2

Architecture of Active Database Systems

Alejandro P. Buchmann

ABSTRACT
The architecture of an active DBMS determines both its functionality and the components that are required for its implementation. This chapter addresses first some issues that have an impact on the architecture of an active database system, and presents the various architectural alternatives. The basic functions of an ADBMS are identified and then related to the architectural alternatives. This chapter discusses rule specification and registration, and rule execution. Special attention is devoted to the execution of transactions and its relationship to the execution of ECA rules.

2.1 Introduction

The architecture of an active DBMS determines both its functionality and the components that are required to realize it. Since an ADBMS by definition must provide the active capabilities in addition to full DBMS features, it can be viewed as an extension of a passive DBMS. There are various properties of the underlying DBMS and of the architectural strategy used for implementing the active extensions that will have an impact on the functionality and the performance of the active DBMS. The major dimensions that need to be considered are:

- the degree of integration between the underlying DBMS and the active capabilities,
- the system architecture of the underlying DBMS, and
- the data model of the DBMS and the programming language used for the active extensions.

In this chapter we will briefly discuss the various architectural alternatives. We will then identify the basic functions and components of an ADBMS and will relate them to the various architectural alternatives and will point out the effect of an architecture on the implementation of a given feature. We will address first the rule specification and registration functionality and then the rule execution subsystem. A central aspect of rule

execution is the integration of the active functionality with the transaction manager of the underlying DBMS. Therefore, special attention is given to the transaction model supported by the underlying DBMS and how the execution of ECA rules is integrated with the execution of transactions. Since existing products and prototypes of active systems are mostly centralized, the discussion will concentrate on centralized active database systems.

2.2 Degree of Integration

An important dimension along which architectures can be distinguished is the degree of integration between the underlying DBMS and the active functionality. The two extremes along this line are a layered architecture in which the active components of the ADBMS are built on top of the existing DBMS in a user process, and a fully integrated architecture that tightly couples the components providing the active features with the rest of the DBMS. Depending on the degree of integration, some of the active capabilities may not be implementable or may be implemented only in a very inefficient manner.

2.2.1 Layered Architecture

Layered architectures are popular because they allow the implementation of the active functionality *on top* of an existing system with little or no modification to the underlying database management system. Event detection and rule execution is done separately from the underlying DBMS and typically on the client-side in the user's address space. Application programs are often preprocessed and modified in such a way that events can be appropriately detected. Since communication between application and DBMS is always via the client, it is relatively easy to generate the events on the client-side when a method is executed. However, the server must communicate transaction events that are required by the active component. Further, the server generally is not aware of the active component and is not prepared to signal the necessary events. Most commercial DBMS's do not allow access to or direct communication with internal components such as the transaction manager, the lock manager, and the access control module. In most cases the interfaces to these components are not laid open. Some commercial systems go as far as isolating the user processes in a proprietary programming environment curtailing the ability to make system calls. In addition, some basic functions that are already performed by the underlying DBMS, such as logging, must be reimplemented in the user space. Other extensions, such as nested transactions, may not be implementable in a layered architecture.

The advantage of low cost extensibility and the fact that the active ex-

tensions can sometimes be used on different DBMS's is balanced by the limitation of functionality derived from the lack of open interfaces of the underlying DBMS and its internal components, and a strong performance penalty.

2.2.2 Integrated Architecture

In a fully integrated architecture, the active functionality is embedded in the basic components of the DBMS. While some of the extensions may be realized on the user side, others must be implemented by extending the basic DBMS functionality. The advantages of a fully integrated architecture are most evident when dealing with the transaction manager, the concurrency control and rollback mechanisms. Important portions of the DBMS that must be adapted when extending a DBMS with active functionality are the dispatch mechanism, the lock manager, and the commit and abort processes. To modify these components it is necessary to have access to the source code of the underlying DBMS. Unfortunately, sources of full-fledged OODBMS's are not readily available to research groups, which has slowed progress in the development of active OODBMS's. Extensions to commercial products have occurred mostly within companies or through partnerships between a research group and a company. An example of this route is the NAOS project and its cooperation with O2 [CCS94]. The use of available OODBMS research prototypes, such as Texas Instruments' OpenOODB [WBT92], is a compromise. OpenOODB is built as a client to the Exodus Storage Server [CDRS86]. While the source code for both systems is made available to research groups, some of the problems remain, since the OODBMS runs as a client to the Exodus server and modifications to the Exodus transaction manager are not trivial. Some of these problems are discussed in the next section.

The advantage of fully integrating the active capabilities with the DBMS lies both in the broader range of functionality that can be provided and in the potential performance gains. The main drawback is the high entry price when building from scratch or the lack of availability of a stable platform to research groups.

2.3 Client-Server Architecture of the Underlying DBMS

All active database systems we know of are implemented as extensions to a DBMS based on the client-server principle.

Relational systems are usually implemented as fat servers, meaning that event detection and trigger execution is essentially carried out on the server side. The old and/or new states of the database on which a trigger operates

are typically handled in the form of delta-relations. Because tuple identification in the relational model is done strictly by value, passing of tuples as parameters is easy.

Object systems are typically distinguished based on their architecture as page-server or object-server systems. The impact of underlying system architecture can be seen clearly when we compare object-server and page-server architectures for object-oriented DBMS's.

In a page-server architecture, the server manages I/O, page buffering, and page locks. The data transfer unit between server and client is a page. The client is responsible for unpacking the page and accessing individual objects. This means that the server does not know how to interpret objects, nor how to execute methods, and object-level locking is not provided by the server. Therefore, most of the active functionality, such as event detection and composition, must be realized on the client-side. On the other hand, transaction control is clearly located on the server-side. Transaction commit and abort is controlled by the server and the corresponding transaction events are generated by the server.

Object servers, on the other hand, unpack pages and do understand objects and can execute their methods. Their transport unit is an object. They can provide either page or object-level locking, which simplifies the implementation of nested transactions. Because the server can execute methods, method events may be generated both at the client-side and the server-side, and event handling and rule execution could be executed on either side. In general, more degrees of freedom exist in an object-server architecture.

Most active OODBMS prototypes we are aware of have been built as extensions to a page-server architecture, either directly at the user level on a commercial OODBMS or as extensions to a prototype OODBMS, such as Texas Instruments' OpenOODB. In the latter case, OpenOODB runs as a client process of Exodus and an application must be bound to an instantiation of OpenOODB. Different applications run on different instances of OpenOODB, thus having different address spaces. This makes the passing of object references impossible and requires that parameter passing be done strictly by value across applications.

2.4 Data Model and Programming Language Issues

The main difference between the relational and an object model when dealing with active capabilities lies in the variety of the events that may have to be detected. In a relational system, the triggering events are usually limited to insert, delete, and update operations with a few systems providing also read access as an event. Therefore, event detection on these predefined operations can be hard-wired into the system. In an object model in which the user may define new classes with arbitrary methods, every method can rep-

resent an event and must be detected. The problem is compounded because of inheritance and the issues of encapsulation. Object-oriented systems and their languages have more complex scoping rules than relational systems.

A major difference with respect to rule definition exists between interpreted and compiled languages. The addition of a new rule is trivial in an interpreted environment, such as Smalltalk, but is a major problem in a statically compiled environment, such as C++. In a C++ environment, relinking is at least required, thus making fully dynamic addition of rules almost impossible. We will pick up these issues as we discuss the individual components of an ADBMS architecture.

2.5 Rule Specification and Registration

Any ADBMS needs some capability for describing rules and registering them with the system. In this section we identify the basic functions involved in the specification of rules and their registration with the system, and discuss the components needed to implement this functionality. The important issues from an architectural point of view are:

- the model and language used,
- the time and method of creation and modification of rules,
- the process of subscription to events, and
- the handling of privileges associated with the definition of rules.

We distinguish here between the functions and components that are part of the DBMS and those that are part of design tools. A function that is typically associated with aDB design tools is the testing for correctness of a rule set, specifically static tests for termination and confluence. We do not address those. However, since new rules that are added may conflict with the current database state, we analyze the implications in the context of rule creation and modification privileges. Some ADBMS's include checking capabilities that include the current state of the database as part of the rule specification and registration subsystem.

Rule definition: ECA rules are defined through a rule specification language that is an extension of the schema definition language. Depending on the data model used, the rule specification language is typically an extension of SQL in the relational case or an extension to a (persistent) OO-programming language. The corresponding precompiler must be provided.

In the case of SQL, ECA rules are specified as triggers that are defined on a specific relation. Triggers are compiled with the corresponding relation and registered in the catalogue. In an object model, the issues are more diverse. Depending on the actual object model used, it may either be possible

to define rules as instances of a single class `Rule`, or the underlying model may in effect require the creation of a single-instance class for each new rule. The implication from an operational point of view will be that the definition of a new ECA rule in the latter case requires the recompilation of the schema. A commonly used subterfuge that has an impact on system architecture is to break up a rule into simple C-functions that are placed into libraries that only need to be relinked when new rules are added.

Rule registration: The registration of a new rule with the ADBMS also implies a series of administrative processes for which the necessary infrastructure must be provided. When a new rule is registered with the system, the event definition must be extracted and the corresponding event handler must be initialized. If the triggering event is an event already known to the system, then the new rule must subscribe to that event. If the event was not previously known to the ADBMS as a triggering event, the corresponding mechanisms for event detection/composition must be initialized. Initialization implies the creation of the necessary event detector/composer structures as well as the necessary metadata entries about coupling-modes, priorities, and activation/deactivation of rules. The initialization process may require changes to the application, for example, to include new wrappers, unless a very flexible approach for the detection of base events is provided. This is discussed in more detail in section 6.1 on primitive event detection.

Privileges: Last but not least are the architectural issues related to privileges associated with rule creation and manipulation. As much as we would like to view rules as any other object, they are metadata that can modify the behavior of other objects. This can range from the addition of new behaviors (e.g., an alarm when a threshold value is exceeded), to substitution of behaviors (e.g., by using the instead modifier provided by several ADBMS's), or (unauthorized) tracking of data usage, and activation/deactivation of consistency constraints. All these are effects that require careful consideration of how and to whom the privilege of rule creation and modification is granted and what support structures must be provided by the ADBMS to make rule definition safe. We illustrate this point by analyzing the effects of creation, activation, and deactivation of constraints modeled by ECA rules. If consistency constraints may be added dynamically or consistency-related rules may be deactivated and reactivated, the ADBMS must provide the mechanisms for validating existing data against the new rule and taking the proper action if a conflict arises between existing data and the new rule. Such action could be the time-stamping of data and rules to be able to reconstruct valid states, the elimination of instances that do not conform to the new constraint, or the flagging of these instances to request human intervention. Some of these actions have a major impact upon system architecture.

Figure 2.1 shows schematically the architecture of the rule registration portion of an ADBMS.

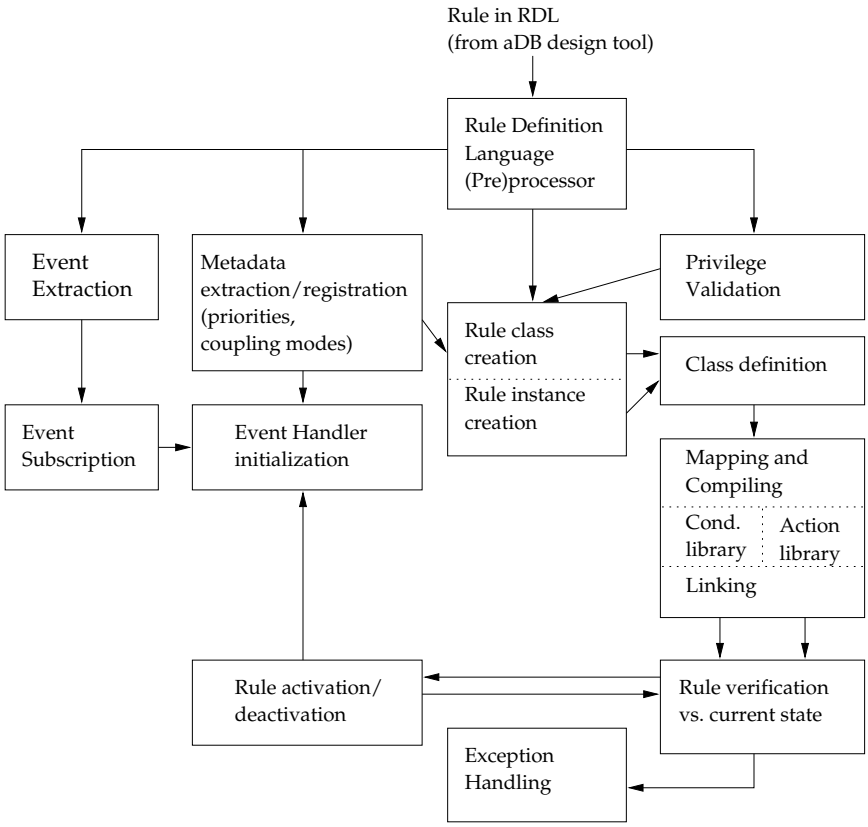

FIGURE 2.1. Rule registration portion of an ADBMS architecture.

2.6 Rule Processing

Once rules have been defined and registered, the rule execution component takes over. The rule execution component of an ADBMS is responsible for

- primitive event detection,
- composition of events,
- signaling of events,
- scheduling of rule execution,
- processing rules and synchronizing them with the execution of user transactions, and
- recovery and garbage collection of events.

Figure 2.2 shows schematically a basic architecture for the run time component of an ADBMS. We will expand this basic architecture in the sequel.

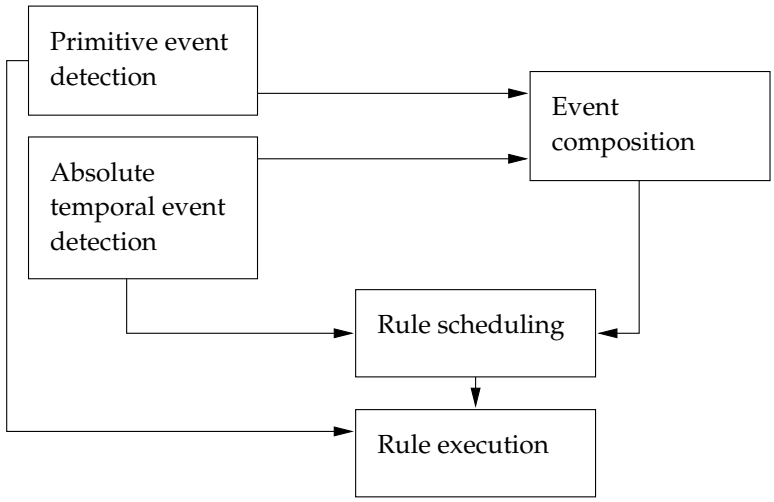

FIGURE 2.2. Schematic representation of the run time component architecture.

2.6.1 Primitive Event Detector Architecture

Detection of primitive events is a basic function in an ADBMS and must be realized in a very efficient manner. How primitive event detection is implemented will have an impact on the overall flexibility of an ADBMS and on the rule registration process. Depending on the underlying model, different primitive events are typically identified.

Relational systems generally handle the modification events *insert*, *delete*, and *update*. Object models consider method invocation as the basic primitive event. To account for the duration of a method execution, modifiers *before* and *after* have been introduced, meaning that a rule subscribing to a method event should execute either before the method is invoked or after it returns. Depending on the particular object model used, state changes may be considered. State change events are typically introduced when an object model distinguishes between attributes that are modified through methods and values that are manipulated through generic accessor functions. Temporal events may be absolute or relative. Absolute temporal events are considered primitive events that are signaled by the system clock. Another class of primitive events that deserves special attention from the architectural point of view comprises the transaction events, such as *begin of transaction*, *end of transaction*, *commit*, and *abort*. Some ADBMS's also

provide explicit external events. We will discuss each class briefly for its architectural implications.

A variety of mechanisms exist to detect primitive method events (we will subsume *insert*, *delete*, and *update* events). Early implementations used an existing DBMS-component, such as the lock manager, to detect update events. The disadvantage of such an approach is that the signaling of a primitive event is tied to another property of the data, such as persistence. Monitoring of primitive events should be independent of other properties of an object. Expressed differently, we want to be able to detect a primitive event, such as a method event, independently of an object being persistent or transient, being a system-object or a user-defined object. This property is known as orthogonality of monitoring and type [BZBW95].

Among the mechanisms that have been proposed for method-event detection, the most popular is method wrapping. Method wrapping consists in bracketing a method with a begin-method and an end-method signal. Depending on whether a rule exists that subscribes to this method event (either before or after), the corresponding primitive method-event is signaled. If no rule subscribed to this event, execution just continues. Figure 2.3 illustrates the principle of method wrapping. The dashed return line indicates that control is immediately returned only if no rule will require this event. If a rule has subscribed, the event is propagated and control is returned after some additional processing.

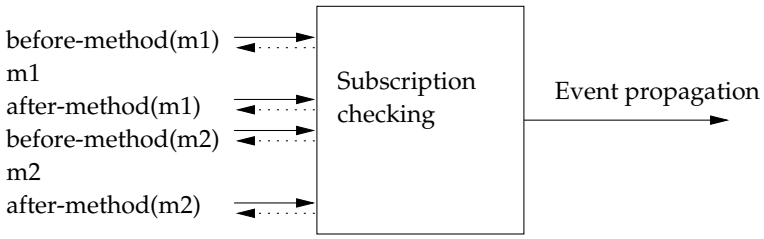

FIGURE 2.3. Method wrapping and subscription checking.

A basic difference from an architectural point of view is whether method wrapping is done manually or by the rule preprocessor, and whether all methods are wrapped or only those methods for which rules are known to exist. Wrapping every method improves flexibility since a new rule that consumes a given method event that has no previous subscription can be added without having to wrap the corresponding method and recompiling the object class definition. This approach was taken in [BZBW95, BDZH95]. However, this flexibility comes at a price. If every method is wrapped independently of whether a rule subscribes to that event or not, some overhead is paid before and after the execution of each rule to check if a subscription exists. Hand-wrapping a method that should be monitored was used

in [GD93, GGD95]. A compromise solution adopted in [CKAK94, Cha97] consists in distinguishing between passive objects, reactive objects, and notifiable objects. The interface for reactive objects is modified and only methods in reactive objects are wrapped.

Detection of state change events requires modification of the generic accessor functions. The detection of transaction events requires redefinition of the transaction bracketing and the commit and abort mechanism to signal the corresponding event to the event handler.

Detection of external events is a somewhat controversial issue. While external events are a needed extension in relational systems that otherwise are restricted to insert, delete, update, and selection events, they are easily represented in an object system through methods. In an object system, anything that is known to the system is represented by an object, an attribute, or a method. Therefore, external events can be easily handled by the method-event detector. In relational systems, external events may be represented explicitly in the database and signaled through an update event. However, many systems, both relational and object-oriented, provide an additional, explicit signaling mechanism for external events that interprets messages.

Absolute temporal events are signaled by the system clock. This mechanism is straightforward if all processes run on a single machine. However, in a client-server environment, in which clients are running on different machines, the options are either to have one master providing the absolute temporal events, in which case different propagation delays are possible, or have multiple clocks which may drift apart. For practical reasons, most prototypes opt for one common clock with a coarse time granularity that is assumed to be greater than the expected propagation delays.

2.6.2 Composite Event Detector Architecture

Event composition is an essential feature of any active system that provides more than the basic active functionality. The basis of composite events is the event algebra that is supported by the ADBMS. Event algebras may vary in expressive power, but they all offer at the minimum the basic composition for sequences of events, disjunctions, and conjunctions. A popular optimization is the closure operator, which accumulates repeated occurrences of the same event and triggers the corresponding rule only once. Negation and other history operators are also common.

From an architectural point of view, the event composer consists of the data structures needed to describe event types and their instantiations, and the operations on these structures. The most popular structures are finite state automata, variations of Petri-nets, and query-graphs. When finite state automata are used, a separate automaton is constructed for each event. A problem of finite state automata is their inability to associate the state of objects. The same is true for basic Petri-nets and is corrected

by colored Petri-nets, for example. In a Petri-net, a primitive event-token enters the Petri-net and progresses through the net according to the state at the decision points. In the case of query-graph-like composers, the structure of the composite event is represented as a tree with the operators of the algebra at the inner nodes. Primitive events and their parameters enter the graph through the leaf nodes and the event composition is completed when the root node is processed.

Garbage collection of semicomposed events is necessary, whenever the validity interval of a composite event expires. This could be either at the time a transaction commits or aborts, or after an interval defined by temporal or other events has lapsed. Once the validity interval has expired, we know that the missing primitive events that are required to complete a composition will never occur. To prevent the system from being swamped with semi-composed events, garbage collection is needed.

A major distinction in event composer architectures is whether the composite event graph is kept as a single monolithic structure or as specialized graphs, one for each event. There are advantages and disadvantages to each approach, particularly with respect to parameter passing, distribution, and garbage collection.

A single event graph minimizes redundancy. This is particularly the case whenever the same primitive event is used by many composite events limited to a single address space. On the negative side, it can be a bottleneck. It makes implementation in a distributed environment difficult and is very time-consuming to garbage-collect, since an extensive graph needs to be traversed and the semicomposed events need to be identified and removed.

Specialized event composers keep a separate event composition graph for each composite event. This carries an overhead in passing the appropriate parameters but makes distribution and garbage collection much easier. Particularly garbage collection is more efficient, since the whole graph can be eliminated once the validity interval has expired. Each specialized event composer corresponds to a specialized event handler. Therefore, each event handler has all the necessary information about the rules that need to be triggered or what other event handlers need to be notified. This eliminates the lookup process in a centralized rule manager.

2.6.3 Event Consumption and Logging

An important architectural issue related to event composition is the event consumption policy that is enforced by the ADBMS. First, it must be clear whether events may participate in multiple composite events or not. It is generally accepted that events may participate in multiple compositions. This requires either event replication or additional bookkeeping mechanisms to decide when an event can be discarded.

When composing events, there exist multiple event consumption strategies that are application-dependent. Events could actually be consumed

in *chronological* order (typical for workflow applications), in a *most recent* manner in which the latest event of a kind supersedes previous occurrences (typical for control applications), *continuous*, in which windows are established by two events and other events of interest in that window are detected (typical for trend monitoring applications), or a *cumulative* policy, in which all instances of the participating primitive events are accumulated and consumed at once when the composition is completed [CKAK94].

To support any of these policies, events and their parameters must be logged. Event logging can be a major performance factor in an ADBMS. For every event of interest, i.e., an event that has at least one subscriber, its occurrence must be recorded with a time stamp, the transaction in which the event occurred (if it is an event that can be associated with a particular transaction), and the necessary parameters that must be passed. Since events and their parameters may be used in a non immediate mode, they must be recorded for later use. This means multiple write operations to the log and a potential bottleneck. To avoid this hotspot, some systems use distributed logging with deferred consolidation of the partial logs [BZBW95].

2.6.4 Guarded Events, Light-Weight Vs. Heavy-Weight Events

An important design decision that is still debated concerns how much information should be attached to events and how much processing should be done by the event detectors. One school of thought (represented, for example, by the HiPAC project [DBM88]) states that events should be as lightweight as possible, i.e., they should only signal that a rule is to be processed and get out of the way. Any further testing as to whether the action ought to be executed should be pushed into the condition part of a rule. This has the advantage of not blocking further event detection, and that the condition is processed as any other query. The other school of thought (represented among others by the Ode project [GJS92]) states that lightweight events cause too many rules to be triggered unnecessarily just to detect in the condition evaluation part that no action is required. Therefore, events are provided with additional conditions, so-called guards, that allow the specification of rather complex conditions on the events. In the extreme, such an approach would make the condition evaluation superfluous and revert to (complex) event-action rules. The implication is also that the event handler must contain filtering mechanisms that otherwise are provided by the query processor.

2.6.5 Rule Scheduling

Once an event is raised, the rule(s) triggered by that event must be identified and scheduled for execution. The identification of the rule(s) that are triggered by an event can be done either by a specialized event handler that knows locally what rules are to be fired once the event has been raised, or through the use of a separate registration mechanism and an additional lookup. The rules that are fired are then scheduled and the requests for their execution must be passed to the DBMS's transaction manager. Condition and action can either be processed together or separate.

Depending on the coupling mode specified and the origin and type of the triggering event, a rule may either execute within the transaction in which the triggering event was raised or outside the user transaction.

If the rule executes within the triggering transaction, it may either be immediately after the event was raised, in which case the transaction's execution is halted until the rule completes execution (immediate coupling mode), or at the end of the triggering transaction and before the triggering transaction commits (deferred coupling mode). If a rule executes outside the triggering transaction in a separate transaction, then it may either begin in parallel and finish independently of the triggering transaction (detached coupling mode), it may begin in parallel but wait for the triggering transaction to commit before being allowed to commit (parallel causally dependent coupling mode), it may have to wait until the triggering transaction commits before being allowed to execute (sequential causally dependent coupling mode), or it may not begin execution unless the triggering transaction aborts (exclusive causally dependent coupling mode).

Rules triggered by absolute temporal events always execute in separate transactions. Rules triggered by events that are composed from events originating in multiple transactions must execute as detached transactions. If they execute in causally dependent detached mode, the causal dependence exists with *all* transactions in which the component events were raised [BZBW95].

Rule scheduling and execution is highly dependent on the DBMS's transaction management and must be synchronized with the execution of user transactions. How the user transactions and rule executions are synchronized depends on the transaction model supported by the underlying DBMS, the interfaces to the transaction manager provided by the underlying DBMS, and the way the active database system takes advantage of the provided facilities. We briefly review the alternatives.

2.6.6 Transaction Models

The two main alternatives when dealing with transaction models are flat transactions and nested transactions.
Within both groups, special extensions are needed to properly execute

rules.

Flat transactions are the transactions commonly supported by today's DBMS's and need little further explanation. However, when building active functionality on top of a flat transaction model, several limitations arise. Flat transaction models support only a single transactional thread of control. Therefore, whenever a rule needs to be executed, it has to be done one at a time without any parallelism. The transaction manager must provide a transaction handle to the rule system. It must further signal the end of the user transaction and transfer control to the rule manager and allow for the execution of deferred rules before committing the transaction and releasing the locks. No locks may be released prior to the execution of the deferred rules. Since rules may require new locks during their execution, two-phase locking would be violated otherwise. Flat transaction models typically do not provide for spawning additional transactions. Extensions for spawned detached transactions are straightforward, as long as no parameters are passed. If parameters that refer to objects modified within the transaction are passed to the spawned transaction, the isolation is compromised and the spawned transaction may execute with dirty data. If extended with the capability of spawning causally dependent transactions, the possibility exists that the spawned transaction competes with the spawning transaction for the same data, thus causing a deadlock. The transaction manager must be modified in such a way that the spawned transaction is always declared the victim when resolving a conflict. In [Mar95], these transactions are introduced as *weak transactions*.

Nested transactions have been proposed to increase intra-transaction parallelism. Common to all nested transaction models is the fact that new subtransactions can be spawned from within a transaction, and that the related transactions are organized in the form of a transaction tree. In the basic nested transaction model proposed in [Mos85], a nested subtransaction is started explicitly by the parent transaction, which is suspended until the nested transaction commits or aborts. Commitment of a subtransaction is conditional and occurs through the top. If the top transaction aborts, the whole transaction tree is aborted. In [HR93], a variation to the basic nested transaction model is proposed. This model allows for exploitation of intra-transaction parallelism through the introduction of downward inheritance of locks. This mechanism modifies the visibility rules for nested transactions and makes it possible for children to access the data manipulated by the parent.

For parallel execution of rules in active databases under a variety of coupling modes, a modified nested transaction model is presented in [DHL90]. The nested subtransactions of [Mos85] are used for the execution of immediate rules. In addition, three more types of nested (sub)transactions are defined. *Deferred subtransactions* are subtransactions whose execution is explicitly delayed until the end of the user's top transaction. If more than one deferred subtransaction is spawned within a user's transaction, they

all execute in parallel at the end of the user's transaction. If any of these deferred subtransactions spawns itself another subtransaction, this is executed immediately if the rule is to be executed in immediate mode or it is deferred until all deferred transactions from the level above have finished. *Nested top-transactions* are top transactions started from within another transaction and are represented by their own tree. However, a nested top-transaction has no privileges with respect to the spawning transaction, i.e., it may not see any non-committed objects and is not automatically aborted when the spawning transaction aborts. *Causally-dependent-top-transactions* (CDtop) are spawned from within another transaction and are like nested top-transactions that have their own transaction tree but are commit-dependent on the parent. Aborting the spawning transaction aborts the CD-top transaction. However, aborting the CD-top transaction has no effect whatsoever on the spawning transaction. A combination of this transaction model with downward inheritance of locks is a meaningful extension.

2.6.7 Rule Execution

Rules that are executed in immediate coupling mode just cause the execution of the user transaction to pause until the rule finishes and then control is returned to the user transaction. For all rules that are executed in a non-immediate coupling mode, the corresponding scheduler elaborates a schedule that is then passed for processing. Depending on the correctness criteria supported by the ADBMS, the transaction model provided and the number of rule processing threads, a scheduler may either create an ordering of the rules or allow for their parallel execution.

The execution of sets of rules triggered by the same event or sets of rules that are to be executed in a deferred mode at the end of the transaction requires either an ordering of the rules and their sequential execution in priority order as an extension of the user transaction, or it requires a nested transaction model to execute the rules as parallel subtransactions. The nested transactions presently offered by commercial DBMS's do provide transaction hierarchies but with *sequential* execution of the subtransactions. Since none of the commercial DBMS's offers *parallel* nested transactions, they must be implemented as part of the active extensions.

Some aOODBMS prototypes [Cha97] implement nested transactions in user space on top of the flat transaction model provided by the server. For example, if the underlying page server offers only flat transactions and page locks, it is possible to implement an additional object-locking mechanism in the user address space and provide the visibility and local commit and abort dependencies of the nested subtransactions proposed in [HR93]. However, in case of failure, the whole user transaction must be rolled back by the server.

Rules that execute in a detached mode as separate transactions can be

scheduled like any other user transaction. However, a major problem arising when executing transactions in any detached mode that allows for parallel execution is the passing of the parameters. Quite often the parameters to be passed include the state of the object that was just modified in the user transaction. If the user transaction has not committed, the lock on the object has not been released. Waiting for the commit of the user transaction before the rule can execute precludes parallelism, but making dirty data visible to other transactions reduces the isolation level. Some prototypes have opted for implementing lock sharing mechanisms between the triggering and the triggered transaction. Others restrict the detached execution of rules to those rules that do not share data between the triggering and the triggered transaction.

The causally dependent coupling modes imply additional commit or abort dependencies that are typically not supported by a DBMS's transaction manager. Therefore, the causally dependent coupling modes are often not implementable in layered architectures in which the active capabilities run completely at the user level or where modifications to the commit and abort process are not possible.

Processing may be done either by a single rule processor or in parallel among multiple rule processors. Most rule processors we are aware of are implemented as user processes or threads within a user process in the user's address space. This may be acceptable for prototypes but poses risks of interference in case of rule failure. Some systems [KRSR97] offer the possibility of dynamically assigning new threads for additional rule processors or giving up rule processors when the load falls below a threshold.

2.7 Recovery

Recovery, i.e., the ability to restore the database to a consistent state in the case of transaction or system failure, is one of the distinguishing features of database management systems. Therefore, active databases must provide the same resilience to failure.

The problem of recovery in active databases is more complex mainly because of three reasons: some events may be non-recoverable, some external actions may be irreversible, and certain coupling modes may allow transactions to commit ahead of the triggering transaction.

Closed nested transactions that always commit through the top are perfectly recoverable and do not require additional precautions. In general, to guarantee recoverability, a transaction should not be allowed to commit unless *both* its database updates and the events signaled by it are logged on stable storage.

The exact meaning of recoverability of events is still an open research issue and might have to be considered in the context of specific applications.

Database events are always recoverable. Other events, such as temporal events, may or may not be recoverable. The event log should contain explicitly all the relevant events, including temporal events, that either triggered a rule or were part of an event composition. As a general rule it is stated in [DHL90] that events signaled by committed transactions and for which the action was executed before the failure occurred should always be recovered. Events that are signaled by committed transactions and whose action had not completed before failing should only be signaled on reexecution if they are recoverable.

Rules that were triggered and are scheduled for execution in a detached mode must be guaranteed execution. An interesting situation arises when a rule that is being executed in a causally dependent transaction does not terminate because of a system failure during its execution while the spawning transaction committed. The active DBMS must provide the mechanisms to guarantee the execution of these rules, since the triggering transaction already committed and the user cannot be aware of the failure of the triggered transaction.

The problem of non-recoverable actions arises when the active database system allows either detached transactions that may commit independently of the spawning transaction, or when external actions that are irreversible are carried out as actions of a rule. To deal with the latter situation, [BBKZ93] introduced the detached sequential causally dependent coupling mode, in which a triggered action may only begin execution once the triggering transaction committed. Detached transactions that have no dependencies upon the spawning transaction should be used only in those situations in which their execution can be tolerated independently of the spawning transaction or where a compensating transaction can be defined.

Figure 2.4 puts the components of the rule processing subsystem together and shows schematically their interplay.

2.8 Conclusion

In this chapter the major factors affecting the architecture of active DBMS's were outlined. These are the degree of integration between the active functionality and the underlying DBMS, the architecture of the underlying DBMS, and the data model and language of implementation used. We discussed the basic subsystems of the active component: the rule specification and registration component, and the event detection and rule execution component. For each component we analyzed its functionality and tried to view it in the perspective of integration, and in terms of the limitations imposed by the underlying DBMS and language of implementation.

As with any generic architecture, much detail had to be omitted. Details about individual components can be found in the corresponding chapters

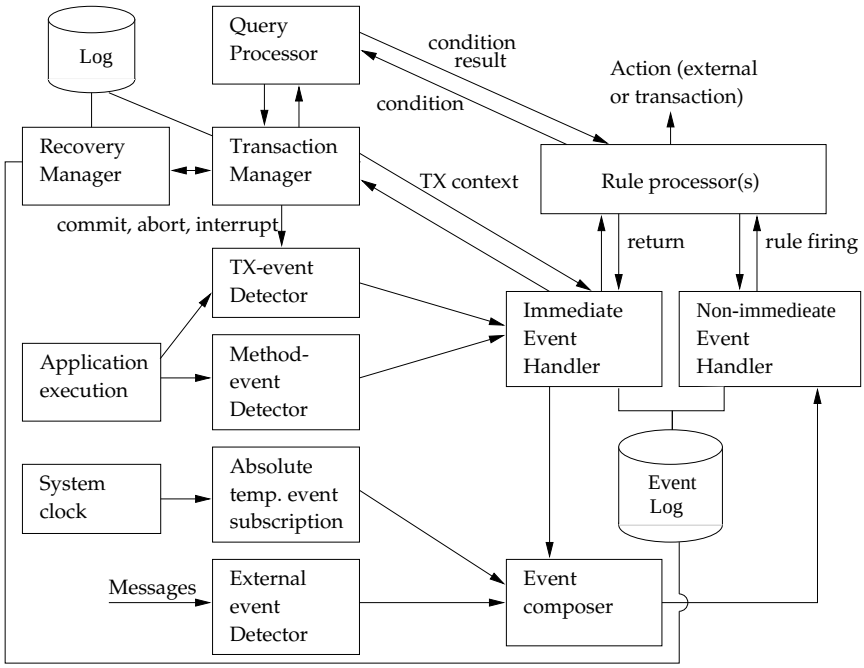

FIGURE 2.4. Schematic representation of the rule execution subsystem architecture.

that will treat each topic in depth. Details on the design decisions that were made in specific systems can be found in the chapters describing each of them. Hopefully this discussion served to set the stage for the more detailed discussions to follow.

2.9 References

[BBKZ93] H.Branding, A.P. Buchmann, T.Kudrass, and J.Zimmermann. Rules in an Open System: The REACH system. In N. W. Paton and M. H. Williams (Eds.), *Rules in Database Systems, Proc. 1st Intl. Workshop on Rules in Database Systems*, August 1993.

[BDZH95] A.P. Buchmann, A.Deutsch, J.Zimmermann, and M.Higa. The REACH Active OODBMS. In *Proc. of the 1995 ACM SIGMOD Intl. Conf. on Management of Data*, May 1995.

[BZBW95] A.P. Buchmann, J.Zimmermann, J.Blakeley, and D.Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proc. 11th Intl. Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[CCS94] C.Collet, T.Coupaye, and T.Svensen. NAOS Efficient and Modular Reactive Capabilities in an Object-Oriented Database System. In *Proc. of the 20th Intl. Conf. on Very Large Data Bases,* Santiago, Chile, September 1994.

[CDRS86] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proc. 12 Intl. Conf. on Very Large Data Bases,* Kyoto, Japan, August 1986.

[Cha97] S.Chakravarthy. SENTINEL: An Object-Oriented DBMS with Event-Based Rules. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data,* Tucson, AZ, May 1997.

[CKAK94] S.Chakravarthy, V.Krishnaprasad, E.Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proc. of the 20th Intl. Conf. on Very Large Data Bases,* Santiago, Chile, September 1994.

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

[DHL90] U.Dayal, M.Hsu, and R.Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proc. of the 1990 ACM SIGMOD Intl. Conf. on Management of Data*, Atlantic City, NJ, May 1990.

[GD93] S.Gatziu and K.R. Dittrich. Events in an Active Object-Oriented Database System. In N. W. Paton and M. H. Williams (Eds.), *Rules in Database Systems, Proc. 1st Intl. Workshop on Rules in Database Systems*, Edinburgh, August 1993.

[GGD95] S.Gatziu, A.Geppert, and K.R. Dittrich. The SAMOS Active DBMS Prototype. In *Proc. of the 1995 ACM SIGMOD Intl. Conf. on Management of Data*, San Jose, CA, May 1995.

[GJS92] N.H. Gehani, H.V. Jagadish, and O.Shmueli. Composite event specification in active Databases: Model and Implementation. In *Proc. of the 18th Intl. Conf. on Very Large Data Bases,* Vancouver, Canada, August 1992.

[HR93] T.Haerder and K.Rothermel. Concurrency Control Issues in Nested Transactions. *VLDB Journal*, 2(1), 1993.

[KRSR97] G.Kappel, S.Rausch-Schott, and W.Retschitzegger. A Tour of the TriGS Active Database System–Architecture and Implementation. Technical report, Department of Information Systems, Johannes Kepler University, Linz, 1997.

[Mar95] J.Marschner. *Non-Standard Transaktionsmanagement in Einem Aktiven Objektorientierten Datenbanksystem.* Dipl. Thesis, Dept. of Computer Science, Darmstadt University of Technology, Darmstadt, 1995.

[Mos85] E.Moss. *Nested Transactions.* MIT Press, Cambridge, MA, 1985.

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