Architecture and Implementation of the Active Object-Oriented Database Management System SAMOS

Andreas Geppert, Stella Gatziu, Klaus R. Dittrich, Hans Fritschi, Anca Vaduva

Institut für Informatik, Universität Zürich
Switzerland[1]

Technical Report 95.29

Abstract

The architecture and the implementation of the active, object-oriented database management system SAMOS[2] are described. SAMOS is a research prototype extending a commercial object-oriented database management system with reactive behavior specified in the form of event-condition-action rules. This paper describes the various components of SAMOS for event detection, rule management, rule execution, and transaction management, and explains how they work and interact upon rule definition and execution. The current architecture of SAMOS is evaluated, experiences with the layered approach are described, and drawbacks are identified.

Keywords: Active Database Systems, Object-Oriented Database Systems, DBMS-Architecture, ECA-rules

1. Authors' address: Institut für Informatik, Universität Zürich, Winterthurerstr. 190, CH-8057 Zürich, Switzerland. Email: {geppert | gatziu | dittrich | fritsch | vaduva}@ifi.unizh.ch
2. **S**wiss **A**ctive **M**echanism-Based **O**bject-Oriented Database **S**ystem

Contents

1 Introduction

Active database management systems (ADBMSs) [e.g., 7, 14, 17, 48] support the specification and implementation of reactive behavior. This functionality is commonly defined in terms of event-condition-action rules (ECA-rules) [17]. The meaning of such a rule is “if the event occurs and the condition holds, execute the action”. Numerous concepts for systems and rule definition languages have been proposed [e.g., 4, 9, 11, 12, 19, 23, 30, 40, 42, 47], whereas less experience exists for the design and implementation of an operational entire active DBMS. Initial approaches, mostly for relational systems, assume rather simple rule definition and/or rule execution facilities [40, 42, 47], but knowledge about the design of an active DBMS with powerful rule definition facilities together with elaborate execution mechanisms (such as nested rule execution) is mostly missing. This is particularly true for *object-oriented*, active DBMSs.

Most approaches for implementing an active DBMS make use of an existing DBMS or parts thereof (ACOOD [2], REACH [5], Sentinel [10], NAOS [11], Ode [23], RDL [40], POSTGRES [42], TRIGS [28], Starburst [47]). Some of them *customize* and *modify* the base system (NAOS, Ode, Postgres, REACH, Sentinel, Starburst), while others add active functionality *on top* of the base system and treat the used DBMS as a black box ([31], ACOOD, RDL, TRIGS) (*layered approach*). It is thus not only an open question how to design an ADBMS in general, but also which architectural style to apply in particular.

This paper investigates the architecture of the active DBMS SAMOS [18] and reports on experiences with a layered approach. In a nutshell, these experiences are as follows:

- Given an appropriate passive base DBMS, the layered approach is beneficial in terms of construction cost, since all the passive functionality can be reused, and newly implemented components can make use of it. The disadvantage is the lacking opportunity to optimize the entire system comprehensively, i.e., runtime performance may be worse than in an integrated architecture.
- The layered approach is beneficial for active *object-oriented* DBMSs if the base system is in turn implemented in an object-oriented way such that functionality to be rewritten can be easily modified or wrapped.

The contribution of this paper is thus twofold. First, it contributes to the knowledge on ADBMS-architectures. Second, to the best of our knowledge, SAMOS is among the first full-fledged operational object-oriented ADBMSs, and this paper is the most comprehensive presentation of a layered ADBMS-architecture so far. It thus helps to evaluate and assess the various architectural styles from different perspectives (functionality, construction cost, runtime performance).

The remainder of the paper is structured as follows. The next section investigates architectural styles for ADBMSs and surveys related work. The subsequent section then deals with specifying reactive behavior in SAMOS. Section 4 describes the SAMOS architecture, followed by a brief description of how applications work with SAMOS. Section 6 contains an evaluation of the SAMOS architecture and implementation. Section 7 concludes the paper.

2 The Architecture of Active Database Management Systems

In this section, we describe requirements and architectural approaches for ADBMSs in general and address related work.

2.1 Requirements

The following list shows which features a database management system should implement in order to be called “active” [17].

1. *An ADBMS must be a DBMS*. All the concepts required for a passive system are required for an ADBMS as well (data modeling facilities, query language, multi-user access, recovery, etc.). That means, if a user ignores all the active functionalities, an ADBMS can be worked with in exactly the same way as a passive DBMS. We then refer to the components realizing the non-active part of the DBMS as “passive components”.
2. *An ADBMS supports definition and management of ECA-rules*. Events (of various types), conditions, and actions should be definable at the ADBMS interface, and the ADBMS should manage a catalogue of defined rules.
3. *An ADBMS must detect event occurrences*. The ADBMS has to implement event detection for all the types of events that can be defined.
4. *An ADBMS must be able to evaluate conditions and to execute actions*. This requirement means that the ADBMS has to implement rule execution.

These features comprise the mandatory features of an ADBMS. Additional optional features improve the usability of an ADBMS; they refer to appropriate tool support for developing and maintaining ADBMS-applications. Starting with section 4, we describe how the mandatory features and some of the optional ones are actually implemented in SAMOS.

2.2 Architectural Styles

The following architectural approaches for ADBMSs exist:

- implementation from scratch,
- integrated architecture,
- layered architecture.

In the first approach, the ADBMS is implemented from scratch, i.e., all the passive components are also newly implemented. This approach is obviously the most costly one.

The second possibility is to use an existing passive DBMS and to modify and extend it internally. Internal modifications require the availability of the source code. Modifications can in principle be made at any place of the DBMS where necessary. The price for this high degree of freedom is that a profound understanding of the architecture and implementation of the passive DBMS is indispensable. For systems as complex as a DBMS, this is usually hard to gain for people who have not built these systems in the first place. Therefore, this approach is typically only feasible in cooperation with the designers/implementors of the passive system.

Some approaches to integrated architectures use customizable DBMSs or DBMS-toolkits. A *customizable* DBMS is a complete DBMS that can be tailored to meet new requirements. Most functionality is predefined (e.g., the data model), while enhancements are possible at well-defined places in the system. This approach is feasible whenever all required customizations are possible at all. Conversely, it is not beneficial whenever the necessary extensions are not supported by the customization framework.

A *DBMS-toolkit* defines an architecture model and a set of libraries whose elements implement certain functionalities of a DBMS. Using a toolkit, reusable components for some tasks of a DBMS can be selected from a library. Functionality that is not covered by any of the library elements must be implemented conventionally. Both, reused and newly implemented components together with some “software glue” are then configured into an entire DBMS. This approach is in general more flexible than using a customizable DBMS. It is also intended to provide for a higher degree of reusability of existing components. The approach is feasible in its ideal form: then most of the components (including the passive parts) needed for the ADBMS are already available from previous constructions and can be reused; they thus need not be newly implemented. It is less beneficial when most of the desired components do not yet exist and thus have to be implemented manually[3].

The last possibility is to use an existing passive database management system and to implement the active behavior on top of this system. In contrast to the other approach, the reused system is just used as a black box and cannot be modified internally. In contrast to the toolkit approach, the entire passive DBMS is reused instead of single components.

2.3 Architecture of Existing ADBMSs

Initial work on the architecture of active DBMSs has been done in the HIPAC project [35] beginning in the mid 80ies. Since then, many proposals for rule definition languages,

3. This is a well-known situation in software reuse in general: reusability starts to pay off only after a significant number of constructions have been performed and the collection of reusable components is then sufficiently large [32].

event detection techniques, and rule execution models have been made. Nevertheless, prototypical implementations and design documentations are available for a few active DBMSs only. Particularly, descriptions of operational ADBMSs are rather scarce.

Postgres and Starburst are relational ADBMSs; they have customized an existing DBMS [42, 47]. In this way, internal interfaces (such as *attachments* in the Starburst case [34]) can be used and internal components can be modified if needed. RDL [40] is another relational system which however has been implemented in the layered style.

Layered approaches for object-oriented systems have been used in [31], in ACOOD, and in TRIGS [28]. Initial work in that area has been reported in [31], where the implementation of an active mechanism (conceptually similar to our proposal) on top of the object-oriented DBMS GemStone [6] is described. In this work, it has not been possible to reflect the precise meaning of coupling modes and rule execution (including intra-transaction savepoints) since GemStone does not provide for nested transactions.

The architecture of the Monet ADBMS [29] is classified as a layered one, since Monet is implemented on top of the Goblin Database Kernel [3]. The active component of Monet provides solely basic support for ECA-rules (i.e., a restricted set of event types and coupling modes). However, since Monet is intended to be a customizable DBMS, more powerful concepts can be implemented using Monet's primitives and can be made available at the end-user interface. Thus, in contrast to other systems surveyed here, Monet itself including its active component is a *customizable DBMS*, where extensibility is supported by the Monet extension language.

Integrated object-oriented ADBMSs that have been constructed by internal modifications of a passive OODBMS are NAOS and Ode. NAOS [11] extends the OODBMS O_2 [15]. Similar to [31], the problem in NAOS has been to implement the precise meaning of rule execution (i.e., coupling modes), since O_2 does not provide for nested transactions. Ode [23] is another example where the passive and the active DBMS are implemented by the same group of developers.

In the newest version of the REACH ADBMS [5], the DBMS-toolkit OPEN OODB [46] has been used. In this way, it is possible to customize DBMS-internal components such as transaction management to the needs of the ADBMS under construction. REACH supports some functionality that is not offered by SAMOS (more coupling modes), it has also traded some functionalities for performance (e.g., rules triggering upon composite events cannot be executed in immediate mode). In order to achieve better performance, multi-threaded transactions have been used for composite event detection in REACH. Nested transactions, however, are not yet supported in REACH.

Sentinel [10] also extends the Open OODB system. It uses some of the modules provided by Open OODB (e.g., the object manager, the persistence manager), but also adds new modules for other tasks (transaction manager, modules for primitive event detection and composite event detection). In contrast to REACH, it supports multiple threads and nested transactions. In Sentinel, therefore, concurrent execution of multiple rules within multiple threads is possible.

3 An Overview of Rule Definition in SAMOS

This section gives an introduction of rule definition facilities in SAMOS, as far as necessary for understanding the rest of the paper. Details of rule specification have been described in [18]. Details of the event definition part of the rule definition language are further presented in [22].

SAMOS is implemented on top of an object-oriented DBMS (the "underlying system"). Hence, when considering the passive part of SAMOS only, it is an object-oriented DBMS. It thus provides for classes, objects with identity, inheritance, methods, transactions, etc. In addition to the data definition language of the underlying system, SAMOS provides a *rule definition language* as a means to specify ECA-rules. It includes constructors for the definition of events, conditions, and actions. Events can be defined separately and used in multiple rules. Rule and event definitions in SAMOS have the following form:

```
DEFINE RULE       <rule_name>
ON                <event_clause>
IF                <condition>
DO                <action>
COUPLING MODE "(" <coupling>, <coupling>")"
PRIORITIES        (BEFORE | AFTER) <rule_name>

DEFINE EVENT      <event_name> <event_def>
```

Figure 1: Part of the Rule Definition Syntax in SAMOS

Rule or event definitions specify *event descriptions*. Event descriptions can be defined implicitly in rules, or explicitly using the `DEFINE EVENT` clause. An event description can be regarded as the intension of a set of similar event instances of interest. Instances of event descriptions are the actual events happening at a given point in time; they are also called *event occurrences*. Subsequently, we simply refer to "events" whenever it is clear whether event *descriptions* or event *occurrences* are meant.

Event descriptions can be *primitive* or *composite*. SAMOS distinguishes the following kinds of primitive events:

- *method event*: occurs at the beginning or the end of a method execution,
- *transaction event*: occurs before or after a transaction operation (begin, commit, or abort transaction),
- *time event*: occurs at a specific point in time (absolute time event), periodically (periodical time events), or as soon as a specified time interval following another event occurrence has elapsed (relative time events), and
- *abstract event*: "occurs" when explicitly signalled from outside the ADBMS (by the user or application).

Event descriptions are parameterized. The actual parameters that represent relevant information are bound to the formal ones after the event occurrence. The set of formal parameters are fixed (except for abstract events). SAMOS supports the following event parameters:

- *occurrence time*: the point in time when the event occurred (`occ_time`),
- *transaction*: the transaction in which the event occurred,
- *object*: the object to which the message has been sent in case of method events,
- *user*: the user who executed the transaction which contained the event occurrence.

Not all these parameters are applicable for all the kinds of event descriptions (see Table 1).

event description	occ_time	object	transaction	user
method	x	x	x	x
transaction	x	-	x	x
time	x	-	-	-
abstract	x	-	x	x

Table 1: Event parameters of primitive events

In addition to the receiver object, method events have the arguments to the method call as parameters.

Furthermore, *monitoring intervals* can be specified for event descriptions. A monitoring interval defines a start and an end point (e.g., time events). If a monitoring interval is given for an event description then only those occurrences are considered that happen within the monitoring interval.

Composite events occur if one or more other (primitive or in turn composite) component events have occurred. Composite event descriptions can be specified by one of the following constructors:

- *conjunction*: occurs when both component events have occurred; its `occ_time` corresponds to the `occ_time` of the component event that happens second,
- *disjunction*: occurs when one of the two components has occurred; its `occ_time` corresponds to the `occ_time` of the component event that happens,
- *sequence*: occurs when the two component events have occurred in the specified order; its `occ_time` corresponds to the `occ_time` of the second component event,
- *negation*: occurs when the component has *not* occurred within a specified time interval; its `occ_time` corresponds to the end point of the interval,
- *times*: occurs when the component event has occurred a given number of times (say `n`) within a certain time interval; its `occ_time` corresponds to the `occ_time` of the `n`-th component event,
- *closure*: occurs when the component event has occurred for the first time within a specified time interval; it is signalled exactly once if the component event occurs at least once. Its `occ_time` corresponds to the `occ_time` of the first component event.

Monitoring intervals are mandatory for negation but optional for the other cases. Furthermore, event parameters are relevant for composite events, too. They can be used to define additional constraints for a composite event and its components. It can be specified in the composite event description that all of its components have to occur

- within the same transaction (`same transaction` restriction),
- within transactions started by the same user (`same user` restriction),
- for the same object (`same object` restriction), which is applicable only for method events as components.

After an event has been detected, the conditions of related rules are checked. If they hold, the corresponding actions are executed. Both, conditions and actions must be specified in the data manipulation language (DML) of the underlying ooDBMS.

The transaction where an event occurrence happens is called the *triggering transaction* of the event. The transaction that is executed in order to perform condition evaluation and action execution is called the *triggered transaction*. The temporal and causal relationships between these two kinds of transactions are specified using *coupling modes* which define when the triggered transaction is executed with respect to the triggering one, and whether abort/commit dependencies exist between them. Each rule defines two coupling modes: one for the event-condition coupling, and one for condition-action coupling. Possible modes for event-condition coupling are

- *immediate*: the condition is evaluated directly after the triggering event has been detected and within the same transaction,
- *deferred*: the condition is evaluated at the end of the triggering transaction, but before commit,
- *decoupled*: the condition is evaluated in a separate transaction which will be scheduled independently by the ADBMS.

For condition-action coupling, the coupling modes have the following meaning:

- *immediate*: the action is executed directly after the condition evaluation and within the same transaction,
- *deferred*: the action is executed at the end of the triggering transaction, but before commit,
- *decoupled*: the action is executed in a separate transaction which will be scheduled independently by the ADBMS.

It may be the case that multiple rules are defined for the same event and with the same coupling mode (`immediate` or `deferred`). In this case, priorities define the order to be imposed on the execution of the rules. *Priorities* form a partial order on rules. Rules that are not (transitively) ordered by priorities are executed in an arbitrary (system-determined) order.

Note that events can also occur during rule execution, i.e., rules can trigger further rules. Thus, SAMOS is able to perform *nested rule execution*.

4 The SAMOS System Architecture

In this section, we describe the SAMOS system architecture. SAMOS is operational on SUN machines running under the SUNOS[4] operating system [44] (i.e., SUN's Unix[5] operating system).

SAMOS consists of three layers (Figure 2):

- the object-oriented DBMS ObjectStore[5],
- the SAMOS kernel implementing the active functionality, and
- a set of tools supporting users in applying the active functionality of the SAMOS kernel.

4. SUNOS is a registered trademark of SUN Microsystems. Unix is a registered trademark of AT&T.
5. ObjectStore is a registered trademark of Object Design Inc.

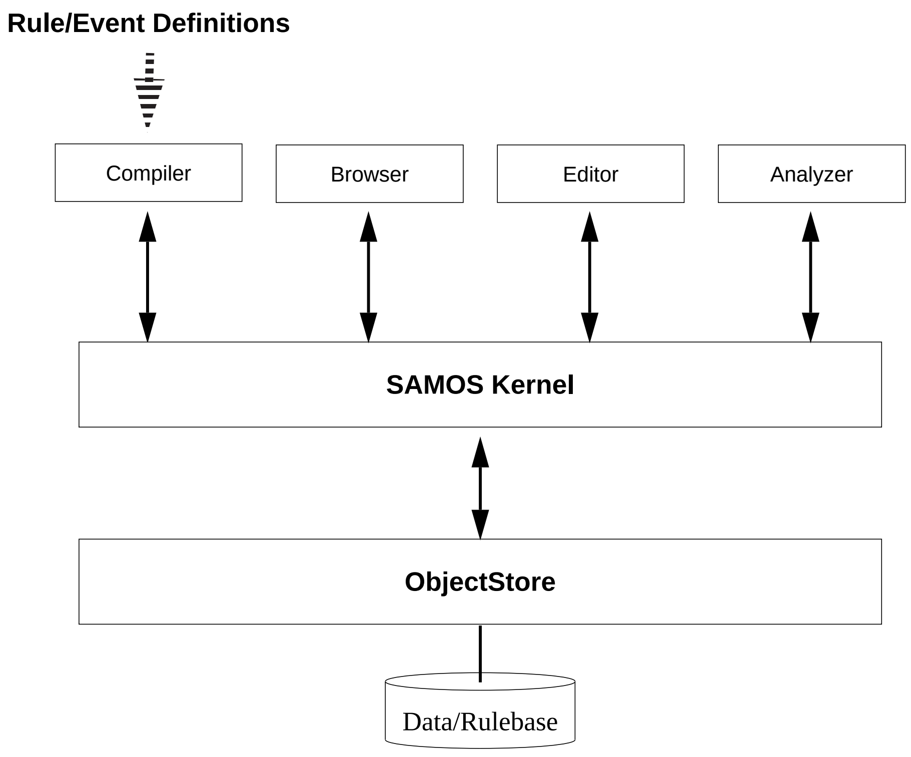

Figure 2: The Architecture of SAMOS

The three layers together implement the requirements given in section 2.1, i.e., the mandatory features of an ADBMS and some of the optional ones.
In the following, each layer will be discussed in detail.

4.1 ObjectStore

ObjectStore [33] is a commercial OODBMS. It extends C++ [43] with database functionality (most important: persistence, a query processor, indexing, clustering, and transaction management).

Specifically, ObjectStore supports closed nested transactions [36]. Nested transactions can in turn execute transactions (which are then called subtransactions). Transactions that do not contain subtransactions (but only database operations) are called *leaf transactions*. Transactions that are not subtransactions of any other transaction are called *root transactions* (*top-level transactions*). Changes performed on behalf of subtransactions become permanent only if and when the root transaction commits. If a nested transaction aborts, all its subtransactions are rolled back as well.

Subtransactions block their parent, and ObjectStore does not support parallelism among sibling transactions (i.e., ObjectStore provides for neither sibling parallelism nor for parent/child parallelism [27]).

4.2 The SAMOS Kernel

The most important components of the SAMOS kernel are (Figure 3):

- a *rule manager* for the storage and retrieval of information about event and rule definitions,
- a *detector* for composite *events*,
- a new class `samTransaction` for SAMOS' own transaction management on top of ObjectStore,
- a *rule execution component* for condition evaluation and action execution.

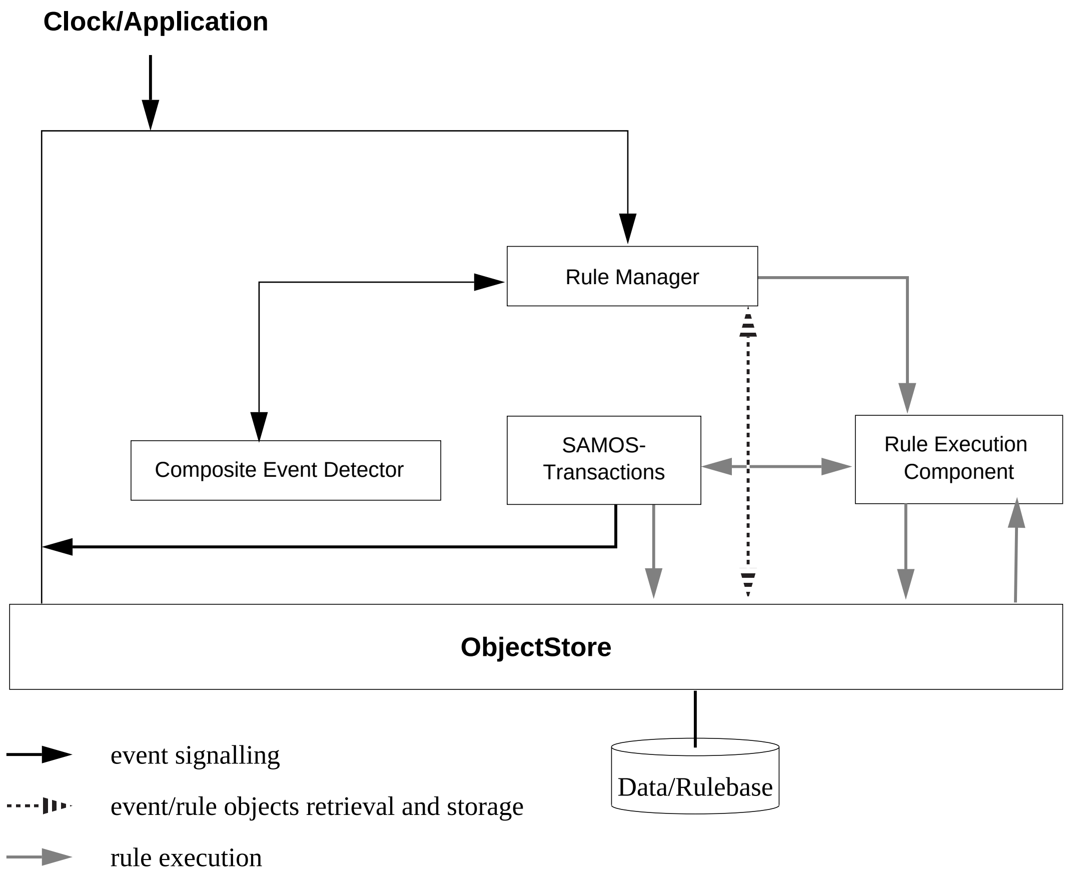

Figure 3: The Component Architecture of the SAMOS Kernel on top of ObjectStore

4.2.1 The Rule Manager

The rule manager is responsible for persistently storing and retrieving information on rules and events. It is represented by an object, an instance of the class `CRuleManager`. Event descriptions and rules are also represented as objects [18][6], called *event objects* and *rule objects*, respectively. They are stored in the so-called *rulebase*, which in turn is a part of the database. Both, event and rule objects have an identity and the rule manager can manipulate and access them like any other object by means of methods. Since all these objects are stored in ObjectStore and manipulated within transactions, atomicity and durability is automatically supported for operations on the rulebase.

Event and rule objects are created using the methods offered by the rule manager (see Figure 4). These methods are used by the rule compiler if the rule definition language has been used to specify rules and events. In this case, rule and event objects are created automatically. Alternatively, a programmer can create these objects “manually” if she uses the interface of the rule manager for the definition of rules instead of the rule definition language.

Event and rule objects are persistently stored in so-called *event extents* and *rule extents*. In ObjectStore terminology, an extent is a collection of currently existing, persistent instances of a specific class. Hash indexes are defined for the event and rule extents. Furthermore, all event objects are clustered in an ObjectStore segment, which also contains

```
class CRuleManager
{
  public:
// some methods for primitive event creation
     TStatus DefAbstrEvent(char EventName[]);
     TStatus DefTransEvent(char EventName[], TTransMode Modus);
     TStatus DefMethEvent(char EventName[], char className[], char MethName[],
                                            char HeaderFileName[], char ImplFileName[],
                                            eventBeforeAfter BeforeAfter);
     TStatus DefAbsTimeEv(char EventName[], time_t occp);
// some methods for composite event creation:
     TStatus DefDisj(event* ev1, event* ev2, char disj[]);
     TStatus DefConj(event* ev1, event* ev2, char conj[], TSameClause option);
//method for rule creation
     TStatus DefineRule(char RuleName[], event* Ev, condition* Cond,
                                action* Act, couplingMode eccMode, couplingMode cacMode);
}
```

Figure 4: Class Definition of the Rule Manager

6. This is similar to the proposals in [1, 4, 13, 16].

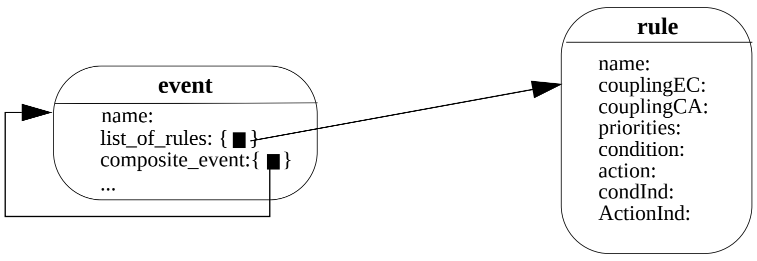

Figure 5: A Part of the Rule Schema

the event extent index. These measures are applied in order to accelerate the access to event objects.

4.2.1.1 Storage of Event Descriptions

All user-defined event descriptions are instances of the class `event` (Figure 5). Event objects can be created, updated and accessed through methods implemented by the rule manager (Figure 4). Various subclasses of `event` are defined for the different kinds of event descriptions (e.g., method events, conjunctions, and so forth). A detailed description of this class hierarchy can be found in [20]. Each event has the attribute `name` (user- or system-defined). The attribute `list_of_rules` determines the rules (i.e., references to objects of the class `rule`) that have to be executed when this event has occurred. Other systems (e.g., ADAM [16], Ode [23]) associate the rule with the objects or classes on whose operations the events are defined. However, this approach is feasible only for (some kinds of) primitive events (e.g., method events), because only then such a rule association exists. In SAMOS we thus associate rules with event descriptions. This approach is more general since it can be used for all sorts of events, primitive and composite ones.

Whenever there are multiple rules triggering upon a specific event, SAMOS has to decide on the order of rule execution upon an occurrence of this event. This execution order is constrained by the coupling modes and can be further determined by user-defined priorities. If there are still multiple possibilities, the execution order is determined by SAMOS. In order to avoid "sorting" of rules at runtime, the structure of `list_of_rules` already reflects execution constraints (condition evaluation coupling mode). For each event, rules with `decoupled` mode are stored at the top of the list, since their triggering transactions are spawned first. The second group of rules are those with `immediate` conditions. `Deferred` rules are stored at the end of this list. In the latter two cases, rules with the same coupling modes are ordered according to their priorities.

Finally, the attribute `composite_event` determines the composite events in which this event participates.

4.2.1.2 Storage of Rules

Rule definitions are stored as instances of the class `rule` (rule objects). The attributes of rule objects include a name, the coupling modes, priorities, and the condition as well as the action. For the latter two, references to files containing the source text of conditions and actions are stored in addition to the storage of these texts in rule objects (attributes `condInd` and `actionInd`). This redundancy has been chosen in order to support rule modification and browsing (using the information in rule objects) as well as compilation of rule implementations equally well (using the source text in files).

Upon rule execution, the executable code fragments for conditions and actions must be found and executed. The textual representation is of no use in this situation. Additionally, it is not feasible to call functions by their name and let the runtime system find the correct main memory address in the linked program code. Since the names of conditions and actions as well as the number of them is not known at the compilation time of the SAMOS kernel, the rule manager itself would have to be re-compiled upon each rule definition, deletion, or modification. In other words, the problem is to enable modifications of ADBMS-behavior at runtime under the condition that the programming language compiler and linker cannot be extended. Note that dynamic linking, on the other hand, is not covered by the C++ standard up to now, and is therefore not considered as a possibility either.

The solution of SAMOS is as follows (see Figure 6). A function type including a fixed set of formal parameters (i.e., the set of all C-functions with a given set of formal parameters) is defined for conditions and actions, respectively. For each condition and action specified in a rule, a C++-function is generated whose signature conforms to that function type and whose body implements the condition (action). The source text of these C++-functions is stored consecutively in two files. References to the functions are stored in an array of function pointers. The index of a function pointer in this array is recorded in an attribute value of the corresponding rule object (the attributes `condInd` and `actionInd` in Figure 5). In order to execute a function, its index has to be found through the corresponding rule object, the function pointer has to be retrieved from the function pointer array, and the pointer finally has to be de-referenced.

In this approach, the following tasks have to be performed by SAMOS whenever a new rule is defined:

- extension and compilation of the condition and action files,

- extension and compilation of the function pointer array,
- linking of the modified parts with the SAMOS kernel.

4.2.1.3 Event and Rule Retrieval

Apart from storing event and rule objects, the rule manager is also responsible for retrieving these objects efficiently.

Retrieval of event objects takes place after the occurrence of one or more events have been *signalled*. The rule manager is informed about event occurrences through the receipt of the message *raise_event*. The various event detectors are the possible senders of this message (see section 4.2.3). The rule manager then reacts by performing the following actions:

- retrieval of the corresponding event objects,
- invocation of the composite event detector,
- retrieval of rule objects, and eventually invocation of the rule execution component.

After an event has been signalled, the rule manager is responsible for determining rules to be executed, and composite events the occurred event participates in. This information is available on the level of *event descriptions*. Therefore, upon signalling of an event, the

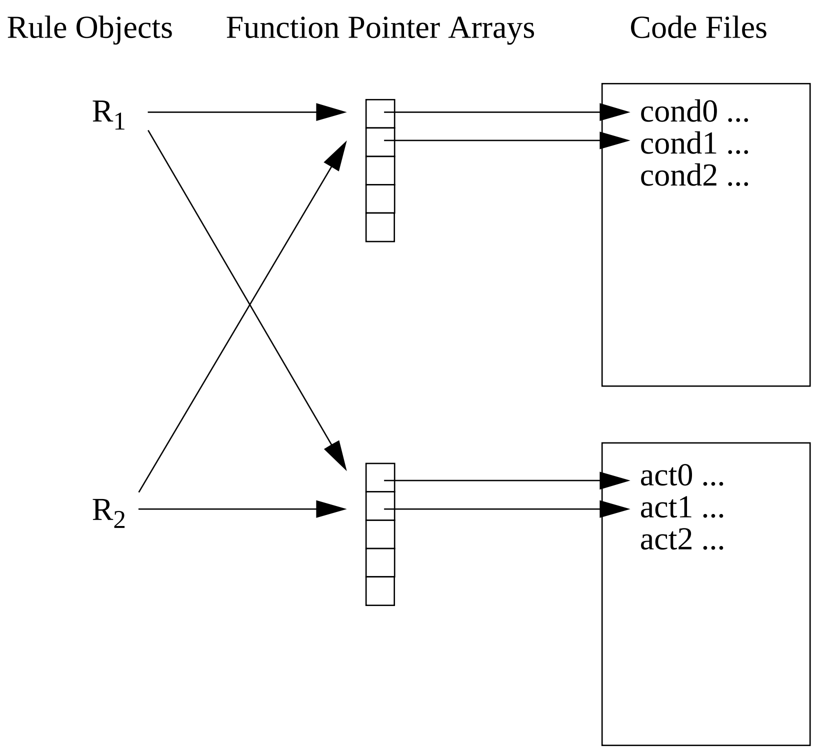

Figure 6: Management of Code Fragements for Conditions and Actions

rule manager has to find the appropriate event description, i.e., the appropriate event object. Parameters of the event signalling function (`raise_event`), such as the event description name, are used for this.

In general, the rule manager will retrieve exactly one event object for the signalled event occurrence. In special cases, however, such a corresponding event object might not exist, because neither a rule has been defined for the event nor does the event participate in a composite event. We call such events "non-interesting" events. Obviously, the signalling of non-interesting events should be avoided for efficiency reasons. In SAMOS, for some sorts of primitive events only interesting events are signalled (e.g., for method events and time events); however, this is not possible for transaction events.

Let us now assume that an event object has been found. In this case, the rule manager next determines whether the event participates in a composite event (expressed through references in the attribute `composite_event`). If yes, the composite event detector is informed about the event occurrence.

Furthermore, the rule manager determines whether a rule is defined for the event, which is expressed through the attribute `list_of_rules`. If yes, the set of rules defined for the event is retrieved. The rule execution component is invoked with this list of rules as a parameter.

Efficient retrieval of event objects is a crucial performance requirement, especially if numerous event descriptions have been defined. In SAMOS, access to the set of event objects is tried to be optimized in three ways:

- retrieving event objects only for interesting events,
- using indexing techniques for the set of event objects,
- clustering event objects together.

4.2.2 SAMOS Transactions

In ObjectStore, each transaction is represented as an instance of the class `os_transaction`. Operations on (sub)transactions are implemented as methods of this class (e.g., `os_transaction::begin` starts a new transaction). For use in SAMOS, the transaction management of ObjectStore must be extended for two reasons:

- transaction events must be signalled,
- rules must be executed before commit in case of the `deferred` coupling mode.

Since it is not possible to override the transaction statements in ObjectStore, SAMOS defines a new class `samTransaction` (Figure 7). This class defines methods to start, commit, and abort transactions. These three methods implement the transaction functionality using

```
class samTransaction {
    public:
    os_transaction       *ostx;             // pointer to ObjectStore transaction
    int                  user;
    int                  level;
    char                 * name;
    os_list<ruleExec*>list_of_rules;        // transaction rule register. ruleExec contains
    ...                                     // a flag "condYetEval" and a pointer to a rule
    samTransaction (int mode);              // the constructor, starts a samTransaction
    static samTransaction begin;            // starts a samTransaction
    static commit(samTransaction * stx);    // commits a samTransaction
    static abort(samTransaction * stx);     // abort a samTransaction
    addRule(Rule * newRule, int condYetEval);
                                            // add newRule to transaction rule register
};
```

Figure 7: Definition of Class `samTransaction`

ObjectStore's transaction operations, but also add code in order to implement event signalling and rule execution properly.

SAMOS requires applications to use the functions of this class instead of those provided by ObjectStore. In other words, SAMOS users are not allowed to use ObjectStore transactions directly. Otherwise, transaction events would not be signalled, and correct rule execution would not be possible.

Each `samTransaction` instance contains a reference to an associated ObjectStore transaction (`ostx`). The instance variable `list_of_rules` of class `samTransaction` references rules with the `deferred` mode which have to be executed before this transaction commits. Furthermore, each element of this list contains a flag determining whether the condition has already been evaluated.

The operation `raise_event` is invoked within each transaction operation (`begin`, `commit`, `abort`). In this way, transaction events are signalled. In addition, the respective operations of class `os_transaction` are called.

Finally, the attribute `level` is used to restrict the height of transaction trees for the following reason. Since SAMOS provides for nested rule execution, it may happen that several rules mutually trigger each other and thus rule execution does not terminate. Since conditions and actions are in turn executed within (sub)transactions, a transaction tree of indefinite depth would be the result. Therefore, restricting the depth of transaction trees breaks cycles during rule execution. Rule execution is thus guaranteed to terminate at least for those cycles that do not contain rules with `decoupled` mode.

The database administrator (DBA) can specify an upper limit for the nesting depth of transactions. For each transaction, SAMOS records its depth (i.e., the number of ancestors it has) in the `level` attribute, and the start of a new transaction is only allowed if its level value does not exceed the limit specified by the DBA. If no limit has been specified by the DBA, transactions can be nested arbitrarily deep, at least as far as SAMOS is concerned.

4.2.3 The Event Detectors

Event detection refers to the task of noticing the occurrences of events and signalling them. Event detection is realized differently depending on the various kinds of primitive event. Additionally, the composite event detector covers composite events.

4.2.3.1 Primitive Event Detectors

In the case of method events, the original method is modified by SAMOS in order to support event detection. The new method contains statements for the signalling of the respective event at its beginning and/or before each return statement. After the modification has been performed, the method implementation needs to be recompiled.

In order to detect time events, appropriate entries are inserted into the `crontab` system file provided by the Unix operating system. These entries are generated automatically by the rule compiler from time event definitions, and are inserted into the crontab file by the compiler. This file is used by the `cron` process to monitor the system clock and to perform specified actions at specific points in time. In SAMOS, a `crontab` entry has the form `(<time>,<action>)` where `time` specifies the point in time when the time event must be signalled. `action` contains the call of a function which informs the rule manager about the occurrence of a time event. Details of detection of time events can be found in [22].

Event detection for transaction events is performed within the `samTransaction` operations described above. Note that SAMOS-transactions signal *any* transaction event, regardless whether the event is actually an interesting one or not.

4.2.3.2 The Composite Event Detector

This component is responsible for the detection of composite events, which can generally be implemented in two alternative ways:

- either the composite event detector stores all occurrences in a log, and upon each new primitive event occurrence queries the log in order to determine new composite event occurrences, or
- the composite event detector performs *stepwise detection* in that it knows for each composite event the ordered sequence of primitive events that have to occur in order to let the composite event occur.

Stepwise detection is considered to be more efficient with respect to execution time and space required for the storage of event occurrences. Once a (primitive or in turn composite) event has been signalled, the detector checks whether a "step forward" in any of the ordered sequences can be made. If the last element of such a sequence has been added, the corresponding composite event is considered to have occurred. It is thus not necessary to consider the entire event log. Furthermore, "garbage collection" of occurrences that cannot be used in the future is more straightforward in this approach.

In SAMOS, we have thus implemented stepwise detection. This approach requires a model which allows the representation of sequences of primitive events and of detection states in the system. In SAMOS, we use Colored Petri Nets (a special kind of Petri Nets). For each kind of composite event *constructor*, a Petri Net *pattern* has been defined. Whenever a new composite event description is created, the appropriate pattern is instantiated for it. All instantiated patterns are combined into a large, not necessarily connected, Petri Net.

For the sake of brevity, we give only an outline of the composite event detection process; details can be found elsewhere [21, 22]. The composite event detector is informed about interesting primitive events by the rule manager. In Petri Net terminology, the composite event detector then marks the input places corresponding to the primitive event. It then begins to play the token game. Tokens have as attributes the event parameters such as the identifier of the triggering transaction, the time of the occurrence, the identifier of the receiver object in case of method events, etc. Each time a place is marked with a token, the detector checks whether a transition can fire. As a result of the token game, output places representing a composite event can be marked. In this case, the composite event detector informs the rule manager about the occurrence of those composite events (through the operation `raise_event`).

A composite event occurrence is composed out of component events. The *consumption mode* [8] thereby determines which events are considered for event composition, and how the attributes of composite events are constructed out of the attributes of the components. The composite event detector in SAMOS implements the *chronicle* consumption mode [8], i.e., if not specified otherwise by event restrictions, always takes the oldest occurrence of an event description. The firing of transitions can be further constrained by restrictions (so called guards), such as the `same transaction` restriction for the component events.

The object-oriented approach of the SAMOS-architecture suggests an object-oriented representation of the Petri Net components (transitions, places and arcs are represented as objects). In addition, since event occurrences must be stored until they are used in the signalling of *all* corresponding composite events, and event occurrences are represented in the Petri Net as tokens, the appropriate tokens must be kept persistent. For this reason, tokens are represented as (persistent) objects as well.

4.2.4 The Rule Execution Component

After an event has been signalled and the rules to be executed have been determined, the rule execution component is invoked in order to actually execute these rules.

Assume an event (primitive or composite) has occurred. The rule manager determines whether rules are defined for this event and retrieves the corresponding rule objects. These objects as well as condition and action code fragments are determined through the references included in *list_of_rules* (Figure 5). Each rule object contains all further relevant information such as coupling modes. The rule execution component (REC) is called with the list of references to rules to be executed and the triggering transaction as parameters.

The tasks of the rule execution component depend on the coupling modes of the rules. In any case, each single rule execution starts with the condition evaluation. The rule execution component reads the coupling mode of the condition, which is stored in the rule object, and works as follows:

1. If the coupling mode is `immediate`, REC determines the index of the condition function and then calls this function directly.
2. If the coupling mode is `deferred`, the triggering transaction is notified to add the condition evaluation in its transaction-specific rule register (recall the class `samTransaction` described in section 4.2.2).
3. If the coupling mode is `decoupled`, a new top-level transaction for evaluating the condition is started.

Multiple rules may trigger upon the same event. In this case, SAMOS determines and enforces the execution order based on the information on coupling modes and priorities. Recall that this ordering is already reflected in the structure of the attribute `list_of_rules` for event objects (section 4.2.1.1).

In the `decoupled` case, independent transactions are started directly after the event occurrence. In contrast to the cases of `immediate` and `deferred` rules, SAMOS does not wait for their termination, and priorities are not controlled by SAMOS. Since it is not possible in ObjectStore to start new top-level transactions from within a running transaction, such independent transactions are started by a demon process.

In the `immediate` coupling mode for condition evaluation, triggered transactions are executed as subtransactions of the triggering transaction. They are started directly after the `decoupled` rules have been spawned (or directly after the event occurrence has been detected, if there are no `decoupled` rules). The execution order of these subtransactions is determined by the priorities of the corresponding rules. Since in the current implementation only one subtransaction can be active at one point in time, priorities also determine the sequential execution order of triggered transactions.

Finally, in the `deferred` coupling mode, SAMOS uses a transaction-specific rule register (the instance variable `list_of_rules` in Figure 7). This rule register implements a queue that includes all references to rule objects that have to be executed at the end of this transaction. The order in the rule register determines the execution order of the rules. Note that, depending on the respective coupling modes, "rule execution" may refer to the execution of condition or action parts only.

4.2.5 The SAMOS Demon

Decoupled rules must be executed in newly started top-level transactions. In ObjectStore, it is not possible to start new top-level transactions within another running transaction. In order to properly implement the `decoupled` mode, SAMOS thus uses a *demon process*.

The process architecture of SAMOS thus comprises two concurrent processes: the *kernel process* which executes the proper SAMOS kernel, and the demon process whose primary task is to execute decoupled rules. The demon process is in fact a "duplicate' of the kernel process because it operates instances of all the components described above for the SAMOS kernel. On a technical level, the major difference between both processes is whether they issue or accept requests for decoupled rule execution. Both, the demon and the kernel process share data- and rulebases. The kernel process uses the inter-process communication facilities of Unix to communicate with the demon (see Figure 8).

After a request for the execution of a decoupled rule has been received by the demon, control remains in the demon until this rule is entirely processed. In particular, if further events occur within the decoupled rule execution, then these events are processed by the demon (and so forth transitively). It can then also happen that the demon executes further rules in any coupling mode. Furthermore, component events occurring during the execution of a decoupled rule can be detected by the demon process. These component events have to be "merged" with the component events detected by the kernel process, in order to maintain a uniform history of events. This merge is performed in that the demon updates the Petri Net upon component event occurrences. Thus, both, the kernel and the demon are

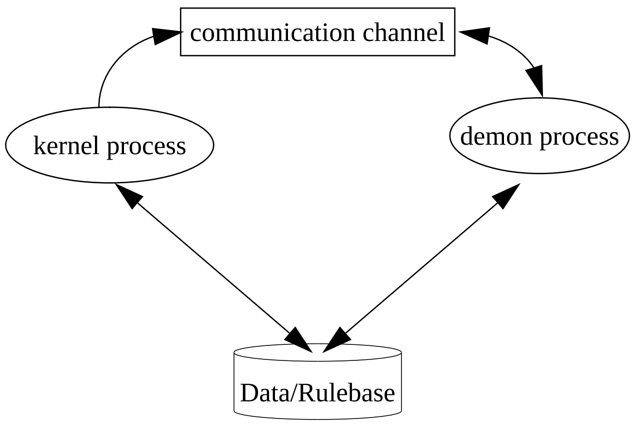

Figure 8: SAMOS Kernel and Demon Process Structure

synchronized via the shared rulebase. The possibility that component events may need to be handled and nested rule execution has to be performed is the reason for the demon to operate all the components described above for the kernel (event detection, event and rule object retrieval, and rule execution).

In order to abstract from the concrete communication facilities used, SAMOS defines a new class `samDemon`, which provides for the necessary functions. First, its methods implement the connection of the kernel and the demon to the communication channel. For the kernel process, the method `samDemon::send` allows to send messages concerning rules to be executed in `decoupled` mode to the demon. For the demon, `samDemon::receive` implements a blocking read of the communication channel. Whenever the demon receives a message via the communication channel, it reacts by executing the specified condition and/or action. The interprocess communication mechanisms currently used are *named pipes* [41].

4.3 Flows of Control in SAMOS

Based on the component descriptions in the previous section, we now describe the control flows through SAMOS after primitive event detection.

4.3.1 Execution of Single Rules

The interaction of the various SAMOS components is best illustrated by considering the following three phases:

- event detection,
- rule retrieval, and

- rule execution.

The simplest case is a primitive event occurrence that (1) does not participate in any composite event, (2) has only one rule defined for it, and (3) is associated to a rule with coupling modes (`immediate, immediate`). According to the execution model of SAMOS, three phases have to be executed in this case (Figure 9):

1. the primitive event detection phase ends with the signalling of the event,
2. the rule manager reacts to the event signalling and performs rule retrieval, and
3. the rule execution component performs the rule execution phase.

In Figure 9 and the following ones, we represent the structure of triggering (user) transactions as transaction trees (the execution order obtained using a depth-first left-to-right traversal of the trees). Triggering user transactions are represented by horizontal lines; subtransactions executed by and within the SAMOS kernel are represented as circles. Circles with a solid shape represent ObjectStore transactions, and bold, dashed circles represent SAMOS transactions. The various tasks refer to composite event detection (CED), rule retrieval (RR), and rule execution (RE). If rule execution is considered in more detail, we refer to condition evaluation (CE) and action execution (AE). Note that signalling of primitive events is done within triggering transactions.

All other cases (for composite events, multiple rules per event, and different coupling modes) are variations of the scheme in Figure 9. First, consider a primitive event occurrence that also contributes to a composite event. In this case, executing the rule attached to the primitive event first and then performing composite event detection would not be correct. For instance, the rule attached to the primitive event could raise further events, which might in turn participate in the composite event. Since the first primitive event occurrence would not yet be handled by the composite event detector, this component would permit primitive event occurrences to overtake each other, and the resulting event history as detected by the composite event detector would not be correct (i.e., would not be a sequence of occurrences ordered according to their occurrence time). In other words, SAMOS

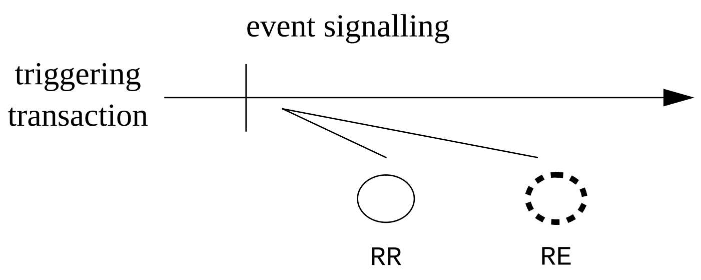

Figure 9: Phases of Active Behavior

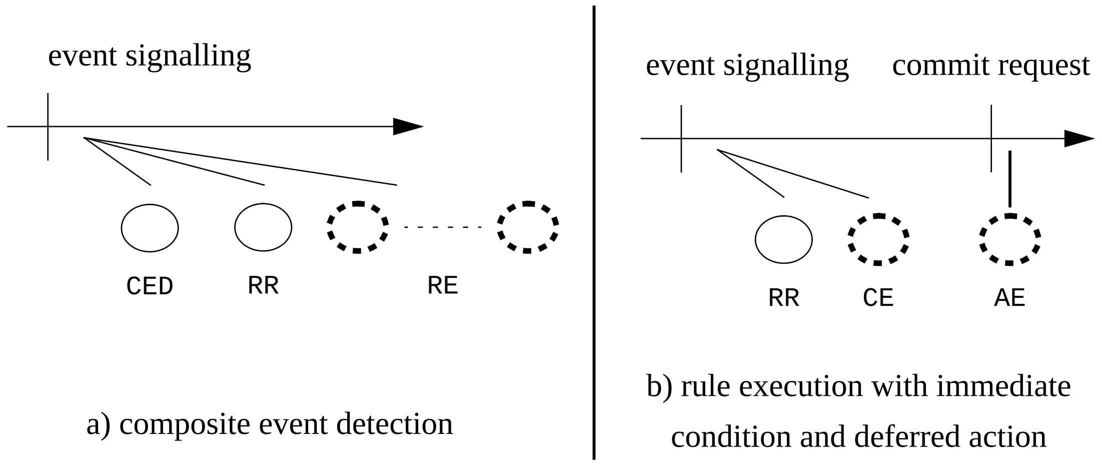
event signalling
CED
RR
RE
a) composite event detection
event signalling
commit request
RR
CE
AE
b) rule execution with immediate condition and deferred action

Figure 10: Execution of Single Rules

would in this case not implement the consumption mode *chronicle* [8] correctly. Out of the same reason it is not correct to signal composite events and to perform rule execution as soon as a composite event has been detected. Instead, the composite event detector has to play the token game until a final state of the composite event detector has been reached. This means that event detection has to be performed completely within the first of the three phases (Figure 10a).

In the second phase — rule retrieval — all rules to be executed for the signalled events are determined. Rule execution begins only after this phase terminates. In other words, both, event detection and rule retrieval are executed in one coherent phase, respectively. This is not always the case for rule execution, as is discussed subsequently.

Execution even of a single rule can be split into two non-successive phases, namely whenever the coupling modes are `(immediate,deferred)` or `(immediate,decoupled)`. In this case, control returns to the triggering transaction after condition evaluation, and rule execution (for this rule) is continued after the commit request (Figure 10b) in the case of a `deferred` action.

If the coupling mode of the condition is `decoupled`, the condition is evaluated in a new top-level transaction started by the demon, which then also performs action execution if necessary at all. If the condition coupling mode is immediate or deferred, but only the action is decoupled, then the condition is evaluated in a subtransaction of the triggering transaction. If the condition evaluates to true, then the demon is notified to execute the corresponding action.

Another case where rule execution is split into several (possibly non-successive) phases is the execution of multiple rules triggered by the same event, as discussed below.

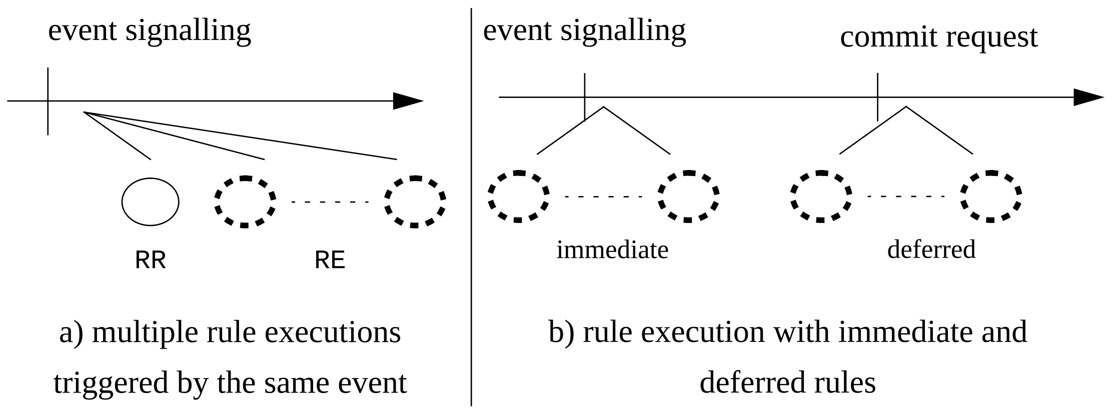
event signalling
RR
RE
a) multiple rule executions triggered by the same event
event signalling
commit request
immediate
deferred
b) rule execution with immediate and deferred rules

Figure 11: Execution of Multiple Rules

4.3.2 Execution of Multiple Rules

Multiple rules with immediate coupling modes, all triggered upon the same event, are executed after the corresponding rule objects for this event have been found (Figure 11a). All rules with condition coupling mode `immediate` are executed one after the other. Their execution order is determined either by user-specified priorities, or by SAMOS. For each condition, its corresponding action is executed directly afterwards if the evaluation has returned `true`. After the last immediate condition has been evaluated and its action has possibly been executed, control returns to the application.

Control returns to REC upon the commit request of the triggering transaction. Then all remaining actions are executed (Figure 11b). Actions in `deferred` mode whose corresponding conditions have been evaluated immediately and that have been evaluated to `true` are then executed. After the last such action has been executed, conditions with coupling mode `deferred` are evaluated. If a specific condition evaluates to true, its action is executed directly afterwards, provided the coupling mode is `immediate` or `deferred`.

4.3.3 Nested Rule Execution

Execution of multiple rules triggered by the same event can be interrupted due to nested rule execution (Figure 12). In this case, an event occurs within the execution of a rule and leads to the execution of conditions and actions as subtransactions of the triggered transaction. In the example, assume that a primitive event `e1` occurs that causes two composite events `e2` and `e3` to occur. Each event has a rule defined (`r1`, `r2`, and `r3`). Within the action execution of `r1`, another event `e4` occurs that triggers rule `r4`. Then, `r4` is executed as a subtransaction of `r1`, i.e., before rules `r2` and `r3`. In other words, while the sequence of de-

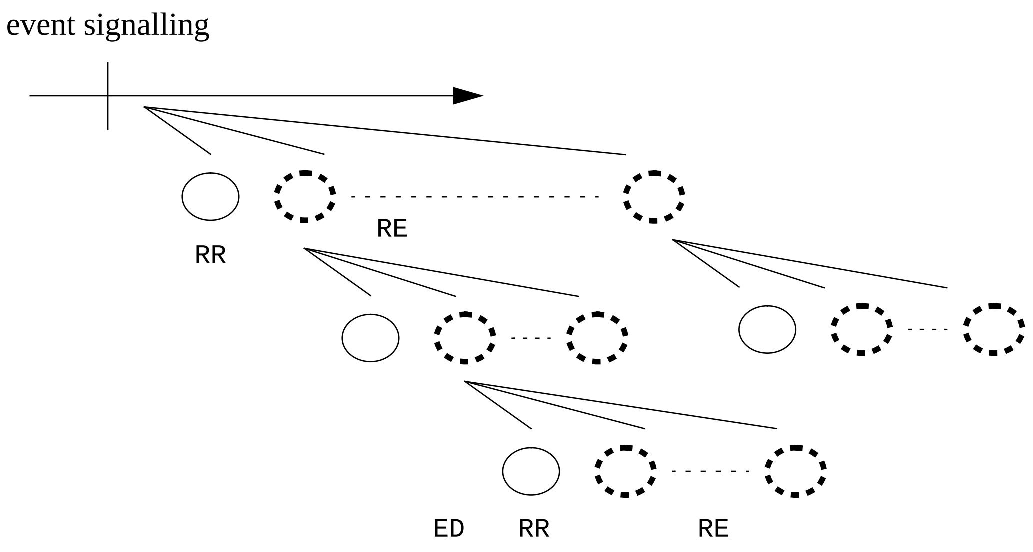

Figure 12: Nested Rule Execution

tected events is `<e1,e2,e3,e4>`, the corresponding rule execution sequence is `<r1,r4,r2,r3>`.

All coupling modes are allowed for nested rules. These coupling mode then refer to their triggering transactions —which are themselves triggered transactions— instead of the triggering transaction of the top-level rule. This is different in NAOS [11]: nested `immediate` rules are executed within the triggering rule execution, while `deferred` ones are executed at the end of the transaction that triggered the top-level rule (recall that NAOS does not support nested transactions).

4.4 The SAMOS Tools

A set of tools supports the user during the development and the maintenance of SAMOS applications. Some of these tools are mandatory to support (a compiler for the rule definition language and an analyzer for checking the correctness of rulebases), while others are "nice to have" (a rule editor and a rulebase browser).

4.4.1 The Rule Compiler

The rule compiler is responsible for the syntactic and semantic analysis of rule definitions. Its input are event and rule definitions in the syntax of SAMOS' rule definition language. If syntactic and semantic analysis have been successful, the compiler uses the interface offered by the rule manager to create corresponding event and rule objects.

4.4.2 The Rule Analyzer

The rule analyzer assists users in checking the termination of rules. It supports static and dynamic analysis. *Static analysis* means that rule definitions are examined at compile time, while *dynamic analysis* investigates the behavior of a set of rules at runtime.

During static analysis, the *relationship graph* of a set of rules R is built. This is a directed, labeled graph where the nodes represent the rules. The edges symbolize the action/ event relationships between the rules, i.e., the fact that an action of one rule lets the event (or part of the event) of another one occur. An edge between two rules is called *firm* if the execution of the source rule definitely causes the execution of the target rule. An edge is termed *potential* if the execution of the source rule *may* cause the execution of the target, e.g., depending on whether the condition is true or false. If the graph includes a cycle and all edges are firm, then R will never terminate; we call R a *hard-nonterminating set*. In this case, the user has to modify the rule definitions in R and rule analysis must be done again from the beginning. The case when at least one edge is potential indicates possible nonterminating rule sequences and requires further dynamic analysis to establish if the cycle is a real one. R is then called a *soft-nonterminating set*. In this case, additional information may be provided through the investigation of the action/condition relationships. We say that an action/condition relationship exists between two rules if the action of one rule may influence the condition of the other one. The consideration of such relationships is often crucial in deciding whether the rules in a soft nonterminating set form a cycle.

Dynamic analysis in SAMOS is based on simulation. The simulator generates a stream of event occurrences of interest for the given event definitions and rule execution scenarios in case of multiple rule definitions. The simulation is applied for some relevant initial database states, which are generated using the set of conditions derived from the given set of rules. The analyzer monitors the behavior of rules during the simulation and checks whether the same rule sequence belonging to a soft-nonterminating set repeats for a certain finite number of steps *n*.

Static analysis provides only incomplete information about the real interactions during the operation of a SAMOS application. Conclusions derived at compile time are only suppositions about interactions which could take place between rules, and in many cases the probability of such interactions is actually very small. Dynamic analysis helps to find more information about real rule interactions. However, rule analysis cannot be done completely. The rule analyzer notifies the user that a certain subset of rules could fall in a cycle and the user can decide himself whether the cycle is a real one or not.

4.4.3 The Rule Editor

The *graphical editor* supports rule definition in a more comfortable way then the rule compiler whose input is pure text. It provides a user friendly interface using icons or visual representations that can be manipulated for generating rule definitions. For example, during the definition of a composite event, the user can choose the appropriate operator icon and the editor automatically demands the needed parameters. Obviously, text is indispensable when defining the body of conditions and actions because source code is required.

4.4.4 The Rule Browser

The browser retrieves and shows individual rules, events, actions and conditions and allows the selection of items which meet various desired criteria. It consists of four parts: a rule, an event, a condition and an action browser. The rule browser shows the list of all rules and for each selected rule the appropriate rule body. If an event, condition or action is selected from the rule body description, the appropriate browser is activated by calling retrieval queries over the rulebase.

5 Working with SAMOS

In this section, we describe how users and application programs can use SAMOS.

After SAMOS has been installed (which implies an operational installation of ObjectStore), databases for storing user data and information on rules must be created. The rulebase is one part of the database. After the database has been created, the rulebase must be initialized, i.e., the segment to store event objects must be created and indexes must be defined.

Initialization is supported by methods of the class `Samos` (see Figure 13). These methods also allow some rudimentary tuning facilities. In order to set up a SAMOS application, two modes can be chosen for initialization:

- A *cold start* creates the internal data structures of SAMOS (rule and event extents including their indexes) as well as an ObjectStore database. The database will be empty afterwards.
- A *cold start for SAMOS only* creates the internal data structures of SAMOS, but leaves other user data untouched.

After SAMOS has been set up, the user can define ObjectStore classes, methods, and application programs via the ObjectStore interfaces. Event descriptions and ECA-rules are defined via one of the SAMOS interfaces (typically using the rule compiler or the rule editor). Rule definitions update the rulebase and also the application code. Applications thus have to be re-compiled after rule definitions.

At the beginning of each SAMOS application, SAMOS must be initialized in a mode called *warm start*. During warm start, SAMOS opens data- and rulebases.
During initialization in either mode, two tuning parameters can be specified. Recall that all the information related to event descriptions is clustered in an ObjectStore segment. ObjectStore allows for the specification that in each access to an object of a segment, the entire segment should be read (and thus transferred to the ObjectStore client's cache). Albeit the first access might then be more expensive, performance is improved whenever multiple objects out of the same segment are accessed successively. Thus, the first tuning parameter specifies whether the entire event segment should be read whenever one event object is retrieved. This parameter can be turned on and off during a SAMOS session. Turning this parameter off means that only individually needed pages are read for each access or query. Reading pages instead of the whole segment can be required if the client cache is small and the event segment is very large. The segment size can be determined using the method `SAMOS::get_info`.

The second tuning parameter specifies whether the entire event segment should be read upon initialization. In this case, it is assumed that the information on event descriptions is already in main memory when the first event in a session occurs.

Admittedly, these tuning techniques are still rather simple. More powerful tuning techniques and physical rulebase design are subjects to future work.

6 Evaluation of the SAMOS Prototype

In this section, we evaluate the implementation of SAMOS with respect to three criteria:

```
class Samos {
    public:
        static void initialize(s_Init_modes mode,             //cold, warm, or Samos only
                       s_Fetch_Policy fMode,                 //fetch segment or pages
                        s_Prefetch_Policy pMode,             //prefetch desired?
                       char *dbname);                        //name of database
        static void set_eseg_policy (s_Fetch_Policy  fMode,//fetch segment or pages
                       os_int32       numBytes);
        static void get_info (int  *  eseg_size,             //size of event segment
                       int  *  num_of_rules,          //number of rules in the rulebase
                       int  *  num_of_events);        //number of events in the rulebase
};
```

Figure 13: The Class Samos and its methods for Initialization and Tuning

1. *functionality*: has it been possible to implement the intended functionality using the layered approach?
2. *construction efficiency*: how costly has it been to design and implement SAMOS?
3. *runtime efficiency*: does the current implementation of SAMOS show satisfying runtime performance?

6.1 Functionality

Advanced *functionality* as specified for SAMOS can be implemented through a layered architecture, provided sufficient support by the underlying system exists. Albeit ObjectStore is a black box for the upper layer, primitive and composite events can be detected. Since ObjectStore provides for nested transactions, the `immediate` and `deferred` coupling modes as well as nested rule execution can be supported in their precise meaning. Additionally, we can provide recovery and concurrency control for triggered transactions (including transaction-internal checkpoints). Note that these features cannot be implemented on top of systems that support flat transactions only [31].

On the other hand, especially in the area of transaction management more functionality of the underlying system would improve the SAMOS kernel. First, a parent transaction is determined “syntactically” in ObjectStore as the transaction in which the request for the new transaction was issued. If ObjectStore would allow to start new top-level transactions in an asynchronous way, then the demon process would no longer be required. In this case, the `decoupled` coupling mode could be implemented more easily and efficiently. This approach would be more elegant since inter-process communication with the demon would no longer be used, and more efficiently, since inter-process communication can be costly.

Second, SAMOS represents the Petri Net component as a complex database object structure, thereby guaranteeing atomicity, durability, and isolation for transactions modifying the event history. However, multiple transactions raising component events of the same composite event description will thus block each other, or may even form deadlocks. We thus would like to implement a more subtle concurrency control protocol for the Petri Net component (comparable to dedicated concurrency control techniques for access paths). This is hard to achieve in a layered approach, because the Petri Net-specific transaction management had to be integrated with the existing transaction management in ObjectStore (recall that we cannot modify ObjectStore’s transaction manager).

Finally, the transaction concept of ObjectStore does not provide for parent/child and sibling parallelism [27]. If the former were possible, a transaction could spawn a subtransaction while proceeding with its own execution. Sibling parallelism allows subtransactions of the same parent to execute in parallel. With parent/child parallelism, a triggering transaction could proceed while its triggered transactions still execute. Likewise, internal tasks such as composite event detection could be performed concurrently to the user transactions, as is done in REACH [5]. Otherwise, a triggering transaction is blocked until the last triggered transaction has terminated. With sibling parallelism, triggered transactions whose executions are not constrained through priorities could be executed concurrently. Otherwise, they must be executed sequentially. Obviously, this restriction increases blocking times of triggering transactions.

6.2 Construction Cost

The current implementation of the SAMOS kernel comprises approximately 20’000 lines of C++-code. Most of the code has been spent for composite and time event detection. For event object and rule object management, the facilities offered by ObjectStore have been exploited (object management, clustering, database management). For retrieval of event and rule objects, ObjectStore features for querying and indexing are used. Ultimately, only slight extensions for transaction management have been necessary.

Summarizing, the layered approach of SAMOS is assessed as being construction efficient, since all the “passive” functionality the SAMOS kernel requires has already been available. Many components providing tasks like recovery or concurrency control which are critical parts in from-scratch implementations are for free in the layered approach on top of a DBMS.

6.3 Runtime Performance

The current SAMOS prototype has been evaluated with respect to performance. For that matter, we have defined a benchmark for ADBMSs, called BEAST[7] [25]. We sketch the most important results here, for a detailed description, see [25]. Two observations can be drawn from these tests:

1. they show how costly active functionality in SAMOS actually is,
2. they reflect the effects of optimizing the SAMOS system.

In the current state, the tests do not allow to asses SAMOS as being “fast” or “slow”. In order to justify such statements, comparisons with other ADBMSs are required but are not

7. BEnchmark for Active database SysTems

yet available. Furthermore, a comparison of functionally equivalent applications (one version implemented in a purely “passive” manner, the other one using active functionality) must be performed, in order to identify the drawbacks or benefits of ADBMSs.

BEAST is a configurable benchmark; it proposes many tests which can be performed for a given system — provided it offers the tested functionality at all. Tests exist for event detection, rule retrieval, and rule execution. Typically, each test raises an event, which is successively detected by the ADBMS and triggers a rule. In general, BEAST tests consider entire rules, since it is assumed that for systems other than SAMOS itself we do not have access to internal interfaces and thus cannot measure event detection or rule retrieval separately. Since most tests focus on one of the three phases, BEAST tests keep the other phases as simple as possible. For instance, tests for event detection have rules attached with a simple condition that always evaluates to `false`.

Test ED-02 has been executed for method event detection. Another test (ED-04) executes two transactions which in turn raise the same set of events such that buffering effects of event objects can be observed. For composite events, BEAST contains tests for `sequence` (ED-06), `negation` (ED-07), the `times`-operator (ED-08), a sequence of in turn composite events (ED-09) and a conjunction with a `same` restriction (ED-11).

Test RM-01 has been executed for rule retrieval, i.e., it focuses on the time it takes to retrieve event and rule objects. Rule execution tests consider different coupling modes: `immediate` (RE-01), `deferred` (RE-02), and `decoupled` (RE-03). Test RE-04 measures the execution of four rules all triggered by the same event. The execution order of these rules is not constrained by means of priorities.

For each test, we consider four different rulebase sizes. The rulebase size is defined in terms of dummy events/rules in addition to the set of events and rules required by BEAST. The empty rule base contains zero dummy events and rules, the small one 50 dummy events and rules, the medium one 250, and the large one 500 events.

In the first test series, the Samos Petri Net has been implemented “relationally”. Instead of using pointers for representing the Net structure, queries (joins) have been necessary in order to traverse the Petri Net during the token game. Performance especially for composite event detection thus has been quite poor.

In the subsequent version, pointers had been used for representing the Petri Net structures (as described in this paper), resulting in reasonable performance gains. In this version, the prime performance problem has been event object retrieval, since the event extent has neither been clustered nor indexed. After indexing and clustering (as described above) have been added, more performance gains have been possible.

The tests have been executed on a SUN server Sparc Model 5 under SUNOS 4.1.3. The size of the ObjectStore cache has been 12 MB. Table 2 summarizes the measured times (CPU-time in milliseconds). This table contains four results for each of the tests:

1. the results of the "non-pointered" version are given in the first row,
2. the second row contains results for the pointered version,
3. the third row reflects results for the indexed and clustered extents,
4. the last row shows results when the prefetching option is used

	Rulebase Size (# Dummy Events), Results in Milliseconds			
Test	empty	small	medium	large
ED-02	150	280	246	878
	147	253	450	840
	109	110	111	135
	101	110	119	108
ED-04a	768	984	1472	1830
	685	905	1438	1634
	444	467	459	476
	433	442	453	439
ED-04b	674	878	1080	1056
	573	740	1039	940
	376	375	380	394
	373	366	380	359
ED-06	532	758	1210	2596
	395	500	680	1066
	317	391	369	384
	308	341	364	355
ED-08	1090	1428	2480	4814
	836	1000	1529	1724
	752	768	756	819
	740	742	769	769

Table 2. Performance Measurement Results for SAMOS

	Rulebase Size (# Dummy Events), Results in Milliseconds			
Test	empty	small	medium	large
ED-09	1102	1494	2566	5350
	819	973	1478	1639
	693	724	739	757
	709	716	753	729
ED-11	428	538	1142	1826
	357	469	800	1076
	325	347	340	350
	309	327	345	328
RM-01	164	176	358	184
	154	256	438	179
	118	124	122	134
	101	118	104	134
RE-01	152	272	588	266
	157	200	562	203
	119	130	122	150
	103	105	104	122
RE-02	178	252	476	232
	157	205	506	197
	113	127	105	150
	102	118	114	129
RE-03	—	—	—	—
	152	216	449	170
	104	118	122	151
	102	109	103	126
RE-04	384	550	496	1296
	325	353	660	1072
	159	169	154	188
	156	134	157	171

Table 2. Performance Measurement Results for SAMOS

Summarizing, significant performance improvements have been achieved. Especially, in the last two versions SAMOS scales better than in the previous ones, i.e., the curve relating execution times to rulebase size has a much smaller slope.

We are currently implementing BEAST on other object-oriented ADBMSs in order to compare the performance of SAMOS with that of other systems. We are also currently comparing active and "passive" implementations of the same application with respect to performance. Additionally, more subtle tuning and clustering techniques in SAMOS are a subject of future work (e.g., clustering of Petri Net parts).

7 Conclusion

This paper has described the comprehensive architecture of the SAMOS active DBMS. We have presented a layered approach on top of an object-oriented DBMS for its implementation and reported on experiences with it. Hence, the overall contribution of this paper is twofold:

- it contributes to the knowledge about how to design active, object-oriented DBMSs, and
- it shows that the layered approach is feasible at least for prototype systems, provided basic DBMS-mechanisms are supported.

Our experience with the layered SAMOS architecture is that the desired functionality can be implemented at reasonable costs. The SAMOS prototype described here is operational and is publicly available. It enables us to experiment with SAMOS in sample application domain requiring reactive behavior [24, 26, 45]. We have full control of the add-on layer and can extend it at our will whenever necessary. Due to the still missing comparative performance figures of other systems, we cannot definitely assess the runtime performance of SAMOS. Nevertheless, we have a vehicle for improving our understanding of performance issues, optimization, and tuning techniques of ADBMSs as well as the operational prototype that allows us to verify our observations and prospective improvements.

After the first release, our future work on SAMOS will focus on several areas:

- we will continue our investigation of *ADBMS-applications* (e.g., banking, workflow management),
- we will address usability aspects of ADBMSs (tool support for ADBMSs),
- we will consider runtime performance in more detail and investigate optimizations, tuning facilities, and redesigns of some components, and
- finally we will develop extensions of the existing prototype, e.g. to support distribution.

8 References

1. E. Anwar, L. Maugis, S. Chakravarthy. *A New Perspective on Rule Support for Object-Oriented Databases*. Proc. ACM SIGMOD, Washington, May 1993.
2. M. Berndtsson, B. Lings: *On Developing Reactive Object-Oriented Databases*. In [7].
3. P.A. Boncz, M.L. Kersten: *Monet. An Impressionist Sketch of an Advanced Database System*. Proc. Basque Intl. Workshop on Information Technology, San Sebastian (Spain), July 1995.
4. H. Branding, A. Buchmann, T. Kudrass, J. Zimmermann: *Rules in an Open System: The REACH Rule System*. In [38].
5. A.P. Buchmann, J.Blakeley J.A. Zimmermann, D.L. Wells: *Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions*. Proc. of the 11th Intl. Conf. on Data Engineering, Taipei, Taiwan, March 1995.
6. P. Butterworth, A. Otis, J. Stein: *The Gemstone Object Database Management System*. Communications of the ACM 34:10, 1991.
7. S. Chakravarthy (ed.): *Active Databases*. Special Issue of the Bulletin of the IEEE TC on Data Engineering 15:1-4, 1992.
8. S. Chakravarthy, V. Krishnaprasad, E. Anwar, S.-K. Kim: *Composite Events for Active Databases: Semantics, Contexts, and Detection*. Proc. of the 20th Intl. Conf. on Very Large Data Bases, Santiago, Chile, September 1994.
9. S. Chakravarthy, D. Mishra: *Snoop: An Expressive Event Specification Language for Active Databases*. Data & Knowledge Engineering 14:1, November 1994.
10. S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, R.H. Badani: *ECA Rule Integration into an OODBMS: Architecture and Implementation*. Proc. of the 11th Intl. Conf. on Data Engineering, Taipei, Taiwan, March 1995.
11. C. Collet, T. Coupaye, T. Svensen: *NAOS: Efficient and Modular Reactive Capabilities in an Object-Oriented Database System*. Proc. of the 20th Intl. Conf. on Very Large Data Bases, Santiago, Chile, September 1994. .
12. U. Dayal et al: *The HiPAC Project: Combining Active Databases and Timing Constraints*. ACM Sigmod Record, 17:1, March 1988.
13. U. Dayal, A. Buchmann, D. McCarthy: *Rules Are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System*. In: K. R. Dittrich (ed.). Proc. 2nd Intl. Workshop on Object-Oriented Database Systems, 1988, LNCS 334, Springer.
14. U. Dayal, E. Hanson, J. Widom: *Active Database Systems*. In W. Kim (eds): Modern Database Systems. ACM Press / Addison Wesley, 1995.
15. O. Deux: *The O2 System*. Communications of the ACM 34:10, 1991.
16. O. Diaz, N. Paton, P. Gray: *Rule Management in Object-Oriented Databases: A Uniform Approach*. Proc. 17th Intl. Conf. on Very Large Data Bases, Barcelona, September 1991.
17. K.R. Dittrich, S. Gatziu, A. Geppert (eds): *The Active Database Management System Manifesto: A Rulebase of ADBMS Features*. In [39].
18. S. Gatziu, A. Geppert, K.R. Dittrich: *Integrating Active Concepts into an Object-Oriented Database System*. Proc. 3rd Intl. Workshop on Database Programming Languages, Nafplion, August 1991.

19. S. Gatziu, K.R. Dittrich: *SAMOS: An Active, Object-Oriented Database System*. In [7].

20. S. Gatziu, K.R. Dittrich: *Events in an Active Object-Oriented Database System*. In [38].

21. S. Gatziu, K.R. Dittrich: *Detecting Composite Events in Active Database Systems Using Petri Nets*. Proc. 4th Intl. Workshop on Research Issues in Data Engineering: Active Database Systems (RIDE-ADS), Houston, Februar 1994.

22. S. Gatziu: *Events in an Active Object-Oriented Database System*. Doctoral Dissertation, University of Zurich, 1994. Verlag Dr. Kovac, Hamburg, Germany, 1994.

23. N.H. Gehani, H.V. Jagadish. *Ode as an Active Database: Constraints and Triggers*. Proc. 17th Intl. Conf. on Very Large Data Bases, Barcelona, September 1991.

24. A. Geppert, K.R. Dittrich: *Specification and Implementation of Consistency Constraints in Object-Oriented Database Systems: Applying Programming-by-Contract*. Proc. GI-Conf. Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), Dresden, Germany, March 1995.

25. A. Geppert, S. Gatziu, K.R. Dittrich: *Performance Evaluation of an Active Object-Oriented Database Management System: 007 Meets the Beast*. In [39].

26. A. Geppert, M. Kradolfer, D. Tombros: *Realization of Cooperative Agents Using an Active Object-Oriented Database Management System*. In [39].

27. T. Härder, K. Rothermel: *Concurrency Control Issues in Nested Transactions*. The VLDB Journal 2:1, 1993.

28. G. Kappel, S. Rausch-Schott, W. Retschitzegger, S. Viewweg: *TriGS: Making a Passive Object-Oriented Database System Active*. Journal of Object-Oriented Programming 7:4, July 1994.

29. M.L. Kersten: *An Active Component for a Parallel Database Kernel*. In [39].

30. A.M. Kotz, K.R. Dittrich, J.A. Mülle: *Supporting Semantic Rules by a Generalized Event/Trigger Mechanism*. Proc. Intl. Conf. on Extending Database Technology (EDBT), Venice, Italy, March 1988. LNCS 303, Springer 1988.

31. A. Kotz-Dittrich: *Adding Active Functionality on an Object-Oriented Database System: A Layered Approach*. Proc. GI Conf. Datenbanksysteme in Büro, Technik und Wissenschaft, Braunschweig, Germany, March 1993.

32. C.W. Krueger: *Software Reuse*. ACM Computing Surveys 24:2, 1992.

33. C. Lamb, G. Landis, J. Orenstein, D. Weinreb: *The ObjectStore Database System*. Communications of the ACM 34:10, 1991.

34. B. Lindsay, J. McPherson, H. Pirahesh: *A Data Management Extension Architecture*. U. Dayal, I. Traiger (eds): Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, San Francisco, CA, May 1987. SIGMOD Record 16:3, 1987.

35. D.R. McCarthy, U. Dayal: *The Architecture of an Active Data Base Management System*. Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, Portland, May/June 1989.

36. J.E.B. Moss: *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.

37. N.W. Paton, O. Diaz, M.H. Williams, J. Campin, A. Dinn, A. Jaime: *Dimensions of Active Behaviour*. In [38].

38. N.W. Paton, H.W. Williams (eds): *Rules in Database Systems*. Workshops in Com-

puting, Springer-Verlag, 1994.

39. T. Sellis (ed): Proc. 2nd Intl. Workshop on Rules in Database Systems, Athens, Greece, September 1995. LNCS 985, Springer 1995.

40. E. Simon, J. Kiernan, C. de Maindreville: *Implementing High Level Active Rules on Top of a Relational DBMS*. Proc. 18th Intl. Conf. on Very Large Data Bases (VLDB), Vancouver, Canada, August 1992.

41. W.R. Stevens: *Advanced Programming in the UNIX Environment.* Addison-Wesley Professional Computing Series, 1992.

42. M. Stonebraker, E.N. Hanson, S. Potamianos: *The POSTGRES Rule Manager.* IEEE Transactions on Software Engineering 14:7, 1988.

43. B. Stroustrup: *The C++ Programming Language*. Addison-Wesley, 1986.

44. *SUNOS Reference Manual*. SUN Microsystems, Inc., 1990.

45. D. Tombros, A. Geppert, K.R. Dittrich: Brokers and Services: Constructs for the Implementation of Process-Oriented Environments. Technical Report 95.03, Institut fuer Informatik, Universitaet Zuerich, August 1995.

46. D.L. Wells, J.A. Blakeley, C.W. Thompson: *Architecture of an Open Object-Oriented Database Management System*. IEEE Computer 25:10.

47. J. Widom, R.J. Cochrane, B.G. Lindsay: *Implementing Set-Oriented Production Rules as an Extension to Starburst*. Proc. 17th Intl. Conf. on Very Large Data Bases, Barcelona, September 1991.

48. J. Widom, S. Ceri (eds): *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, 1995.