Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions *

A. P. Buchmann J. Zimmermann

Dept of Computer Science
Tech. University Darmstadt
Darmstadt, Germany 64293

J. A. Blakeley D. L. Wells

Computer Science Laboratory
Texas Instruments, Inc.
Dallas, TX 75265

Abstract

Active OODBMSs must provide efficient support for event detection, composition, and rule execution. Previous experience, reported here, building active capabilities on top of existing closed OODBMSs has proven to be ineffective. We propose instead an active OODBMS architecture where event detection and rule support are tightly integrated with the rest of the core OODBMS functionality. After presenting an analysis of the requirements of active OODBMSs, we discuss event set, rule execution modes, and lifespan of events supported in our architecture. We also discuss the coupling of event composition relative to transaction boundaries. Since building an active OODBMS ex nihilo is extremely expensive, we are building the REACH active OODBMS by extending Texas Instrument's Open OODB Toolkit. Open OODB is particularly well suited for our purposes because it is the first DBMS whose architecture closely resembles the active database paradigm. It provides low-level event detection and invokes appropriate DBMS functionality as actions. We describe the architecture of the event detection and composition mechanisms, and the rule firing process of the REACH active OODBMS, and show how these mechanisms interplay with the Open OODB core mechanisms.

1 Introduction

Active database systems have been proposed as a new data management paradigm to satisfy the needs of many applications that require a timely response to critical situations. Frequently mentioned examples of such applications are power and communication network management, computer-integrated manufacturing, commodity trading, air-traffic control, plant and reactor control, tracking, monitoring of toxic emissions, and multimedia applications. Common to all is the need to respond in a timely manner to external events. Other applications that could benefit from active database technology are applications in which large volumes of data must be analyzed to detect relevant situations although time constraints are less important, as is the case with intelligence applications and the monitoring of environmental pollutants and their long-term effects. Workflow management is another application domain of active databases that is rapidly gaining importance. It combines the need for event-driven activities with temporal constraints. Finally, a domain for active database technology is the DBMS itself, since the same mechanisms can be applied for unified handling of consistency constraints in homogeneous as well as heterogeneous systems, materialized views, access control, and complex transaction models required for heterogeneous systems and workflow management.

So far, the promises of active database technology have not been realized, particularly in an OO-framework. The main reason, in our opinion, is the lack of a readily available and stable platform that incorporates all the functionality identified to be relevant for many of the above applications. Many good but partial solutions have been proposed to individual aspects of active database technology, such as event detection and composition, rule execution strategies and execution models, and the application of active database techniques for limited domains, such as consistency enforcement and materialized views. Many of the systems that have been built address subsets of the functionality that has been identified as relevant. For example, most implemented systems use sequential rule execution with one coupling mode, either immediate or deferred. They are limited in the set of events that can be detected to basic database operations,

*This work is sponsored in part by the ACT-NET research network on active databases which is part of the program "Human Capital and Mobility" of the European Community.

and the actions are constrained to database updates or transaction aborts (e.g., Postgres [SHP88, SHP89], Starburst [WF90], Ariel [Han92], Exact [DPG91]). Other systems, such as HiPAC [DBB88, CBB89], although ambitious in the defined functionality, have not been fully implemented.

The implementation of an active DBMS with full functionality requires the availability of a full-fledged DBMS into which the active capabilities can be integrated. This fact makes the entry price very high, particularly for active OODBMSs. Building the active capabilities as a layer on top of an existing commercial system, results in difficulties when trying to modify the underlying transaction model or trapping method execution events. Further, basic requirements for efficient implementation of active capabilities, such as threads and their parallel execution, are not provided by many popular operating system implementations.

The REACH project[1] [BBK92, BBK93] attempts to provide a stable research platform based on an extensible object-oriented DBMS, Texas Instruments' Open OODB [WBT92]. Open OODB uses the EXODUS storage manager [CDR86] and is implemented in an extensible way that follows closely the event-driven paradigm of active databases. In this paper we describe the architecture and basic design decisions that are required for a seamless integration of a wide range of active database capabilities into an object-oriented DBMS. The driving motivation behind many of these decisions is to provide an efficient and stable platform without sacrificing essential active capabilities, such as a rich event set with temporal and composite events in addition to the basic database operations, multiple coupling modes, and parallel rule execution. The contributions of this paper are threefold:

1. We clarify the semantics of event composition relative to transaction execution. This issue has not been properly addressed previously, but is essential for efficient event detection and composition.
2. We present an experience report summarizing the difficulties we faced when trying to build an active OODBMS on top of two closed commercial OODBMSs. We conclude from that experience that it is necessary to build the active capabilities *into* the OODBMS and identify important features of the underlying system.
3. We show how the Open OODB sentry mechanism is a key feature in the efficient support of active database management, and present the architecture of the REACH ECA-managers.

[1] REal-time ACtive and Heterogeneous mediator system under development at T.H. Darmstadt

Section 2 characterizes the essential features we expect in an aDBMS. Section 3 discusses events and rule execution in REACH. Section 4 reports and analyzes our negative experiences trying to build a layered system on top of two closed, commercial OODBMSs. Section 5 introduces the Open OODB meta-architecture as the basis for the REACH implementation. Section 6 addresses the architectural decisions needed to perform efficient event monitoring and rule firing, while Section 7 addresses open issues, future work and conclusions.

2 Characterization of Active Database Features

The concept of an active database system is elusive. With commercial relational DBMSs offering basic trigger capabilities and the standards bodies planning on adding them to the SQL-3 standard, the question arises what the basic features of an aDBMS are. In REACH we aim at supporting complex applications, possibly in open environments, for which the primitive triggers offered by today's RDBMSs are not sufficient. In the HiPAC project [DBB88,CBB89], one author participated in the requirements analysis of complex applications (process control, network management, command and control, CIM, and air-traffic control). As a result of that analysis, the primitives and basic abstractions of HiPAC were defined. Recently, we started a study of applications in the areas of power-plant maintenance and operations and telecommunication network management. Preliminary results confirm the validity of the HiPAC primitives and have lead to the identification of some additional capability needed to deal with recovery in open systems [BBK93]. We therefore require from an aDBMS the following basic features:

Flexible rule invocation/triggering: The flexibility of rule invocation is achieved through a clear separation of the triggering event from the condition and action parts of the rule, the richness of the event set, and the power of the algebra that composes those events. In an aOODBMS an event set must comprise arbitrary method invocation events, temporal events, and flow-control events. These events must be composable through an event algebra [DBM88]. While the required richness of the event algebra may be application dependent, we specify that basic compositions, such as, sequence, conjunction, disjunction, negation, closure and history be provided. Good examples of such an algebra are [GD93a, GD93b, CM91, GJ92].

However, a sophisticated event composition algebra is of little use if a rich collection of event types cannot be detected and composed *efficiently*. Therefore, a balance must be found between the expressiveness and complexity of the event algebra and the efficiency with which expressions of this algebra can be evaluated.

Efficient rule invocation/triggering: The efficiency of rule triggering depends on the efficiency of primitive event detection, the efficiency of event composition, and the overhead incurred in deciding whether a rule must be triggered or whether normal processing can proceed. Efficient event detection depends on the availability of a low level trapping mechanism, such as hardware interrupts, virtual memory page faulting, dispatch redefinition, or in-line wrappers. It depends on being simple and parallelizable. Further, overhead derived from storing parameters and garbage-collecting incomplete events must be minimized. Finally, the event composition process should be executed asynchronously with normal processing to avoid unnecessary delays while a decision is made whether normal method execution can proceed or if a rule must be executed.

Flexible rule execution: The flexibility of rule execution is guaranteed through the definition of various coupling modes that define the execution of rules or parts thereof relative to the triggering user-submitted transaction. HiPAC recognized four coupling modes, namely *immediate*, *deferred*, *detached* and *parallel detached causally dependent* [HLM88, CBB89]. In [BBK93] we showed that for closed database applications these coupling modes are sufficient. For open environments, i.e., situations in which rules may cause non-recoverable side effects, two additional coupling modes are required, namely *sequentially detached causally dependent* and *exclusive detached causally dependent*.

Full database capability: An active database system by definition is a full-fledged DBMS. Therefore, persistence, query processing, concurrency control and recovery are essential features.

Efficiency and tight integration of DBMS functionality and ECA-rule execution: The complex applications that are targeted require a tight integration of the active capabilities with the DBMS. The loose coupling between rule management at the application level and a passive DBMS, as has been provided by expert system shells that interface with relational database systems, has proven to be rather inefficient because of the expert system's rule invocation modus (either explicit or through a recognize-act cycle) and the frequent crossing of the application and database interfaces.

Definition of timing constraints: Many applications do require the specification and enforcement of timing constraints. Although we do not view this as an essential requirement of an active database system, we consider timing constraints an important feature for which the necessary provisions should be made early on.

The goal of the REACH project is to provide a stable platform that satisfies the above requirements. Such a stable platform is a necessary condition for the development of meaningful applications using active database technology.

3 Events and Rule Execution

The above requirements determine many of the design decisions that were made for REACH. This section summarizes four important aspects related to rule invocation and execution:

- the primitive and composite events, that are handled;
- the composition of events relative to transaction boundaries;
- the lifespan of composite events;
- the event consumption policies.

Space limitations preclude a full discussion of all these issues. Therefore, we refer to previously published research whenever possible.

3.1 Event Set

The event structure of REACH has been described elsewhere [BBK92]. It is similar in many aspects to other event hierarchies and algebras [DBM88, CM91, GD93b, GJ92]. We describe only briefly the main features needed for a better understanding of the following sections.

REACH recognizes both primitive and composite events. Primitive events can be either method-invocation events, state-change events, flow-control events (such as transaction-related events), and absolute temporal events. Explicit user signals can be modelled as method-invocation events. Temporal events can be either absolute or relative, periodic or aperiodic. A good characterization of primitive events is given in [GD93a].

Events can further be composed through an event algebra. A variety of event algebras of varying complexity have been proposed [DBM88, CM91, GD93b, GJ92]. The REACH algebra inherits the sequence, disjunction and closure of the HiPAC algebra with the same semantics [DBM88]. In addition, it takes from SAMOS the notion of validity interval for an event

and uses SAMOS's negation, conjunction and history operators [GD93b]. In addition, we defined a special kind of temporal event, *milestones*, which are used for time-constrained processing and can be applied to tracking the progress of a transaction relative to its deadline. If the transaction does not reach a milestone in time, the probability of missing its deadline is high and a contingency plan can be invoked [BBK93]. The first REACH prototype supports the event classes for method events, DB-internal events (e.g. commit or persist), time events, and composite events. Future extensions will consider special events, such as milestones, and state change events, which require other low-level event detection mechanisms, such as virtual-memory faults.

It is not the goal of this paper to expand on the semantics of events and the event algebra, as these have been described elsewhere. However, we provide more detail on the semantics of event composition relative to transaction boundaries, as this has not been properly addressed in the literature.

3.2 Event Composition Relative to Transaction Boundaries

Events can be composed using either finite state automata [GJS92], (colored) Petri nets [GD93b] or syntax graphs (e.g., [CKA93], [Deu94]).

A crucial issue for the architecture is the composition of events relative to transaction boundaries and the valid execution strategies of rules, depending on the kind of event. This issue has not been properly addressed elsewhere. Events can be:

- Simple method events (both application-method invocations and transaction-related events, such as BOT, EOT, Commit, Abort).
- Simple temporal events (absolute points in time).
- Complex events where all the primitive method-events originate in the same transaction.
- Complex events where the primitive events originate in different transactions.

REACH distinguishes six coupling modes. In the *immediate* mode a rule is executed, possibly as a subtransaction, at the point at which an event was detected within another transaction. In *deferred* mode, the rule is executed as a subtransaction after the triggering transaction completes its execution but before it commits. In *detached* mode the rule is executed in an independent transaction. In *parallel causally dependent* mode, the rule executes in a separate transaction which may begin in parallel but may not commit unless the triggering transaction commits. In *sequential causally dependent* mode, the rule executes in a separate transaction which may initiate only after the triggering transaction has committed. Finally, in *exclusive causally dependent* mode a rule may execute as a separate transaction which may commit only if the triggering transaction aborts [BBK93].

Not all the combinations of the four kinds of events and the six coupling modes are semantically meaningful. Some that are semantically correct may not be practical for performance reasons. Therefore, we first identify which combinations are reasonable to support in the REACH architecture. Table 1 summarizes the supported combinations.

Single-method events can always be related to the transaction in which they were raised. Therefore, rules triggered by a single-method event can be executed under any coupling mode. Conversely, simple temporal events occur independently of transactions. Therefore, rules triggered by purely temporal events may only be executed in a detached mode.

Composite events where all primitive events originate in a single transaction can be related to that transaction. Therefore, any coupling mode would be semantically correct. However, if a method-event is raised and composite events are allowed to trigger rules in immediate mode, the normal flow of execution must be stopped every time a method event is raised until the event composers have signaled that no complex event that triggers an immediate rule has been completed. This overhead is prohibitive (and collides with our requirement that event composition can be carried out asynchronously). Therefore, we do not support in REACH the combination of complex, single-transaction events with immediate coupling mode.

If the composite event is composed of events originating in more than one transaction, immediate and deferred coupling modes are not well defined, since an ambiguity exists as to which transaction is meant. Therefore, this combination is disallowed. The various detached coupling modes are allowed, provided that the commit/abort dependencies are respected for *all* transactions from where the primitive method events originated.

It should be observed that for any of the detached execution modes, references to persistent objects as well as values can be passed as parameters. References to transient objects are not allowed since these objects may disappear as soon as the originating transaction completes. However, transient objects may be passed by value.

	Single Method	Purely Temporal	Composite 1 TX	Composite n TXs
Immediate	Y	N	(N)	N
Deferred	Y	N	Y	N
Detached	Y	Y	Y	Y
Par.caus.dep.	Y	N	Y	Y (all commit)
Seq.caus.dep.	Y	N	Y	Y (all commit)
Exc.caus.dep.	Y	N	Y	Y (all abort)

Table 1: Supported combinations of event categories and coupling modes.

3.3 Event Life-Span

The next important issue when dealing with composite events is to determine how long to keep alive a partially composed event. We distinguish between composite events that are composed from primitive events originating in a single transaction and those that are composed across transactions. The life-span for composition of events that originate within a single transaction is the duration of a transaction. Once the transaction is either committed or aborted, the event composition is discarded. This avoids cluttering the system with semi-composed events. For events that are composed from primitive events that originate in different transactions we require a validity interval. This may be given either for the whole composite event, or it may be determined by the smallest validity interval of the composing events. Composite events without an explicit or implicit validity interval are illegal. Clearly defining the life-span of events enables to clean the system from semi-composed events in an efficient manner.

3.4 Event Consumption Policy

The third important issue when composing events is the event consumption policy. The problem arises from multiple instances of primitive events arriving at an event composer and the resulting ambiguity. Let us consider that the sequence $E_3 = (E_1; E_2)$ of primitive events E_1 and E_2 is being composed. The primitive event instances e_1, e_1', e_2 arrive in this order. Should the composer use e_1 or e_1' in the composition? This problem was addressed in SNOOP [CM91], defining four contexts: recent, chronicle, continuous, and cumulative. In recent mode, typical for sensor monitoring, the most recent occurrence of a primitive event is used in the composition. In chronicle mode, typically used in workflow applications, the primitive events are consumed in chronological order. In continuous mode, useful in financial applications, such as monitoring of the Dow Jones index, each occurrence of a primitive event opens a new window that stays open for a specified period. Finally, in cumulative context all occurrences are used up to the point where the composite event is raised. For details see [CM91]. We consider these consumption policies to be the best so far defined. Although complex, they all appear to be useful for some application. As a minimum, a system must support recent and chronological event consumption policies. These are the consumption policies currently supported in REACH. The issues of consumption policy and life-span of the event composition process are orthogonal.

4 Limitations of Layered Architectures

Because an aDBMS combines full database functionality with active capabilities the entry price one must pay in terms of infrastructure before any meaningful applied research can be done is extremely high. A layered implementation of active database capability *on top* of a commercial OODBMS appeared, at first sight, to be a reasonable compromise between functionality and effort. Giving up the efficiency of a tight coupling between the OODBMS and the rule manager, one could try to implement the active database capabilities on top of a closed commercial system hoping that the provided functionality would eventually be migrated into the system by its developers. We attempted this route using two well-known OODBMSs, O_2 and ObjectStore, for which we had licenses but no source code available, and encountered a series of limitations which would have resulted in too many sacrifices of functionality and prompted us to abort this course of action in favor of an integrated approach. Here we summarize the main problems.

Event detection: Method invocations are basic database events in an OODBMS . To detect method-calls it becomes necessary to wrap the methods and modify the dispatcher to signal the pertinent events whenever the method is called. Implementing the detection of method events in a closed OODBMS is difficult at best. It requires redefinition of all the classes for which method invocations generate events. This

results in a parallel class hierarchy of active classes that must be maintained by the application programmer. Since active classes also require system-provided classes, it requires the reimplementation of many of these classes as well. Alternately, each single method-body must be modified to signal invocation and return. None of these alternatives is satisfactory since it either imposes a large overhead or forces applications to announce the events. More details on this issue are given in Section 6.

Another major problem results from the distinction between values and objects made by some object models. Smalltalk, for example, supports strict encapsulation. Therefore, all the interaction occurs through methods. C++ and other object models being used in OODBMSs include state in their public interface. While objects are modified through methods only, value changes are detected through low-level system functions that could not be modified by us. Thus, changes of state could not be detected as events.

Transaction model and manager: The implementation of the coupling modes that we identified as essential to satisfy the more demanding applications requires a flexible transaction model. For parallel rule execution it is necessary to have a nested transaction model. Without a nested transaction model only serial execution of triggered rules is possible in the immediate and deferred modes. One of the commercial systems we attempted to use only provides flat transactions, while the other does provide closed nested transactions. For the implementation of the detached and the detached causally dependent modes (parallel, sequential, and exclusive) it becomes necessary to spawn new top-level transactions. All four modes require the forking off of a new transaction, which caused problems with one OODBMS's license manager. Furthermore, to enforce the dependencies in the three detached causally dependent modes, it is necessary to have access to information handled by the transaction manager, such as transaction identifier, commit and abort signals, and in the case of the exclusive causally dependent mode it becomes necessary to transfer resources from one transaction to the other once it is determined that the spawning transaction is to be aborted. Neither of the commercial OODBMSs we used provided us with the necessary access to transaction-manager information nor did they allow us to gain control as needed, or to redefine the commit and abort methods.

Persistence model: The persistence model used by an OODBMS has a major impact on the coupling mode in which rules can be executed upon deletion of an object. In the case of O_2 that implements persistence by reachability without an explicit delete we encountered serious problems in triggering rules on delete of an object. This problem, although not insurmountable, requires much additional information to be carried (externally) to support the firing of deletion-triggered rules. In the case of persistent C++ systems this problem does not exist since invocation of the destructor methods can be detected by the event detector (provided the issues discussed under detection of method-events have been solved).

Closed environments: When implementing the rule system on top of a closed OODBMS one is forced, in many cases, to work in the OODBMS's proprietary programming and run-time environment. The closed nature of the O_2 programming environment caused major problems, since access to the operating system level was not possible whenever we needed it.

We were initially encouraged by UBILAB [KOT93], a prototype that was implemented in a layered architecture in a Smalltalk environment. However, this implementation has a reduced functionality in terms of events, execution model and performance and its architecture hinges on some features that are Smalltalk-specific. The kind of applications that we want to address, the compatibility with existing applications and the performance requirements push us to a C++ object model. None of the commercial OODBMSs we have access to provides us with the necessary features and access to internal levels that is required for robust and efficient implementation of the full range of active capabilities. We have therefore turned to Texas Instruments' Open OODB, an extensible C++ based OODBMS as the basic platform we want to extend with our active DBMS capabilities in an integrated architecture.

5 The Open OODB Platform

Texas Instruments' Open OODB system [WBT92] is based on an open, extensible architecture for constructing a family of DBMSs. By abstracting orthogonal dimensions of database functionality, the Open OODB system allows tailoring each of these dimensions for particular applications within a common, incrementally improvable framework that can serve as a common platform for research.

The Open OODB system is based on a computational model that transparently extends the behavior of operations in application programming languages [WBT92]. Invocations of these operations are examples of primitive method events. Open OODB uses the type systems of conventional object-oriented programming languages, currently C++ and Common Lisp, as

alternative data models for applications. The computational model is realized by a meta-architecture module that implements the concepts of *event*, *sentry*, and *policy manager interface*.

Figure 1 illustrates the architecture of the Open OODB. The meta-architecture module provides the extensibility mechanisms in Open OODB. It plays the role of a "software bus" on which database components (policy managers) can be plugged in. This meta-architecture module is philosophically close to the active database paradigm. An *event* is the entry point to extensibility in Open OODB. Any operation performed within the context of a programming language can be an event. Object dereference and method invocation are two operations whose behavior is often extended to enable persistence, distribution, or access control support. A mechanism, the *sentry*, tracks primitive events and invokes (activates) the appropriate *policy manager* (PM) which implements the extended behavior. Different sentries (see Section 6) may invoke a variety of policy managers, e.g. a Persistence PM, a Transaction PM, a Distribution PM, a Change PM, and an Indexing PM. There must be at least one policy manager per database function, but one of the interesting features of this architecture is the possibility of exchanging or adding new policy managers and thus evolve and extend the system. A Nested Transactions PM and a Rule PM to support active database capabilities can be added. The meta architecture also contains a collection of support modules including *address space manager* (ASM), *communications*, *translation*, and *data dictionary*. ASMs may be passive or active. A passive ASM is simply a data repository (e.g., a file system). An active ASM allows computation which, in an object-oriented environment, is essential to the execution of methods. In an Open OODB system configuration, at least one ASM must be active. If more than one address space exists, there must be *communications* and *translation* mechanisms to effect object transfer between them. A *data dictionary* serves as a globally known repository of system, object, name, and type information.

Open OODB does not implement all its modules from scratch. The Exodus storage manager [CDR86] is used as an ASM for permanent storage of objects, and the Volcano optimizer generator to generate the query optimizer [BMG93]. For our purposes, the use of multiple threads, preferably on a multiprocessor platform, for event composition and rule firing in the active DBMS is essential. Therefore, we committed early to a Solaris 2.x platform. This required the porting of Exodus to the Solaris environment. Currently, a version of Open OODB runs on top of Solaris 2.3 and Exodus 2.2.

6 The ECA-Oriented Architecture

The efficiency of an aDBMS depends critically on the efficiency of event detection, event composition, and rule triggering after the event is raised. These mechanisms must be well integrated with the type system of the DBMS. At the same time they should be implemented in a modular way since there are many architecturally equivalent alternatives for each. Specifically, there are many ways to detect primitive events, each of which can be made compatible with any of a number of algebras. Similarly, depending on the predominant kind of rule and process topology of a particular system, different organizations of rules can be exploited by the rule dispatcher. Further, detection, composition, and triggering interact with other parts of the system in distinct ways. Event detection must be tightly integrated with the type system and operations of the programming language of the application being monitored or with the hardware platform on which it runs. By contrast, event composition is independent. Rule triggering must have some capability to execute operations, possibly spawning processes and transactions.

This section describes the need for orthogonality of event monitoring relative to the type system, the low-level primitive event detection mechanism used in Open OODB, the event composition and rule execution mechanisms of REACH, and their interplay.

6.1 Event monitoring orthogonal to type

The REACH system uses the C++ type system as its underlying data model. To illustrate how REACH rules are mapped onto C++ classes we use an environmental rule that must be enforced in power-plant operation. Whenever the water level of the river from which the cooling water is drawn reaches a lower mark and the water temperature is above a maximum temperature and the heat-load given off is above a threshold, then the Planned Power Output must be reduced by 5%:

```
#include "River.h"
#include "Reactor.h"
rule WaterLevel {
  prio   5;
  decl   River *river, int x, Reactor *reactor named "BlockA";
  event  after river->updateWaterLevel(x);
  cond   imm x < 37 and river->getWaterTemp() > 24.5
                   and reactor->getHeatOutput() > 1000000;
  action imm reactor->reducePlannedPower(0.05);
};
```

This rule is mapped onto one rule object and two C functions for condition evaluation and action execution. All those functions are archived in a shared

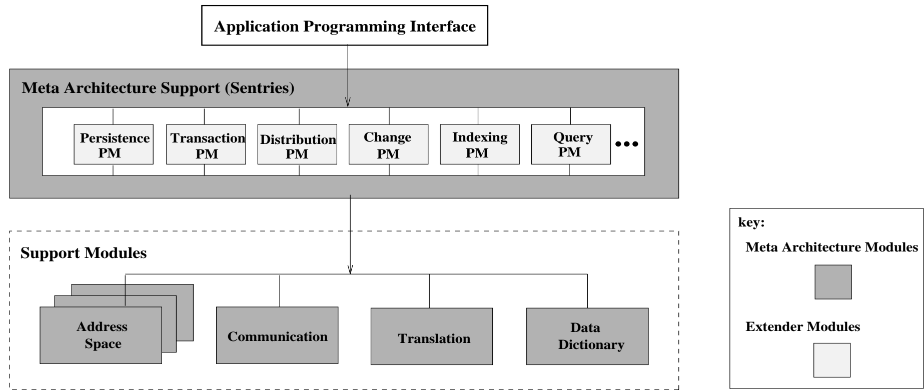

Figure 1: Open OODB Architecture.

library and are extracted using the naming convention that the rule's name is appended by "`Cond`" and "`Action`", respectively. The base class `Rule` contains methods `evalCond()` and `execAction()` which call the C functions belonging to the relevant rule object. Specialized rule classes for consistency management, replication management, and so forth can be derived from this base class.

```
#include "Reach.h"
#include "River.h"
#include "Reactor.h"
extern "C" boolean_t WaterLevelCond (void **args) {
   River        *river = (River*) args[0];
   int           x = (int) *args[1];
   Reactor   *reactor = (Reactor*) OpenOODB->fetch("Block A");
   boolean_t  rc;
   if (x < 37 && river->getWaterTemp() > 24.5 &&
       reactor->getHeatOutput() > 1000000) rc = B_TRUE;
   else rc = B_FALSE;
   args[0] = (void*) reactor;      // reorganize argument list
   args[1] = (void*) NULL;
   return rc;
}
```

This example reveals several requirements when extending an existing programming language type system with active capabilities:

- Rich data types must be sentried. The sentry mechanism must be able to monitor instances of C++ classes, structs, arrays, and unions uniformly, since irregularities make systems very difficult to use properly.
- Monitoring of events must be possible regardless of other object properties such as persistence, distribution, or versioning.
- Trapping member function invocation must be supported. We require that C++ virtual member functions of classes, and class and struct accessors be trappable.

There are several places where a poorly designed sentry could become apparent. It is better to "transparently" detect events than to force applications to "announce" them, since the former does not clutter a program or force changes in existing code. Further, it is not always known in advance which events may be of interest to a particular Open OODB extension or aDBMS rule set. Therefore, we require that:

- Type declarations for monitored types must be the same as for the equivalent unmonitored types.
- Object manipulation and function invocation must be the same for monitored types and for the equivalent unmonitored types. The mechanism must support all C++ pointer conversions, accessibility of state variables (public, private or protected), inheritance hierarchy including multiple inheritance and base class information, friend functions/classes, copy constructors and assignment operators, constructor/destructor functions, and function properties such as virtual, non-virtual, and static.

6.2 Efficient Detection of Primitive Events

Detection of events is performed by the Open OODB sentry mechanism which is used internally by Open OODB as described in Section 5. Because Open OODB relies heavily on sentries, both in support of applications and for its own internal activities, performance overhead associated with sentries is critical. The same holds for active DBMSs. There are three categories of overhead associated with sentries: *useful overhead* is caused by a sentry that must always trigger an extension; *useless overhead* is caused by a sentry that will never trigger an extension; *potentially useful overhead* is caused by a sentry that is not currently triggering an extension but might in the future. Ideally, useless overhead should be avoided. This can be achieved by filtering low-level events and aggregating them before invoking a rule or policy manager.

Many sentry-like mechanisms exist in a variety of domains. Examples are hardware interrupts, illegal type tags, virtual memory traps, language mechanisms and redefinition of function dispatch, traps defined in root classes, and in-line wrappers. The goal is finding a sentry mechanism that is applicable in a wide variety of situations and adopting it as the prime sentry mechanism. However, the architecture must allow the use of other sentry mechanisms for special purposes.

Hardware interrupts and *illegal type tags* can be used to interrupt the flow of machine instructions. Such interrupts are machine-specific, only cover a small category of possible events and are defined at a different level of abstraction (machine language instructions) than the application programs to be extended, which are written in C++.

Virtual memory traps [LLOW91] allow objects to be transparently retrieved. This technique, while fast, has three drawbacks: it is hardware-specific; it cannot trap function invocation, precluding its use for several kinds of extensions; and it is dependent upon the physical memory layout of data.

Programming languages like Lisp and its extension, the Common Lisp Object System (CLOS), provide *language mechanisms* to dynamically modify the behavior of certain functions associated with particular classes of objects. The CLOS Meta Object Protocol, which allows *redefinition of function dispatch*, can be used to ensure that any declared extensions are executed as part of "normal" dispatch. In C++, it is possible [BCL93] for different instances of a class to have distinct virtual function dispatch tables (instead of the usual one per class), each of which can be independently populated with modified virtual functions that extend or change behavior. This is compiler-dependent, and cannot trap memory access, which precludes its use to ensure residency of objects whose state is accessed without using a virtual function; however, it does appear to have excellent properties for other kinds of language extensions. Managing a wide variety of such tables can make the application's code size increase dramatically.

Traps defined in a root class(es) can be inherited by application classes. Unfortunately, this changes simple single inheritance into more complex multiple inheritance, which, while supported by C++, is idiosyncratic in definition and implementation, thus making certain operations that are legal under single inheritance not work under multiple inheritance. A surrogate object stands in for some other object that may or may not be present, intercepts all messages directed at the actual object, and performs any necessary actions before forwarding the original message to the actual object for execution (possibly including instantiating the object by retrieving it from a database). Since in C++ the state of an object can be manipulated without using a member function, it is possible to affect the object without activating the sentry, a semantic error that would cause the behavioral extensions to be omitted.

Open OODB implements, for C++, a sentry via *in-line wrappers*, that treats all extendible classes as *logically* inheriting from a *conceptual* base class of `extendible-objects`. This conceptual base class defines the structures and functions necessary to make all classes inheriting from it sentried. Unlike the use of normal inheritance to implement sentries, the class `extendible-objects` is never seen by a C++ compiler. Rather, all classes logically inheriting it are modified by a language preprocessor to insert the defined structures and functionality into each actual class. These *augmented* classes are then compiled as usual. By placing the sentrying code before the compiler sees the application, it is possible to avoid artificially created multiple inheritance, and to use normal C++ compilers to ensure that pointer conversion, base class offsets, friend functions, etc. work properly. They work properly because, as far as the compiler is concerned, it is compiling a normal C++ program, not an augmented one. The generation process takes a C++ program as input and preprocesses it to collect type information and to generate an equivalent program with extensions added.

Preliminary performance figures for the implemented sentry mechanism in Open OODB comparing useful overhead, useless overhead, and execution of unmonitored types is reported in [WSTR93].

6.3 Efficient Event Composition

The clue for an efficient event management is to keep event composition simple and to execute it in parallel. We believe that large, monolithic event managers that are based on a single graph should be avoided. Instead, many small compositors that can be executed by parallel threads should be supported. This approach makes the garbage-collection of semi-composed events much simpler. Since in REACH the life-span of composite events is well-defined, when the life-span of a semi-composed event elapses, the whole composition graph instance for that event occurrence is simply removed. Many small event composers are also a necessary step toward distributed event detection/composition.

Event hierarchy: Each ECA-manager for a primitive event type contains two kinds of information: the

rules that are directly fired by the primitive event and the composite events that contain the primitive event. Primitive events are thus passed first to the rules for firing and then to other ECA-managers for composition of the corresponding composite events. ECA-managers know what parameters must be passed with a primitive event, such as OID of the object to be acted upon, transaction-id, timestamp, and other attributes that can be taken from the method invocation message. The collection of all the ECA-managers serves as a repository of event specifications.

Event history: ECA-managers create an event object and keep local histories of the created event occurrences. The maintenance of a highly distributed history eliminates the bottleneck that would result from centrally logging the occurrence of events. The price one pays for the distributed histories is an overhead when the effects of a rule must be compensated. Therefore, a global history is maintained by a background process after a transaction has committed or has been aborted.

6.4 Efficient Rule Firing

To make rule firing efficient the crucial part is to minimize the search for the rule that is to be fired, reduce the levels of indirection (messages) needed between the point of event detection and the firing of the rule, and to eliminate unnecessary delays in the firing of rules triggered by simple events due to the presence of rules waiting for the composition of complex events.

To provide an efficient and highly selective rule firing mechanism, we use the ECA-managers. ECA-managers are dedicated to a given event type. Therefore, they know which set of rules is fired by an event. If a rule can be triggered by a simple event, the ECA-manager passes the event and fires the rule. The coupling mode between a rule and the user-submitted transaction is specified as part of the rule but the ECA-manager is aware of what rules are fired and in what mode. If a primitive event is part of a composite event, the primitive event is passed along to the corresponding event composer. To eliminate a major performance-killer we decided that only rules that are fired by primitive events can be executed in an immediate coupling mode, because otherwise the execution of a program must be halted until it is clear that no composite event that could fire an immediately-coupled rule will be completed by the detected primitive method event. This wait for negative acknowledgements is not acceptable. In the case of the rules that are triggered by the primitive events, the primitive event ECA-manager knows whether a rule must be executed in an immediate coupling mode and can give the application program the go-ahead. Rules that are fired by composite events can be executed in the coupling modes deferred, detached, or detached with causal dependencies.

Figure 2 shows schematically the flow of information between the actual primitive event detectors (implicitly sentried), the corresponding primitive and composite ECA-managers, and the rule objects. Arrows represent messages. For example, a method-event is detected and a message is sent to the corresponding ECA-manager. This manager produces the corresponding event object, passes it to the rules that are fired by it (if any) and to the composite ECA-managers (if any). The primitive ECA-manager sends a message to the execution engine as soon as it is clear that no immediately-coupled rules are fired.

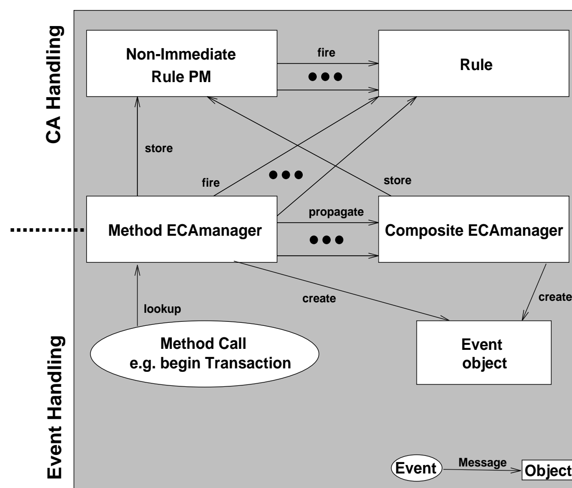

Figure 2: ECA-oriented architecture (method part).

When rules are fired, there are two situations in which the system must deal with parallelism: a) when multiple rules are fired by the same event, and b) when multiple rules are fired by different events and are to be executed in deferred mode.

In REACH we aim at providing parallel rule execution. In this case, the triggered rules can execute as sibling subtransactions. However, at present, Open-OODB does not provide nested transactions. Therefore, we are providing in the first prototype a mechanism for mapping a set of rules that could potentially be executed in parallel to an ordered firing-sequence. This decision, taken out of necessity to proceed with the implementation while the transaction model is being extended, has the advantage that we will be able to perform actual measurements comparing the gain of parallel rule execution with the overhead incurred for setting up the parallel subtransactions. It

also enables us to provide the best execution strategy depending on the actual requirements of an application. At present, the issues of termination, the need for determinism in rule execution and the development of correctness criteria, such as confluence for rules in an object-oriented environment are still evolving [AWH92], [BCW93], [VS93]. Tools for testing and debugging are just emerging [DJ93]. Therefore, we want to provide the possibility of enforcing different rule execution strategies.

When multiple rules are fired by a single event, in the absence of nested transactions, the ECA-manager must determine the order in which they are to be fired. Since we do not consider rules fired by a composite event in immediate coupling mode, only one ECA-manager is involved at a time and control resides with it. Rules can be prioritized and the ECA-manager will execute them in this order. Even if the rules are conceptually fired in parallel or no priorities have been defined, an ordering is required at a lower level for creating child processes and the initialization of Solaris threads. In the absence of priorities or in the case of a tie, the ECA-manager uses the rule's timestamp to enforce an oldest rule first (default) or a newest rule first (optional) tie-break policy.

When multiple rules are fired in deferred mode by many different events, the control over the execution cannot any longer reside with a single ECA-manager. Instead, the control now resides with the transaction policy manager who knows that at commit-time the deferred rules can be executed. Again, priorities can be enforced as the main ordering criterion, but in addition to the previous two tie-breaking policies a third policy that fires rules with simple events ahead of rules with complex events can be enforced.

7 Conclusions and Future Work

We presented the architecture of REACH, a tightly integrated active OODBMS. We focused on a clean integration of database management and active capabilities. To this end we extended TI's Open OODB, an extensible DBMS that supports low-level event detection as the basic mechanisms for providing extensibility. The main contributions of this paper are:

- A characterization of the requirements of active OODBMSs.
- A description of the crucial issues of flexible and efficient event detection, composition, and rule processing.
- A clarification of the semantics of event composition relative to transaction execution.
- An experience report summarizing the pitfalls we encountered while trying to build an active OODBMS on top of two closed commercial database management systems.
- A clean integration of the Open OODB sentry mechanism with the REACH ECA managers. This is the core architectural component of a tightly integrated active OODBMS which we expect to become a stable platform for future research.

The REACH active OODBMS is being built at the Technical University of Darmstadt. A first prototype became operational in August 1994 based on Open OODB's alpha release 0.2. Open OODB is still evolving, hence, to start working on the development of the active capabilities we had to make some design decisions that accommodate some of the missing features, such as nested transactions. We are developing a nested transaction model for Open OODB which will provide the parallel execution of rules that is part of the execution model, and enable us to obtain actual performance results for sequential and parallel rule executions. Ongoing work is concerned with efficient event composition comparing different strategies, with efficient garbage-collection of semi-composed events, performance measurement and the implementation of a GUI for rule definition and management. We plan on further developing Open OODB's intrinsic event-orientation to express other system properties such as index maintenance PMs with the active database paradigm. This includes extending the set of sentry mechanisms to include virtual memory traps. Another area of great interest is the combination of the ECA-rule description with Open OODB's query language, OQL[C++]. At the application level, different applications will be tested once an operational platform is available. Feedback from these applications will drive further development of the REACH prototype, particularly the user interface and timing constraints.

References

[AWH92] Aiken, A., Widom, J., Hellerstein, J.M.; Behavior of database production rules: termination, confluence, and observable determinism, Proc. ACM SIGMOD 1992.

[BCW93] Baralis, E., Ceri, S., Widom, J.; Better Termination Analysis for Active Databases, in [PW93].

[BMG93] Blakeley, J. A., McKenna, W. J., Graefe, G.; Experiences Building the Open OODB Query Optimizer. Proc. ACM SIGMOD, 1993.

[BCL93] Bracha, G., Clark, C. F., Lindstrom, G., and Orr, D. B. Modules as values in a persistent object store. Dept. of CS, University of Utah.

[BBK93] Branding, H., Buchmann, A., Kudrass, T., Zimmermann, J.; Rules in an Open System: The REACH Rule System, in [PW93].

[BBK92] Buchmann, A., Branding, H., Kudrass, T., Zimmermann, J.; REACH: a REal-time ACtive and Heterogeneous mediator system, Bulletin of the TC on Database Engineering, Vol. 15, Dec. 1992.

[CDR86] Carey, M.J., DeWitt, D.J., Richardson, J.E., Shekita, E.J.; Object and File Management in the EXODUS Extensible Database System, Proc. 12th VLDB, 1986

[CW91] Ceri, S., Widom, J.; Deriving Production Rules for Incremental View Maintenance. Proc 17th VLDB, 1991.

[CBB89] Chakravarthy, S., Blaustein, B., Buchmann, A., Carey, M., Dayal, U., Goldhirsch, D., Hsu, M., Jauhari, R., Ladin, R., Livny, M., McCarthy, D., McKee, R., Rosenthal, A.; HiPAC: A Research Project in Active, Time-Constrained Database Management. Final Tech Report, XAIT, July 1989.

[CM91] Chakravarthy, S., Mishra, D. An Event Specification Language (SNOOP) for Active Databases and its Detection. TR-91-23, U. Florida, 1991.

[CKA93] Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.-K; Anatomy of a Composite Event Detector. TR-93-039, U. Florida, 1993.

[DBB88] Dayal, U., Blaustein, B., Buchmann, A., Chakravarthy, S. , Goldhirsch, D., Hsu, M., Ladin, R., McCarthy, D., Rosenthal, A.; The HiPAC Project: Combining Active Databases and Timing Constraints, SIGMOD RECORD, 17(1), March 1988.

[DBM88] Dayal, U., Buchmann, A., McCarthy, D. Rules are Objects Too: A Knowledge Model for an Active Object-Oriented Database System, 2nd Workshop on OODB, Bad Münster, Germany, 1988

[Deu94] Deutsch, A. Detection of Method and Composite Events in the Active DBMS REACH, Tech. University Darmstadt, M.S. Thesis, 1994.

[DJ93] Diaz, O., Jaime, A.; DEAR: A DEbugger for Active Rules in an O-O Context, in [PW93].

[DPG91] Diaz, O., Paton, N.W, Gray, P. Rule Management in Object-Oriented Databases: A Uniform Approach, Proc. 17th VLDB, 1991.

[GD93a] Gatziu, S., Dittrich, K.R.; Eine Ereignissprache für das aktive, objektorientierte Datenbanksystem SAMOS, Proc. BTW, Germany, 1993.

[GD93b] Gatziu, S., Dittrich, K.R. Events in an Active Object-Oriented Database System, in [PW93].

[GJ91] Gehani, N., Jagadish, H.V.; ODE as an Active Database: Constraints and Triggers, Proc. 17th VLDB, 1991.

[GJS92] Gehani, N., Jagadish, H.V., Shmueli, O. Composite Event Specification in Active Databases: Model & Implementation, Proc. 18th VLDB, 1992.

[Han92] Hanson, E. N.; Rule Condition testing and Action Execution in Ariel, ACM SIGMOD, 1992.

[HLM88] Hsu, M., Ladin, R., McCarthy, D.; An Execution Model for Active Database Management Systems, Proc. 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem, 1988.

[KOT93] Kotz-Dittrich A.; Adding Active Functionality to an Object-Oriented Database System - a Layered Approach, Proc. BTW, Germany, 1993.

[LLOW91] Lamb, C., Landis, G., Orenstein, J., Weinreb, D.; The ObjectStore Database System. CACM 34(10), Oct. 1991.

[OODB93] Open OODB Project. Open OODB Toolkit: Release 0.2 (Alpha). Texas Instruments, Inc., Sept. 1993.

[PW93] Paton, N., Williams, M. (Eds.); Rules in Database Systems, Proc. 1st Int. Workshop on Rules in Database Systems, Edinburgh, 1993.

[SHP88] Stonebraker, M., Hanson, E., Potamianos, S.; The POSTGRES Rule Manager, IEEE TSE, Vol 14, No. 7, July 1988.

[SHP89] Stonebraker, M., Hearst, M., Potamianos,S. A Commentary on the POSTGRES Rules System, SIGMOD RECORD, Vol 18, No. 3, Sept., 1989.

[VS93] van der Voort, M.H., Sibes, A.; Enforcing Confluence of Rule Execution, in [PW93].

[Wel91] Wells, D. L. ARPA Open Object-Oriented Database Meta Architecture Support Module Specification. TR Vers 6, ARPA Open OODB Project, CSL, Texas Instruments, Inc., Nov. 1991.

[WBT92] Wells, D. L., Blakeley, J. A., Thompson, C. W.; Architecture of an Open Object-Oriented Database Management System. Computer 25(10), Oct. 1992.

[WSTR93] Wells, D. L., Srivastava, A., Thompson, C. W., Ramey, J.; A Mechanism for Extending and Evolving C++ Programs. TR, ARPA Open OODB Project, CSL, Texas Instruments, Inc., Oct. 1993.

[WB94] Wells, D. L., Blakeley, J. A.; Distribution and Persistence in the Open Object-Oriented Database System. In Distributed Object Management, M. T. Özsu, U. Dayal, and P. Valduriez (Eds.), Morgan Kaufmann, 1994.

[WF90] Widom, J., Finkelstein, S.J. Set-Oriental Production Rules in Relational Database Systems, Proc. ACM SIGMOD, 1990.