**University of Florida**
Computer and Information Sciences

# ECA Rule Integration into an OODBMS: Architecture and Implementation

**S. Chakravarthy**
**V. Krishnaprasad**
**Z. Tamizuddin**
**R. H. Badani**

email: sharma@cis.ufl.edu

**Tech. Report UF-CIS-TR-94-023**

May 1994

(Submitted for publication)


**Department of Computer and Information Sciences**
**Computer Science Engineering Building**
**University of Florida**
**Gainesville, Florida 32611**

# Contents

# ECA Rule Integration into an OODBMS: Architecture and Implementation

**S. Chakravarthy** **V. Krishnaprasad** **Z. Tamizuddin** **R. H. Badani**

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

May 1994

**Abstract**

Making a database system active entails not only the specification of expressive ECA (event-condition-action) rules, algorithms for the detection of composite events, and rule management, but also a viable architecture for rule execution that extends a *passive* DBMS, and its implementation. In this paper, we propose an *integrated* active DBMS architecture for incorporating ECA rules using the Open OODB Toolkit (from Texas Instruments, Dallas). We then describe the implementation of the composite event detector, and rule execution using a nested transaction model for object-oriented active DBMS. Finally, the functionality supported by this architecture and its extensibility are analyzed along with the experiences gained.

## 1 Introduction

During the last decade, database management systems (DBMSs) have evolved considerably to meet the requirements of emerging applications. ECA rules (or event-condition-action or situation-action rules) generalize the forms of monitoring supported earlier (e.g., On conditions in programming languages, triggers and assertions in DBMSs, and signals in operating systems). 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 which can be either primitive or composite. The condition can be a simple or a complex query on the current database state or on the previous and current states, or even on historical data. Actions specify the operations to be performed when an event has occurred and the condition evaluates to true. Although event specification has received considerable attention [3, 19, 9, 14, 7, 8], other aspects of active databases, such as techniques suited for supporting ECA rules, architectures for incorporating rules/events in an existing passive DBMS, and implementation issues have received very little attention.

In this paper, we propose extensions to an object-oriented DBMS (the Open OODB Toolkit from Texas Instruments, Dallas [18]) and indicate the functionality supported by the extended architecture. The implementation of – composite event detection and nested transactions for rule execution – using the design proposed in [1, 6] is the main contribution of this paper. The environment/model into which ECA rules are incorporated has a bearing on the implementation of the above. Event detection is considerably complex in an object-oriented environment as compile time and runtime issues need to be taken into account. Parameter computation is also complicated for the object-oriented model. Support for rule execution was designed and implemented within the limitations imposed by the Open OODB and Exodus.

This paper is organized as follows. Section 2 proposes an integrated active DBMS architecture for supporting ECA rules and their processing. In section 3, we describe the ECA rule format, translation of event and rule specification, composite event detection along with parameter computation, and implementation of the nested transactions for concurrent rule processing. Section 4 presents conclusions. Appendix A contains a detailed example of an application specification for Sentinel and its translation.

# 2 Sentinel Architecture

We use an integrated approach for Sentinel.[1] In other words, we have enhanced the open OODB by incorporating primitive event detection and support for nested transactions as part of its kernel. In addition, we have added support for composite event detection, and rule management as separate modules. Before we describe the Sentinel architecture, we outline the requirements for events and rule processing in an object-oriented framework.

## 2.1 Support for Events

- *Primitive and Composite event detection:* Any method of any object class is a potential primitive event. Further we permit before- and after-variants of method invocation as events. Composite events are formed by applying a set of operators to primitive events and composite events. Both primitive and composite events need to be detected by the system. The detection of composite events entails not only the time at which the composite event occurs, but also keeping track of the constituent event occurrences.

- *Parameter computation:* The parameters of a primitive event corresponds to the parameters of the method declared as a primitive event. The processing of composite events entails not only its detection, but also the computation of the parameters associated with a composite event. The parameters of a composite event need to be collected, recorded and passed on to condition and action portions of a rule by the event detector. Furthermore, these parameters need to be recorded in such a way that they can be interpreted by the condition and action components of a rule.

- *Online and batch detection of composite events:* The composite event detector needs to support detection of events as they happen (online) when it is coupled to an application or over a stored event-log (in batch mode).

- *Inter-application (global) events:* In addition to rules based on events from within an application, it is useful to allow composite events whose constituent events come from different applications. This is especially useful for cooperative transactions and workflow applications. This entails detection of events that span several applications.

## 2.2 Support for Rules

- *Multiple rules:* An event (primitive as well as composite) can trigger several rules. Hence, it is necessary to support a rule execution model that supports concurrent as well as prioritized rule execution.

[1] An active OODBMS being developed at the University of Florida.

- *Nested rules:* When rule actions raise events which trigger other rules there is nested execution of rules. Rules can be nested to arbitrary levels.
- *Coupling modes:* The three coupling modes (immediate, deferred and detached) discussed in HiPAC were introduced to support application needs. Sentinel architecture should be able to support all of them.
- *Rule scheduling:* In the presence of multiple rules and nested execution, the architecture need to support prioritized serial execution of rules, concurrent execution of all rules, or a combination of the two. Further, the system, should allow the application designer to choose from among the above alternatives.

The above requirements as well as the OO model into which active capability is being incorporated affect the design of both the rule processing subsystem and the event detector. Below, we present the Sentinel architecture in terms of extensions to the Open OODB system and discuss how the above requirements are supported in our current implementation. For some of the requirements that are being implemented, we discuss the available alternatives and the rationale for our choice.

## 2.3 Architecture

The Sentinel architecture proposed in this section extends the *passive* Open OODB system [12]. Concurrency control and recovery for top-level transactions are provided by the Exodus storage manager. A full C++ pre-processor is used by the Open OODB for transforming the user class definitions as well as the application code.

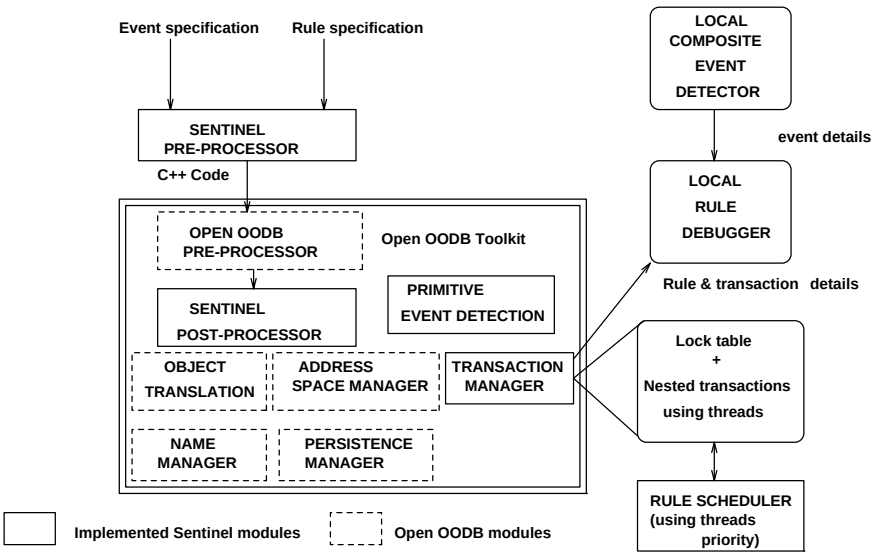


Figure 1: Sentinel Architecture

Figure 1 indicates the modules of the open OODB and the extensions incorporated so far to make it active. These extensions include:

- Implementation of a Sentinel pre-processor (and a Sentinel post-processor) to transform the ECA rules specified either as part of a class definition or as part of an application; these

processors are different from the C++ pre-processor used by the Open OODB. Sentinel pre- and post-processors convert the high-level user specification of ECA rules into appropriate code for event detection, parameter computation, and rule execution,

- Detection of primitive events by notifying the local composite event detector from within each wrapper method if that method is identified as an event. Open OODB provides a wrapper method into which this notification is added by the Sentinel post-processor,
- Implementation of a local composite event detector for detecting composite events (within an application) and parameter computation in various contexts [13, 5]. There is a local composite event detector for each open OODB application or client (each application of Open OODB is a client to the Exodus server),
- Implementation of a transaction manager for supporting nested transactions used for concurrent execution of rules. Light weight processes are used both for prioritized and concurrent rule execution.
- Implementation of a rule debugger for visualizing the interaction among: rules, events and rules, and rules and database objects. This will not be discussed further in this paper. See [16] for details.

Figure 2 shows how the class lattice of the Open OODB has been extended. The classes outside the dotted box have been introduced for providing active capability. This figure also shows the kernel level enhancements to the Open OODB modules to accommodate nested subtransactions.

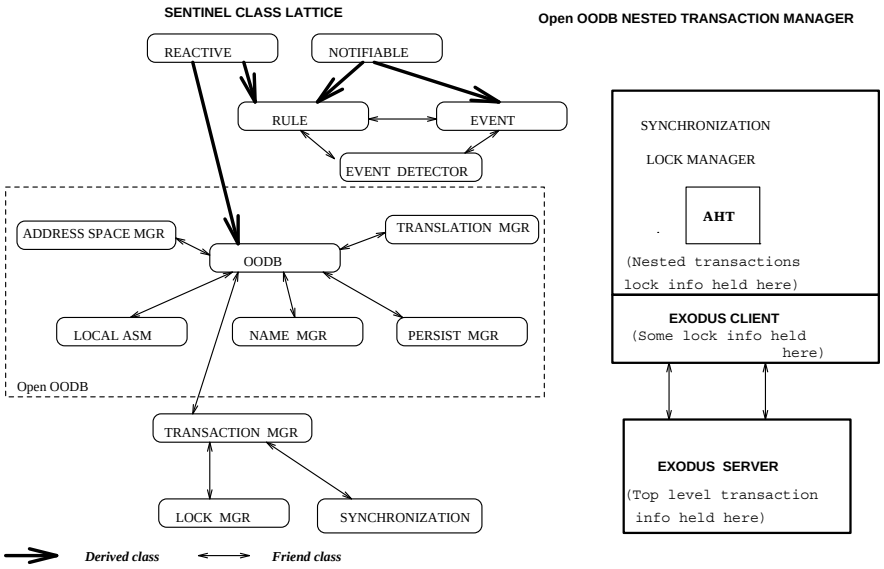


Figure 2: Class lattice and transaction manager of Sentinel

The architecture (new classes and modules) shown in Figures 1 and 2 supports the following features: i) detection of primitive events, ii) detection of local composite events, iii) parameter computation of composite events, iv) clean separation of composite event detection with application execution, v) execution of rules in immediate and deferred coupling modes, and vi) prioritized and concurrent rule execution. The control flow for supporting the above features are further elaborated in Figure 3 and are described below.

Our primitive event detection is based on the design proposed in [1]. Primitive events are signaled by adding a notify procedure call in the wrapper method by the Sentinel post-processor. Also, appropriate calls for the parameter collection are added at this stage. Appendix shows the details of the original program and the transformed program that includes these calls. Both primitive and local composite events are signaled as soon as they are detected. However, the detection of a composite event may span a time interval as it involves the detection and grouping of its constituent events in accordance with the parameter context specified. A clean separation of the detection of primitive events (as an integral part of the database) from that of composite events allows one to: i) implement a composite event detector as a separate module (as has been done) and ii) introduce additional event operators without having to modify the detection of primitive events.

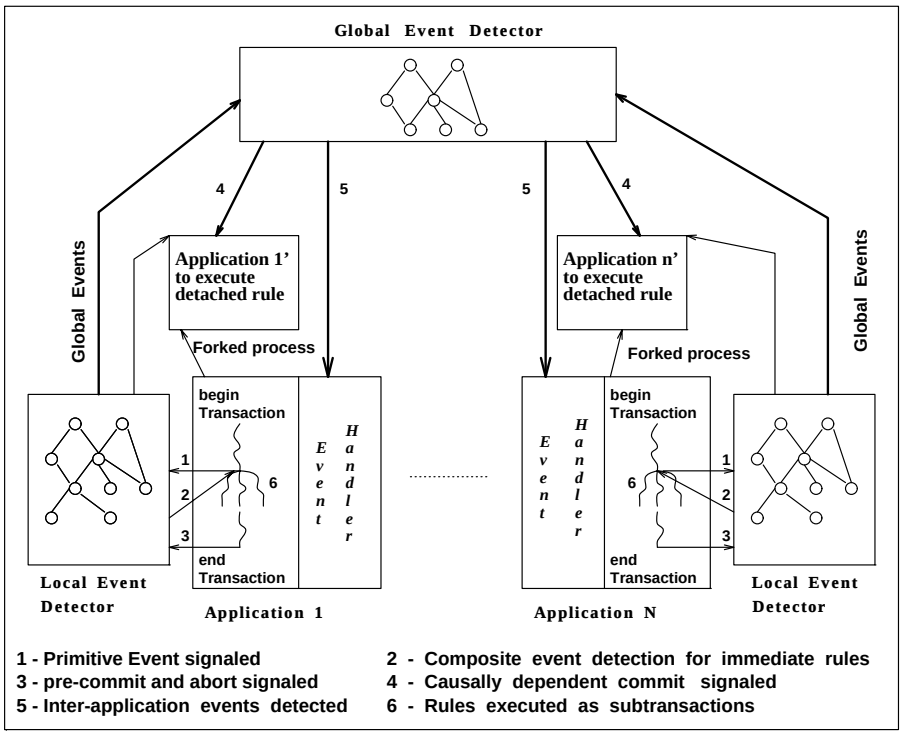


Figure 3: Local and Global Event Detector Architecture

Each application has a local composite event detector (Figure 3) to which all primitive events are signaled. Our implementation uses threads (light weight processes), instead of processes, for separating composite event detection (as well as for the execution of rules) from application because: i) threads communicate via shared memory rather than a file system, thus allowing applications to share the same address space, ii) the overhead involved in creating threads and inter-task communication is low, and iii) it is easy to control the scheduling and communication of threads by assigning priorities. When a primitive event occurs it is sent to the local composite event detector and the application waits for the signaling of a composite event that is detected in the *immediate* mode. The local composite event detector and the application share the same address space and our event detector uses an event graph similar to operator trees [5].

Parameter computation for composite events raises additional problems in the object-oriented framework. The lack of a single data structure (such as a relation) makes it extremely difficult to

identify and manage parameter computation even within an application. As a first cut, we include the identification of the object (i.e., oid) as one of the event parameters and other parameters which have atomic values. However, no assumptions are made about the state of the object (when the oid is passed as part of a composite event) as the detection of a composite event is over a time interval. A linked list that contains the parameters of each primitive event (as a list) that participates in the detection of the composite event is computed and passed to the rule associated with that event. Complete support for parameters of composite events may require versioning of objects and related concurrency control and recovery techniques. Optimization of condition and action components are currently not done as they are C++ functions.

A rule specified to be executed in the deferred mode is rewritten at the source code level into a rule in immediate mode by the Sentinel pre-processor. Our event specification language Snoop [6] supports a number of operators of which $A^*$ monitors the cumulative effect of an event occurrence within a specified interval. For example, if we need to accumulate all insert events in a transaction, we can specify the event as $A^*$(begin_transaction, insert, end_transaction). Using this operator, we can readily model the deferred coupling mode in terms of the immediate coupling mode by using *begin* and *pre-commit* transaction events and postpone the execution of the rule to the end of the transaction. In Sentinel, the begin transaction event is always signalled at the beginning of a transaction and the pre-commit is signalled before the commit of a transaction. Using the $A^*$ operator, a rule in deferred mode with an (arbitrary) event E is transformed by the Sentinel pre-processor to $A^*$(begin-transaction, E, pre-commit-transaction). This causes a deferred rule to be executed exactly once even though its event may be triggered a number of times in the course of that transaction execution. This formulation handles the net effect variant of deferred rule execution.

For rule execution, a nested transaction manager is implemented with its own lock manager. This is in addition to the concurrency control and recovery provided by the Exodus for top-level transactions. Each rule (i.e., condition and action portions of a rule) is packaged into a subtransaction. A number of subtransactions are spawned as a part of the application process. This is further elaborated in the implementation section. The order of rule execution is controlled by assigning appropriate priorities to each thread based on the priority of the rule and the priority of the triggering rule (if there is one). Support for multiple rule execution and nested rule execution entails that the event detector be able to receive events detected within a rule's execution in the same manner it receives events detected in a top level transaction. This is accomplished relatively easily by separating the local composite event detection from the application as shown in Figure 3. This separation also readily supports both online and batch (or after-the-fact) detection of composite events.

Finally, in the presence of composite events, it is possible for the events to cross transaction boundaries (within the same application). Currently, we provide a mechanism to flush all events generated by a transaction when it commits. More work is required to understand the semantics of rule execution whose events span transaction boundaries.

### 2.4 Planned Extensions

We distinguish between local (composite) events all of whose constituent events are generated by the same application and global (composite) events whose constituent events can come from different applications. Although the event specification language remains the same for both local and global events, they differ significantly in their implementation. Global events can be used to model rules

that cross not only application boundaries, but also transaction boundaries.

Supporting rules whose events are global requires not only the detection of global events spanning several applications, but also dealing with address space issues. In the relational model, it is easier to handle this as the data dictionary has the type information of all objects and furthermore attributes values are atomic. In the object-oriented model, interoperability across applications is extremely complicated on account of the component objects, pointers, and virtual functions. These issues are currently being addressed by OMG and Corba [17].

We envision each application having a thread (a global event-handler thread shown in Figure 3) that handles the execution of rules with global events (whose events span applications or transactions). The global event detector communicates with the local event detectors for receiving events detected locally and with the application's global event handler for signaling the detection of global events for executing tasks based on global events. Again there is a clean separation between the events detected by the local event detector and the global event detector.

The implementation of detached coupling mode stipulates that a new top-level transaction be started for executing a rule. There are two alternatives for starting a new top-level transaction: i) by starting a separate process and ii) by forking process from the triggering transaction. The first alternative has severe ramifications in the object-oriented model where the rule's condition and action could be arbitrary functions requiring the declaration of all classes. Unlike the relational model, creating an independent transaction for a rule in an object-oriented case, may be limited by the host environment (e.g., objective C, C++, Common Lisp, SmallTalk). However, for this alternative, the commit dependencies (causally detached mode) can be modeled by events spanning applications. Each transaction can signal a pre-commit and (possibly) an abort event which can be used by the global event detector to enforce abort dependencies between two top-level transactions. However this is subject to the address space issues discussed above. For the second alternative the detached rule could be forked as a process that evaluates the condition and executes the action function of the rule. In this approach, the commit dependencies can be easily established between the parent and child processes. Signals inform processes of asynchronous events. The forked process would have the copy of the resources of the parent process and hence the address space issues do not arise in this case.

Finally, we plan on using a declarative query language, such as ZQL[C++] (supported by Open OODB) in the future. We intend to combine rule evaluation with event detection when the coupling mode permits and rules are non-procedural and optimize the entire tree as a whole.

# 3 Implementation

The Sentinel pre- and post-processors, the local composite event detector, a transaction manager for supporting nested transactions, and the rule manager have been implemented. First we discuss the rule format and how we translate a high level rule specification to Sentinel system calls followed by the details of our implementation. A detailed example can be found in the appendix, parts of which are used in this section for explanation.

## 3.1 Sentinel Rule Format and Explanation

The syntax of a Sentinel event/rule specification is:

event [begin (eventid)] [&& end (eventid)] method_name

| | | |
|---|---|---|
| event eventName | = | event_expression |
| rule ruleName([eventName =] | | event_expression \| eventName, |
| | | condition_function, action_function |
| | | [[, parameter_context][, coupling_mode] |
| | | [, priority][, rule_trigger_mode]]) |

In Sentinel it is necessary to identify, as part of the class definition, the methods that generate primitive events. Both begin-method (by indicating **begin(***eventName***)**) and end-method events (by indicating **end(***eventName***)** are supported. This event interface specification is pre-processed and calls to notify the event detector are added to the wrapper methods. The eventName specified is optional and either only the begin or end of a method can be designated as an event. By default end of a method is taken to be the event. For primitive events specified as part of the interface, the user is allowed to use them directly in the application program by prefixing the classname (as *className_eventName*) for defining additional event expressions. Below, examples of events and rules specified at the class level are shown:

```
class STOCK : public REACTIVE  {
      public:
      event end(e1) int sell_stock(int qty); /* end primitive event */
      event begin(e2) && end(e3) void set_price(float price);
      int get_price();
      event e4 = e1 ^ e2; /* AND operator */
      rule R1[e4, cond1, action1, CUMULATIVE, DEFERRED, 10, NOW]; }; /* class level rule */
```

*Event expressions* specify primitive and composite events using the event specification language described in [6] which supports a number of event operators (e.g., and, or, sequence, aperiodic). The BNF of the event specification language can be found in [14]. We allow an optional *eventName* to be specified within the event/rule definition to allow the users to name an event expression for subsequent usage. When an event expression is processed, calls for creating the event graph for that event expression are added to application code. Currently, the condition and action component of a rule are global functions.[2] The condition function returns a boolean value to indicate whether the condition evaluates to true or not. The action function does not return any value.

Parameter contexts indicate the order in which successive occurrences of the same events are grouped. In the absence of any composite events, the notion of parameter contexts is not necessary. The optional parameter *parameter_context* provides the context for detecting an event expression as well as for computing its parameters. Although the parameter context is meaningful only to an event (either primitive or complex, for its detection), specifying it along with the event limits the utility of an event definition. If several rules need to be defined on the same event in different parameter contexts, then the event has to be duplicated for each context. By allowing a late binding of the parameter context (i.e., at the rule specification time instead of at event specification time), reusability of events is readily supported. Furthermore, common event sub-expressions are represented only once in the event graph (a graph representing an event expression; this is analogous to an operator graph) reducing the total number of nodes. However, this has a bearing on the event detection algorithm and the data structures employed as the same event may

[2]Currently, only functions are used for specifying condition/action. In the current host environment (i.e., C++), methods cannot be used for condition/action as their invocation is tied to an object which is not known at compile time. But these condition and action functions can access stored objects as well as objects in the main memory.

have to be detected in multiple contexts. Of the four parameter contexts allowed, namely, *Recent*, *Chronicle*, *Cumulative*, and *Continuous*, the Recent context is assumed to be the default due to its low storage requirements. The specified context is propagated to all the nodes of the event graph associated with the rule to facilitate event detection and parameter collection.

*Coupling_mode* refers to the execution points. Currently, immediate and deferred coupling modes are supported between event and condition-action pair. We use *priority* classes for specifying rule priority. An arbitrary number of priority classes can be defined and totally ordered. A rule is assigned to a priority class either by indicating its number or the name of the class. As our implementation supports concurrent and nested rule execution, priority of rules need to be resolved at different levels of execution. Our current approach provides a global conflict resolution mechanism among the priority classes and concurrent execution of rules that belong to the same priority class. This approach combines the advantages of both integer priority schemes and precedes/follows schemes. This approach allows us to change rule priority categories based on the context or inherit priorities from users/applications.

We allow rule specification at class definition time and as part of an application. We also support rule activation and deactivation at runtime. Moreover, named events can be reused later. This implies that a number of rules may be defined on the same event expression and the event expression might have been defined prior to the rule definition time. As a result, it is possible that a rule gets triggered by event occurrences that temporally precede the rule definition time itself. As this might not be desirable in all situations, we provide an option (*rule_trigger_mode*) for specifying the time from which event occurrences to be considered for the rule. Two options, NOW (start detecting all component events starting from this time instant) and PREVIOUS (all component events are acceptable) are supported as rule triggering modes, with NOW being the default. For rules specified within a class, the semantics of NOW and PREVIOUS are the same.

The user-level rule specification given above is pre-processed into C++ statements that create rule and event objects. This specification helps to hide the details of rule/event implementation from the user. Furthermore, the syntax of a rules is the same for both **class level** and **instance level** rules. A class level rule satisfies the inheritance property. As part of the application, rules having primitive events can be specified as applicable to an entire class or an instance of that class as shown below.

```
REACTIVE Stock;
Stock IBM;
event any_stk_price('any_stk_price', 'Stock', 'begin', 'set_price(float price)');
event set_IBM_price('set_IBM_price', IBM, 'begin', 'set_price(float price)');
```

Here the character string 'Stock' in the event definition denotes a class and IBM denotes the instance of that class. The primitive event any_stk_price defines a class level primitive event. This event will be detected for all instances of this class whenever the method 'set_price' is invoked. The event set_IBM_price will be invoked only when the same method set_price is invoked on the IBM object. A rule defined on any_stk_price will be a class level rule and a rule on set_IBM_price will be an instance level rule. The specification of class/instance at the primitive event level allows us to have event expressions with class level as well as instance level events and hence rule specification which has mixed instance specification. Note that the event name is different although both the events are specified on the same method set_price. A rule which contains all constituent primitive events as class level primitive events is termed a class level rule. Likewise a rule declared on only

instance level primitive events is an instance level rule. Any class whose events are used in rules (either class level or instance level) need to be reactive (i.e., subclass of the REACTIVE class). Any class whose events are used in rules (either class level or instance level) need to be reactive (i.e., subclass of the REACTIVE class). When a user-defined reactive class is pre-processed, appropriate primitive events and rule declarations are generated and inserted in the application program. Since this rule will subscribe to an event expression that is specified on a class level, this rule will be notified whenever any object of this class invokes the method that are potential event generators.

## 3.2 Implementation of Event Detection

All objects that can signal events must be derived from the global REACTIVE class. The extensibility of the system is achieved by making the system class of Open OODB (namely OODB) REACTIVE. We specify an event interface to make the methods beginTransaction and commitTransaction of this class generate events which result in actions used for supporting Sentinel features, such as deferred rule execution, flushing of all the event occurrences from the event graph. Although rules are subclasses of the Notifiable class, methods of the Rule class can themselves be event generators. As shown in Figure 2, a rule class can be both reactive and notifiable.

### 3.2.1 Primitive Event Detection

For methods that can generate primitive events, the wrapper class methods generated by the Open OODB are modified. Open OODB renames the original method as user_method and creates a wrapper method which has the original method name. The calls inserted by the Sentinel post-processor into the wrapper method does the required signaling to notify the local composite event detector before and/or after the invocation of the user_method (according to the event interface specification). Each wrapper method which can generate an event is further extended extended by adding code for parameter collection and notification to the event detector. An example of a wrapper method after Sentinel post-processing for the class STOCK and event set_price is shown below.

```
void STOCK::set_price(float price)
{
    /* Parameters are collected in a linked list */
    PARA_LIST *set_price_list = new PARA_LIST();
    set_price_list->insert("price", FLOAT, price);

    /* Notify begin of method */
    Notify(this, "STOCK", "void set_price(float price)","begin",set_price_list);

    /* The original set price method is invoked here */
    user_set_price(price);

    /* Notify end of method */
    Notify(this, "STOCK", "void set_price(float price)","end", set_price_list);
}
```

Since conditions are assumed to be side-effect free, we have to avoid detecting events that may be generated during condition execution. This can happen if conditions invoke methods that are declared as event generators in the event interface. To disable the signaling of a primitive event

when the condition function is executed, we maintain a global variable to indicate whether the events signaled should be acknowledged or not.

### 3.2.2 Composite Event Detection

Composite event specifications are pre-processed and code is inserted into the main program for generating event graphs at execution time. Leaf nodes of the event graph corresponds to primitive or external events. Internal nodes correspond to event sub-expressions. Each node has a list of subscribers to whom it has to notify once the event denoted by that node is detected. For example, a primitive event will have a list of subscribers which may contain rules and other event expressions in which it takes part. The same mechanism is uniformly used for composite events as well. Since primitive events, composite events as well as rules are derived from a base NOTIFIABLE class, the subscribers' list is implemented as a linked list by specifying it as an attribute of the NOTIFIABLE class. Separate lists are maintained for composite event and rule subscribers to allow for optimization as part of future enhancements. Every node of the event graph has outgoing edges equal to the number of subscribers it has. For example, for the events shown in the STOCK class, the following code is inserted in the main program to generate the event graph. The code below also indicates how the rule R1 (in the STOCK class) is translated:

```
/* Main program */
STOCK IBM, DEC, Microsoft;
LOCAL_EVENT_DETECTOR *Event_detector

main()
{
    ..................
    /* Creating the local event detector */
    Event_detector = new LOCAL_EVENT_DETECTOR();

    /* Creating primitive events */
    EVENT *STOCK_e1 = new PRIMITIVE("STOCK_e1", "STOCK", "end", "int sell_stock(int qty)");
    EVENT *STOCK_e2 = new PRIMITIVE("STOCK_e2", "STOCK", "begin", "void set_price(float price)");
    EVENT *STOCK_e3 = new PRIMITIVE("STOCK_e3", "STOCK", "end", "void set_price(float price)");

    EVENT *STOCK_e4 = new AND(STOCK_e1, STOCK_e2); /*Composite event AND */

    /* Creating Rule R1 */
    RULE *R1 = new RULE("R1", STOCK_e4, cond1, action1, CUMULATIVE);
    R1->set_coupling_mode(DEFERRED);
    R1->set_priority(10);
    R1->set_trigger_mode(NOW);
    R1->set_mode(DEFERRED);
    ..................
}
```

The Event Detector is implemented as a class and hence we have a single instance of this class per application (termed the local event detector). Each of the primitive events defined is maintained as a list based on the class on which it is defined. When the local event detector is notified of a method invocation for a class by the DBMS, it is propagated only to the primitive events defined for that class. The local event detector maintains separate lists for temporal and explicit events. Once a primitive event node is notified it checks the method signature with the one

that has been sent. If it matches, it notifies all its subscribers. Similarly once a complex event node is notified, it is activated based on the operator semantics [5], and notifies subscribers in its list. A rule node, in addition to notification, creates a thread with the condition and action function as a unit to be executed when the thread is scheduled. The local event detector schedules these threads. Our implementation based on event graphs is demand-driven (analogous to a data-flow scheme) and does not propagate parameters to irrelevant nodes. Furthermore, this approach efficiently supports subscribe/unsubscribe of rules to events as insertion/deletion in its subscriber's list. Our implementation of the local composite event detector supports:

1. *Multiple contexts in a single event graph:* Our implementation uses the same event graph to detect an event in different contexts. Each node of the graph maintains all the contexts in which it has to collect parameters as well as to whom it has to propagate the parameters. Whenever a rule is defined its context is propagated to all the nodes in its event graph. The counter for that particular context is incremented. If the counter was previously 0, the set of nodes corresponding to the event expression starts detecting events in this context. Specifying PREVIOUS (for rule_trigger_mode) for this rule will not have any effect. Introduction of this mechanism for event detection in the presence of contexts helps avoid detecting events in the continuous and cumulative mode as they have significant storage requirements. Once a rule is disabled or deleted the event expressions are again notified and the respective counter is decremented. If the counter is reset to 0 events are no longer detected in that context.

2. *Parameter passing:* Composite events pose additional problems for parameter computation. The difficulties involved in passing complex data types as parameters across applications has been detailed in the previous section. To avoid these pitfalls, currently, we pass only simple data types as parameters. Although it is possible, copying the values of complex data types will add considerable storage overhead. The parameters and component events are all maintained as linked lists at the relevant nodes and hence there is no copying of data. Only the pointers have to be adjusted thereby increasing the efficiency of event detection.

3. *Events crossing transaction boundaries:* The logical unit of work in a DBMS is a transaction. To maintain the correctness of this concept we have to ensure that events (as well as parameters associated with the event) are not carried over across transaction boundaries. This is especially important in the presence of composite events whose detection can span an arbitrary time interval. Consider the case when Transaction1 invokes certain methods and is later aborted. These methods might have triggered certain primitive events whose parameters are recorded in the event graph. If these events (and their parameters) are not flushed when a transaction is aborted (or committed), these events can participate in composite events for another transaction. If we allow events to span transactions, a second transaction might cause the firing of a rule which has constituent primitive events and parameters from a previously aborted transaction. This means that the condition and action functions access parameters which in the database sense does not exist at all (since the previous transaction was aborted, all its effects would have been rolled back in essence making it seem like that method was never executed). The above situation can arise for committed transactions as well although the parameter values may be consistent in this case.

   We provide a flush operation that can either flush the event graph selectively for an event expression or for the entire graph. This is invoked as an action of a rule on abort and commit events. However, these can be easily modified by deactivating these rules if events across transaction boundaries need to be detected.

### 3.2.3 Rule Execution and Scheduling

All primitive event signaling is done by invoking methods of Event Detector class. Since this object is visible to the entire application, the nested triggering of rules by the execution of action function is also readily accomplished. As detailed above, when a primitive event is signaled the local event detector determines which of the primitive event nodes should be notified. Once this is done, the events propagate to the root nodes of the event graph. Whenever a rule is triggered in immediate coupling mode, it gets a free thread_id from a pool of free threads and transforms the function which checks the condition and performs the action to a thread with the appropriate priority. Once all the immediate rules are in the form of threads, the main application is suspended and the rule scheduler is invoked. Once all the rules are executed, the triggering transaction resumed execution. Figure 4 shows the psuedocode for scheduling and how the thread function is packaged.

```
Initiate_thread()
{
 If (there is rule to be scheduled) {
    priority = assign_priority();
   If (free thread is available) {
      thread_id = get_thread();
      update_thread_pool();
      create_thread(     thread_id, priority,
              cond_action, rule_cond,
              rule_action );
     } } }
```

```
Condition and action packaged as
 the body of the thread

Cond_action(rule_cond, rule_action)
{
  void (*cond)() = &rule_cond;
  void (*action)() = &rule_cond;
sub = begin_subtransaction(current)        ;
 if((*cond)() == True)
     (*action)();
  end_subtransaction(sub);
 }
```

Figure 4: Rule execution using threads

Since a deferred rule is executed as a transformed rule (with an $A^*$ event) in immediate coupling mode, it is triggered only when pre_commit primitive event is signaled by the transaction manager and hence it is treated in the same way as an immediate rule. Consider

REACTIVE Stock;
event any_stk_price('any_stk_price','Stock','begin', 'set_price(float price)');
rule R1(any_stk_price, checkSalary, resetSalary, CHRONICLE, DEFERRED);

The above rule is translated internally (and rule R1 is modified to reflect immediate mode) to

event def_rule_event = A*(beg_trans, any_stk_price, pre_commit);
rule R1(def_rule_event, checkSalary, resetSalary, CHRONICLE, DEFERRED);

The event to be monitored is changed to an $A^*$ event and the rule subscribes to the new $A^*$ event created. The nested rule triggering is handled by assigning priorities to threads based on their level and the priority of the rule that triggered them. We currently support depth first execution of rules using the priority class of the triggering rule and the priority class of the triggered rule to compute a new priority value.

### 3.3 Implementation of Nested transactions

Although rules can be executed as part of the triggering transaction (either as in-line code or a procedure call), this does not support concurrent execution of rules. Furthermore the unit of recovery will be the entire triggering transaction. We have taken the approach of executing rules as subtransactions. To support nested execution of rules as subtransactions we have adopted the Nested transaction model [15]. In nested transactions, flat transactions are enhanced by a hierarchical control structure. Each nested transaction consists of either primitive actions or some nested transactions (called subtransactions of the containing transaction). This allows a dynamic decomposition of a transaction into a hierarchy of transactions, hence preserving all properties of a transaction as a unit. Subtransactions follow all the ACID properties of a flat transaction except durability, due to the fact that the commit of a subtransaction is conditional upon its parent's commit. As shown in [11], the nested transaction concept fits well with the semantics of rule execution in immediate mode and some extensions are necessary to model other coupling modes. Concurrent execution of several rules can be provided using a concurrency control strategy for an implementation of the nested transaction model. We have implemented a lock-based algorithm for supporting nested transaction model in Sentinel. [2] discusses nested transaction implementation in detail.

The limitations of the Open OODB architecture for implementing nested execution of rules are:

- The underlying storage manager (Exodus) does not support nested transaction calls. Although Exodus supports multiple clients (each as a thread), each client supports only a flat transaction model,
- All the lock information about objects is maintained by the Exodus server and client. No lock information is present in the Open OODB. Furthermore, in the version used, lock modes cannot be currently specified to Exodus through the interface (it can only be set globally).
- Currently, all objects that are fetched by the Open OODB from Exodus, are always written back to the server when the transaction commits. The lock is escalated into an exclusive lock at the commit time. Hence in case of data conflict among applications, only one transaction will commit and the rest are aborted,
- The transaction calls (beginTransaction, commit, abort) of Open OODB in the current architecture are tightly coupled with the same calls in the Exodus client interface. That is, there is a one-to-one correspondence between the transaction calls of Open OODB and Exodus. This implies that Open OODB has only a skeletal transaction manager which maps calls to Exodus utilizing the underlying transaction model provided by Exodus.

To overcome one of the above problems, we have introduced two functions namely fetch_for_read() and fetch_for_write(). When we fetch an object from the Exodus server using fetch_for_read() we do not write back the object to the Exodus server. This allows a read only transaction to concurrently execute and commit with other transactions.

With the above limitations, we had two alternatives for supporting nested transactions: i) extend Exodus to incorporate a nested transaction model, or ii) extend the transaction manager of Open OODB to incorporate nested transactions without including recovery. Although the first option is more desirable in the long term, we opted for the second choice as it involves developing the transaction layer in the Open OODB in a shorter time to support concurrent execution of

rules. The implementation entailed extending the skeletal transaction manager of Open OODB to a full-fledged transaction manager.

The subtransaction calls are managed by the transaction manager of Open OODB. An initial connection to Exodus is established when the the top-level transaction is invoked. If there is an object fault in the nested call (subtransaction), the object is fetched through this connection. Although the lock information of this (top-level) transaction is maintained by the Exodus server, it is also maintained in the Open OODB as well to enforce concurrency control among substransactions. A separate lock manager is implemented in the Open OODB to handle concurrency control among nested transactions. The lock manager contains the lock information for all transactions including the top level transaction. The modification to the Open OODB is illustrated in the Figure 2.

Our approach does not alter the semantics of concurrent top-level transactions and at the same time supports nested subtransactions within each application/client. The condition-action portions of a rule are packaged into a procedure (by the pre-processor) that is scheduled as a thread by the rule scheduler as described in an earlier section. Below we briefly highlight the implementation of the transaction subsystem to execute rules.

### 3.3.1 Transaction manager

The transaction subsystem consisting of the following components.

**The Transaction manager container Class (Trans_Mgr)** : This Class maps the user function calls to the appropriate transaction manager calls of the kernel. Two function calls were introduced to distinguish between the top level transactions and the subtransactions namely *begin_transaction* and *begin_subtransaction*. The latter takes the transaction pointer of the parent as the input parameter and both return transaction pointers as a result of the invocation. This container class provides an interface by which we can seamlessly integrate different policies of the transaction system. Only one instance of this class is created per application. The constructor of this class does the following: (i) create and initialize the semaphores which would be used for synchronization. (iii) create an instance of the global *Counter* class. This keeps track of the transaction_id of the previous transaction based on which we calculate the current transaction_id, and (iii) initialization of the lock table. The methods of this class take care of allocation and freeing of semaphores. The following is the interface provided by this class

1. *begin_transaction():* This calls begins the transaction in the Exodus storage manager. Every operation from now on is within the scope of this transaction as far as Exodus is concerned. In addition this function creates an instance of the transaction class. While creating the transaction instance its transaction_id is calculated appropriately. It is calculated with respect to the previous top-level transactions already created.

2. *begin_subtransaction():* It takes the pointer of the parent transaction object as the input parameter. An invocation of this call creates an instance of the transaction class. The transaction_id of the subtransaction is calculated with respect to the parent transaction.

3. *Set_lock() :* Invocation of this call allows for setting the locks in appropriate modes on particular object for a particular subtransaction. Presently only two lock modes (read and exclusive) are supported.

4. *set_transaction():* This call sets the current transaction to the transaction supplied as the argument (transaction_id) to the function. This function is needed to switch the context of

the transaction for scheduling concurrent subtransaction.

5. *Transaction Class:* This is the component of the kernel which implements the nested transaction model. Every transaction in an application is an instance of this class. Among other methods the critical member functions which implement the nested transaction model are Set_lock, Upgrade_lock and Release_lock.

### 3.3.2 The Lock manager

We have implemented lock the table in the local address space (which is shared by all subtransactions) for each instance of the Open OODB. We assume the basic locking rules proposed in [10]. Enforcement of the basic locking rules requires maintaining retain mode and lockmodes as a minimum. However, to efficiently implement *distributed nested spheres of control* and *distributed disjoint spheres of control* [10, 2], it is necessary to search the entire tree to make sure locks can be granted in an appropriate mode. To avoid an exhaustive search of the transaction tree, we have additional information in each node of the transaction tree. This reduces the search to the nodes along the path to the root (in the worst case when the closest common ancestor is the root node). Below, we describe the holdmodes introduced for that purpose.

**Lockmodes & Holdmodes:** Lockmodes specify the mode (Shared, Exclusive or Read only) in which a lock is is held on an object. To avoid excessive search, we introduce the following **holdmodes** which are in addition to the lockmodes: Hold(x)/holdsub(x)/retain(x)/retainsub(x)/grantmode. These fields define the status of a lock and its availability to the transactions in the sphere of control and to the transactions outside. In our algorithm we have used these fields along with the lockmodes to determine the availability of lock in a particular mode. Below we briefly describe these fields.

- **Hold(x):** Indicates if the transaction in question is holding a lock (shared, Exclusive or Read only) on a particular object 'x'.
- **Holdsub(x):** It is a numerical value indicating the number of subtransactions holding the lock on object 'x'. If the lock held is shared or read only, then it will be the number of (sub)transactions holding the lock on the object. If the lock being held is exclusive, then this value will be one.
- **Retain(x):** When a subtransaction commits, locks are retained by its parent thereby forming a sphere of control. This field indicates the sphere of control.
- **Retainsub(x):** It is a numerical value indicating the number of subordinate transactions retaining the lock on the object. By checking this field of the parent/ancestor, it is possible to obtain the information about the existence of one or more spheres of control.
- **Grantmode:** Gives the lockmode in which the request can be granted.

The use of these holdmodes avoids exhaustive search of the hierarchy of the nested execution. In fact the algorithm guarantees that the search can be limited to the path from the node requesting the lock to the root of the transaction tree containing that node.

**Lock table:** Unlike in flat transactions where each node in the lock table (which is a hash table on the object-id as the hash attribute) corresponds to a unique object, an object may be used by several subtransactions resulting in multiple nodes within the same bucket of the hash table for

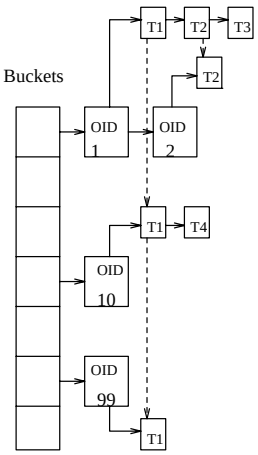


Figure 5: Anchored Hash Table

a given object. In our implementation, if a transaction acquires a lock on an object, then all its superiors also (along with the transaction itself) will be represented by nodes in the lock table. Hence in order to determine the grantability of a lock mode for an object for a transaction (node) we need to traverse along a path from itself (node) to the top level transaction(root). A naive implementation of this would be inefficient since the number of nodes in a bucket of the hash table increases by a number depending upon the depth of the tree. Also, the number of times the lock table is accessed is increased by the same amount (each acquire and release would require us to update the information to all superiors). To circumvent this problem we use an 'Anchored Hash Table' (AHT) for implementing the lock table. An AHT will have an anchor node for each object that hashes to that bucket. If more than one object hashes to the same bucket in the lock table, then we have a linked list of anchor nodes which in turn has a linked structure for the object. This anchor node serves as the virtual node for that object. The hash table is illustrated in Figure 5. In the figure, the broken line represents the linked list of nodes belonging to the same transaction. With the AHT, the number of nodes accessed in the worst case is

**depth of tree * number of nodes with non-null hold field + m**

as compared to a naive implementation whose worst case complexity is
$\sum_{i=1}^{m}$ **(depth of tree * number of nodes with non-null hold field)**

where, m is the number of objects hashing to the same bucket.

### 3.3.3 Synchronization

We have used semaphores for acquiring and releasing of locks atomically. When a requested lock is held in a conflicting mode, the (sub)transaction is required to wait on that lock using a semaphore. In a flat transaction model, for a shared lock, a transaction can wait on a single semaphore with all other transactions which are waiting for the same shared lock. In case of an exclusive lock we need a unique semaphore for each transaction. In a nested transaction model, even the transactions waiting for a shared lock cannot be blocked on the same semaphore. This is because, when an existing (shared) lock on an object is released, not all transactions waiting on

the same shared lock can acquire it (due to nested spheres phenomenon). Hence even for a shared lock mode each transaction has to wait on a unique semaphore. Therefore the total number of semaphores required in a transaction increases and is equal to the total number of transactions (1 + the number of subtransactions).

# 4 Conclusions

In this paper, we have presented an integrated architecture for an active OODBMS and described its implementation. We have discussed the implementation details of ECA rule transformation, composite event detection, rule execution and nested transactions. The underlying concepts behind our architecture and implementation can be easily adapted to the relational model as well. For example, the implementation of composite event detection can be easily tailored to a relational model since the individual linked lists maintained by the composite event detector can be viewed as tuples. Our future work includes expanding the rule management support to public, private, and protected rules, investigating efficient ways of providing the semantics of detached rule execution, implementation of a global event detector, and recovery at the rule/subtransaction level.

The implementation of Sentinel has uncovered several problems, which are specific to the host environment (C++ in our case). The implementation of the detached coupling mode entails generating an entire application with all the class definitions in the triggering application as the condition and action functions might refer to both program and database objects. The problems being addressed by OMG and Corba need to be resolved for the implementation of detached mode. An alternative is to extend the nested transactions semantics to include detached execution of rules. Implementation of recovery for the nested subtransactions requires considerable enhancements to the Exodus storage manager.

### Acknowledgment

This work is supported, in part, by the National Science Foundation IRI-9011216, by the Office of Naval Research and the Navy Command, Control and Ocean Surveillance Center RDT&E Division, and by the Rome Laboratory. We would like to thank S. Karpuram for his comments on an earlier version of the paper. We thank Ms. Anwar for implementing Sentinel pre- and post-processors.

# References

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

[2] R. Badani. Nested transactions for concurrent execution of rules: Design and implementation. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, October 1993.

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

[4] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Anatomy of a composite event detector. Technical Report UF-CIS-TR-93-039, University of Florida, E470-CSE, Gainesville, FL 32611, December 1993. (Submitted for publication.).

[5] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proceedings, International Conference on Very arge Data Bases*, August 1994. (To appear.).

[6] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 1994. (To appear).

[7] S. Gatziu and K. R. Dittrich. Events in an Object-Oriented Database System. In *Proc. of the 1st International Conference on Rules in Database Systems*, September 1993.

[8] S. Gatziu and K. R. Dittrich. Detecting Composite Events in Active Databases Using Petri Nets. In *Proc. of the 4th International Workshop on Research Issues in data Engineering: Active Database Systems*, pages 2–9, February 1994.

[9] 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.

[10] T. Haerder and K. Rothermel. Concurrency Control Issues in Nested Transactions. IBM Research Report RJ5803, Aug. 1983.

[11] M. Hsu, R. Ladin, and D. McCarthy. An Execution Model for Active Data Base Management Systems. In *Proceedings 3rd International Conference on Data and Knowledge Bases*, Washington, D.C., Jun. 1988.

[12] Texas Instruments. Open OODB Toolkit, Release 0.2 (Alpha) Document, September 1993.

[13] V. Krishnaprasad. Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, March 1994.

[14] 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, August 1991.

[15] J. Moss. Nested Transactions: An Approach To Reliable Distributed Computing. MIT Laboratory for Computer Science, MIT/LCS/TR-260, 1981.

[16] Z. Tamizuddin. Rule execution and visualization in active oodbms. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, May 1994.

[17] S. Vinoski. Distributed object computing with corba. *C++ Report*, pages 33–38, July-August 1993.

[18] D. Wells, J. A. Blakeley, and C. W. Thompson. Architecture of an open object-oriented database management system. *IEEE Computer*, 25(10):74–81, October 1992.

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

# Appendix A. A detailed example

## Original program

```
class STOCK : public REACTIVE
{
    private:
    ......
    public:
    ......
```

```
        event end(e1) int sell_stock(int qty);
        event begin(e2) && end(e3) void set_price(float price);
        int get_price();
        event e4 = e1 ^ e2; /* AND operator */
        rule R1[e4, cond1, action1, CUMULATIVE, DEFERRED]; /* class level rule */
};

int STOCK::sell_stock(int qty) { ..... }
void STOCK::set_price(float price) { ..... }
int STOCK::get_price() { ..... }

/* Main program */
STOCK IBM, DEC, Microsoft;
main()
{
        .....
        /* Creating instance level primitive event */
        event set_IBM_price("set_IBM_price",IBM,"begin","void set_price(float price)");
        event seq_event = STOCK_e4 >> set_IBM_price; /* SEQUENCE operator */

        /* Creating a rule which contains both class level and instance level events */
        rule R2[seq_event, cond2, action2,,,20, PREVIOUS];
        .....

        OpenOODB->beginTransaction();
                IBM.set_price(115.00);
                DEC.set_price(100.00);
                Microsoft.sell_stock(200);
                DEC.get_price();
                IBM.set_price(75.95);
        OpenOODB->commitTransaction();
}
```

## Preprocessed program

```
class STOCK : public REACTIVE
{
    private:
    .....
    int user_sell_stock(int qty);
    void user_set_price(float price);
    public:
    .....
    int sell_stock(int qty);
    void set_price(float price);
    int get_price();
};

int STOCK::sell_stock(int qty)
{
    int ret_value;
    /* Parameters are collected in a linked list */
```

```
    PARA_LIST *sell_stock_list = new PARA_LIST();
    sell_stock_list->insert("qty", INT, qty);

    /* The original sell stock method is invoked here */
    ret_value = user_sell_stock(qty);

    /* Notify end of method */
    Notify(this, "STOCK", "int sell_stock(int qty)","end",sell_stock_list);
    return(ret_value);
}

int STOCK::user_sell_stock(int qty)
{
    /* original sell_stock method */
}

void STOCK::set_price(float price)
{
    /* Parameters are collected in a linked list */
    PARA_LIST *set_price_list = new PARA_LIST();
    set_price_list->insert("price", FLOAT, price);

    /* Notify begin of method */
    Notify(this, "STOCK", "void set_price(float price)","begin",set_price_list);

    /* The original set price method is invoked here */
    user_set_price(price);

    /* Notify end of method */
    Notify(this, "STOCK", "void set_price(float price)","end", set_price_list);
}

int STOCK::user_set_price(float price)
{
    /* original set_price method */
}

int STOCK::get_price(char *n1) { ..... }

/* Main program */
STOCK IBM, DEC, Microsoft;
LOCAL_EVENT_DETECTOR *Event_detector

main()
{
    .....
    /* Creating the local event detector */
    Event_detector = new LOCAL_EVENT_DETECTOR();

    /* Creating primitive events */
    EVENT *STOCK_e1 = new PRIMITIVE("STOCK_e1", "STOCK", "end", "int sell_stock(int qty)");
    EVENT *STOCK_e2 = new PRIMITIVE("STOCK_e2", "STOCK", "begin", "void set_price(float price)");
    EVENT *STOCK_e3 = new PRIMITIVE("STOCK_e3", "STOCK", "end", "void set_price(float price)");
```

```
    EVENT *STOCK_e4 = new AND(STOCK_e1, STOCK_e2); /*Composite event AND */

    /* Creating Rule R1 */
    RULE *R1 = new RULE("R1", STOCK_e4, cond1, action1, CUMULATIVE);
    R1->set_mode(DEFERRED);

    /* Creating instance level primitive event */
    PRIMITIVE *set_IBM_price = new PRIMITIVE("set_IBM_price", IBM, "end", "void set_price(float price)");

    EVENT *seq_event = new SEQ(STOCK_e4, set_IBM_price); /* Composite event SEQUENCE */

    /* Creating Rule R2 */
    RULE *R2 = new RULE("R2", seq_event, cond2, action2, RECENT);
    R2->set_priority(20);
    R2->set_trigger_mode(PREVIOUS);

    OpenOODB->beginTransaction();
        IBM.set_price(115.00);
        DEC.set_price(100.00);
        Microsoft.sell_stock(200);
        DEC.get_price();
        IBM.set_price(75.95);
    OpenOODB->commit();
}
```

This example illustrates the wrapper methods introduced and conversion of application level event specification to system calls during preprocessing stage. It also illustrates the use of class level and instance level events/rules. Three class level primitive events, e1 as end-method event of sell_stock(), e2 as begin-method and e3 as end-method event of set_price() are declared. A class level composite event e4 is defined which is an AND of e1 and e2. A class level rule R1 is defined on event e4. Instance level primitive event *set_IBM_price* is defined for Stock object IBM. A composite sequence event is defined which is a combination of an instance level and class level event and finally rule R2 is defined on the sequence event(seq_event). Notice that after preprocessing the user defined methods 'sell_stock' and 'set_price' are renamed as 'user_sell_stock' and 'user_set_price' and wrapper methods 'sell_stock' and 'set_price' are introduced. As seen from the example appropriate code is introduced in the wrapper methods to notify the events. Also the application level rule and event specification are preprocessed to appropriate code for generation of event and rule objects along with the relevant parameters.

Regarding the detection of events Rule R2 will be fired first because it is in *immediate* mode with parameters {{DEC, price, FLOAT, 100.00}, {Microsoft, qty, INT, 200}, {IBM, price, FLOAT, 75.95}}. Rule R1 will be fired later since it is in *deferred* mode with parameters {{IBM, price, FLOAT, 115.00}, {DEC, price, FLOAT, 100.00}, {Microsoft, qty, INT, 200}}. Both DEC and IBM prices will be parameters to Rule R1 since its context is specified to be CUMULATIVE. Refer to [4] for details on parameter computation for various contexts.