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

S. Chakravarthy
Database Systems Research and Development Center
Computer and Information Science and Engineering Department
University of Florida, Gainesville, FL 32611
Email:sharma@cise.ufl.edu
URL: http://www.cise.ufl.edu/~sharma

1 Introduction

Active or reactive database management systems (DBMSs) provide an event-based rule capability that can be used to support a number of database functionality (e.g., integrity enforcement, view materialization, management of index structures, applicability of compiled query plans when access methods change) in a uniform way. Rules used for supporting active capability consists of: an event expression, one or more conditions, an action, and a set of attributes. An event expression specifies the sequence of events whose occurrence triggers the evaluation of the condition. A condition is a side effect-free boolean computation (set computation in general) on the database state and an action is an arbitrary sequence of operations. Attributes, typically, specify rule characteristics such as coupling mode, event consumption mode, and precedence relationship among rules. A rule with these components is termed an ECA or event-condition-action rule in the literature [3].

2 Paradigm Differences

Clearly, there is a paradigm shift when we move from the relational model to an object-oriented one. The differences between the two data models profoundly influence how the concepts and techniques are carried over from one model to the other. Below, we enumerate some of the differences between the data models that led to the design choices made for *Sentinel*.

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

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

3. The principle of inheritance and its effect on rule incorporation, and

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

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

Another dimension of choice is the object-oriented environment into which ECA rules are being incorporated. For example, statically compiled languages, such as C++ may make implementation of some features extremely difficult (or even impossible) as compared to dynamically interpreted languages such as Common Lisp or Smalltalk. Support for dynamic addition of rules is relatively easy in the latter environments whereas it is not possible to support it without relinking unless a C++ interpreter is used. This choice of environment may limit the capability of the system in some ways.

3 Design Choices

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

1. Support primitive (e.g., database, temporal, and external) events as well as composite events using a small set of event operators. Support a broad set of event consumption modes to meet the requirements of composite events for a large number of applications.

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

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

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

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

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

6. Provide a uniform mechanism for associating rules to all instances of a class as well as individual instances of one or more classes.

7. Support visualization of events/rules and interactive debugging to facilitate the design of rules

8. Provide a way to support rules based on global events (primitive and composite events defined and detected in other applications) for distributed active environments

9. Provide support for defining rules dynamically to the extent possible. This requires at the least relinking of the application with a library of rules declared outside of the application.

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

The last three items augment the basic active functionality to make it useful for larger classes of applications. We strongly believe that the utility of the system will be significantly enhanced with the availability of appropriate tools.

3.1 Object Classification

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

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

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

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

4 Snoop – The Event Specification Language

Events are broadly classified into: i) primitive events: events that are pre-defined in the system, ii) Composite events: events that are formed by applying a set of operators to primitive and composite events. The event classification is shown in Figure 1.

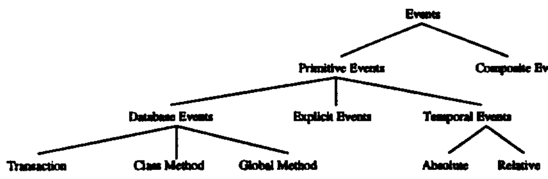

Figure 1: Event Classification

4.1 Snoop Event Operators

Below, we summarize the operators supported in Snoop with brief explanations:

1. AND ($\wedge$): Conjunction of two events E_1 and E_2, denoted $E_1 \Delta E_2$, occurs when both E_1 and E_2 occur (the order of occurrence of E_1 and E_2 is irrelevant).

2. OR (|): Disjunction of two events E_1 and E_2, denoted $E_1 \nabla E_2$, occurs when either E_1 or E_2 occurs (simultaneous occurrence is currently excluded).

3. SEQ ($\gg$): Sequence of two events E_1 and E_2, denoted $E_1 \gg E_2$, occurs when E_2 occurs provided E_1 has already occurred. This implies that the time of occurrence of E_1 is guaranteed to be less than the time of occurrence of E_2.

4. NOT (!): The NOT operator, denoted $!(E_1, E_2, E_3)$, detects the non-occurrence of the event E_2 in the closed interval formed by E_1 and E_3. It is rather similar to the SEQ operator except that E_2 should not occur between E_1 and E_3.

5. A: One can express the occurrence of an aperiodic event in the half-open interval formed by E_1 and E_3. An aperiodic event is denoted as A(E_1, E_2, E_3), where E_1, E_2 and E_3 are arbitrary events. The event A is signaled each time E_2 occurs during the half-open interval defined by E_1 and E_3.

6. A^*: This is a cumulative variant of A expressed as $A^*(E_1, E_2, E_3)$. It is useful when a given event is signaled more than once during a given interval, but

rather than detecting the event and firing the rule every time the event occurs, the rule has to be fired only once. A^* is detected when E_3 occurs and accumulates the occurrences of E_2 in the half-open interval formed by E_1 and E_3.

7. P: A periodic event is defined as an event E that repeats itself within a constant and finite amount of time. It is denoted as P(E_1, E_2, E_3), where E_1 and E_3 are any types of events and E_2 is a relative temporal event. P occurs for every amount of time specified with the time string of E_2 in the half-open interval (E_1, E_3]. The time string should be positive and should not have any wild card to prohibit continuous occurrences of P.

8. P^*: P^* is a cumulative variant of P and is denoted by P^*(E_1, E_2, E_3). P^* occurs only once when E_3 occurs and accumulates the time of occurrences of the periodic event whenever E_2 occurs.

9. PLUS (+): Sequence of an event E_1 after a time interval TI, denoted E_1 *+ [TI]* occurs when TI time units are elapsed after E_2 occurs.

4.2 Parameter Contexts or Event Consumption Modes

The notion of parameter contexts is introduced in Snoop to capture application semantics for computing the parameters or consuming event occurrences (of composite events) when they are not unique. These contexts are precisely defined using the notion of initiator and terminator events. An initiator of a composite event is a constituent event which can start one detection of the composite event, and a terminator is a constituent event which can detect an occurrence of the composite event.

- **Recent:** In this context, *not all occurrences/instances* of a constituent event will be used in detecting a composite event, only the most recent occurrence of the initiator for any event that has started the detection of that event is used. When an event occurs, the event is detected and all the occurrences of events that cannot be the initiators of that event in the future are deleted (or flushed). Furthermore, an initiator of an event (primitive or composite) will continue to initiate new event occurrences until a new initiator occurs.

- **Chronicle:** In this context, for an event occurrence, the initiator, terminator pair is unique (after a detection, the initiator and the terminator are flushed). The oldest initiator is paired with the oldest terminator for each event (i.e., in chronological order of occurrence).

- **Continuous:** In this context, each initiator of an event starts the detection of that event and is saved until a terminator occurs. A terminator is paired with every initiator. Thus the terminator may detect one or more occurrences of the same event. The initiator and the terminator are discarded after an event is detected. This context is especially useful for tracking trends of interest in a moving window governed by the initiator event.

- **Cumulative:** In this context, all occurrences of an event type are accumulated as instances of that event until a terminator occurs (that is, the event is detected). Thus all the occurrences of the event detection are packaged in timely order. Whenever an event is detected, all the occurrences that are used for detecting that event are deleted.

5 Sentinel Architecture

Sentinel is based on the Open OODB [11]. Sentinel's ECA rule support enhances the Open OODB from a passive OODB to an active one.

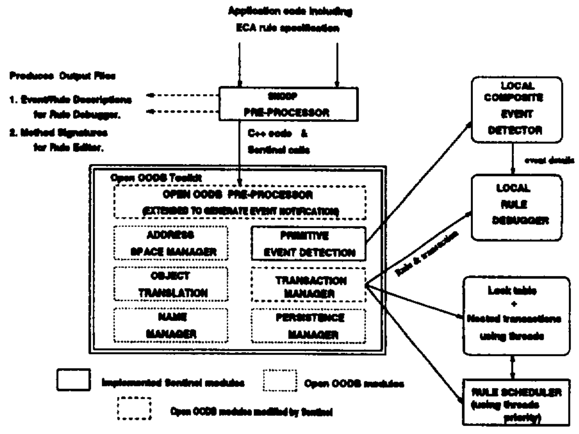
Application code including
ECA rule specification
Produces Output Files
1. Event/Rule Descriptions for Rule Debugger.
2. Method Signatures for Rule Editor.
SNOOP PRE-PROCESSOR
C++ code & Sentinel calls
Open OODB Toolkit
OPEN OODB PRE-PROCESSOR
(EXTENDED TO GENERATE EVENT NOTIFICATION)
ADDRESS SPACE MANAGER
PRIMITIVE EVENT DETECTION
OBJECT TRANSLATION
TRANSACTION MANAGER
NAME MANAGER
PERSISTENCE MANAGER
LOCAL COMPOSITE EVENT DETECTOR
event details
LOCAL RULE DEBUGGER
Rule & transaction
Lock table + Nested transactions using threads
RULE SCHEDULER (using threads priority)
Implemented Sentinel modules
Open OODB modules
Open OODB modules modified by Sentinel

Figure 2: Sentinel Architecture

Figure 2 indicates the functional modules of the open OODB and the extensions for Sentinel. These extensions include:

- Primitive event detection: A method can be specified as a primitive event, and the occurrences of the primitive events are notified to the local event detector when the method is invoked. We have modified the Open OODB preprocessor to wrap the method invocation with the notifications to the local event detector in order to detect primitive events.

- Composite event detection: Composite events defined within an application are detected by using a sequence of primitive events detected according to the specified parameter context of the composited event [4, 8]. Each Open OODB application has its own local event detector.

- Nested transactions: The transaction manager in the client address space supports nested transactions [2] for concurrent execution of rules. Solaris threads are used both for prioritized and concurrent rule execution.

- Snoop preprocessor: The Snoop preprocessor transforms the ECA rules specified either as part of a class definition or as part of an application. The preprocessor converts the high-level user specification of ECA rules specified in Snoop language [6] into appropriate code for event detection, parameter computation, and rule execution.

Figure 3 shows how the class lattice of the Open OODB has been extended by Sentinel. 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.

5.1 Implementation Status

Current release of Sentinel (Sentinel release 0.9) has the following functionality:

1. A full event detector (both primitive and composite events in all the contexts proposed in the sentinel reports, viz. recent, chronicle, continuous, cumulative, and unrestricted). This has been implemented as a separate module (and can be executed as a separate thread in each application/client).

2. Support for rules and events both at the class and instance level. Sentinel (user-friendly) specification of rules is translated by a sentinel pre-processor written at UF which transforms the source code. This is fed to the Open OODB pre-processor which generates wrappers for methods in addition to other translations.

3. A dynamic rule editor for specifying rules outside of the application in an interactive manner. These rules can be enabled and disabled requiring only relinking (and not recompilation of the application). For details, refer to [10]. Condition and action portion of a rule can either be C++ functions, OQL (Object Query Language) statements, or a combination thereof.

4. A global event detectior for detecting and managing events across applications. The global event detector communicates with the local event detectors through RPC and socket-based communication to detect global events (not shown in Figure 2). For details, refer to [9]

5. Rules are currently executed as nested transactions using the priority information supplied along with rules. Both concurrent execution of rules using nested transactions and serial execution using priorities are supported by Sentinel. Nested transactions are supported in the client address space and a separate lock table is maintained by Sentinel. This essentially gives a two level transaction management; top level transaction concurrency is controlled by Exodus at the server and the nested transaction concurrency (without recovery) is controlled by the client.

6. Rule execution can be visualized using a visualization tool [7]. This tool shows both rule execution and event detection graphically. This tool can be used on a dump created by Sentinel (along the lines of Xdb but meant as a rule/event debugger, not a low level one). We are currently extending this tool to support: i) online visualization (as the application is being executed) and as a post-execution visualization tool, ii) interactive tool to enable/disable event detection and subscribe/unsubscribe rules. This is a tool for debugging rule-rule interaction, rule-database object interaction, and interaction among database objects. Locking and other information is displayed on demand.

7. Two applications are being developed: a planning application for the Navy and a "clean-room" manufacturing application. The planning applications is being extended to include weather data and its effect on unit objects and plans. The "clean-room" application is being developed to stress-test the ECA rule capability.

6 Demonstration

The following Sentinel features will be demonstarted at Sigmod'97: a) Application development with ECA rule capability, b) Local event detector, c) Global event detector, d) Dynamic rule editor, e) rule visualization and debugging environment

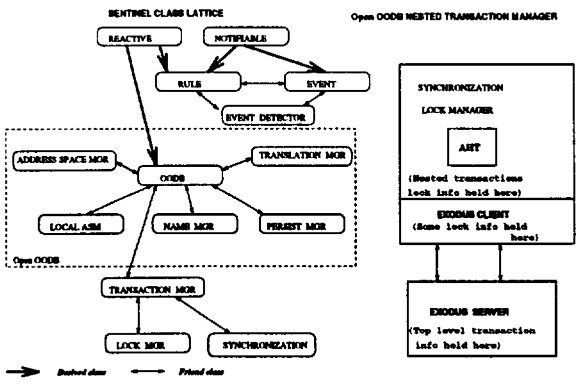

Figure 3: Class lattice and transaction manager of Sentinel

References

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

[2] R. Badani. Nested Transactions for Concurrent Execution of Rules: Design and Implementation. Master's thesis, CIS Department, University of Florida, 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. Composite Events for Active Databases: Semantics, Contexts, and Detection. In *VLDB Proceedings*, pages 606–617, August 1994.

[5] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. Badani. ECA Rule Integration into an OODBMS: Architecture and Implementation. In *ICDE Proceedings*, pages 341-349, Feb. 1995.

[6] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(10):1–26, October 1994.

[7] S. Chakravarthy, Z. Tamizuddin, and J. Zhou. SIEVE: An Interactive Visualization and Explanation Tool for active Databases. In *RIDS '95 Proceedings*, pages 179–191, October 1995.

[8] H. Lee. Support for temporal events in sentinel: Design, implementation, and preprocessing. Master's thesis, CISE, University of Florida, Aug 1996.

[9] H. Liao. Global events in sentinel: Design and implementation of a global event detector. Master's thesis, CISE, University of Florida, January 1997.

[10] P. Madhabushi. Dynamic rule editor for sentinel: Design and implementation. Master's thesis, ECE, University of Florida, March 1997.

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