SAMOS in Hindsight: Experiences in Building an Active Object-Oriented DBMS

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

Technical Report 2000.05

Database Technology Research Group
Department of Information Technology, University of Zurich
Email: {dittrich, fritsch, gatziu, geppert, vaduva}@ifi.unizh.ch

Abstract

Active object-oriented database management systems incorporate object-oriented database technology and active mechanisms such as event-condition-action rules (ECA-rules). SAMOS has been among the first representatives of this class of systems. During the development of SAMOS, numerous then open research questions have been addressed. In this paper, we present a "historical" perspective of the SAMOS project and report on lessons and experiences we gained in the project. We identify requirements, present the solutions we devised, and report on experiences we draw from this project. In particular, we describe the rule model of SAMOS, which represents a smooth integration of ECA-rules into an object-oriented data model. We also discuss the implementation and architecture of the SAMOS prototype on top of a passive object-oriented database system. Furthermore, we report on performance and usability issues. In order to analyze performance, we have developed a benchmark; we discuss the experiences (and improvements) we made by running the benchmark on SAMOS and by comparing the results to those obtained for other systems. Usability issues have been investigated with respect to tool support for designing SAMOS applications and analyzing rule systems. Finally, we discuss experiences in implementing SAMOS and the conclusions we have drawn for the implementation of other types of event-based persistent systems as well as a development method for active systems in general.

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

Table of Contents

1 Introduction

Active database management systems (ADBMS) [1, 16, 19, 24, 56, 58, 59, 75] are able to react in a predefined way to specific, predefined situations occurring in the database or its environment. The execution of appropriate reactions is automated and therefore does not require explicit requests issued by users or applications. Situations are usually specified as event/condition pairs, and together with actions, they form so-called *event-condition-action rules* (*ECA-rules*). The meaning of an ECA-rule is

- *when* the event occurs, and
- *if* the condition holds
- *then* execute the action.

The *rule model* of an ADBMS defines how ECA-rules can be specified and what their semantics is. Rule models differ with respect to the supported event types (e.g., update of a data item in the database, or temporal events) as well as condition and action definition (e.g., predicates or queries for conditions). The *execution model* of an ADBMS defines the semantics of rule execution, e.g., with respect to the point in time when rule execution starts, scheduling of multiple rules, etc.

Active database research has initially focussed on the integration of active behavior into relational DBMSs (Starburst [74], Ariel [48], POSTGRES [65]). A second generation of projects (including Ode [38], Sentinel [17] and SAMOS, later followed by REACH [6, 7], ACOOD [4], NAOS [22], TriGS [50], Chimera [15], and [51]) investigated *object-oriented* ADBMS.

Reactive behavior as offered by ADBMS can be beneficially used by numerous application areas, such as financial applications [20], network management [3], workflow management [42], medical applications [5], integrity constraints [12, 40], maintenance of materialized views [13], and coordination of heterogeneous, distributed systems [14].

In this paper, we present an overview of the object-oriented ADBMS SAMOS which we have developed since 1990. We describe the rule and execution model of SAMOS as well as its implementation on top of a passive DBMS. A first complete prototype has been finished in 1994 and demonstrated [35] in 1995; since 1996, SAMOS has been publicly available [34]. We report on lessons learned and experiences gained during SAMOS' design and implementation. In particular, we discuss opportunities and limitations of the layered approach used to build SAMOS on top of a passive DBMS.

Following the development of SAMOS, we have investigated usability issues in various respects, such as functional evaluation based on use cases, performance evaluation, and the development of required tools. This work taught us further lessons of how to provide and implement ADBMS-functionality. In particular, the most important experience has been that active mechanisms need to be offered in a very flexible way and that they need to be customizable to the appli-

cation area at hand. Based on these findings, we started to investigate flexible and systematic approaches to the implementation of active systems in general.

The remainder of this paper is organized as follows. In Section 2, we introduce the rule and execution models of SAMOS and describe the design and implementation of the SAMOS prototype. Section 3 discusses three aspects of usability, namely sample application areas, performance evaluation, and tool support for ADBMS-applications. Section 4 briefly introduces the ongoing research in construction techniques for active systems. Section 5 concludes the paper.

2 Design and Implementation of SAMOS (1990-1994)

When the SAMOS project started in 1990, the notion of trigger had been around for some time. However, among the prominent DBMS products only RDB and Sybase supported such a concept. Research had been done in the context of relational database systems [e.g., 48, 66, 74] and the Hipac project [23]. With the notable exception of Hipac, the expressiveness of the rule models has been rather restricted. Thus, when SAMOS started, one of the major research questions was how to make rule and execution models more powerful. For instance, which further types of events (beyond simple database update events) would be needed by ADBMS-applications? How could complex situations (consisting of more than one constituent event) be modeled? How could they be efficiently detected?

In 1990, object-oriented database systems were *the* emerging database technology—a manifesto incorporating the ideas of several researchers representative for the field had been published shortly before [2], and products were just entering the market [e.g., 25, 52]. Thus, the second research question has been how to integrate active mechanisms into an object-oriented data model—how would a rule model in an object-oriented context look like? For instance, how to relate ECA-rules and class definitions: should rules always be a part of the class definition, or can ECA-rules be independent from classes? Should ECA-rules be subject to class inheritance?

Finally, how could an object-oriented, active DBMS be implemented? How can new components (such as event detectors) be added to a passive DBMS? Apparently (internal) components of a DBMS need to be adapted (such as the transaction manager, the object manager, etc.). Does this imply that a from-scratch implementation is necessary, or can a passive system be extended towards an active one?

We addressed these questions as follows:

- *A powerful event definition language.* Together with the languages proposed in the Sentinel [17] and in the Ode project [38] the SAMOS language was among the first expressive event definition languages. Our event language concentrates on the definition of complex events and offers a small number of event constructors that allow expressive event definitions in combination with the concept of monitoring intervals [26, 32].

- *Detection of complex events.* In contrast to several other techniques for detection of complex events [7, 18, 39], the event detector of SAMOS is based on Petri Nets. Generally, Petri Nets allow both powerful and succinct descriptions of many complex systems. It turned out that they are suitable for the modeling and the detection of complex event definitions [33].
- *Integration of active and object-oriented features in one system*. SAMOS investigates the co-existence of active and object-oriented features in one system. In this context, we have addressed issues concerning the nature of events specified on database operations and the association between rules and classes [30].
- *Architecture and implementation*. We ruled out a from-scratch implementation, because based on our previous experience, from-scratch-implementations imply a high overhead (which is hard to bear in a research environment), in particular when only a small fraction of the system components is of a major research interest. When the project started, we also considered to use extensible database management systems. However, the data model and query language of Exodus (called Extra/Excess) [8] have no longer been available back then, and systems such as Open OODB [73] and Shore [11] have not yet been available when the SAMOS implementation started. We therefore decided to follow a layered approach, where we tried to add active functionality on top of a passive system. By doing this, our research interest was in finding out whether a fully-functional active DBMS could be built in this way, what the limitations are, and what kind of performance can be achieved in this approach.

2.1 ECA-Rules in SAMOS

In this section we introduce the major features of the SAMOS rule language [32]. A rule in SAMOS has the following form:

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

A rule definition specifies an event description (also called *event type*[1]), a condition, an action and execution constraints (priorities and coupling modes). The event and action parts are mandatory while the definition of a condition is optional. In case no condition is explicitly specified, it is assumed to be always true. Events, conditions and actions can be named and defined separately (outside rule definitions). In this way, it is possible to reuse event types, conditions and actions across rule definitions.

1. Note that in SAMOS we distinguish between event types and event occurrences.

2.1.1 Primitive and Composite Events

Events can be either *primitive* or *composite*. Primitive events in SAMOS are

- *time events* (which occur at a specific point in time, periodically, or relative to some other event),
- *message sending events* (which occur at the beginning or the end of a method execution),
- *value events* (which occur before or after the modification of an object's value),
- *transaction events* (which occur before or after a transaction operation), and
- *abstract events* (which have to be signalled explicitly by the application or the user).

Composite events are constructed out of primitive or other composite events (called component events). The event constructors supported by SAMOS are

- *sequence* (E1; E2): all component events occur in the prescribed order.
- *conjunction* (E1, E2): all component events occur in any order.
- *disjunction* (E1 | E2): one of the component events occurs.
- *negation* (NOT E): the component event has not occurred during an interval defined with the negation event (see below).
- *reduction*: multiple occurrences of the component event are collapsed into a single occurrence.

The reduction constructors allow the repeated occurrence of an event `E` to be signalled only once. In case of the closure constructors `*E` and `last E`, the event `E` is signalled only after its first respectively last occurrence. The TIMES-constructor `TIMES(n, E)` is another kind of reduction constructor; it is signalled after each n^{th} occurrence. Instead of a fixed number, it is also possible to specify ranges (such as `n1-n2`, or `>n`, where `n`, `n1`, and `n2` are natural numbers).

The consumption mode underlying event composition is *chronicle*. This means that whenever multiple eligible component event occurrences exist, the oldest one is chosen.

2.1.2 Monitoring Intervals

As mentioned above, it is sometimes required that a (primitive or composite) event `E` is signalled only if it has (not) occurred during a specific time interval `I`. Therefore, we have introduced in SAMOS *monitoring intervals* for those time intervals during which the event has to occur in order to be considered relevant. Monitoring intervals are mandatory for the definition of negation and reduction events because it is necessary to restrict the monitoring time of the occurrence of an event.

Various options exist for the definition of start and end points of intervals:

- both can be defined as absolute points in time,

- the start point can be defined as an implicit point in time, i.e., as an event occurrence, and the end point can be defined as an implicit, absolute or relative (referring to the start) point in time,
- they can be specified as an interval of absolute points in time that reappears periodically, e.g., `EVERY WEEK [MO,18:00-FR,24:00]`,
- they can be computed from other time intervals using the operators *overlap* and *extend* defined for monitoring intervals.

2.1.3 Conditions and Actions

A condition clause in SAMOS is an expression in the query language of the underlying object-oriented database system. The meaning of the condition clause is that if the query produces any data (a non-empty set of database objects), then the condition is satisfied. The main reason for expressing a condition as a query instead of a predicate is that the non-empty answer is thus available to the action which may exploit it appropriately.

An action can be any executable program written in the data manipulation language of the underlying database system. It may send messages to objects for the execution of methods or even abort the execution of the (user-defined) transaction in which the corresponding event occurred. Aborting a transaction is required, e.g., in cases where the rule expresses an integrity constraint that is not satisfied. The fact that the action can manipulate other objects might cause the occurrence of other events and lead to nested rule execution.

2.1.4 Event Parameters and Restrictions

For powerful rule execution, information on event occurrences must be passed between the constituents of a rule. In SAMOS, this is accomplished through *event parameters*. The set of event parameters is event type-specific and fixed (except for abstract events). Each event (except a time event) carries so-called *environment parameters*, such as the occurrence time or the identifier of the transaction that triggered an event (the *triggering transaction*). Method events have the object identifier of the object executing the method as a parameter. Using event parameters, rule execution can refer to the actual database state, i.e., to the state when the condition is evaluated or the action is executed.

Event parameters for composite events depend on those of the component events and the composite event constructor. For example, a sequence or a conjunction takes the union of the parameters of their components as parameters. Events defined using the *-operator take the parameters of the first occurring event. Composite events defined as `TIMES(n,E)` take the union of the parameters of the *n* occurring events as parameters. Composite events defined as `TIMES([>n],E) IN [s-e]` or TIMES([n1-n2], E) take as parameters the union of the parameters of the component events and the number of these events.

Event restrictions can be specified to further restrict composite events to actually interesting ones. They are conditions to be met by the event parameters of component events. For example, the *same transaction* restriction attached to a composite event requires that all component events must be triggered by the same transaction. For message sending events, it can be specified that they have to occur for the same receiver object (*same object*) in order to be eligible constituents of a composite event. Furthermore, the *same user* restriction requires that the component events have occurred in transactions issued by the same user.

2.2 Execution Model

Events occur within a transaction (except time events, which occur outside transactions), called the *triggering transaction*. Triggered rules are also executed within transactions, called *triggered transactions*. Coupling modes define when triggered transactions are started with respect to the triggering transaction, and which kind of dependencies among triggered and triggering transactions exist, if any. SAMOS allows two coupling modes per rule to be specified: the event/condition coupling mode specifies when the condition is evaluated with respect to the event detection, and the condition/action coupling mode defines when the action is executed with respect to the condition evaluation. For both of them, SAMOS offers the following choices: `immediate` (in a subtransaction executed directly after the event has been detected), `deferred` (in a subtransaction executed at the end of the triggering transaction) and `decoupled` (in a separate, independent transaction). These coupling modes have been adopted from the HiPAC project [23].

Since *multiple rules* may be defined for the same event and with the same coupling mode, priorities are required to specify the order to be imposed on the execution of these rules. SAMOS supports relative priorities: for each rule, it can be specified if it should be executed before or after specific other rules associated to the same event. Priorities thus form a partial order on rules. Rules that are not (transitively) ordered by priorities are executed in an arbitrary (system-determined) order.

2.3 Example

We now give a short example for the administration of a schools and students. Students have to take intermediate exams; if they repeatedly fail these exams, they are not allowed to continue their studies.

```
DEFINE EVENT RELEGATION =
        (Student.registerExam; TIMES(Student.failedExam, 2): same object): same object

DEFINE RULE RELEGATE
ON      RELEGATION
DO      oid->notify ("We regret to inform you that you will be relegated since you definitely
                      failed the intermediate exam");
        oid->relegate
COUPLING    [immediate, deferred]
```

Example 1. Example SAMOS Rule

2.4 The SAMOS Prototype

In the prototype implementation of SAMOS, all components implementing the active behavior are built "on top" of the passive object-oriented database system ObjectStore [55], which is left unmodified and is treated as a black box for SAMOS. This is the major feature of layered architectures in contrast to integrated architectures, where the kernel DBMS can be internally modified.

The SAMOS prototype consists of three building blocks (Figure 1):

- the object-oriented DBMS ObjectStore,
- a layer on top of ObjectStore implementing the active functionality which consists of a number of components like a rule manager, a detector for composite events and a rule execution component, and

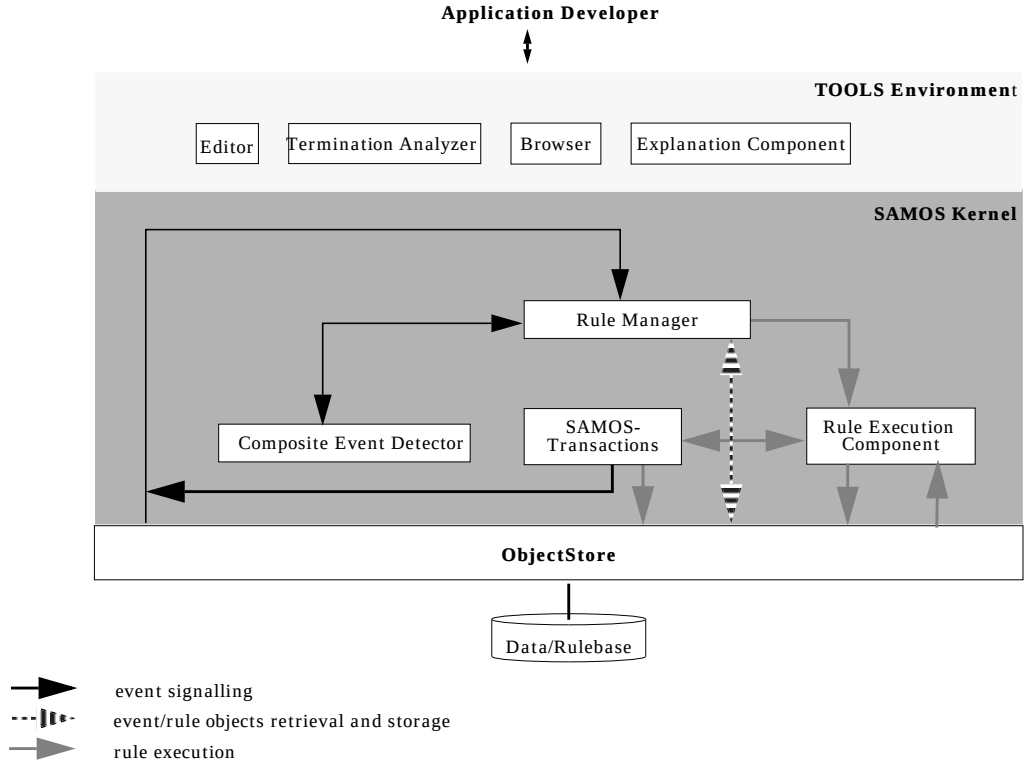

Figure 1. An Overview of the SAMOS Architecture

- a set of tools such as a rule compiler, a rule analyzer, a rule editor and a rule explanation component which are discussed in Section 3.3.

Below we describe the design and implementation of the middle layer, which is implemented by a collection of C++-classes.

2.4.1 Defining and Storing Rules and Events

Event and rule definitions are managed by the rule manager (see Figure 2 for the most important parts of the rule manager's interface). SAMOS users define or modify ECA-rules by using the SAMOS rule definition language. All rule definitions are in a first step syntactically and semantically analyzed and in a second step persistently stored. In a third step, for each defined event the appropriate event detector is initialized.

After a rule definition has been successfully checked, its internal representation is stored in the rulebase. The rule compiler uses the interface offered by the rule manager (as a set of methods) to store this information.

SAMOS uses the means of the underlying DBMS to provide persistence of the rulebase. A part of the rule schema is illustrated in Figure 3 (for space reasons, we omit subclasses of `composite_event` and `time_event` in this figure). Rule definitions are stored as instances of class `rule`, so-called *rule objects*. All the event types are objects of class `event` (so-called *event objects*). Each event object maintains a list of references to objects of class `rule`; these rules have to be executed when this event has occurred. Other systems (e.g., ADAM [56], Ode [38]) associate rules with the objects or classes on whose operations events are defined. However, this approach is feasible only for some primitive event types (e.g., method events), because only then such a rule association exists. Our approach is more general since it can be used for all sorts of events, primitive and composite ones. Each event object also has a list of references to other event objects representing composite events. The meaning of such a reference is that the origin of the reference can participate as a component event in the target event. Subclasses of `event` are defined for the various kinds of event descriptions (e.g., method events, complex events, and so forth).

Event descriptions are persistently stored in a so-called *event extent*. In ObjectStore terminology, an extent is a collection of currently existing, persistent instances of a specific class. In order to accelerate access to event objects, system-provided means like indexing over the event extent and clustering the event objects in specific segments are applied.

Conditions and actions basically consist of rule schema information (e.g. action name, textual representation of the respective operations to be performed etc.) and the code fragments to be carried out upon rule execution. It is not a viable option to include the source code of conditions and actions into a module statically linked to SAMOS, because this would require recompilation of the entire SAMOS system whenever new rules are defined. In order to separate the SAMOS kernel from these code fragments, each condition/action is represented as a C++-function which can be

```
class CRuleManager {

public:
    TStatus ColdStart(char DBName[], int replace, os_int32 mode = 0664);
    TStatus HotStart(char DBName[]);

    rule*   ExistsRule(char RuleName[]);
    void    GetListOfRules(os_set* RuleSet);

    event* ExistsEvent(char* ID);
    composite_event* GetComplEvent(char* ID);
    void    GetRulesForEvent(event* evt, os_set* Rules);

    TStatus DefineRule(char RuleName[], event* Ev, condition* Cond, action* Act,
                       couplingMode eccMode, couplingMode cacMode, int prio = 10);
    TStatus DefCondition(char name[], char query_stmt[]);
    TStatus DefAction(char ActName[], char dml_stmt[]);

    TStatus DefAbstrEvent(char EventName[]);
    TStatus DefTransEvent(char EventName[], char TransName[], TTransMode Modus);
    TStatus DefMethEvent(char EventName[], char className[], char MethName[],
                       char HeaderFileName[], char ImplFileName[], eventBeforeAfter BeforeAfter);
    TStatus DefAbsTimeEv(char EventName[], time_t occp);
    ... // similar methods for other types of time events
    TStatus DefConj(event* ev1, event* ev2, char conj[], TSameClause option);
    TStatus DefDisj(event* ev1, event* ev2, char disj[]);
    TStatus DefSeq(event* ev1, event* ev2, char seq[], TSameClause option);
    TStatus DefStarOp(event* ev, char StarOp[], TSameClause option);
    TStatus DefHistOp(event* ev, char HistOp[], int times, TSameClause option);
    TStatus DefSimpleInterval(event* start, event* end, event* ev, char interval_ID[]);
    TStatus DefcompositeInterval(TInterval mode, event* s1, event* e1,
                          event* s2, event* e2, event* ev,
                          char interval_ID[]);
    TStatus DefSimpleNegat(event* start, event* end, event* ev, char not_ev[]);
    TStatus DefcompositeNegat(TInterval mode, event* s1, event* s2,
                       event* e1, event* e2, event* ev, char not_ev[]);

    TStatus DeleteRule(rule*& RuleToDelete);
    TStatus DeleteEvent(event*& EvToDelete);
    TStatus DeleteCondition(condition*& CondToDelete);
    TStatus DeleteAction(action*& ActToDelete);
};//CRuleManager
```

Figure 2. Interface of the Rule Manager

loaded dynamically during rule execution. As part of its state, each condition/action-object maintains the information necessary to determine the appropriate function. More details can be found in [43].

In a further step of rule compilation, actions to enable proper event detection at runtime must be executed. This is done in a way specific to the various kinds of events:

- in case of method events, the operation *raiseEvent(event-name)* is added at the beginning and/or at the end (before each `return` statement) of the method body. This has to be done manually. The file which contains the method body then has to be recompiled.
- in case of time events, operating system utilities (*crontab* and *atq*) are used to signal occurrences of time events. The necessary entries are generated automatically based on the event specification, and are inserted into operating system tables by the compiler.

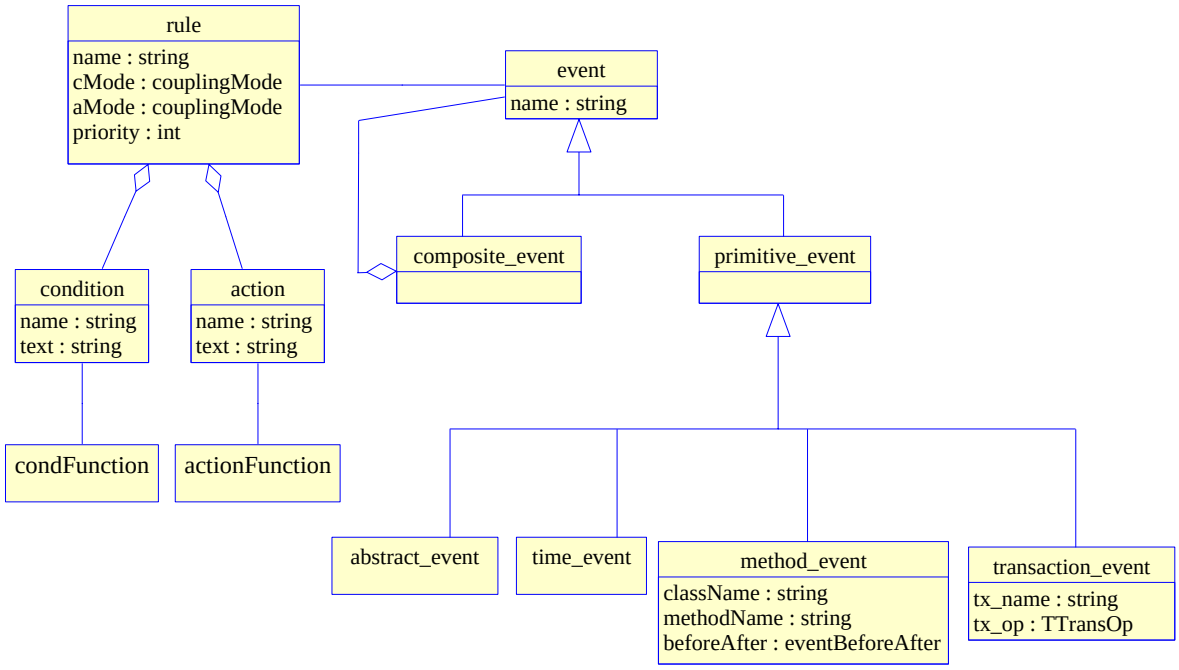

Figure 3. The SAMOS Rule Schema

- in SAMOS, we use Colored Petri Nets [33] for the detection of composite events. We have defined Petri Net types for all event constructors. Based on these types, SAMOS builds the Petri Net instance for each composite event description whenever a new composite event is defined. All these Petri Net instances together form the event detector for composite events.

No specific steps are necessary at rule definition time to enable the correct signalling of transaction events.

2.4.2 Rule Processing

Rule processing is subdivided into two phases:

- *event detection and signaling.*
- *rule scheduling and execution.*

In the following we discuss specific aspects of the implementation of rule processing on top of ObjectStore. Since event detection and rule processing need appropriate support from transaction management, we shortly discuss SAMOS transactions first.

2.4.2.1 SAMOS Transactions

SAMOS uses ObjectStore's model of closed nested transactions [54], which allows conditions and actions to be executed as subtransactions. Transaction management must be extended for three reasons:

- transaction events must be signalled,

- rules must be executed before commit processing takes place in case of the `deferred` coupling mode,
- the height of transaction trees needs to be restricted to control rule execution.

In ObjectStore, each transaction is represented as an instance of class `os_transaction`. Operations on transactions are implemented as methods of this class (e.g., `os_transaction::begin` starts a new transaction). SAMOS defines a new class `samTransaction` (Figure 4), which "wraps" ObjectStore's transaction class and offers the additionally needed functionality. From a programmer's point of view, this class defines methods to start, commit, and abort transactions. These three methods implement the transaction functionality using ObjectStore's transaction operations, but also add code in order to implement event signalling and rule execution properly.

```
class samTransaction {
    public:
    os_transaction      *ostx;              // pointer to ObjectStore transaction
    int                 user;
    int                 level;
    char                * name;
    os_list<ruleExec*>  list_of_rules;      // rules to be executed before commit.
    ...
    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
                                            // condYetEval specifies whether only action remains to
                                            // be executed
};
```

Figure 4. Definition of Class samTransaction

Transaction events (more precisely, those that actually occur in a composite event or are associated to an ECA-rule) are signalled within the corresponding transaction operation. Each transaction maintains a list of rules with `deferred` coupling mode which it has to execute before it commits. For each of these rules it is also recorded whether only the action is to be executed, or whether the condition needs to be evaluated first.

Finally, the height of transaction trees might need to be restricted as a stopgap solution to the rule execution termination problem. Since SAMOS provides 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 might 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. The start of a new transaction is only allowed if its level value in the transaction tree does not exceed this limit. Note that this is not the only way to ensure termination of rule processing; static analysis may be used at buildtime (see Section 3.3.1.3).

2.4.2.2 Event Detection

Event detection and signalling start as soon as the rule manager receives the message *raiseEvent* from a primitive event detector. The rule manager then retrieves the appropriate event object from the database, and determines whether rules are associated to this primitive event. These rules are then scheduled for execution.

In a second step, the rule manager checks whether the primitive event is used in an event composition. If so, the composite event detector is notified and the Petri Net component begins to play the token game. Using Petri Nets, composite events are detected in a stepwise manner. Each time a primitive event occurs, the detector checks if a step forward in detecting one or more composite events can be made. If yes, it marks the corresponding positions. As a result, one or more composite events may be signalled. The token game continues until no more composite events can be detected. For each detected composite event, it is also determined whether rules are defined for it. A detailed description of the detection of composite events can be found in [33].

Implementing the composite event detector in an object-oriented environment (using ObjectStore) enables 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 are kept persistent as well. Therefore, tokens are also represented as (persistent) objects.

2.4.2.3 Rule Scheduling and Execution

This phase processes the rules associated to one of the events detected in the previous phase. Each rule is scheduled according to its coupling mode:

- those with coupling mode `immediate` are executed instantly,
- those with coupling mode `deferred` are scheduled for execution before commit processing of the triggering transaction takes place, and
- those with coupling mode `decoupled` are executed in separate transactions.

Multiple rules are scheduled according to the sequence of the events by which they are triggered. In case multiple rules are triggered by the same event occurrence, these rules are scheduled according to their priorities. If no priority is specified, SAMOS chooses a rule arbitrarily.

Immediate Conditions or Actions

Conditions with the coupling mode `immediate` are executed directly after the event has been detected. For condition evaluation, a new subtransaction of the triggering transaction is started. If the condition evaluation yields a non-empty result, the action execution is scheduled. If the condition/action coupling mode is also `immediate`, the action is executed directly after the condition evaluation.

Deferred Conditions or Actions

In case the coupling mode of a condition is `deferred`, the rule manager informs the (SAMOS-)transaction that the condition evaluation is still pending. The condition will then be evaluated before the transaction commits. If it then returns a non-empty result and the condition/action coupling mode is either `immediate` or `deferred`, the action will be executed immediately.

If only the condition/action coupling mode is `deferred` (while the condition coupling mode is `immediate`), the rule manager will inform the triggering transaction to execute only the action before commit, because in this case the condition has already been evaluated.

Decoupled Conditions or Actions

`Decoupled` rules must be executed in newly started top-level transactions. In ObjectStore, it is not possible to start new top-level transactions from within another running transaction. In order to implement the `decoupled` mode, SAMOS uses a *demon process* that executes the `decoupled` rules (see Figure 5). Whenever a condition or action is to be executed in `decoupled` mode, the kernel's rule manager informs the demon to execute the condition/action. Processes use the inter-process communication facilities of Unix to communicate with the demon.

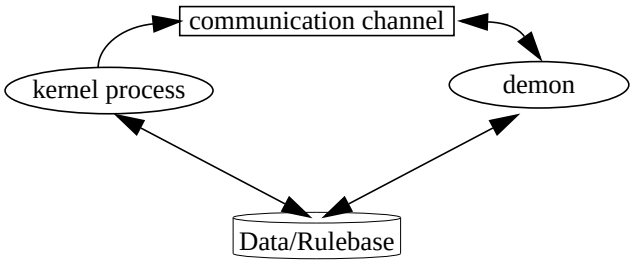

Figure 5. SAMOS Kernel and Demon Process Structure

During the action execution (regardless of coupling modes), further events may be signalled, leading to *nested rule execution*. Depending on the coupling modes of the newly triggered rules, nested rules are executed within the action execution.

2.5 Discussion and Evaluation

We conclude this section with a discussion and evaluation of SAMOS' rule language and implementation.

2.5.1 Functionality

The rule language of SAMOS offers a rich set of event types and constructors. It therefore is a powerful language which allows a broad range of real-world situations to be expressed. In our perspective, an advantage of the rule language is that expressive power does not compromise ease of use.

Instead, the set of event constructors is rather small. Furthermore, event constructors are orthogonal—only very few restrictions for the application of event constructors exist.

In addition, SAMOS offers three different coupling modes, which allow to express various intended semantics of rule execution. Using these coupling modes, the point of time when rules are executed as well as the relationship of triggered to triggering transactions (i.e., whether triggered transactions are commit-dependent) can be specified. We will report on a more detailed functional evaluation in Section 3.2.

2.5.2 Implementation

The evaluation of the implementation of the SAMOS prototype includes three different aspects:

- has it been possible to implement all the features prescribed by the rule and execution models?
- how efforteous has the implementation been (i.e., what are the construction costs)?
- how efficient is the implemented prototype?

We discuss performance evaluation in the next chapter and elaborate on the first two aspects immediately.

By applying the layered approach in SAMOS, the time needed to implement SAMOS was significantly shorter when compared to an implementation from scratch. This is because the “passive” part of SAMOS (persistence, queries, indexing, transaction management, etc.) could be reused and required no implementation efforts on our side. The current implementation of the SAMOS prototype 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. Therefore, the construction cost of the SAMOS prototype has been very low.

Pursuing the layered approach on top of ObjectStore, it has been possible to implement most, yet not all, of the rule and execution models' features in the SAMOS prototype. In particular, value events and class-internal rules have not been implemented. Implementing value events would have required to override assignment operators in ObjectStore's language. We would not only have had to do this for a significant number of operators (because assignment operators are specific to the type of the arguments), this technique would also have incurred tremendous performance degradation, because any update would have been signalled. We therefore concluded not to implement value events. In those cases where they would be needed, we recommend to encapsulate the update within methods, and to define appropriate method events instead. In this way, no real loss of functionality was experienced.

Value events are an excellent example of functionality whose implementation can only then be reasonable done if either the platform already provides it (as do current relational DBMS), or if internals of the underlying DBMS such as the object manager can be modified. This possibility would also have enabled us to add more functionality, or to implement some functions more efficiently.

As a first example in this respect, if ObjectStore would allow to start new top-level transactions in an asynchronous way, 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 and efficient since expensive inter-process communication with the demon would no longer be used.

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 block each other, or may even produce deadlocks. Improvements of this situation require the ability to customize internal components of ObjectStore. The possibility to add a more subtle concurrency control protocol for the Petri Net component (comparable to specialized concurrency control techniques for access paths) would not only help to prevent deadlocks, but would also increase performance in the presence of concurrent triggering transactions.

Finally, the transaction concept of ObjectStore does not provide for parent/child and sibling parallelism [49]. 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 [7]. 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.

3 Using SAMOS (1994-1998)

After the SAMOS prototype was completed, our focus turned towards using and evaluating it.

Obviously, the first question users would ask is what the additional runtime costs of using an active OODBMS are. Although it is clear that additional functionality very likely incurs some performance degradation, no performance measurements have been available at the time. Moreover, benchmarks for active DBMS did not exist. Therefore, we encountered the research question of how to evaluate and measure the performance of (object-oriented) ADBMS.

The second question is whether active mechanisms as proposed by SAMOS are useful at all. Can applications benefit from active OODBMS in such a way that they can implement tasks that were not doable before, or implement functions in a better or faster way than before? Thus, we faced the challenge to evaluate the functionality of SAMOS from an application perspective.

Finally, SAMOS offered a powerful mechanism to define ECA-rules. How can users make effective use of this functionality? In other words, how could the design of rule systems be supported? How could users be helped in understanding the meaning of ECA-rules (especially those that have been defined by other users)? Moreover, how can users be assisted in making sure that the rules they have defined exhibit the intended behavior? Finally, since actions of ECA-rules could easily generate further events, it could easily happen that rule execution ran into a cycle. Therefore, termination analysis became an issue that had to be treated in the context of usability.

We have addressed these questions in SAMOS as follows:

- *Performance issues of active object-oriented DBMS:* We have developed the Beast benchmark [41, 46] which helped evaluating the performance of SAMOS and other ADBMS.
- *Usefulness*: We evaluated SAMOS in various application scenarios, such as banking, consistency constraints, and workflow management.
- *Tools to support users during the implementation of ADBMS-applications:* The SAMOS tools [69, 70, 71] provide graphical interfaces supporting both, buildtime activities (performed during rule specification) such as rule editing, browsing, rule termination analysis, and runtime activities (performed at runtime, during the execution of an application) such as the understanding of rule behavior.

3.1 Performance Evaluation of ADBMS

The Beast[2] benchmark has been designed to help analyze the performance of SAMOS. We sketch the most important results here, for a detailed description, see [41].

Beast allows the performance of object-oriented ADBMS to be measured. It uses the schema and databases of the 007-benchmark for object-oriented DBMS [10]. It concentrates on the active functionality; for analyzing only passive functionality, 007 itself can be used. Beast has been used extensively to measure (and improve!) the performance of SAMOS [41]. A comparison with the performance of other ADBMS has also been pursued; results are reported in [46].

The tests proposed by Beast are subdivided into three classes:

- event detection tests,
- rule management tests, and
- rule execution tests.

2. BEnchmark for Active database SysTems

Event detection tests measure the performance of primitive and composite event detection. In composite event detection tests, the common event constructors such as conjunction, sequence etc. are used. Composite event detection tests also allow the performance of event restrictions (such as `same transaction`) to be measured. Event detection tests are particularly useful to compare different approaches to (composite) event detection.

Rule management tests have been devised to determine the overhead of rule and event object retrieval. Such an analysis is important mainly if rules and event types are (internally) represented as database objects.

The last group of tests focuses on performance of the rule execution component.

All tests have been designed in a way that the rule processing phase to be tested is stressed in the test (e.g., tests interested in event detection specify the simplest possible condition and action parts). For each test, the corresponding event is generated, and the elapsed time the ADBMS needs to process it (i.e., to detect possibly complex events, evaluate conditions, and execute actions) is measured.

Beast also allows to determine the impact of the number of rules and event types on ADBMS-performance. To that end, an arbitrary number of event types and rules can be created (in our tests, we considered 0, 50, 250, and 500 events and rules in addition to those needed by Beast itself). Although these events and rules are not needed in any of the Beast tests, they still can degrade performance of rule management and composite event detection.

Running Beast on several versions of SAMOS helped us significantly to find bottlenecks and to validate performance improvements. In an early version of SAMOS, the Petri Net implementation has been identified as a bottleneck, because the Petri Net structure had not been represented as an object graph, but as a set of independent objects. Therefore, numerous joins had to be executed in order to reconstruct the Petri Net. In the subsequent version, pointers had been used for representing the Petri Net structures, resulting in significant performance gains. In this version, event object storage and retrieval have been determined as major bottlenecks. The event extent (the set of all existing event types) had neither been clustered nor indexed, so that the rule manager had to scan the entire extent whenever an element of the extent was requested. After indexing and clustering have been added, more performance gains have been achieved. In the most recent version, we observed that event history management has an impact on performance. In particular, if the `same transaction` restriction is specified for a composite event, component occurrences cannot be used after their triggering transaction commits. If they are garbage collected, they will not be considered as candidate components during composite event detection, and therefore composite event detection is faster.

Further performance-related experiences have been made by comparing SAMOS with other systems (Acood [4], Naos [22], and Ode [38]). Here it turned out that the performance of primitive event detection and rule execution in SAMOS is comparable (or even slightly better) with that of the other systems. Composite event detection has been less efficient in SAMOS. This is because

SAMOS' consumption mode is *chronicle*, which implies a higher overhead for storing and retrieving event occurrences that sometime later might be used as elements of composite events. The other reason is that composite event detection itself based on Petri Nets, due to its complex structure of fine-grained objects, seems to be less efficient than other approaches. We therefore considered other approaches as well in subsequent implementations of other active systems (such as EvE [45], see below).

3.2 Use Cases

We used (or considered to use) SAMOS in a variety of application scenarios. We will discuss three of them.

In the first scenario, we evaluated the use of SAMOS for other database functions. In this case, advanced database functionality is available to users, while active mechanisms themselves are hidden from them. In particular, we considered consistency constraints in OODBMS as an application [40]. It should be noted that (to date) OODBMS provide only rudimentary support for consistency constraints. For constraint specification, we adapted Meyer's Programming by Contract [53]. Pre- and postconditions can be specified for methods and transactions. Invariants can be specified for single classes or across classes (i.e., so-called inter-object constraints). The active mechanism of SAMOS has been used to check and enforce these constraints. For the sake of brevity, we do not describe the detailed transformation process of constraints into ECA-rules. It should suffice to mention that method and transaction events as well as composite (sequence) events could be beneficially used. For invariants, we ideally would have used value events; since these are not available, we had to use the aforementioned workaround. Since invariants can be violated within transactions, but have to hold at commit time, we beneficially used the `deferred` coupling mode for this kind of constraint.

In the second scenario, we evaluated the usage of SAMOS in "active applications", i.e., those that use ECA-rules for addressing domain-specific problems. In particular, together with a major Swiss bank we considered ECA-rules in a banking environment [27], where ECA-rules can be used to perform automated portfolio management. A comprehensive design of an active application has been pursued for this application. We thereby found the rule language of SAMOS quite adequate. Putting this application into operation would have required that the already existing data and applications were migrated on top of ObjectStore, which however was not a serious option for the bank.

Finally, we considered the use of SAMOS as a platform for other software systems, namely workflow management and process-oriented environments [42, 67]. The objective hereby was to design a purely event-based system, so that a workflow/process is executed only by participants generating and reacting to events. While composite events as provided by SAMOS were mostly adequate, it turned out that SAMOS did not provide the right set of primitive events (e.g., for coordi-

nating processing entities in a workflow system) for this purpose. Furthermore, it was not clear how to extend SAMOS to a distributed system, and we were concerned about the performance of composite event detection. We therefore finally implemented a new (distributed) event engine, EvE [45], which provided for composite events most of which were also available in SAMOS, but used a different composite event detection technique. EvE is not designed as a full-fledged active database management system, and so we were free to tailor its functionality and implementation to distributed, event-driven systems.

The major conclusion we draw from (especially the last two of) the examples is that active mechanisms in general and the rule model of SAMOS in particular are adequate for a broad range of applications. The problems in using the prototype were mainly due to the fact that its functionality could not easily be extended and ported to another platform, and that the implementation could not be adapted (e.g., replacement of event detection techniques). This lead us to reconsidering the construction methods of active system in general, which will be discussed in Section 4.

3.3 The SAMOS Tools

A set of tools (illustrated in Figure 1) supports SAMOS users during the development and the maintenance of applications. We distinguish between two kinds of tools:

- buildtime tools which support activities performed during rule specification such as rule editing, browsing, rule termination analysis,
- runtime tools which support activities performed at runtime (i.e., during the execution of an application) such as the testing of rule behavior.

Buildtime tools communicate with the rule manager, while runtime tools exchange data with the event detector and the rule execution component. Subsequent sections will discuss the various SAMOS tools and the interconnections between them.

3.3.1 Buildtime Tools

The browser and the editor are graphical interfaces to retrieve and insert rule definitions from the rulebase in a user-friendly way. They have been implemented using the object-oriented application framework ET++ [72]. The termination analyzer is responsible for detecting rules that may generate nontermination as a consequence of their interactions. The way buildtime tools are working is illustrated in Figure 6.

3.3.1.1 Browser

The task of the browser is to support navigation through the rulebase. The required information is provided by the rule manager which makes use of the retrieval facilities of the underlying DBMS ObjectStore for querying the rulebase. The returned information includes:

- lists of all event-, condition- action- and rule definitions,

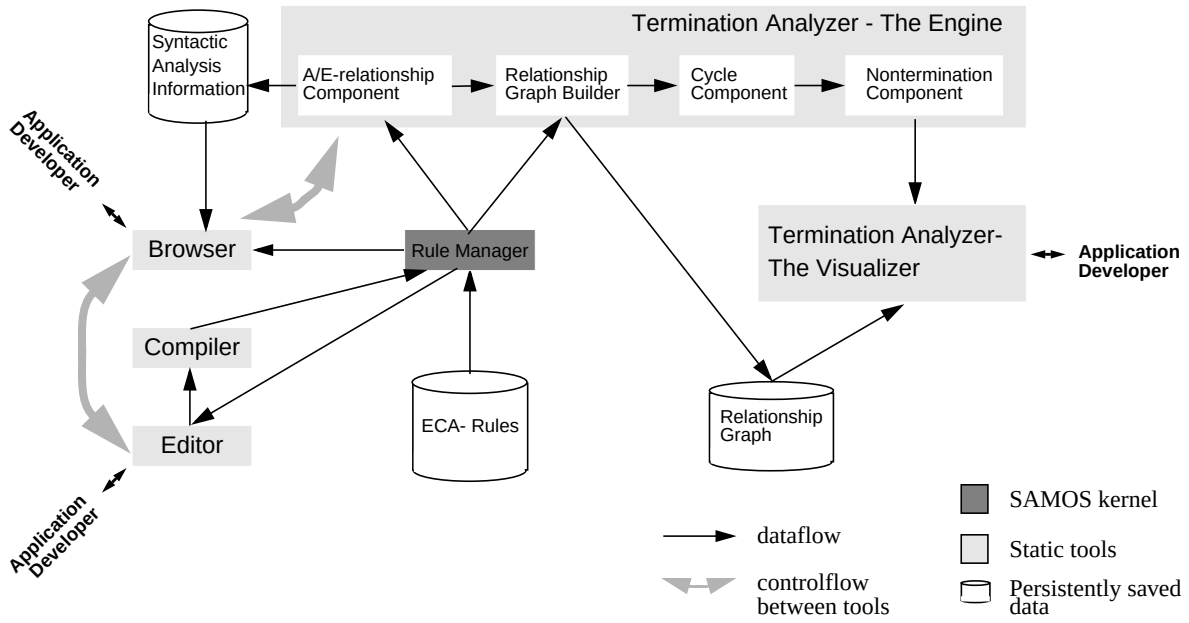

Figure 6. Buildtime Tools

- lists of definitions fulfilling certain criteria (e.g., list of composite events, list of time events, and so on),
- dependencies between parts of rule definitions, including the list of rules an event may trigger or the list of composite events in which an event is participating, etc.

The browser additionally provides users with information produced by the termination analyzer (during syntactic analysis) such as the primitive events that must occur in order to trigger a certain rule and the primitive events that may be raised by action execution of a certain rule. This information is stored persistently to avoid the repeated reanalyzing of rule definitions.

3.3.1.2 Editor/Compiler

The editor offers a graphical interface for the definition, modification and deletion of rules and their constituent parts. The developer may either specify parts of rules or reuse existing ones and then choose between available operators in order to create (or change) a rule definition. The editor translates the input in the rule language of SAMOS and invokes the rule compiler. The rule compiler is responsible for the syntactic and partly semantic analysis of rule definitions. The semantic analysis of events is mainly performed for interval-based events. If an interval does not make sense (e.g., the lower bound is larger than the upper bound), an error message is displayed. If syntactic and semantic analysis have been successful, the compiler uses the interface offered by the rule manager to create corresponding objects that are persistently stored in the rulebase.

3.3.1.3 Termination Analyzer

The termination analyzer is responsible for static rule analysis and assists users in checking the termination of rules.

During static analysis the *relationship graph* of a set of rules R is built. This is a directed, labeled graph where nodes represent rules. Edges represent the action/event relationships (also called A/E-relationships) between 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 only some component events are triggered or not.

If the graph includes a cycle and all edges are firm, then R will never terminate. 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 analysis to establish if the cycle is a real one. Finally, the rule sets for which termination cannot be proven are changed and analysis has to be repeated.

The termination analyzer (also illustrated in Figure 6) consists of an *engine* and a *visualizer*. The engine contains components that are responsible for the steps to be performed during termination analysis. The *A/E-relationship component* performs syntactic analysis in order to derive the relationships between the rules, i.e., it checks actions to find out if they may signal primitive events. The output of the A/E-relationship component is passed on to the *relationship graph builder* which constructs the relationship graph. Then, the *cycle component* detects all cycles in the graph. Finally, the *nontermination component* filters the output of the cycle component: it eliminates “false” cycles containing composite events by means of methods presented elsewhere [68, 69] and returns the remaining rule sets that may probably cause nontermination.

This output of the engine is displayed to the rule developer by means of the *visualizer*. The visualizer supplies the graphical interface for illustrating relevant information like the relationship graph and the cycles causing potential nontermination. It is implemented based on the interactive graph visualization system daVinci [29] which provides a universal layout tool for directed graphs.

The engine is also responsible for the connection between the visualizer and other static components of the environment, i.e., the browser and the editor. Nodes of the graph visualization may be selected and the browser and/or editor opened for them. In this way, the specification of the chosen rule may be examined or changed.

3.3.2 Runtime Tools

The SAMOS runtime tools, the testing and the explanation component, support checking of whether rule behavior corresponds to the intended functionality. Figure 7 illustrates the architecture and connections between the various components of the runtime tools.

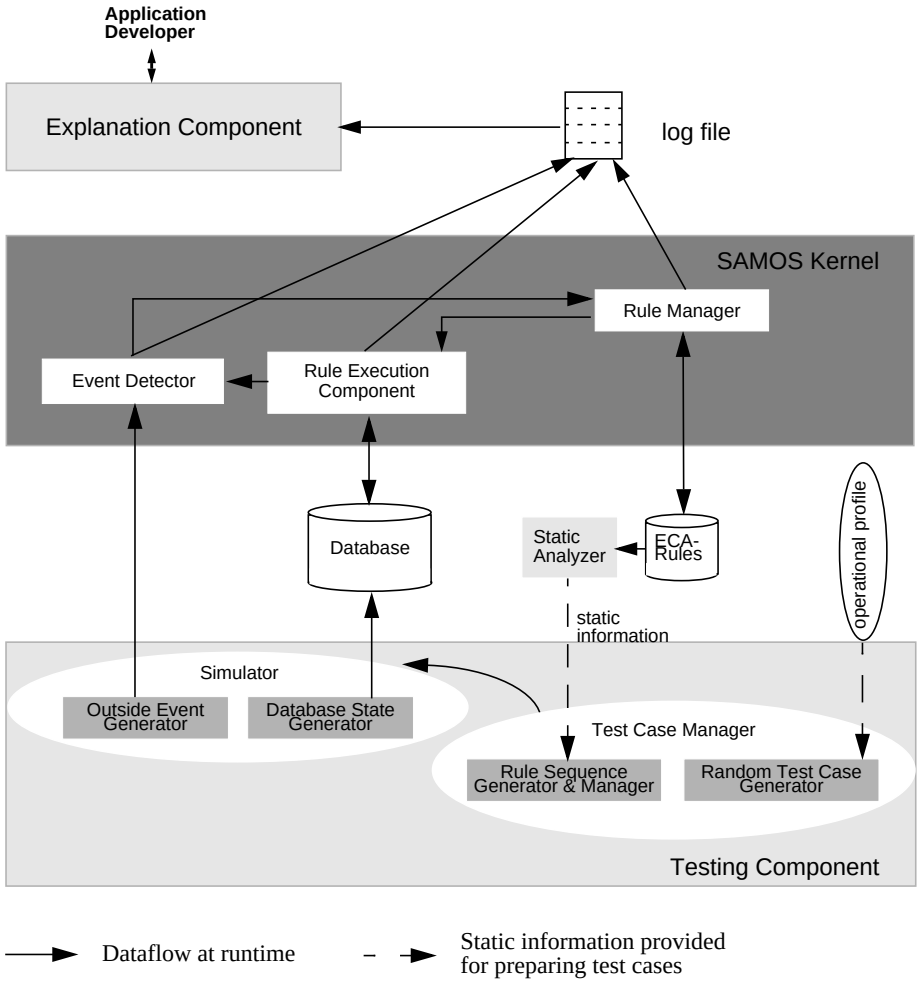

Figure 7. Runtime Tools

3.3.2.1 Testing Component

The aim of rule testing is to check if rule behavior exhibits the desired functionality. The rule testing process considers the execution of rule sequences based on test cases. For each test case, test data (i.e., the input data that start the execution of the test case) have to be generated. In rule testing we call these test data *test situations*. A situation s_i consists of a pair (e_i, db_i) reflecting that the event e_i occurred when the database state was db_i. Success or failure of each test case is established by comparing actual output results with expected results. Expected results are deduced from an *oracle (*the user or the rule requirements), which is a means to unequivocally decide whether the produced output is correct.

In SAMOS, we propose two methods for rule testing which are complementary to each other:

- a systematic method, the so-called *rule sequence coverage method* and
- a method based on random testing.

The *testing component* contains the infrastructure for effective and efficient testing such as tools to evaluate test cases, to generate test situations (event and database state generations).

One of the building blocks of the testing component is the *test case manager* which generates and manages test cases during the testing process. It consists of two components, one for each testing method: the *rule sequence generator* and the *random test case generator*. The rule sequence generator uses static information (e.g., the relationship graph produced by the termination analyzer[3]) to generate relevant rule sequences which must be tested. The random test case generator gets information from the operational profile of the rules.

The *rule sequence manager* administrates the set of all test cases that have to be considered. At the beginning of the test process, it builds the set of all test cases excluding irrelevant rule sequences (e.g., containing rules which can never trigger each other). During the testing process, the rule sequence manager removes rule sequences that have been executed.

The *simulator* receives the output of the test case manager and is responsible for the generation of events and database states. Generated events are always primitive events modeling interactions with the environment (e.g., user or device input). Therefore, the first task of the event generator is the identification of event definitions which may produce relevant event occurrences. Then, during the testing process the generator produces instances corresponding to each event definition. An instance is characterized by a timestamp and a list of parameters which depend on the event definition. Timestamps either already exist (for time events) or are generated for each event definition using modeling heuristics (like the discrete-event technique [60]). All generated event instances are collected into a calendar which is a chronological list of future event occurrences. Instead of waiting for events to be signalled in the environment, the simulator picks one element after the other from the calendar and processes it.

Concerning database state generation, the idea is to have for each rule sequence an initial database state that allows, when the appropriate events occur, the execution of the whole rule sequence. This requires all conditions of the rule sequence to be fulfilled at the moment when their corresponding rule is triggered. In this case, actions that may influence the evaluation of conditions have to be considered as well.

3.3.2.2 Explanation Component

The explanation component provides support for understanding and debugging the database system activity when using SAMOS. It is an autonomous component that may be transparently used either during the execution of an application or when the simulator is in use. Its task is to visualize rule behavior when rules are executed. Thus, the developer gets an idea about existing rule interactions and the context in which rules are triggered and executed. In particular, the explanation component graphically traces the signalling of events, shows which rules are consequently triggered and which actions executed. The required information is available in a log file that documents the output of the event detector, rule manager and execution component in a cer-

3. In this case rule testing corresponds to dynamic rule analysis

tain format and using a certain (primitive) language. The explanation component takes as input the log file and “translates” its content from a textual into a graphical language offering elements like boxes, links between them and colors which play an important role for explaining rule behavior.

The connection between SAMOS and the visualization tool is provided by UNIX pipes which allow the information written in the log file to be redirected as input to the explanation component. The explanation component has been implemented on the basis of the application framework ET++ [72].

3.4 Discussion

Experiences and lessons we extracted from the work on usability aspects of SAMOS can be summarized as follows.

We observed that in general active mechanisms are useful for a variety of purposes, ranging from DBMS-internal tasks such as constraint maintenance to event-driven systems such as workflow management (see also [61][4]). Tool support to assist users in developing ADBMS-applications (e.g., testing analysis) can be effectively provided. Note that tool support has been identified as one major obstacle of succesful use of ADBMS [64].

Nevertheless, the successful usage of ADBMS is restricted by three drawbacks. First, active mechanisms are of little use if they are not provided for the right passive platform, i.e., if they are not available for the (passive) DBMS operated by users to store their data. Second, different applications (such as those considered in our case studies) pose varying requirements to the functionality of ADBMS, and there is no general “one-size-fits-all” solution. Third, unless performance is satisfying for all of the ADBMS-components, users will refrain from using them [c.f. 64]. These three lessons lead us to reconsider construction techniques of ADBMS, to be discussed in the next section.

4 Construction of Active Systems Revisited (1998-)

Although research in active DBMS somewhat has cooled down during the late 1990s, event-based systems were becoming ubiquitous in this period. In addition to already mentioned application areas such as constraint maintenance and workflow management, new ones such as notification systems [62] (e.g., publish/subscribe), data warehouse refreshment [21], etc. have emerged.

As a consequence, we faced the question of how to support a broad spectrum of applications with active mechanisms. Some of these applications, such as workflow management, had already

4. While we agree with the authors of [61] that ECA-rules are not suitable means for the *specification* of workflows and processes, we have shown elsewhere that generalized event-based systems are indeed a viable option for workflow *execution* [45].

been addressed in our projects, yet the challenge was to come up with a general solution to provide active mechanisms in a way that they can be tailored to the specific requirements of the application area. Second, we had concluded that offering active mechanisms on a fixed passive platform significantly restricts the usability of SAMOS. We therefore faced the research problem of how to provide active mechanisms so that it can be used for ideally any passive platform. Finally, since performance of some parts of SAMOS has been a problem, the obvious challenge was how to allow parts of the ECA-subsystem to be exchanged (e.g., by more efficient ones).

All these questions and requirements refer to the way an ECA-subsystem is designed and implemented, and how tightly it is integrated with the passive part. Hence, in the final part of the SAMOS project, we have been reconsidering construction issues. The requirements described above are addressed as follows:

- the “active” subsystem is designed as completely independent from the passive part,
- a construction method allows to tailor active mechanisms to actual needs of applications, and then to integrate the active subsystem with the passive platform.

The first step is a prerequisite for the ability to combine specialized active subsystems with passive platforms. To that end, active subsystems have to be available in an “unbundled” form [47]:

- functions typically offered by active subsystems are extracted from ADBMS,
- all the services needed (from the passive platform) are identified and abstracted into general interfaces.

As a result, active mechanisms are represented in a general, abstract way independent of any concrete rule or execution model. Unbundling also helps abstracting from the concrete platform used (the passive part). In [36], we have described how active functionality can be unbundled.

During rebundling, rule and execution models are implemented according to the application needs. Viable implementation techniques (such as adequate, efficient event detection) are chosen, and the active subsystem is “hooked” together with the chosen passive platform. Thus, rebundling is the inverse activity of unbundling—unbundled components are specialized, instantiated according to the application requirements, and finally plugged together into a coherent active system. Thereby, the rule and execution model can be designed to meet the application needs (e.g., concerning the required event constructors).

Apparently, such an approach should rely on proven software engineering practices. In this respect, we have chosen (component) frameworks as the underlying paradigm. In the FRAMBOISE project [28], we investigate the design and implementation of a component framework intended to enable the construction of software systems, so-called ECAS, that allow the definition and execution of active mechanisms interoperating with specific DBMSs.

We are currently using FRAMBOISE for the construction of ECAS supporting active functionality required in the SIRIUS project [37]. In SIRIUS, active mechanisms are used for refreshment in data warehouses.

5 Conclusion

In this paper, we presented an overview of the SAMOS project and system. We have described the rule and execution models and the tool environment. We have also reported on the evaluation of the functionality and performance of the SAMOS prototype. The lessons and experiences we draw from the SAMOS project can be summarized as follows:

- from a functional perspective, the SAMOS model and implementation meets most of the requirements of a broad range of applications;
- runtime performance of several parts of the prototype is not really satisfying;
- the layered approach to the SAMOS implementation is feasible for experimental, research purposes, yet not for products.

Various use cases showed that the SAMOS prototype would have to be customized or extended in order to fully meet the encountered requirements. This led us to reconsidering the architecture of active (database management) systems. In the ongoing Framboise project, we therefore exploit componentware technology to achieve a higher degree of flexibility in the construction and architecture of this class of system.

Another lesson that—as we feel—not only applies to ADBMS but to experimental DBMS in general refers to the platform used for the development of (research) prototypes. Layered approaches often imply many workarounds; they may prevent some functionality to be implemented at all and/or may also imply performance degradation. Construction of experimental systems from scratch is not a viable strategy in academia, because this also requires the implementation of DBMS-subsystems which are not interesting in the respective context. Extensible database systems allow to focus on the really interesting parts. However, these systems are often not available over a sufficiently long period of time, and therefore researchers face the risk that the maintenance of such a system is abandoned in the course of their project. A lesson we therefore draw from the SAMOS project (and also from EvE) is that the database community should establish some kind of “open database software foundation” and that making systems (such as Exodus [9], Shore [11], Open OODB [73]) available should be rewarded much more than today.

6 References

1. The ACT-NET Consortium. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. *ACM Sigmod Record, 25:3*, September 1996.
2. M.P. Atkinson, F. Bancilhon, D.J. DeWitt, K.R. Dittrich, D. Maier, S.B. Zdonik. The Object-Oriented Database System Manifesto (a Political Pamphlet). *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan, December 1989.
3. E. Baralis, S. Ceri., G. Monteleone, S. Paraboschi. An intelligent database system application: The design of EMS. *Proc. 1st Int'l Conf. on Applications of Databases*, Sweden, June 1994.

4. M. Berndtsson, B. Lings. On Developing Reactive Object-Oriented Databases. In [16].

5. A. Blue, B.M. Brown, W.A. Gray. An Implementation of Alerters for Health District Management. *Proc. 6th British National Conference on Databases*, Cardiff, Wales, July 1988.

6. H. Branding, A.P. Buchmann, T. Kudrass, J. Zimmermann. Rules in an Open System: The REACH Rule System. In [57].

7. A.P. Buchmann, J.Blakeley J.A. Zimmermann, D.L. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. *Proc. 11th Int'l Conf. on Data Engineering*, Taipei, Taiwan, March 1995.

8. M.J. Carey, D.J. DeWitt, S.L. Vandenberg. A Data Model and Query Language for EXODUS. *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Chicago, Illinois, June 1988.

9. M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, J.E Richardson, E.J. Shekita, M. Muralikrishna. The Architecture of the EXODUS Extensible DBMS. In K.R. Dittrich, U. Dayal, A.P. Buchmann (eds). *On Object-Oriented Database Systems*. Springer 1991.

10. M.J. Carey, D.J. DeWitt, J.F. Naughton. The 007 Benchmark. *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Washington, DC, May 1993.

11. M.J. Carey, D.J. DeWitt, M. Franklin, N.E. Hall, M.L. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O.G. Tsatalos, S.J. White, M.J. Zwilling. Shoring Up Persistent Applications. *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Minneapolis, May 1994.

12. S. Ceri, J. Widom. Deriving Production Rules for Constraint Maintenance. *Proc. 16th Int'l Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990.

13. S. Ceri, J. Widom. Deriving Production Rules for Incremental View Maintenance; *Proc. 17th Int'l Conf. on Very Large Data Bases*, Barcelona, Spain, September 1991.

14. S. Ceri, J. Widom. Managing Semantic Heterogenity with Production Rules and Persistent Queries; *Proc. 19th Int'l Conf. on Very Large Data Bases*, Dublin, Ireland, September 1993.

15. S. Ceri., P. Fraternali, S. Paraboschi, L. Tanca. Active Rule Management in Chimera. In [75].

16. S. Chakravarthy (ed). Active Databases. *Special Issue of the Bulletin of the IEEE TC on Data Engineering 15:1-4*, 1992.

17. S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, R.H. Badani. ECA Rule Integration into an OODBMS: Architecture and Implementation. *Proc. 11th Int'l Conf. on Data Engineering*, Taipei, Taiwan, March 1995.

18. S. Chakravarthy, V. Krishnaprasad, E. Anwar, S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection. *Proc. 20th Int'l Conf. on Very Large Data Bases*, Santiago, Chile, September 1994.

19. S. Chakravarthy, J. Widom. *Proc. 4th Int'l Workshop on Research Issues in Data Engineering: Active Database Systems*, Houston, February 1994.

20. R. Chandra, A. Segev. Active Databases for Financial Applications. In [19].

21. S. Chaudhuri, U. Dayal. An Overview of Data Warehousing and OLAP Technology. *ACM SIGMOD Record* 26:1, March 1997.

22. C. Collet, T. Coupaye, T. Svensen. NAOS: Efficient and Modular Reactive Capabilities in an Object-Oriented Database System. *Proc. 20th Int'l Conf. on Very Large Data Bases*, Santiago, Chile, September 1994.

23. U. Dayal et al. *The HiPAC Project: Combining Active Databases and Timing Constraints. ACM Sigmod Record, 17:1*, March 1988.

24. U. Dayal. Ten Years of Activity in Active Database Systems: What Have We Accomplished? *Proc. Int'l Workshop in Active and Real-Time Database Systems*, Skövde, Sweden, 1995.

25. O. Deux. The O2 System. R.G.G. Cattell, eds.. Special Issue on Next-Generation Database Systems. *Communications of the ACM* 34:10, October 1991.

26. K.R. Dittrich, S. Gatziu. Time Issues in Active Database Systems. *Proc. Int'l Workshop on an Infrastructure for Temporal Databases*, Arlington, Texas, June 1993.

27. U. Flück. FAME for SAMOS, Implementation of an Application for Active Object-oriented Database Systems. *Diploma Thesis*, Department of Computer Science, University of Zurich, 95 (in german).

28. H. Fritschi, S. Gatziu, K.R. Dittrich. FRAMBOISE—an Approach to Framework-Based Ac-

tive Database Management System Construction. *Proc. 7th Int'l Conf. on Information and Knowledge Management*, Washington, November 1998.

29. M. Fröhlich, M. Werner. Demonstration of the Interactive Graph Visualization System daVinci. *Proc. DIMACS Workshop on Graph Drawing*, Princeton, 1994.
30. S. Gatziu, A. Geppert, KR. Dittrich. Integrating Active Mechanisms into an Object-Oriented Database System. In P.C. Kanellakis, J.W. Schmidt (eds). *Proc. 3rd Int'l Workshop on Database Programming Languages* (DBPL), Nafplion, Greece, August 1991.
31. S. Gatziu, K.R. Dittrich. SAMOS. An Active, Object-Oriented Database System. In [16].
32. S. Gatziu, K.R. Dittrich. Events in an Active Object-Oriented Database System. In [57].
33. S. Gatziu, K.R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In [19].
34. S. Gatziu, H. Fritschi, A. Vaduva. *SAMOS an Active Object-Oriented Database System: Manual*. Technical Report 96.02, Department of Computer Science, University of Zurich, 1996.
35. S. Gatziu, A. Geppert, K.R. Dittrich. The SAMOS Active DBMS Prototype (Demonstration). *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, San Jose, CA, May 1995.
36. S. Gatziu, A. Koschel, G. von Bueltzingsloewen, H. Fritschi. Unbundling Active Functionality. *ACM SIGMOD Record 27:1*, March 1998.
37. S. Gatziu, A. Vavouras, K.R. Dittrich. SIRIUS: An Approach for Data Warehouse Refreshment. Technical Report 98.07, Department of Computer Science, University of Zurich, June 1998.
38. N.H. Gehani, H.V. Jagadish. *Ode as an Active Database: Constraints and Triggers. Proc. 17th Int'l Conf. on Very Large Data Bases*, Barcelona, Spain, September 1991.
39. N.H. Gehani, H.V. Jagadish, O. Shmueli. Composite event specification in active databases: Model and implementation; *Proc. 18th Int'l Conf. on Very Large Databases*, Vancouver, August 1992.
40. 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, Springer Verlag.
41. A. Geppert, S. Gatziu, K.R. Dittrich. A Designer's Benchmark for Active Database Management Systems: 007 Meets the BEAST. In [63].
42. A. Geppert, M. Kradolfer, D. Tombros. Realization of Cooperative Agents Using an Active Object-Oriented Database Management System. In [63].
43. A. Geppert, S. Gatziu, K.R. Dittrich, H. Fritschi, A. Vaduva. Architecture and Implementation of the Active Object-Oriented Database Management System SAMOS. Technical Report 95.29, Department of Information Technology, University of Zurich, November 1995.
44. A. Geppert, M. Berndtsson (eds). *Proc. 3rd Intl. Workshop on Rules in Database Systems*, Skövde, Sweden, June 1997. LNCS 1312, Springer 1997.
45. A. Geppert, D. Tombros. Event-based Distributed Workflow Execution with EVE. *Proc. Middleware '98*. The Lake District, England, September 1998.
46. A. Geppert, M. Berndtsson, D.F. Lieuwen, C. Roncancio. Performance Evaluation of Object-Oriented Active Database Management Systems Using the BEAST Benchmark. *Theory and Practice of Object Systems* (TAPOS) 4:4, October 1998.
47. A. Geppert, K.R. Dittrich. Bundling: Towards a New Construction Paradigm for Persistent Systems. *Networking and Information Systems Journal 1(1)*, June 1998.
48. E. Hanson. The Design and Implementation of the Ariel Active Database Rule System. *IEEE Transactions on Knowledge and Data Engineering* 8:1, February 1996.
49. T. Haerder, K. Rothermel. Concurrency Control Issues in Nested Transactions. *The VLDB Journal* 2:1, 1993.
50. 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.
51. 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.

52. C. Lamb, G. Landis, J. Orenstein, D. Weinreb. The ObjectStore Database System. *Communications of the ACM* 34:10, 1991.
53. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 1988.
54. J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.
55. Object Design. *ObjectStore - Manuals for Release 3.0 SunOS*, 1993.
56. N.W. Paton, O. Diaz, M.H. Williams, J. Campin, A. Dinn, A. Jaime. Dimensions of Active Behaviour. In [57].
57. N.W. Paton, H.W. Williams (eds). *Rules in Database Systems*. Workshops in Computing, Springer-Verlag, 1994.
58. N.W. Paton (ed). *Active Rules in Database Systems*. Springer, New York, NY, 1999.
59. N.W. Paton, O. Diaz. Active Database Systems. *ACM Computing Surveys* 31:1, 1999.
60. U. W. Pooch, J. A. Wall. *Discrete Event Simulation: A Practical Approach*. CRC Press Boca Raton, Florida 93.
61. J. Reinert, N. Ritter. Applying ECA-Rules in DB-Based Design Environments. *Proc. CAD 98*, Darmstadt, Germany, March 1998.
62. D.S. Rosenblum, A.L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. *Proc. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, September 1997. LNCS 1301, Springer.
63. T. Sellis (ed). *Proc. 2nd Int'l Workshop on Rules in Database Systems*, Athens, Greece, September 1995. LNCS 985, Springer 1995.
64. E. Simon, A. Kotz-Dittrich. Promises and Realities of Active Database Systems. *Proc. 21st Int'l Conf. on Very Large Data Bases*, Zurich, Switzerland, September 1995.
65. Stonebraker, E.N. Hanson, S. Potamianos. The POSTGRES Rule Manager. *IEEE Transactions on Software Engineering 14:7*, 1988.
66. M. Stonebraker, A. Jhingran, J. Goh, S. Potamianos. On Rules, Procedures, Caching, And Views in Data Base Systems. Proc. ACM SIGMOD Int'l Conf. on Management of Data, Atlantic City, NJ, May 1990.
67. D. Tombros, A. Geppert, K.R. Dittrich. Design and Implementation of Process-Oriented Environments with Brokers and Services. In B. Freitag, C.B. Jones, C. Lengauer, H.-J. Schek (eds). *Object-Orientation with Parallelism and Persistence*. Kluwer Academinc Publishers, 1996.
68. A. Vaduva. Rule Development for Active Database Systems. *Doctoral Dissertation*, University of Zurich, Switzerland, August 98.
69. A. Vaduva, S. Gatziu, K.R. Dittrich. Investigating Termination Analysis for Expressive Rule Languages. In [44].
70. A. Vaduva, S. Gatziu, K.R. Dittrich:. Graphical Tools for Rule Development in the Active DBMS SAMOS (Exhibition Paper). *Proc. 13th Int'l Conf. on Data Engineering*, Birmingham, UK, April 1997.
71. A. Vaduva, S. Gatziu, K.R. Dittrich. Rule Termination in Active Databases: Solved and Unsolved Problems. Submitted for publication.
72. A. Weinand, E. Gamma, R. Marty. Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2), 1989.
73. D.L. Wells, J.A. Blakeley, C.W. Thompson. Architecture of an Open Object-Oriented Database Management System. IEEE Computer 25:10, October 1992.
74. J. Widom, R.J. Cochrane, B.G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. *Proc. 17th Int'l Conf. on Very Large Data Bases*, Barcelona, Spain, September 1991.
75. J. Widom, S. Ceri (eds). *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, 1995.