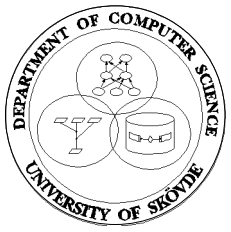

UNIVERSITY OF SKÖVDE
Department of Computer Science

ACOOD Essentials

Henrik Engström (henrike@ida.his.se)
Mikael Berndtsson (spiff@ida.his.se)
Brian Lings (brian@dcs.exeter.ac.uk)

Technical Report

HS-IDA-TR-97-010

ACOOD Essentials

Henrik Engström
Mikael Berndtsson
Brian Lings
University of Skövde, Sweden
{henrike,spiff}@ida.his.se
brian@dcs.exeter.ac.uk

ABSTRACT

This paper describes the active object-oriented database system ACOOD, developed at the universities of Skövde and Exeter. ACOOD adds active functionality on top of the commercially available Ontos DB. The active behaviour is modelled by using Event-Condition-Action (ECA) rules. ACOOD offers all essential functionality associated with an active database. The semantics and user interface have been clearly defined in order to produce a prototype that can be used to develop database applications.
The historical background of active databases and the development of ACOOD are covered in the paper together with a detailed description of the latest, redesigned version of the system. There is also a discussion of experience gained through the work with ACOOD and a comparison with similar systems.

Keywords: active databases, object-oriented databases

1 Introduction

Active Database Management Systems (ADBMS) have been put forward as an approach to support reactive behaviour in database systems. Reactive behaviour is commonly supported through event-condition-action (ECA) rules, with the following semantics: upon the occurrence of an event, a condition is evaluated and, if the condition is satisfied, an action is executed. Pioneering work on active databases and triggers and rules in databases appeared during the ‘70s and ‘80s, but it was not until the early ‘90s that extensive research on active databases was carried out.

Active database research field has developed into a relatively mature area. An indication of the degree of maturity is the significant number of working research prototypes, e.g., Sentinel [AMC93], Samos [GD93], Ode [GJS92], Reach [BZ+95], ACOOD [BL92], and the publication of “The Active Database Management System Manifesto” [ACT96]. Furthermore, the ECA-rule formalism is now being adopted in a number of other research areas such as real-time systems [BH95], cooperative problem solving [BCL96], [CK+93], and workflow systems [BJ94].

Although several ADBMS research prototypes have been constructed, few have been fully implemented for realistic use. A natural continuation of the work with active databases is to implement prototypes that can be used to solve real problems, [Day95]. By experimenting with the architectural components of these systems, it is possible to analyse the benefits and problems with proposals from the community. Implementations of realistic applications give insight into which features are useful and which could be omitted.

By contrast, support for active rules in commercial database systems is limited. Simple forms of active rules (triggers, alerters etc.) are currently provided in many commercial relational database systems, but commercial object-oriented database systems do not yet support active rules. However, there is a large body of work currently being undertaken on research prototypes for active object-oriented databases. An important step towards transferring knowledge from the academic

research field to commercial object-oriented databases is to build research prototypes for realistic use.

ACOOD (Active Object-Oriented Database system) [BL92] is an extensive research prototype of an ADBMS built on top of Ontos DB, which is a commercial object-oriented DBMS. The project is jointly carried out between the universities of Skövde (Sweden) and Exeter (UK). The first prototype of ACOOD was built in 1991 [Ber91], and during the first years it was used to implement and experiment with various theoretical proposals. The most significant projects and areas in which ACOOD has been used are:

- **Rule indexing** - When using rule indexing each event contains information on which rules to signal when the event occurs. In this way only the affected rules are retrieved in contrast to the naive approach where all rules have to be examined on each event occurrence [Ber94a], [Ber94b].
- **Logical events** - A logical event is an event which is said to occur only under a defined condition. This means that the event will not be recorded if the associated condition evaluates to false. This has importance in systems design, where conceptual events can be more closely modeled, and maintained. There are also potential efficiency gains, in that fewer events are reported, and there are fewer invocations of the rule manager [BL95], [Schw95].
- **Inheritance of events and rules** - Events and rules in ACOOD can be associated with classes. This means that, as with other aspects of object definition, they should be inherited. Thus an event defined on a base class will be inherited in a specialization of that class, and will be generated when the method is invoked. However, override of both event and rule definitions may occur. Hence it is possible, for example, to trigger an inherited rule by a differently defined event and/or under different conditions [Schu96].
- **Benchmarking** - Since 1995, ACOOD has been in involved in the BEAST benchmark project [Ekl95], [GM+96] that aims to tune the performance of active DBMSs. This work is also a part of the endeavour to make the system useful for realistic tasks. In order to measure the performance and identify potential bottlenecks in the design, a benchmarking system, BEAST, has been used. BEAST is tailor made for active object-oriented databases. In general, the benchmark results show that the design is scalable. That is, the response times are not worse than linear with respect to database size.
- **Cooperative problem solving** - An application area where ACOOD has been used is Cooperative Problem Solving (CPS). The results from these studies show that an ADBMS can be useful in the modeling of agent communication as well as in the internal work of an agent [BCL96], [Hag96], [BCL97a], [BCL97b].
- **Practical assignments** - Apart from the research projects, ACOOD has been used in different courses and student projects, e.g. a course on Advanced Database Systems given to MSc students. This has given useful feedback on the operational aspects of the DBMS.

The experiences gained between 1991 and 1996 lead to a complete redesign and re-implementation of ACOOD in 1996. The aim was to keep the good features of previous versions but with a uniform architecture and a distinct user interface. In addition, support for new functionality (not previously implemented), such as rule priorities, and passing of event parameters, were to be implemented.

This paper describes the design and functionality of ACOOD. The description follows the "ADBMS Manifesto" which is also used to compare ACOOD with other systems.

The rest of this paper is organized as follows: The next section gives the background of ADBMSs. Section three explains the functionality of ACOOD and is followed by a section on how it can be used by the application programmer. In section five some implementation details are given followed by a discussion of experience gained from the development of ACOOD. The next section compares a number of active object-oriented database systems. The paper ends with some conclusions and suggestions for future work.

2 Active Databases

> "An active DBMS is characterized by its ability to monitor and react to both database events and nondatabase events in a timely and efficient manner." [Cha89]

An active database system (ADBMS) can automatically react to events such as database transitions, time events, and external signals in a timely and efficient manner. This is in contrast to traditional database systems which are passive by nature. That is, a passive database system execute queries and transactions only when it is explicitly requested to do so, [CB+89], [WC95].

2.1 History

The semantics of *active behaviour* that appear in today's active database systems have their roots in the areas of Artificial Intelligence (AI), programming languages and databases.

In AI, active behaviour is supported by daemons and active objects [BS83], production rules [FM87] and procedural attachment in frames [Min75]. AI-systems like KEE [HS87] and LOOPS [BS83] use active values and production rule systems to provide the user with active features. However, these systems assume a small number of objects and they do not support sharing, consistency, and concurrent execution of transactions.

In the field of programming languages, active mechanisms or behaviour can be found in ACTOR [Hew77], which was one of the first programming languages to provide active behaviour for objects. Logic programming is another area where, primarily the PROLOG programming language has been used to enhance database systems.

Active features appeared in DBMS as early as 1973 in the CODASYL [CM94] proposals, which proposed the *ON condition* clause. An ON clause specifies the action to be executed immediately after its triggering event (operation) is performed. The user can define multiple ON clauses with the same triggering event (operation). If multiple ON clauses are specified, then they are executed in the order they where defined. A trigger in CODASYL can fire other triggers. Active behaviour can also be found in System R [Esw76], where Eswaran suggested that production rules could be used for: integrity constraints, authorization checking and maintenance of derived data. System R allows the user to define special triggers in the form of assertions. Assertions are SQL predicates that the system enforces against all updates to the database. Assertions can be triggered by any update that can change the value of a predicate.

The term active databases was coined in [Mor83], which presented a system that could support automatic update of derived data and views when base data is updated.

Previous work on enhancing a DBMS has mostly been on adding some type of rule based technique to the database. Rules have been shown to be useful for specifying constraints etc. Representation of rules is one of the key factors within active databases. The history of active database systems

can be summarized as in Figure 1. The diagram is not an attempt to capture the whole spectrum of database systems but rather to show the main influences on today's active database systems.

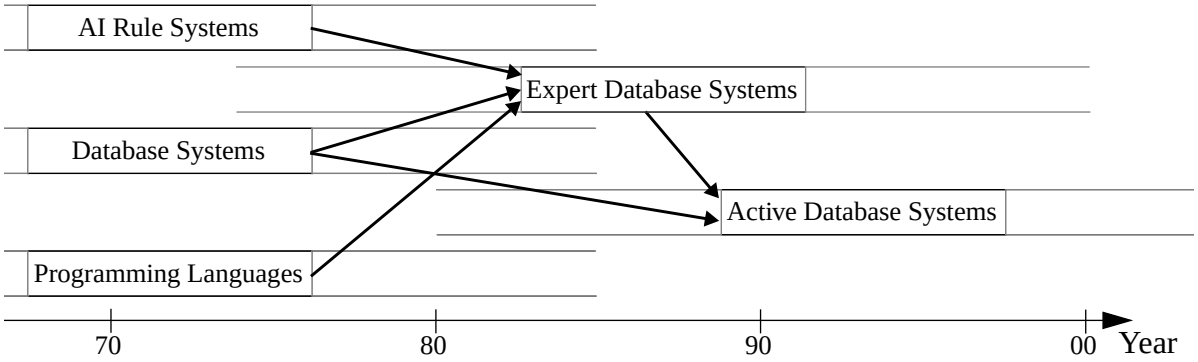

Fig. 1 The history of ADBMS

The three aforementioned fields, i.e. AI, programming languages and databases, have affected the research on active databases in several ways. As we saw previously, these three areas have provided the user with several kinds of active mechanisms with varying power.

In the mid 1980s research teams gathered around the topic of expert databases. There was a need for a new generation of databases, since the traditional systems could not support new applications such as CAD/CAM, CIM, and process control. Research results from Expert Systems (ES), programming languages and the database field were brought together in proposals for a new class of databases referred to as *expert database systems*. Much of the research in this field concerned the coupling of PROLOG with a conventional database. Active database systems have emerged from the ideas in the expert database area.

The HiPAC [CB+89], POSTGRES [SJ+90], and ETM [DKM86] projects were among the first to pick up the idea of active mechanisms in database systems. They were also the first to make extensive implementation attempts to realise the ideas of active databases. HiPAC and POSTGRES have laid the foundation for many other research projects around the world.

2.2 Previous Support for Monitoring

There are several applications, such as shop floor control (SFC) and computer integrated manufacturing (CIM), that require automatic situation monitoring of, for example, the production environment. In such situations, it is important to react in a timely and efficient manner in response to events such as machine break-down or change in inventory.

Previous approaches to support automatic situation monitoring with conventional DBMS can be broadly classified into:

1. Periodically polling the database
2. Embedding or encoding event detection and related action execution in the application code.

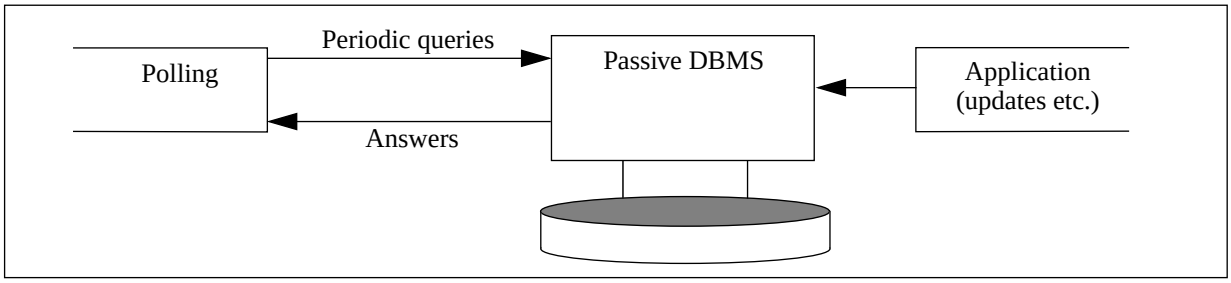

Fig. 2 The polling approach

The first approach (Figure 2) implies that the queries must be executed exactly when the event occurs. The frequency of polling can be increased in order to detect the event, but if the polling is too frequent, the database is overloaded with queries that will, most of the time, fail. Thereby it will lower the system's performance, since a lot of unnecessary queries are running. On the other hand if the frequency is too low, the system may fail to detect an event.

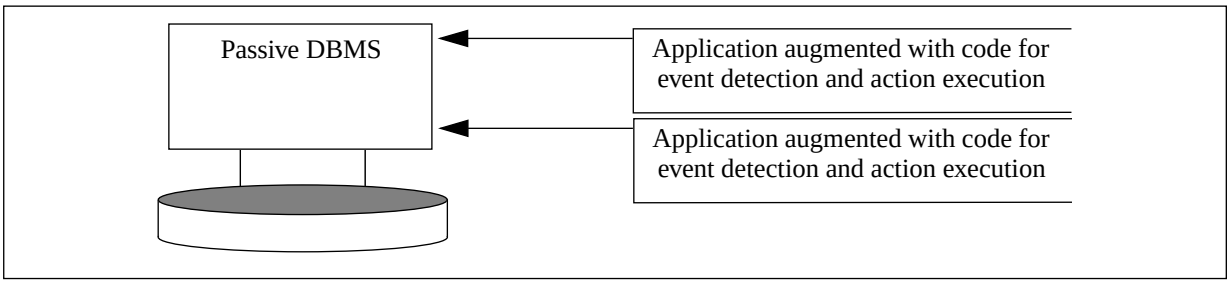

Fig. 3 Applications augmented with code for event detection and action execution

The second approach (Figure 3) implies that every application that updates the database should be augmented with code that detects the events. From a software engineering point of view this approach is inappropriate, since a change in a condition specification implies that every application that uses the affected objects needs to be updated. If rules are embedded in applications, it is difficult to change and inspect the rulebase. Inconsistency among the rules may also arise if the applications are updated in a non-uniform manner.

Thus, neither of the two previous approaches can efficiently support reactive behaviour in a database context.

2.3 The Architecture of an Active Database System

An active database system (Figure 4) has several advantages compared with the two previous approaches. Instead of waiting for explicit user or application requests, an active database system reacts automatically in response to predefined events. Hence, with an active database the drawbacks with polling and application embedded rules can be avoided.

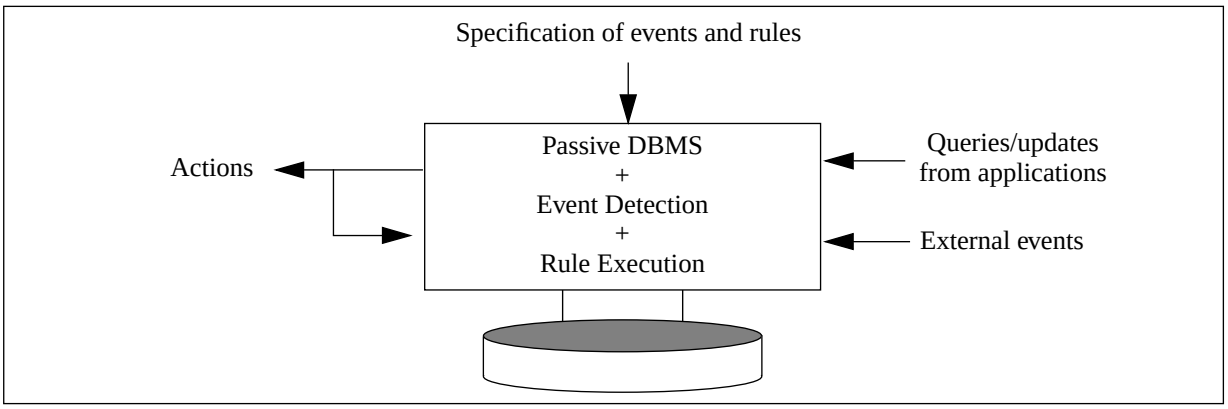

Fig. 4 Active database system

Active databases can be seen as an approach to efficiently support automatic situation monitoring and reactive behaviour. Reactive behaviour in an active database system is expressed by event-condition-action (ECA) rules. The semantics of ECA-rules are: when an event E occurs, evaluate condition C, and if condition C is satisfied, execute action A. Events can be classified into: primitive events, and composite events

Primitive events refer to elementary occurrences which are predefined in the system such as transaction events and database events (e.g. due to data updates). A *composite event* is a set of primitive events combined through event operators such as disjunction, conjunction, and sequence.

Active databases open up a new paradigm within database research, where the database is not seen as a slave to the application, but more like an active module: a peer. The cooperation between the DBMS and the application can now be viewed as a two way communication, where the DBMS can support and control the application's activities in various ways.

There has been intensive research activity concerning active databases for more than ten years. The interested reader can find more information on active databases in the following sources: [Cha92], [PW93], [Wid94], [Sel95], [WC95], [ACT96].

3 Characteristics of ACOOD

This section describes the characteristics of ACOOD following the outline in the “Active Database Management System Manifesto” [ACT96]. The description concerns the latest version of ACOOD which has been redesigned and re-implemented in order to provide a well defined user interface and a clean implementation. The functionality has been extended with rule priorities and the passing of event parameters, which were lacking in previous versions.

A source of confusion when describing the characteristics of a research prototype is the mismatch between the designed and the implemented system. The implementation often reveals design errors and practical problems that are hard to foresee in the design phase. The description below focuses on the implemented version of ACOOD.

3.1 The Properties of ACOOD

"*Feature 1:* **An ADBMS is a DBMS**."

The first required feature states that an active database should not offer less functionality than its passive counterpart.

ACOOD is built as an extension of Ontos DB and all "passive" features are available to the user. Ontos is primarily accessed through an application programming interface (API) which means that database applications are developed by using library functions and classes. ACOOD adds a number of functions to the API without hiding any of the existing ones. ACOOD is backwards compatible with Ontos, i.e. any application written for the latter will run on the former.

"*Feature 2:* **An ADBMS has an ECA-rule model**."

The second required feature implies that there should be means for the user to define events, conditions and actions. The manifesto leaves a door open for ADBMSs that use variations on the ECA-model, e.g. event-action or condition-action rules.

Reactive behaviour in ACOOD is modelled using ECA-rules.

"*Feature 2.a:* **An ADBMS has to provide means for defining event types**."

Both primitive and composite events are supported in ACOOD. There are two different primitive event types in ACOOD namely *explicit* events and *method* events. An explicit event is signalled by the program, or by some other process, and it can be used by the application programmer to model other event types, e.g. temporal events (by adding a timer-process). The method events can be defined to be signalled before or after a method invocation and they are generated automatically by the system, which means that the applications does not have to signal method events explicitly.

Composite events in ACOOD can be constructed by combining other primitive or composite events by using one of the following event operators: *disjunction*, *conjunction* and *sequence* [Eri93]. With regard to the conjunction and sequence operator it is possible to restrict the composition to events originating from the *same object*. This means, for example, that a conjunction will trigger only if all of its components have occurrences originating from the same object.

"*Feature 2.b:* **An ADBMS has to provide means for defining conditions**."

Conditions in ACOOD are defined by specifying a method in a persistent class. The condition is invoked either on a pre-defined object or on the one that triggered the rule. The condition can be omitted in the rule definition in which case the rule becomes an event-action rule (equivalent to having an "always true" condition).

"*Feature 2.c:* **An ADBMS has to provide means for defining actions**."

When a rule is created in ACOOD the action is specified as a method. The object to use for action execution can be specified in the same manner as the condition object.

"*Feature 3:* **An ADBMS must support rule management and rulebase evolution.**"

An ADBMS supports rulebase management if it is possible for the user application to retrieve information about the rules. It should also be possible to change the definitions of events, conditions and actions (rulebase evolution).

"*Feature 3.a:* **An ADBMS has to support rulebase management**."

All event and rule definitions in ACOOD are represented as persistent objects and can hence be managed as any other object in the database. Currently ACOOD supports retrieval of information through a graphical browser.

"*Feature 3.b:* **An ADBMS has to support rulebase evolution**."

ACOOD supports dynamic rulebase evolution which means that rules and events can be defined and deleted at run-time. The only limitation to this is that methods have to be declared in a certain manner to be eligible for method events. The actual definition of the event can, however, be done dynamically. In theory, all methods could be prepared to be used as method generators while just a few are generating events. The overhead associated with the preparation is small compared to the cost of event generation, which means that it is reasonable to prepare all methods.

"*Feature 3.c:* **An ADBMS has to support enabling and disabling of rules**."

When a rule is disabled the definition remains in the database but it will not be triggered. Enabling and disabling of rules makes it easier to control the rule base and is potentially more efficient than creation and deletion of rules.

A rule in ACOOD is passive upon creation and it can be enabled and disabled by the user.

"*Feature 4:* **An ADBMS has an execution model**."

The execution model determines the relation between events and rules and the semantics of rule execution. For example, events can be generated automatically or explicitly, the event can be associated with one instance or a set of instances. Information on the event occurrence should be available when the condition and action are executed.

"*Feature 4.a:* **An ADBMS must detect event occurrences (situations)**."

One key ability of an active database is the automatic detection of events. If the database does not detect events automatically it cannot be considered active. In addition to automatic detection users can be allowed to manually signal some event types.

ACOOD supports automatic method event detection.

"*Feature 4.b:* **An ADBMS must support binding modes**."

The binding mode of the ADBMS determines which kind of entity the event occurrence is associated with. The binding mode could for example be instance-oriented or set-oriented.

When an event occurs in ACOOD it is bound to the object that caused the event. This can be characterized as instance-oriented binding mode.

"*Feature 4.c:* **An ADBMS must be able to evaluate conditions**."

"*Feature 4.d:* **An ADBMS must be able to execute actions**."

Associated with the binding mode is the execution of condition and action. The execution should have access to the information associated with the event occurrence.

In ACOOD the condition and action methods have access to the object identifier of the triggering event and to the parameters of the method (in the case of method events). If the triggering event is a composite event, the condition and action may only access the information from the terminating event. This is due to difficulties in aggregating and storing volatile references.

"*Feature 5:* **An ADBMS must offer different coupling modes**."

The scheduling of a rule with respect to the triggering transaction is determined by the coupling mode. The evaluation of the condition could, for example, be done as soon as the event occurs or it can be delayed. The action can be executed within the triggering transaction or in a separate transaction. It is desirable that several coupling modes should be available to the rule specifier.

The current version of ACOOD supports the immediate coupling mode. This means that the condition of a rule is evaluated and its action executed as soon as the event is detected and within the same transaction as the triggering event.

"*Feature 6:* **An ADBMS must implement consumption modes**."

For composite event detection one or more consumption modes could be specified. These specify how event occurrences should be used during composition. Different consumption modes may be required for different kinds of applications and the manifesto suggest that a number of modes should be offered.

Currently ACOOD supports the *recent* consumption mode but other modes are planned to be added.

"*Feature 7:* **An ADBMS must manage the event history**."

Event occurrences have to be stored in order to generate composite events. The lifetime of an event occurrence is dependent on the consumption mode and some other factors.

Event occurrences in ACOOD are recorded locally for each composite event. Occurrences that may be used for future compositions are stored. The lifetime of an occurrence may span several transactions.

"*Feature 8:* **An ADBMS must implement conflict resolution**."

If several rules are to be triggered at the same instance in time, the ADBMS should perform conflict resolution. The manifesto states that the user should be offered the possibility to control conflict resolution.

Conflict resolution in ACOOD is supported by rule priorities. Upon an event occurrence rules are selected in the following way:

- All rules triggered by the event (directly or through a composite event) are collected into a firing sequence ordered on rule priority
- The rules are executed in sequence
- If the execution of a rule generates a new event occurrence, the associated rules are collected into an ordered sequence which is concatenated after the original sequence
- The cascaded rule triggering can continue to arbitrary depth

"*Feature 9:* **An ADBMS should support a programming environment**."

In order to be useful the ADBMS should offer a number of tools such as a rule browser, a rule designer, a rulebase analyser and a debugger.

The rule definition "language" of ACOOD is a set of functions, used by the application programmer. A simple debugger has been implemented that uses the API of ACOOD. Higher level tools can be constructed in a similar way. There is an object browser for Ontos that can be used to inspect events and rules.

"*Feature 10:* **An ADBMS should be tunable**."

The performance of an active database should be comparable to a passive counterpart. It should be possible to tune the rulebase without affecting the semantics.

The performance and design of ACOOD has not been compared with any passive system (e.g. Ontos). It has, however, been benchmarked using the BEAST benchmark tests [GD93]. The main objective with the test is to identify bottlenecks in the system and to examine if it scales well with the size of the rulebase. The result for ACOOD shows that the time to handle events and rules is

almost independent of the size of the rulebase. The benchmarking also shows that the treatment of composite events are relatively efficient compared with rule execution [GM+96].

4 The User's View of ACOOD

An active database management system, like any other system, is intended to be used by one or more users. There can be a more or less distinct separation between the implementation and usage of the system. In ACOOD, an important design principle is to have a distinct user interface. The current version has an API, which makes it relatively easy for programmers to develop database applications that uses active mechanisms.

ACOOD is implemented on top of Ontos and a user application can access the database by communicating directly with Ontos or via ACOOD. This "layered approach" to extend the functionality of Ontos is illustrated in Figure 5.

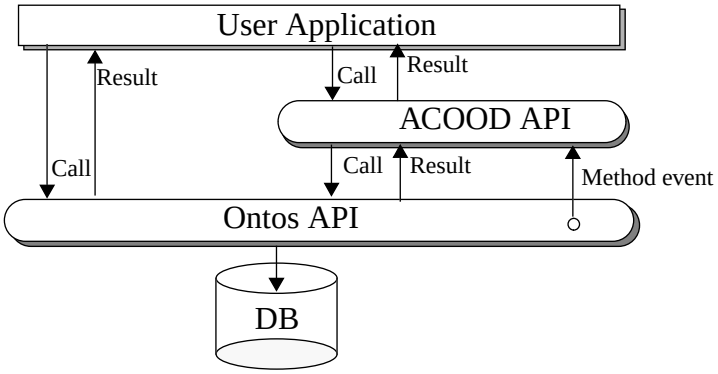

Fig. 5 The layered system-architecture

The advantages and disadvantages of a layered approach are discussed in detail in [GG+95].

4.1 Ontos User's Interface

Ontos [Ont94] is an object-oriented database management system (OODBMS) which means that the basic entities of the database are objects that can be stored, retrieved and modified through different operations. The API of Ontos is an extension of C++ and an object that should be stored in the database should belong to a subclass of "OC_Object".

Persistent classes are "classified" in order to provide Ontos with the necessary type information. An object belonging to a classified class can be persistently stored in the database by calling the member function "putObject". The user application can use Ontos classes and functions to store and retrieve objects, and perform other database operations.

Ontos provides several "aggregate classes" that can be used to cluster associated objects. An aggregate can, for example, be a list, a set or an array. They differ in the way objects can be inserted and retrieved. By using aggregates it is possible to handle groups of objects, a feature which gives the database some set-oriented capabilities.

Persistent objects are identified through a system-generated identifier and possibly through a user assigned name. An object with an assigned name can be retrieved from the database through an Ontos function. Objects without names have to be referenced by other objects or be members of an aggregate.

References to other objects should be treated with extra care when dealing with persistence. In C++ it is common to use pointers to reference other objects, but as they are volatile it is pointless to store them in a database. Ontos provides the programmer with a reference class that is used for persistent references. If an object is referenced using the reference class it will be retrieved from the database automatically when needed.

Ontos offers a way to invoke operations on persistent objects by specifying the name of the method (e.g. the name can be specified through a string variable). By using this feature ("OC_invokeByName") it is possible to trap the method invocations and to perform additional actions before and/or after the actual body.

4.2 ACOOD User's Interface

Events and rules are defined in an application by calling the user functions of ACOOD (as shown in appendix A). A unique name is given to an event or rule upon creation. This name is used to refer to the object in subsequent operations.

The choice of user functions instead of user classes is deliberate. Functions gives a compact and simple user interface and make it easy to get a distinct separation between the user view and the implementation details. It should be noted, however, that it is trivial to replace the current user interface with a class-based one.

4.2.1 Events

There are functions for defining explicit events, method events and composite events.

The definitions of an explicit event contains the name of the new event. In addition a method event requires the name of a method and an indication of whether it should be generated before or after the execution of the method.

Method events are limited to methods with the signature "void (void *)". The reason for this is that it is difficult to extract the parameters from a method with arbitrary signature, and pass them to the condition and action. Note that the limitation on the signature does not limit the expressive power of the ADBMS. The application programmer has to pack the parameters and the return value into a structure or object and use a pointer to it as the only parameter to the method.

The automatic triggering of method events requires that methods are invoked by using "OC_invokeByName".

A composite event is defined by specifying the names of two sub-events and an operator. Arbitrary complex expressions can be formed by combining composite events. A composition may be restricted to event occurrences from the same object. For example, let the composite event CE1 be defined as a conjunction of two method events, E1 and E2. If the following events occur:

e_1^1 - E1 is generated by an invocation on the object o1

e_2^1 - E2 is generated by an invocation on the object o2

e_2^2 - E2 is generated by an invocation on the object o1

Then CE1 will be generated when e_2^1 occurs if there is no restriction. If CE1 is restricted to the same object it will occur when e_2^2 is generated.

It is not possible to restrict a disjunction to the same object as it will trigger each time a sub-event occurs (which makes it meaningless to talk about "same object").

4.2.2 Rules

A rule is defined by specifying an event, a condition, an action and a priority. The condition and priority can be omitted. Through user functions it is possible to define, activate, deactivate and delete rules.

The condition and action are specified as names of methods that take a pointer to an "ACOOD_parameter"-object as their only argument. A condition returns an integer value (where 0 means false) and the action returns no value.

The object to use for condition and action execution can be named directly in the rule definition, i.e. "o1::myCondition" where o1 is the name of a persistent object. If no object is specified, the triggering object will be used.

4.2.3 Parameters

The "ACOOD_parameter"-object that is sent to the condition and action methods, contains information about the triggering event. This information can be accessed through the user functions.

4.3 An Example

Figure 6 shows a complete example of how ACOOD can be used in an application. The example can be seen as a simplified banking scenario where certain accounts are supervised by different rules that signals "events of interest". The conditions and actions for different accounts may differ depending on the owner and the type of account.

In the example, the class "MyMoney" is persistent and contains an integer value and a method "diff" that is used to change the value. The methods eligible for method events have to be invoked through the "OC_invokeByName"-function. The easiest way to do this is to wrap the method. In the example, "active_change" is wrapped by "change". In the current version the wrapping has to be done by the application programmer, but it is possible to let a pre-processor do it in the future.

The "negative"-method, which returns true if the argument is negative, is used as a condition and "alert" is used as an action (it displays a warning message).

The main function creates a method event, E1, that triggers before the execution of "active_change". The rule R1 has the following definition:

R1: ON E1, IF negative, DO alert

When the program is executed, the value of the "testObject" is decreased with 10 units. This triggers the rule, the condition is evaluated to true, and the message "Somebody tries to take your money!" is displayed.

MyMoney.h

```
class MyMoney: public OC_Object {
 public:
   /* The constructor */
   MyMoney();
   /* A condition method. It must have this signature.  */
   int negative(ACOOD_Parameters *);
   /* An action method. It must have this signature.  */
   void alert(ACOOD_Parameters *);
   /* This method should be eligible for method events ... */
   void active_change(void *);
   /* .. it has to be wrapped for automatic event detection.
    This could be performed by a pre-processor. */
   void change(void *);

   /* This method is needed by Ontos, it returns the type of the object*/
   OC_Type* getDirectType();
   /* Used for activation of a persistent object */
   MyMoney(OC_APL* theAPL);
 private:
   int theMoney; /* The amount available*/
};
```

MainProg.cxx

```
#include "acood.h"
#include "MyMoney.h"
main()
{
   OC_open("myDatabase");
   OC_transactionStart();
   MyMoney testObject;
  /* We have to write the name of the wrapped method. */
   ACOOD_defineMethodEvent("E1","MyMoney::active_change",PRE);
   ACOOD_defineRule("R1","E1","negative","alert");
   ACOOD_activateRule("R1");
   int diff=-10;
/* A call to the active method with the parameter -10 */
   testObject.change(&diff);
   OC_transactionCommit();
   OC_close();
}
```

MyMoney.cxx

```
#include "MyMoney.h"
MyMoney::MyMoney():OC_Object()
{
   theMoney=0;
}
int MyMoney::negative(ACOOD_Parameters * param)
{
   // We take out the parameters, that was sent to the method
   int *x=(int *)ACOOD_getMethodParameters(param);
   return (*x)<0; //Check if the value was negative
}
void MyMoney::alert(ACOOD_Parameters * param)
{
   // The action is just to print a warning
   cout<<"Somebody tries to take your money!"<<endl;
}
void MyMoney::active_change(void *diff)
{
   // Change the amount with diff.
   theMoney=theMoney+*((int *)diff);
}
void MyMoney::change(void *x)
{
   // argList is used when methods are invoked "by name"
   OC_ArgumentList argList;
   // Sets the object of the method invocation
   argList.setElement(0L,this);
   // Set the argument of the invocation
   argList.setElement(1L,x);
   // Invokes the named method
   OC_invokeByName("active_change",&argList);
}
OC_Type* MyMoney::getDirectType()
{
   return (OC_Type *)OC_lookup("MyMoney");
}
MyMoney::MyMoney(OC_APL* theAPL)
{
   cout<<"Object read from database"<<endl;
}
```

Fig. 6 An example of an application using ACOOD

5 Implementation

The implementation of ACOOD has been added to Ontos [Ont94] without accessing its source code which has put restrictions on possible solutions. Ontos, however, offers low level routines that are used to implement the automatic generation of method events. When the methods are invoked with "OC_invokeByName" it is possible to trap the call and to generate the appropriate event.

5.1 The Class Hierarchy

Rules and events in ACOOD are implemented as classes. The different types of events form a hierarchy. In addition to the event and rule classes there is a parameter class and a manager class. The class hierarchy is shown in Figure 7 (OC_Object is the base class for all persistent objects).

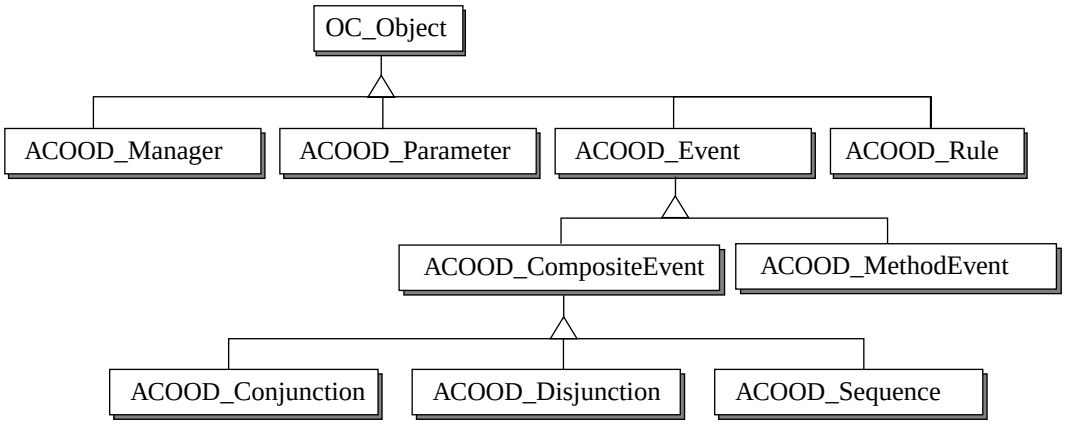

Fig. 7 The class hierarchy

5.2 Events

All events have lists of subscribing rules and composite events. The subscribers of an event are notified when the event occurs. A simplified model of the event class is shown in Figure 8.

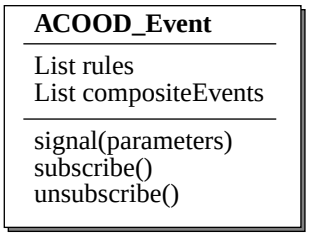

Fig. 8 The event class

Explicit events are represented as instances of the "ACOOD_Event"-class. They are always triggered "explicitly" by the application by calling the "ACOOD_raiseEvent"-function. Method events, on the other hand, will be triggered automatically when the corresponding method is invoked.

The "signal"-method receives a structure containing the object that caused the event and the parameters sent to the method (in the case of method events). It is possible to have parameters to explicit events, but the default is to have none.

A composite event contains references to its two sub-events. When a sub-event occurs it notifies the composite event by calling the "triggerComposite"-method which determines whether an occurrence of the composite event should be generated. The conjunction and sequence objects have histories, where they store event occurrences. Each event-object maintains its own history. Figure 9 shows a simplified model of the conjunction class.

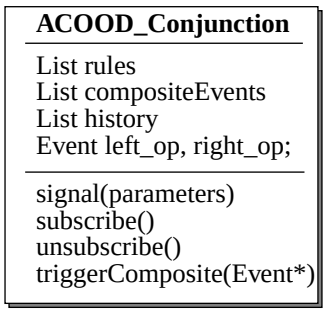

Fig. 9 The conjunction class

5.3 Rules

A rule object contains the names of the associated condition and action and a reference to the event object. The rule class defines methods for rule activation and deactivation. These methods call subscribe and un-subscribe respectively on the associated event. When the event occurs, the manager will invoke the “trigger”-method on the rule (if it is active).

All rules have an integer priority which is used for conflict resolution.

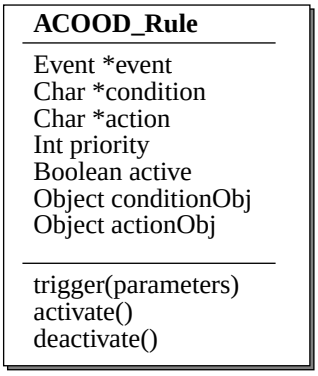

Fig. 10 The rule class

The object used for invocation is either the triggering object or the object specified in the rule definition (which will be referenced by “conditionObj” and “actionObj” respectively).

5.4 Parameters

The parameter objects are used to pass information on event occurrences to conditions and actions. A parameter object contains a reference to the triggering object and a pointer to the parameters (if the event was caused by a method invocation). Figure 11 shows a simplified model of the parameter class.

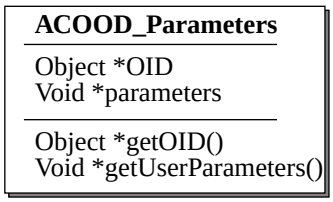

Fig. 11 The parameter class

5.5 The Rule Manager

The manager class is instantiated once. When an event occurs it tells the manager to start a collection of rules. The event reports all rules to the manager by using the “addRule”-method, and it signals all composite events. If a composite event is completed through this event, it reports its rules to the manager. Eventually all triggered rules have been found, and the original event calls the “endCollection”-method. All rules collected are sorted according to priorities and executed.

In the case of recursive rule triggering, the sequences are queued in a FIFO manner. A busy flag is used to indicate that the triggering of rules has started. All rules associated with one occurrence will be executed before any other rules are processed. Arbitrary recursion depth is possible. A simplified model of the manager class is shown in Figure 12.

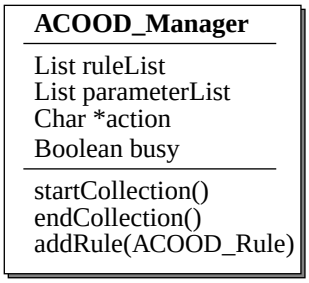

Fig. 12 The rule manager class

6 Experiences

Through the work with ACOOD we have gained experience on useful techniques and tractable features as well as difficulties and pitfalls. This section tries to summarize experiences from several projects dispersed over five years.

6.1 Useful Features

One implementation detail that has been shown to be useful for efficiency is the aforementioned rule indexing, i.e. letting each event contain information on which rules to signal upon event occurrence. In this way the overhead associated with rule triggering is minimized: only the affected rules are activated (i.e. retrieved from the database). If a rule is disabled the event is notified in order to remove the rule from the list of “subscribers”.

Another feature of ACOOD that has been shown to be useful is the dynamic creation of rules and events. In many systems (e.g. Samos, Ode, Sentinel) the event and rule definitions have to be done

before compilation, which means that rules may only be activated or deactivated (if the system allows it). By contrast ACOOD allows the user to add and delete events and rules at runtime, which enables the system designer to update the rulebase without time-consuming compilations. The reduction of compilation-time is one advantage but the major argument for dynamic rule creation is that it enables user-applications to use the rule-engine to perform user-tasks that are non-static. An example of this is given in [Hag96] where a CPS-negotiation is modelled by using rules.

When an event occurs, the parameters associated with the occurrence (e.g. the time and the object that caused the event) are collected and forwarded to the rules. These parameters can be used in the condition and the action and affect the behaviour of the rule. This has been shown to be quite useful as it increases the expressiveness of the rule language and makes it more fine-grained. In the example application above (Section 4.3 on page 12) the condition uses parameters to the method (that causes the event) to determine whether the rule should be triggered. In previous versions of ACOOD, that lacked event parameters, it was not possible to express a rule like this which is obviously a severe restriction. In addition, the possibility to limit composite event detection to occurrences originating from the same object has been shown to be useful. In this way it is possible to define which composite occurrences are of interest. This possibility could be further extended by letting the user specify other entities to use for restriction (e.g. same parameter).

Finally, an important observation from the usage of ACOOD is that the user-interface is important even in a research prototype. With a distinct user interface the DBMS-constructor is forced to define the responsibility of the DBMS-user which may reveal design errors and awkward solutions. The user interface also makes it easier to control and demonstrate the usefulness of the ADBMS.

6.2 Identified Difficulties

The development of ACOOD has given good insight into the difficulties of constructing an ADBMS. One of the key features in an active database is the ability to automatically detect event occurrences [ACT96]. In an OODBMS this includes the detection of method events which has proven to be a non-trivial task. In C++ a method is invoked by specifying an object, the name of a method, and possibly additional arguments. Internally this call is performed by locating the corresponding piece of executable code and invoking it with the object as the first argument. If such a call shall be trapped at runtime it requires that the location of the code is found and replaced with some wrapping instructions that generates the event and calls the original method. This is not easily performed as it requires manipulation of the internal look-up-tables.

The treatment of the method parameters causes additional problems. Firstly, there has to be some restrictions on the use of pointers as they are volatile and references to them in e.g. a condition may cause memory-violation-error if they no longer are valid. Secondly, there has to be some well defined way in which the parameters are referenced in the condition and action. If arbitrary signatures are allowed for methods that generate events, the arguments have to be packed and delivered to the rule in some way. All these problems indicate that event-parameters are complex to handle in an ADBMS.

In ACOOD an additional difficulty has been the layered-system architecture. With access to the source code of the DBMS it would have been easier to implement e.g. transaction events, detached coupling modes and method event generation.

7 Related Work

During the last decade a number of ADBMS prototypes have been constructed. This section gives a brief comparison between some of the active object-oriented database management systems

(AOODBMS). The intention is to focus on characteristics of the implemented prototypes rather than the "paper machines".

Ode [GJS92], [LGA96] is a project at AT&T research laboratories. Samos ([GD93], [GGD94], [GG+95]) was developed at the University of Zurich. Sentinel ([AMC93], [CK+94], [CK+95]) was developed at the University of Florida. Table 1 shows a comparison between the systems. The comments on each topic are very short and a column should be considered as a quick overview rather than a complete description of a system.

Table 1: Comparison of AOODBMS research prototypes

Feature	ACOOD	Ode	Samos	Sentinel
Is a DBMS	Yes, Ontos	Yes	Yes, ObjectStore	Yes, OODB
Has ECA-rules	Yes	Ode uses Triggers that can be used to simulate ECA-rules	Yes	Yes
Event Types	Method, explicit, conjunction, disjunction, sequence, same object restriction	Method, transaction, sequence, disjunction, conjunction, negation, relative	Method, temporal, transaction, abstract, conjunction, disjunction, sequence, negation, reduction, same object and same transaction restriction	Method, Temporal,explicit,disjunction,conjunction,sequence,aperiodic,periodic,not
Conditions	Method	Conditions are formulated as masks which can be any side-effect-free statement	Code fragment	Function
Actions	Method	Statement	Code fragment	Function
Rulebase management	Objects	Declarations	Objects	Objects
Rulebase Evolution	Dynamic creation of events and rules	Static (requires recompilation)	Static (compiled rules)	Static
Enabling and disabling of rules	Yes	Yes	No	Yes
Event detection	Automatic (Ontos "triggers")	Automatic	Automatic (method events have to be wrapped)	Automatic (post processor)
Binding modes	Instance-oriented	Instance-oriented	Instance-oriented	Instance-oriented

Table 1: Comparison of AOODBMS research prototypes

Feature	ACOOD	Ode	Samos	Sentinel
Evaluation of condition	OID, parameters	OID, parameters are available for mask evaluation	OID, parameters, transaction identifier	OID, parameters
Action Execution	OID, parameters	OID, parameters	OID, parameters, transaction identifier	Parameters
Coupling modes	Immediate	Immediate, independent, dependent, deferred	Immediate, deferred, decoupled in some versions	Immediate, deferred, detached
Consumption modes	Recent	Chronicle	Chronicle	Recent, chronicle, continuous, cumulative
Event history	Persistent local	Persistent global	Persistent global[a]	Transaction bound, local
Conflict resolution	Priorities	Arbitrarily	Priorities in some versions	Priorities
Programming environment	Browser	-	Browser, editor	Browser
Tunable	Benchmarking	Benchmarking	Benchmarking	-

a. only for unconsumed component events.

Ode uses triggers which are somewhat different from the ECA-rules used in the other systems. A trigger is bound to the class in which it is defined.

Note that benchmarking does not imply tunability but it may be an indication that performance is on the agenda.

An event history is considered to be local if each composite event operator maintains the information on event occurrences.

8 Conclusions and Future Work

8.1 Conclusions

This paper has explained the architecture and shown some implementation details of ACOOD, an active database management system. The background and history of ACOOD have been given together with a detailed description of the latest version. The system is simple but offers the functionality needed to implement ADBMS applications.

Great effort has been made to define the user interface in order to get a realistic environment for applications development. This work has shown that it is difficult to handle parameters from method events in a transparent way.

The ideal ADBMS would allow any method to be used as an event generator and the arguments would be automatically packed and delivered to the condition and action. It would handle pointers to dynamic structures even if they are moved or deleted. This would apparently cause a great overhead which implies that all ADBMS will make restrictions to the ideal solution. In ACOOD the limitations are quite strict, e.g. methods used for event generation, condition and action must have a certain signature. Parameters are not collected for all sub-events in a composite event and the number of consumption- and coupling-modes are reduced to a minimum. The support of composite events is in many ways minimalistic. Even with these strict limitations it is still possible to perform "non-trivial" tasks. A good verification of this is the work done in [Hag96], where ACOOD is used to implement a cooperative problem solving (CPS) scenario.

8.2 Future Work

There are several issues regarding the current system that need further investigation. These can coarsely be divided into three main areas: the functionality, the implementation and the verification of the system.

The most important extension of the functionality is to allow distributed events and rules, e.g. that events can be signalled from the clients to the server and vice versa. In this way rules can be executed in the environment where they logically belong. This will be in tune with the trend to move away from centralized systems towards distributed and federated computer environments.

More tools and utilities should be implemented to increase the usability of the system. With a pre-processor it is possible to release the application programmer from tasks such as the wrapping of active methods. Another possibility is to let the pre-processor pack the arguments for active methods, which will make more methods eligible for method event generation.

Graphics-based tools for different purposes, such as rule and event browsing, are desirable.

A good way to verify the correctness and usability of ACOOD is to use it for different kinds of applications. Cooperative problem solving, and computer integrated manufacturing are two areas where ACOOD has been and could be further used. The benchmarking of ACOOD is useful in looking to enhance performance, as it gives feedback on different implementation decisions.

Acknowledgements

A number of people have been involved in the ACOOD project since 1991. The authors are very grateful to Bo Brimark, Olof Jansson, Dennis Johansson, Joakim Eriksson, Andreas Eklund, Andreas Schuller, Wieland Schwinger, and Ivar Hagen for their contribution to the ACOOD project.

We would also like to thank Andreas Geppert and Daniel Lieuwen for their valuable comments on the comparison of AOODBMSs.

References

[ACT96] The ACT-NET Consortium. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. *ACM Sigmod Record*, vol 25(3), September 1996.

[AMC93] E. Anwar, L. Maugis and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. In *Proceedings of the International Conference on Management of Data,* pages 99-108, May 1993.

[BCL96] M. Berndtsson, S. Chakravarthy, and B. Lings. Cooperative Problem Solving: A New Direction for Active Databases. *International Symposium on Cooperative Databases for Advanced Applications*, Japan, December 1996.

[BCL97a] M. Berndtsson, S. Chakravarthy, and B. Lings. Result Sharing Among Agents Using Reactive Rules. In *Proceedings of the First International Workshop on Cooperative Information Agents (CIA-97)*, LNAI vol. 1202, Springer, pages 126-137, 1997.

[BCL97b] M. Berndtsson, S. Chakravarthy, and B. Lings. Task Sharing Among Agents Using Reactive Rules. In *Proceedings of the Second IFCIS Conference on Cooperative Information Systems (CoopIS-97)*, 1997.

[Ber91] M. Berndtsson. ACOOD: An Approach To An Active Object Oriented DBMS. *Master's thesis*, Department of Computer Science, University of Skövde, Sweden, 1991.

[Ber94a] M. Berndtsson. Management of Rules in Object-Oriented Databases. In *Proceedings of the Baltic Workshop on National Infrastructure Databases*, vol. 1, pages 78-85, Vilnius, Lithuania, 1994.

[Ber94b] M. Berndtsson. Reactive Object-Oriented Databases and CIM. In *Proceedings of the 5th International Conference on Database and Expert System Applications (DEXA'94)*, Lecture Notes in Computer Science, pages 769-778, Springer-Verlag, September 1994.

[BH95] M. Berndtsson and J. Hansson (editors). *Proceedings of the First International Workshop on Active and Real-Time Database Systems (ARTDB-95)*, Workshops in Computing Series, Springer-Verlag, 1995.

[BJ94] C. Bussler and S. Jablonski. Implementing Agent Coordination for Workflow Management Systems Using Active Database Systems. In *Proceedings of RIDE-ADS'94*, pages 53-59, 1994.

[BL92] M. Berndtsson and B. Lings. On Developing Reactive Object-Oriented Databases. *IEEE Quarterly Bulletin on Data Engineering, Special issue on active databases*, vol. 15 (1-4), pages 31-34, December 1992.

[BL95] M. Berndtsson, and B. Lings. Logical Events and ECA Rules. Technical Report HS-IDA-TR-95-004, Department of Computer Science, University of Skövde, 1995.

[BS83] D. G. Bobrow and M. Stefik. The Loops Manual, Intelligent Systems Laboratory, Xerox Corporation, 1983.

[BZ+95] A. P. Buchmann, J. Zimmermann, J. A. Blakeley, and D. L. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design, and Design Decisions. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 117-128, 1995.

[CB+89] S. Chakravarthy, B. Blaustein, A. Buchman, M. J. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Juahari, M. Livny, D. McCarthy, R. McKee and A. Rosenthal. HIPAC: A research project in active, time-constrained database management, Final Technical report. Technical Report XAIT-89-02, Xerox Advanced Information Technology, August 1989.

[Cha89] S. Chakravarthy. Rule Management and Evaluation: An active DBMS prospective. *SIGMOD Record, Special Issue on Rule Management and Processing in Expert Database Systems*, vol 18 (3), pages 20-28, Sept. 1989.

[Cha92] S. Chakravarthy (ed). Special Issue on Active Databases. *IEEE Quarterly Bulletin on Data Engineering, Special issue on active databases*, vol. 15 (1-4), December 1992.

[CK+93] S. Chakravarthy, and K. Karlapalem, S. B. Navathe, and A. Tanaka. Database Supported Cooperative Problem Solving. *International Journal of Intelligent and Cooperative Information Systems*, 2(3):249-287, November 1993.

[CK+94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proceedings of the 20th VLDB Conference*, pages 606-617, 1994.

[CK+95] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. Badani. ECA Rule Integration into an OODBMS: Architecture and Implementation. In *Proceedings of the Eleventh International Conference on Data Engineering*, 1995.

[CM94] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language For Active Databases. *Knowledge and Data Engineering Journal*, vol. 14, pages 1-26, 1994.

[CODA73] CODASYL Data Description Language Committee. *CODASYL Data Description Language Journal of Development*, June, 1973. NBS Handbook 113(1973).

[Day95] U. Dayal. Ten Years of Activity in Active Database Systems: What Have We Accomplished? In *Proceedings of the First International Workshop on Active and Real-Time Database Systems (ARTDB-95)*, Workshops in Computing, pages 3-22. Springer-Verlag, 1995.

[DKM86] K.R. Dittrich, A.M. Kotz, and J.A. Mulle. An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases, *Sigmod Record*, 15(3):22-36, 1986.

[Ekl95] A. Eklund. Performance Evaluation of an Active Database System. Master's thesis, Department of Computer Science, University of Skövde, Sweden, 1995.

[Eri93] J. Eriksson. CEDE: Composite Event Detector in an Active Object-Oriented Database, Master's thesis. Department of Computer Science, University of Skövde, Sweden, 1993.

[Esw76] K. P. Eswaran. *Specifications, Implementations, and Interactions of a Trigger Subsystem in an Integrated Data Base System.* IBM Research report RJ1820, August 1976.

[FM87] C. L. Forgy and J. McDermott. Domain-Independent production System Language. In *Proc. of the 5th International Conference on Artificial Intelligence*, Cambridge, MA, 1987.

[GD93] S. Gatziu and K. Dittrich. Events in an Active Object-Oriented Database System. In *Proceedings of the 1st Workshop on Rules in Database Systems*, pages 23-39, Edinburgh, August 1993.

[GGD94] S. Gatziu, A. Geppert, and K. Dittrich. The SAMOS Active DBMS Prototype. Technical Report 94.16, Department of Computer Science, University of Zurich, 1994.

[GG+95] A. Geppert, S. Gatziu, K. R. Dittrich, and H. Fritschi. Architecture and Implementation of the Active Object-Oriented Database Management System SAMOS. Technical Report 95.29, Department of Computer Science, University of Zurich, 1995.

[GJS92] N. Gehani, H. V. Jagadish and O. Smueli. Event Specification in an Active Object-Oriented Database. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 81-90, San Diego, June 1992.

[GM+96] A. Geppert, M. Berndtsson, D. Lieuwen, and C. Roncancio. Performance Evaluation of Object-Oriented Active Database Management Systems Using the BEAST Benchmark. Technical Report 96.07, Department of Computer Science, University of Zurich, 1996.

[Hag96] I. Hagen. Active Rules in Cooperative Problem Solving. Master's thesis, Department of Computer Science, University of Skövde, Sweden, 1996.

[Hew77] C. Hewitt. Viewing Control Structures as Patterns of Messages, *Artificial Intelligence*, vol. 8, pages 323-364, 1977.

[HS87] S. Hedberg and M. Steizner. Knowledge Engineering Environment (KEE) System: Summary of Release 3.1, Intellicorp Inc, Mountain View, CA, July 1987.

[LGA96] D.F.Lieuwen,N. Gehani, and R.Arlein. The Ode Active Database: Trigger Semantics and Implementation. In *Proc. of the International Conference on Data Engineering*, New Orleans, 1996.

[Min75] M. Minsky. A Framework for Representing Knowledge. The Psychology of Computer Vision (P. Winston, ed) McGraw-Hill, New York, 1975.

[Mor83] M. Morgenstern. Active Databases as a Paradigm for Enhanced Computing Environments. In *Proc. of the 9th International Conference on VLDB*, pages 34-42, 1983.

[Ont94] Ontos Inc. *Ontos DB 3.0 documentation.* 1994.

[PW93] N. W. Paton and M. W. Williams (eds). Rules in Database Systems. *Proceedings of the 1st International Workshop on Rules in Database Systems (RIDS'93)*, Workshops in Computing, Springer-Verlag, 1995.

[Schu96] A. Schuller. *Inheritance of Events and Rules in Active Object-Oriented Database Management Systems*. Edition Wissenschaft, Reihe Wirtschaftswissenschaften, Bd. 79, Tectum Verlag Marburg, ISBN 3-89608-179-9, 1996.

[Schw95] W. Schwinger. Logical Events in ECA Rules. Master's thesis, Department of Computer Science, University of Skövde, Sweden, 1995.

[Sel95] T. Sellis (ed). Rules in Database Systems. *Proceedings of the 2nd International Workshop on Rules in Database Systems (RIDS'95)*, Lecture Notes in Computer Science 985, Springer-Verlag, September 1995.

[SJ+90] M. Stonebraker, A. Jhingran, J. Goth and S. Potamianos. On Rules, Procedures, Caching and Views in Data Base Systems. In *Proc. of the ACM SIGMOD Inter national Conference on Management of Data*, pages 281-290, Atlantic City, New Jersey, May 1990.

[WC95] J. Widom, S. Ceri (ed). *Active Database Systems- Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, ISBN 1-55860-304-2, 1995.

[Wid94] J. Widom. Research Issues in Active Database Systems: Report From the Closing Panel at RIDE-ADS'94. *ACM SIGMOD Record*, 23(3):41-43, September 1994.

Appendix A- ACOOD User Functions

ACOOD_errorCode **ACOOD_defineMethodEvent**(char*eventName, char*methodName, ACOOD_triggerType mode)

ACOOD_errorCode **ACOOD_defineExplicitEvent**(char *eventName)

ACOOD_errorCode **ACOOD_raiseEvent**(char *eventName, void *parameters=NULL,OC_Object *OID=OC_null)

ACOOD_errorCode **ACOOD_deleteEvent**(char *eventName)

ACOOD_errorCode **ACOOD_defineCompositeEvent**(char*eventName,char*leftEvent,char*rightEvent,
ACOOD_operatorType theOperator, ACOOD_consumptionModeType mode=RECENT,
ACOOD_compsiteRestrictionType restriction=NONE)

ACOOD_errorCode **ACOOD_defineCompositeEvent**(char *eventName,char **events, ACOOD_operatorType
theOperator, ACOOD_consumptionModeType mode=RECENT,
ACOOD_compsiteRestrictionType restriction=NONE)

ACOOD_errorCode **ACOOD_defineRule**(char *ruleName, char *eventName, char *conditionName, char *actionName,
int priority=0)

ACOOD_errorCode **ACOOD_activateRule**(char *ruleName)

ACOOD_errorCode **ACOOD_deactivateRule**(char *ruleName)

ACOOD_errorCode **ACOOD_deleteRule**(char *ruleName)

void ***ACOOD_getMethodParameters**(ACOOD_Parameters *theParameters)

OC_Object ***ACOOD_getObjectID**(ACOOD_Parameters *theParameters)

void **ACOOD_displayError**(ACOOD_errorCode error)