The SAMOS Active DBMS Prototype

Stella Gatziu, Andreas Geppert, Klaus R. Dittrich

Institut für Informatik, Universität Zürich[1]

Technical Report 94.16

Abstract

We describe SAMOS, an active object-oriented database management system prototype. SAMOS offers a powerful rule definition language, including a small yet powerful set of event definition facilities. It is able to detect primitive and composite events automatically and efficiently. Upon event detection, SAMOS executes rules attached to the occurred events.

1 Introduction

Active database management systems (aDBMSs) [1] have recently found great interest as a topic of database research, and restricted aDBMS-functionality is even offered by some commercial relational systems.

An aDBMS implements "reactive behavior" in that it is able to detect situations in the database and to perform corresponding actions specified by the user. Applications that require such reactive behavior are thus freed from performing "polling" in order to detect interesting situations and from encoding reactions (possibly redundantly) in applications.

SAMOS[2] [2, 3] is our home-grown active object-oriented database management system whose development has been started in 1991. To a large extent, SAMOS is operational now and thus is among the first demonstratable active object-oriented DBMSs.

The prototype implementation of SAMOS is characterized by a *layered architecture*. All active components are built "on top" of a passive object-oriented database system, which is left unmodified and is a black box for SAMOS. This is in contrast to *integrated architectures*, where the kernel DBMS can be internally modified. The reason for applying the layered approach in SAMOS is the shorter implementation time in comparison to the integrated architecture, since the passive part can be reused in its entirety. Additionally, an open or configurable DBMS has not been available at the time the project started.

---

1. Authors' address: Institut für Informatik, Universität Zürich, Winterthurerstr. 190, CH-8057 Zurich, Switzerland. Fax: +41-1-363 0035, Email: {gatziu | geppert| dittrich}@ifi.unizh.ch

2. **S**wiss **A**ctive-**M**echanism based **O**bject-oriented database **S**ystem

The prototype implementation of SAMOS uses ObjectStore™ [8] as the underlying passive DBMS. ObjectStore is a commercial object-oriented database management system developed by Object Design as a persistent extension of C++. SAMOS currently runs on SUN machines under UNIX.

The remainder of this paper is organized as follows. The next section describes the notion of ECA-rules in SAMOS. Section 3 gives an overview of the SAMOS architecture and implementation, and section 4 gives a small example. The paper concludes with the description of the current state of SAMOS and of the subjects for future work.

2 ECA-Rules in SAMOS

SAMOS is an *object-oriented* active DBMS. Active features have been integrated into an object-oriented data model. Concepts like classes, object identity, modelling of behavior, inheritance, have been exploited [2]. In addition to the data definition language of ObjectStore, SAMOS provides a *rule definition language* as a means to specify Event-Condition-Action-rules (ECA-rules). It provides constructors for the definition of events, conditions, and actions. 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>
```

Example 1. Syntax of Rule Definitions

A rule definition specifies an *event description*, a *condition*, an *action* and execution constraints (*priorities* and *coupling modes*).

Once a set of rules has been defined, SAMOS monitors the relevant events. In this way, it *detects* the occurrence of each relevant event and afterwards it notifies the component responsible for the rule execution about this occurrence (*event signalling*). Consecutively, all rules which contain this signalled event are *triggered* and must be *executed*. The rule execution incorporates the condition evaluation and the action execution. First, the rule's condition is evaluated and —if it is met— the action is executed.

An event description defines a pattern of events. Each time such a pattern can be instantiated by concrete events, the event is said to have *occurred*. Note therefore that two levels of abstraction are important with respect to the notion of "event": the notion of event description corresponds to the type level, while the notion of event occurrence refers to a particular event instance. Subsequently, we simply use the term "event" whenever the concrete meaning is clear.

Events can be primitive (time events, message sending events, value events, transaction events, or abstract events) or composite events. The latter can be constructed out of primitive or other composite events. SAMOS supports a variety of composite event constructors (see Table 1).

name	signature	composite event is signalled when
sequence	$e_1 ; e_2$	e_1 and then e_2 occur
conjunction	e_1 , e_2	e_1 and e_2 occur regardless of order
disjunction	$e_1 \mid e_2$	e_1 or e_2 occurs
negation	$! \, e_1$	e_1 does not occur in a given time interval
reduction	times (e_1, n)	e_1 occurs n times
	$* \, e_1$	e_1 occurs for the first time

Table 1. Event Constructors in SAMOS

The *event history* in SAMOS consists of all occurrences of the defined event descriptions. Each database contains all relevant event descriptions and rule definitions. The history begins at the point in time when the first event description is defined and ends when all event descriptions are deleted. Usually, the history lasts over many sessions and obviously over several transactions. Thus, it is possible to signal a composite event based on events that have occurred in different application sessions or transactions.

Some event constructors require a *time interval* to be specified for the event pattern (the last three in the table). Time intervals are defined through a start and an end time. For the first three constructors, time intervals are optional. When specified, a time intervals defines when the component events have to occur in order to let the composite event occur.

Furthermore, *event parameters* can be used in SAMOS. Examples for event parameters are the identifier of the transaction that triggered an event, or the identifier of the object that received a certain message (in case of method events). Event parameters can be used to further restrict composite events to actually interesting ones. For example, the same transaction condition attached to a composite event requires that all the component events must be triggered by the same transaction.

Both, conditions and actions must be programmed in the data manipulation language (DML) of ObjectStore (i.e., extended C++). *Coupling modes* specify the point of time when condition evaluation and action execution begin. The transaction that contains an event occurrence is called the *triggering transaction* of the event. The first coupling mode of a rule specifies when the condition part of the rule are executed with respect to the triggering transaction. The second coupling mode determines when the action part is executed with respect to the condition evaluation. SAMOS supports the following coupling modes:

- *immediate*: directly after the detection of the triggering event / directly after the condition evaluation,
- *deferred*: at the end of the triggering transaction, but before commit / directly after the condition evaluation,
- *decoupled*: in a separate transaction.

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

More details on rule specification in SAMOS can be found in [2], details of the event definition part of the rule definition language are described in [4].

3 The SAMOS Prototype

3.1 Overview

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, and
- a set of tools.

The top-most SAMOS layer consists of tools, such as a compiler and a rule analyzer. The compiler takes ECA-rules (defined in the syntax given above) as input, checks their syntactic and semantic correctness, and updates the rule base appropriately. The analyzer supports the checking of further semantic properties for rulebases, such as termination.

The intermediate layer consists of a number of (runtime) components (Figure 1):

- a *rule manager* for the 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.

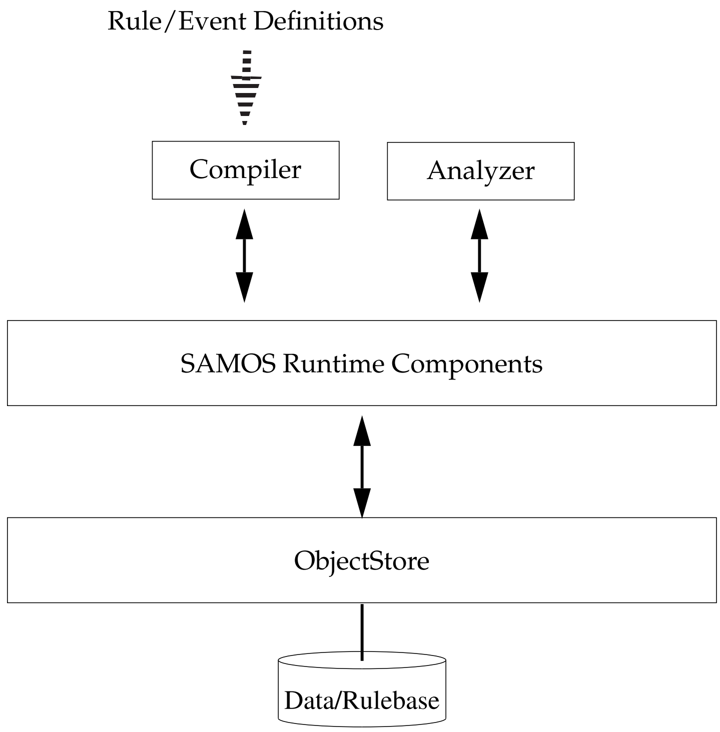

Figure 1. The Architecture of SAMOS

These components together with ObjectStore provide for the two major tasks of SAMOS:

- the mapping of rule definitions onto ObjectStore data structures, and the appropriate initialization of SAMOS for event detection and rule execution,
- the detection of events and the execution of rules.

In the following, both aspects are discussed.

3.2 Rule Definition and Storage

The compiler/analyzer performs syntactic and semantic analysis of rule definitions. To make this information persistent, it has to be a part of the *rulebase*. The structure of the rulebase is described by the *rule schema.* Since SAMOS is an object-oriented database system, event descriptions and rules are represented as objects [4]. Thus, for such an object, all object-oriented characteristics are available. Since all these objects are stored in ObjectStore, atomicity and durability is supported for operations on the rulebase. Moreover, events that participate in compositions are made persistent whenever necessary, and SAMOS can even monitor composite events whose component span multiple sessions.

All user-defined event descriptions are objects of the class event_description (Figure 3). Henceforth, these objects are called "event objects". A detailed description of the class hierarchy starting with *event_description* can be found in [4]. The attribute list_of_rules de-

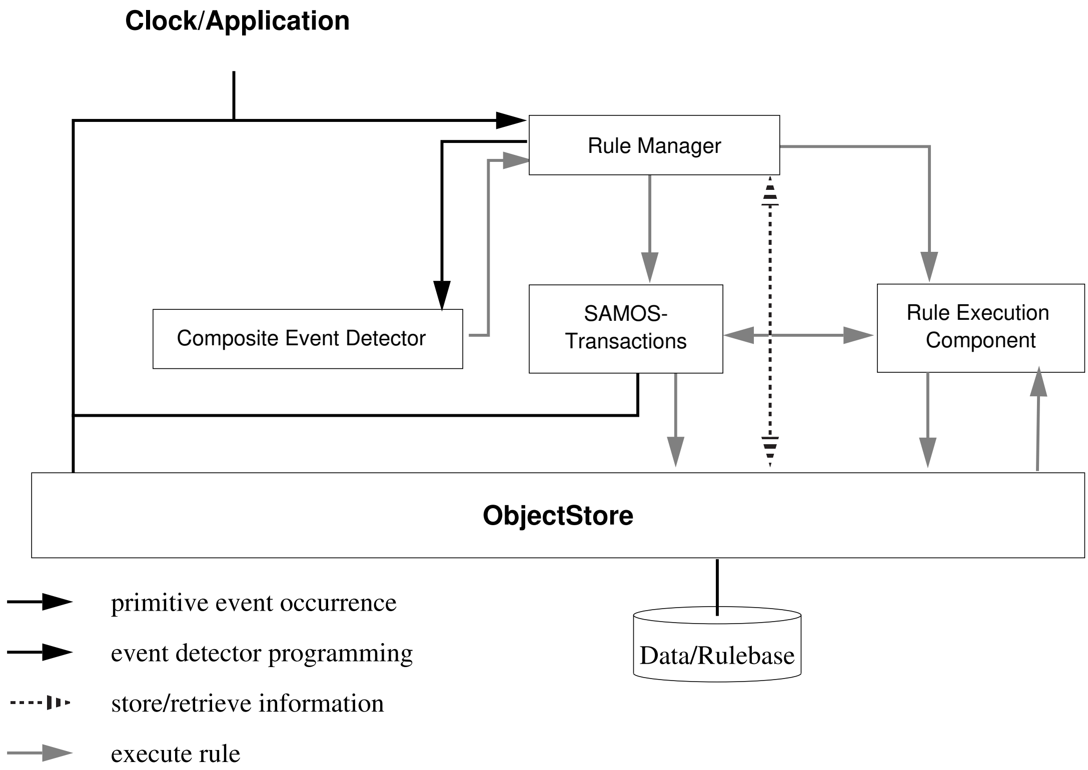

Figure 2. The Intermediate Layer of SAMOS

termines the rules (i.e., references to objects of the class rule) that have to be executed when this event has occurred. The order in list_of_rules depends on the rule execution constraints (condition evaluation coupling mode). The attribute composite_event determines the composite events in which this event participates.

Rule definitions are stored as instances of the class rule ("rule objects"). Their attributes include a name, the coupling modes, and the condition and action. Conditions and actions are compiled, resulting in two executable C++ code fragments. SAMOS transaction operations (see below) are added to the condition and the action code, respectively. The executable code fragments are linked to the program code of SAMOS, and references of rule objects to these fragments are defined.

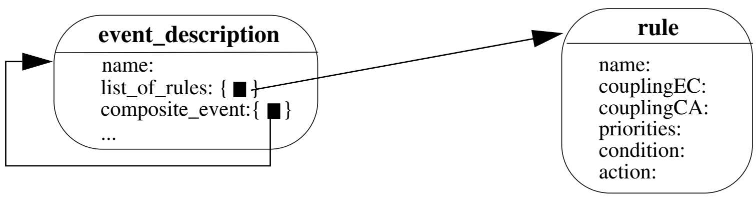

Figure 3. A Part of the Rule Schema

3.3 Event Detection

3.3.1 Initialization of Event Detection

After an event description has been defined, SAMOS must be initialized to detect occurrences of this event. The system (actually the rule manager) "knows" only about the occurrence of an event after it has received the message raise_event. This message is sent from the appropriate event detector.

Concerning the detection of composite events, *step-by-step* (or incremental) detection is applied. In this case, the event detector needs to know for each composite event the ordered sequence of all primitive events that need to occur in order for the composite one to happen. Thus, each time a primitive event occurs, the detector can check if a "step forward" in any of these ordered sequences can be made. If yes, it "marks" the attained positions. As a consequence, we can talk about the actual state of detection processes which is represented by the marked positions. The event detector can signal the occurrence of a composite event (in that the message raise_event is called) as soon as the last element of the appropriate sequence order gets marked. The step-by-step detection 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, and we have defined Petri net types for all constructors of the event algebra. Based on these types, we can build the Petri net instance for each user-defined composite event description. All these Petri net instances together form the event detector for composite events. A detailed description of the detection of composite events can be found in [5].

3.3.2 Activities Upon Primitive Event Detection

Given the facilities to define events and rules and to compile these definitions, we now turn towards the activities upon the occurrence of a primitive event. These activities comprise the *composite event detection* and *rule execution*.

The occurrence of a primitive event is signalled in that the message raise_event is sent to the rule manager. The rule manager then retrieves the ObjectStore object representing the signalled event. Efficient retrieval of such event objects is a crucial performance requirement, especially if numerous event descriptions have been defined. In SAMOS, this retrieval is efficient for three reasons. First, in the case of method events, time events, and abstract events the raise_event operation is only performed for *events of interest*, i.e., for those events that actually occur in rule definitions (i.e., no "useless" event signalling). Still, the set of such events can become rather large. Therefore (and second), SAMOS exploits the indexing facilities of ObjectStore to efficiently map event names to objects representing events. Third, event objects are clustered, using again the appropriate facilities provided by ObjectStore. Based on this clustering of event objects, the likelihood of page I/O in this case is decreased.

After the event object has been found, the rule manager has to determine:

- whether this primitive event is used in an event composition. Each event object contains an instance variable composite_event which stores the composite event(s) in which the appropriate event participates (Figure 3). If the event is used in composite event definitions, the composite event detector is informed. The Petri net begins to play the token game. As a result, one or more composite events may be signalled. In this case, the system behaves as if a primitive event has occurred, i.e., the object corresponding to the composite event description is retrieved from the rulebase.
- whether rules are defined for this event. The objects representing the rules to be executed are determined through the references included in *list_of_rules* (Figure 3).

3.4 Rule Execution

The rule execution component is responsible for rule execution and is called with the list of references to rules to be executed, the actual event parameters and the triggering transaction as parameters.

To implement the rule execution in the framework of the transaction model of ObjectStore, we defined a new class *samTransaction*. Each instance of *samTransaction* contains a reference to an ObjectStore transaction and further informations required by the rule execution like a rule register which implements a queue that includes all references to rule objects that have to be executed at the end of this transaction (i.e., deferred). This class defines methods for the transaction operations *start*, *commit*, and *abort* which must be used in the application programs for SAMOS instead of the transaction operations proposed by ObjectStore.

We will now elaborate on rule execution under consideration of coupling modes. If the coupling mode of the condition is immediate, the message execute_rule is sent to the rule execution component as soon as the respective event occurrence has been detected.

If condition evaluation has to be deferred, the condition evaluation must be run after the last (user-specified) operation, but before committing the triggering transaction. The rule object is added to the transaction *rule register* upon event detection.

Finally, decoupled condition evaluation is realized in that the rule execution component starts a new top-level transaction. In order to execute triggered top-level transactions, the SAMOS process architecture contains an additional *demon process*, whose purpose is to execute decoupled triggered transactions. Whenever a rule has to be executed in decoupled mode, the demon process is informed through interprocess communication facilities.

The same principles as for condition evaluation coupling modes apply to the different coupling modes for action execution.

Multiple rules may trigger upon the same event. SAMOS determines and enforces the execution order in case of more than one triggering rule based on the information on coupling modes and priorities.

If multiple rules with different coupling modes trigger, the triggered transactions with coupling mode decoupled are started first, followed by the immediate triggered transactions. Last, deferred transactions are added to the transaction rule register.

The process sketched above may continue recursively: triggered transactions can act as triggering transactions themselves (*nested rule execution*). Note that ensuring the termination of nested rule execution is a problem, but so far is not addressed by the implementation of SAMOS (i.e., the user is responsible for ensuring the termination of rule execution).

4 An Example

We will now consider a banking application for stock management and use SAMOS for its implementation. A concrete bank records information about its stock dealers, the stocks it deals, customers, and stock sales. Assume the class Dealer as given in Example 2.

```
class Dealer{
    public:
        persistent<stockDB> os_Set<Dealer*> *extent;          // the class extension
        Dealer(char* name)    ;                                // the constructor
        ~Dealer();                                             // the destructor
        Deal * makeDeal(Stock *theStock, Customer *cus, int quantity, int price);
                                                               // method for making deals
        int getProfits (int range);         // computes the profit of the last   range  deals
        int getBonus(int amount);

        char *Name;
        os_List<Deal*> myDeals;                        // the deals made by the dealer
    };
```

Example 2. Class Definition for the Stock-Application

Consider now the active part of the application that is described by rule R1: let a dealer be rewarded whenever he/she performs at least 50 sales within one week and the total profit of the deals is more than 25 millions.

```
DEFINE EVENT FiftyDeals
    TIMES (50, Dealer.makeDeal): same object IN EVERY WEEK [MO - FR]

DEFINE RULE R1
ON          FiftyDeals
IF
            if (oid->getProfits(50) >= 25000000)
                    return 1;
            else
                    return 0;
DO
            oid->getBonus(1000000);

COUPLING (deferred, deferred)
```

Example 3. Sample ECA-Rule for the Stock Example

A simplified sample application program for the stock application is shown in Example 4. This programs collects input information for a single stock deal, and calls the appropriate method defined for the class Dealer. After this method has been called 50 times for a dealer, the composite event FiftyDeals occurs and the rule is executed.

```
main( int argc, char **argv) {
        Dealer * d; Customer * c; Stock * s; int q; int val; Deal * de;
        objectstore::initialize();
        os_collection::initialize();

        samTransaction* stx = samTransaction::begin();
        ... // get input values for the dealer etc.
        de = d->makeDeal (s, c, q, val);
        samTransaction::commit(stx);
        db->close();
    }
```

Example 4. An Application Program

5 Project Status and Future Work

The actual state of the SAMOS prototype and project is as follows. The major parts of SAMOS (including rule definition, persistent storage of rule and event objects, event detection, and rule execution) are operational. The prototype demonstration will illustrate rule definition, event detection, and rule execution in SAMOS with a sample application.

We are currently investigating performance of active object-oriented DBMSs [7] and are comparing SAMOS with other aDBMSs from a performance point of view. One objective of the developed benchmark is to identify and to repair performance weaknesses in SAMOS.

We are also looking into some potential applications for active database systems. These application domains include DBMS-internal tasks such as consistency maintenance [6] as well as applications on top of DBMSs, such as banking, workflow management, and software engineering environments.

6 References

1. U. Dayal: *Active Database Management Systems*. Proc. 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem, 1988.
2. S. Gatziu, A. Geppert, K.R. Dittrich: *Integrating Active Mechanisms into an Object-Oriented Database System*. Proc. of the 3rd Intl. Workshop on Database Programming Languages (DBPL), Nafplion, Greece, August 1991.
3. S. Gatziu, K.R. Dittrich: *SAMOS: An Active, Object-Oriented Database System*. Bulletin of the IEEE TC on Data Engineering 15:1-4, 1992.
4. S. Gatziu, K.R. Dittrich: *Events in an Active Object-Oriented Database System*. In N.W. Paton, H.W. Williams (eds): Proc. Workshop on Rules in Database Systems, Edinburgh, UK, September 1993 (Workshops in Computing, Springer-Verlag, 1994).
5. S. Gatziu, K.R. Dittrich: *Detecting Composite Events in an Active Database Systems Using Petri Nets*. Proc. of the 4th Intl. Workshop on Research Issues in Data Engineering: Active Database Systems, Houston, February 1994.
6. A. Geppert, K.R. Dittrich: *Specification and Implementation of Consistency Constraints in Object-Oriented Database Systems: Applying Programming-by-Contract*. Technical Report 94.14, Computer Science Department, University of Zurich, October 1994.
7. A. Geppert, S. Gatziu, K.R. Dittrich: *007 Meets the Beast: Performance Evaluation of an Active Object-Oriented Database Management System*. Technical Report, Institut fuer Informatik, Universität Zürich, October 1994.
8. C. Lamb, G. Landis, J. Orenstein, D. Weinreb: *The ObjectStore Database System*. CACM 34:10, 1991.