode

This rule can be expressed in Ode by declaring a basic event that is raised *before* the execution of the method Withdraw. Furthermore, a mask can be defined which checks whether the current balance is greater than the amount to be withdrawn. This mask shields all occurrences of the execution of the Withdraw method when the balance can satisfy the amount to be withdrawn. The trigger is declared in the Account class as shown in Figure 0.56.

```
class Account
{
 float balance;

 public :
    Deposit (float amount);
    Withdraw (float amount);

 trigger:
    T1() : perpetual before Withdraw(float amount) && balance  < amount  ==> tabort;
 };
```

Figure 0.56.

ADAM

This rule is easily modeled in ADAM because the *before* notion is supported. The event object is created by sending the following message :

```
new ( [OID, [
                active-method ([Withdraw]),
                when ([before])
]]) => db-event
```

Figure 0.57.

The rule object is then created with the event attribute having the identity of the object created above. The active-class attribute is *Account* and it is enabled for all instances. The condition checks the if the balance is less than zero while the action prevents the execution of the method *Withdraw* if the condition is satisfied. The rule object is created by the code given in Figure 0.58.

```
new ( [OID, [
                    active-method ([Withdraw]),
                    when ([before])
]]) => db-event
```

Figure 0.58.

Sentinel

The *Withdraw* method is declared as a primitive event generator inside the Account class. In this particular example the key words *event begin* are inserted before the declaration of the method Withdraw. The rule is a class level rule and hence is defined inside the class definition. This is illustrated in Figure 5.27.

```
class Account : Reactive
{
 float balance;

 public :
    Deposit (float amount);
    event begin Withdraw (float amount);

 Rules:
    R: AllowWithdraw
    E: new Primitive ("begin Account::Withdraw(float amount)")
    C: CheckBalance()
    A: abort
   };
```

Figure 0.59.

0.6.5 Example Five

To illustrate complex events consider an operating system that uses a deadlock avoidance policy. After a process requests and then acquires some resource, a deadlock detection algorithm is executed. If a deadlock is detected, the process is terminated. This rule cannot be expressed in ADAM since it does not provide the support necessary for detecting complex events. Hence, we consider Ode and our approach only.

Ode

This event can be specified in Ode using the relative operator. Thus a trigger needs to be defined that is fired when the method Acquire is executed after the

method Request is executed. This complex event is specified in the Resource class definition as illustrated in Figure 5.28.

```
class Resource
{
  int units;

  public :
     Request (int x);
     Acquire (int x);
     Detect-Deadlock ();


  trigger :
     T1() : perpetual relative(after Request, after Acquire) ==> Detect-Deadlock();
};
```

Figure 0.60.

Sentinel

The two methods which generate primitive events are *Request* and *Acquire*. They are specified as event generators in the resource class definition shown in Figure 5.29.

```
class Resource : Reactive
{
  int units;

  public :
     event end Request (int x);
     event end Acquire (int x);
     Detect-Deadlock ();
};
```

Figure 0.61.

The event, in this example, is a complex event constructed by applying the sequence operator to two primitive events. The two primitive events are EOM of Request and EOM of Acquire. The code fragment given in Figure 5.30 specifies the creation of the complex event and the rule object.

The OS event object has the structure given in Figure 5.31.

0.6.6 Example Six

Let us now consider another rule that requires the monitoring of events spanning over several different classes. The rule is triggered when the interests rates at a

```
class Sequence : Event
{
  Event* E1, E2;
  int occurred-el, occurred-e2;

  public :
      Sequence (Event* event1, Event* event2)
      {
        E1 = event1;
        E2 = event2;
      }
};
```

```
Event* pe1 = new Primitive ("end Resource::Request (int x)" );
Event* pe2 = new Primitive ("end Resource::Acquire (int x)" );
Event* OS =  new Primitive Sequence (pe1, pe2);
Rule* OSRULE (OS, "Detect-Deadlock()", "abort");
```

Figure 0.62.

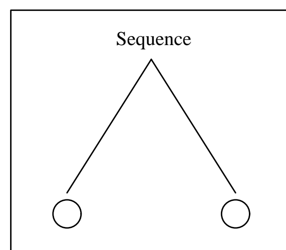


Figure 0.63.

bank drop followed by a decrease in the price of real estate. Therefore, this is a complex event constructed by using the sequence operator. Although Ode supports the sequence operator, this event cannot be expressed in Ode. This is because the event spans over two classes. ADAM cannot express this event since it does not support complex events. Therefore, we consider our approach only.

Sentinel

Firstly, the methods which generate primitive events should be declared as event generators. The methods *InterestDec* and *RealEstateDec*, defined in the bank and real estate classes respectively), 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 sequence class and is created as shown in Figure 5.32. After that a rule object is created which subscribes to the bank and real estate instances that should be monitored.

```
Event* interest = new Primitive ("end bank::InterestDec(float amount)");
Event* estate = new Primitive ("end realestate::RealEstateDec(float amount)");
Event IntEst = new Sequence(interest, estate);
```

Figure 0.64.

0.7 CONCLUSION AND FUTURE RESEARCH

0.7.1 Contributions and Conclusions

In this paper we presented a generic framework for the integration of rules in an OODBMS, thereby achieving active functionality. By capitalizing on the features provided by the OO paradigm, rules are incorporated into an OODBMS without resorting to the introduction of ancillary mechanisms. The major contribution of this paper is in providing a mechanism by which objects may monitor and react to changes occurring to the state of other objects. In addition, the asynchronous operations initiated by an active object, as a result of changes to its own state as well as changes to the state of other objects, can be changed at run-time. Furthermore, this work improves upon existing active OODBMSs by supporting the specification and detection of complex as well as primitive events. Moreover, the approach taken here provides the flexibility for supporting both compile time as well as run-time rules. These features were provided with special emphasis given to providing a modular and extensible system.

To summarize, the contributions of this paper are :

- Adopting an external monitoring view point, thereby allowing an object to react to changes occurring to other objects as well as changes occurring internally to itself.
- Allowing an object to dynamically determine which objects to monitor and react to. This is accomplished by introducing the subscription and notification mechanism. This mechanism also reduces the amount of rule checking necessary.
- The seamless incorporation of ECA rules in an OODBMS. Rules are treated as first class objects which can be created, deleted and modified in the same manner as other objects.
- Supporting the immediate and deferred coupling modes proposed in HiPAC. In the future we aim to support the detached coupling mode.
- Supporting the specification and detection of complex as well as primitive events (a subset of what was proposed in Snoop[**?**]). Furthermore, events are supported in a uniform manner without resorting to the introduction of ancillary mechanisms.
- Allowing rules to be associated to all instances of a class as well as to a subset of those instances.
- Supporting both compile time as well as run-time rules.
- Providing a comprehensive comparison of the active functionality provided by current active databases.
- Finally, implementation of the above in Sentinel.

Comparison of Object Oriented Active Databases

System	Monitoring View Point	Event Scope	Rule Scope	Complex Events	Events as Objects	Rules as Objects	Inheritance of Rules	Coupling Modes	Specify Rules on Rules	Environment
HiPAC	Internal			Yes	Yes	Yes	Yes	I, Df, Det	No	Lisp, smalltalk C
ETM	Internal			Yes	No	No	No	I	No	DAMASCUS
Ode	Internal	1	3,4,5	Yes	No	No	Yes	I, Df, Det	No	C++
ADAM	Internal	1	3	No	Yes	Yes	Yes	I	Yes	PROLOG
OIR	Internal	1,2	3,4,5,6	No	No	Yes	Yes	I	No	O_2
OSAM*	Internal	1,2	3	No	No	Yes	Yes	I, Df	No	C++, ONTOS
Sentinel	**Internal, External**	**1,2**	**3,4,5**	**Yes**	**Yes**	**Yes**	**Yes**	**I,Df**	**Yes**	**C++**

1 Intra-Class 2 Inter-Class 3 Class-Level 4 Instance-Level 5 Global 6 Local

I Immediate Df Deferred Det Detached

0.7.2 Future Research

The task of incorporating rules in an OODBMS involves numerous issues. In this thesis we have attempted to address most issues; however some areas remain unaddressed. These areas offer future directions for research and are:

- Our approach presents a low level implementation for providing active behavior in an OODBMS. Future work can entail transforming any given specification language into ECA rules and then using our approach for providing active functionality.
- Our work supports the specification and detection of complex events. However, we have only supported a subset of the events specified in Snoop[Mis91] and the most recent context for parameter computation. Additional research is required in order to identify the requirements for supporting the chronicle and cumulative contexts for parameter computation as well as support for temporal, periodic and aperiodic events.
- Rule evaluation imposes a large overhead on system performance. Hence, rule optimization strategies as well as techniques for reducing rule checking are imperative for realizing an efficient active OODBMS.
- Currently, rule operations are conceived and implemented as methods. The set of rules defined in the system can be potentially very large hence, efficient rule management techniques need to be employed.
- Currently, the condition and action components of ECA rules are required to be known at compile time. This is because they are implemented as methods. Additional work is required in order to specify the condition and action parts dynamically.
- Currently, applications are limited to sharing data objects only and cannot communicate with each other. Our future effort will address communication among application processes using the active database paradigm.

REFERENCES

[Boo91] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.

[C+89] Sharma Chakravarthy et al. HiPAC: A Research Project in Active, Time Constrained Database Management. Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA, July 1989.

[CM91] S. Chakravathy and D. Mishra. An event specification language (snoop) for active databases and its detection. Technical Report UF-CIS TR-91-23, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, Sep. 1991.

[DBM88] U. Dayal, A. Buchmann, and D. McCarthy. Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database Management System. In *Proceedings 2nd International Workshop on Object-Oriented Database Systems*, Bad Muenster am Stein, Ebernburg, West Germany, Sept. 1988.

[DPG91] O. Diaz, N. Paton, and P. Gray. Rule Management in Object-Oriented Databases: A Unified Approach. In *Proceedings of 17th International Conference on Very Large Data Bases*, Barcelona (Catalonia, Spain), Sept. 1991.

[GJ91a] N. H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proceedings of 17th International Conference on Very Large Data Bases*, pages 327–336, Barcelona (Catalonia, Spain), Sep. 1991.

[GJ91b] N. H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers, Technical Report. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1991.

[GJS92a] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1992.

[GJS92b] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event Specification in an Active Object-Oriented Database. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1992.

[GJS92c] 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.

[Mis91] D. Mishra. Snoop: An event specification language for active databases. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, Aug. 1991.

[MP90] C. Medeiros and P. Pfeffer. Object Integrity Using Rules. Technical Report. Technical report, GIP-ALTAIR, 1990.