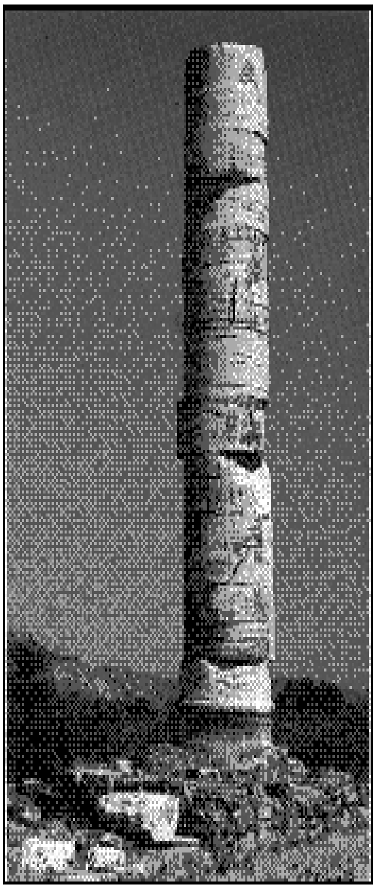
Institut für Informatik der Universität Zürich

# SAMOS an Active Object-Oriented Database System: Manual

Stella Gatziu, Hans Fritschi, Anca Vaduva

Nr. 96.02 February 96



# SAMOS

## An Active Object-Oriented Database System

## Manual


Dr. Stella Gatziu

Hans Fritschi

Anca Vaduva

Database Technology Research Group
Prof. Dr. Klaus R. Dittrich
Computer Science Department
University of Zurich

# List of Contents

# 1 Introduction

*Active database systems* were introduced approximately ten years ago and have been of increasing interest in the database research area ever since. In addition to the functionality supported by a conventional database system (which could be called *passive*), an active database system registers events, conditions, actions and the associations between them by means of ECA-rules (Event-Condition-Action). An event indicates a point in time when the database system has to react, and a condition relates to a database state; it is evaluated when the occurrence of the corresponding event is signalled. If the condition holds, the associated action is executed. An action can be any executable program, including database operations.

The need for active behavior in a database system was recognized some time ago and first approaches were presented in the early 1970's. However, a true research field emerged relatively recently in the mid-to-late 1980's with the pioneer project HiPAC. Recently the research area of active database systems has received a lot of attention and is growing very quickly. In early 1991 the first projects that dealt with active object-oriented database systems were started: Ode at AT&T, Sentinel at the University of Florida and SAMOS at the University of Zurich. SAMOS is now operational and thus is among the first demonstrable active object-oriented DBMSs.

SAMOS has been implemented based on a layered architecture where all the components that implement the active behavior are built as a layer "on top" of a conventional passive database system. For the prototype implementation of SAMOS we chose the commercial object-oriented database management system ObjectStore as a base.

SAMOS Release 1.0 which is available as public domain offers:

- the whole functionality of an object-oriented database management,
- a powerful rule definition language, including a small yet powerful set of event definition facilities,
- a compiler for defining rules and events,
- event detection automatically and efficiently,
- upon event detection, execution of rules attached to the occurred events

- a browser for navigating through rule and event definitions, and
- a simplified explanation component.

### The History of the SAMOS Project

The SAMOS project started at the beginning of 1991 in the database technology research group at the University of Zurich. The results obtained from 1991 to 1995 have been consolidated in the Ph.D. dissertation of Stella Gatziu [3]. In January 1995, two new Ph.D. students, Anca Vaduva and Hans Fritschi, joined the project. Anca focuses on the development of tools to support the user during the development and the maintenance of an active application, and Hans focuses on a new "open" SAMOS architecture. Hans has mainly participated in the development of the SAMOS prototype kernel. Furthermore, several master students have contributed with their semester and diploma theses to the implementation of SAMOS. The project is supervised by the head of the database technology research group at the University of Zurich, Prof. Dr. K. R. Dittrich.

### Acknowledments

Many thanks to Andreas Geppert who has contributed numerous interesting ideas to the project over all these years.

Many thanks also to the students who worked on the project in the context of semester or diploma theses, and in particular to Thomas Meyer for his contribution to the implementation of the composite event detector and the implementation of rule execution.

Finally, we would like to thank the UBS Information Technology Laboratory which has funded the SAMOS project until 1995.

### Outline of the Handbook

This documentation begins with instructions for installing SAMOS on your site. Then we discuss the main features of SAMOS. The next chapter shows step-by-step how SAMOS can be used for implementing active applications. The handbook concludes with the presentation of the syntax of the SAMOS rule language.

# 2 Installation Guide

SAMOS for SPARC SunOS is distributed as a compressed 'tar' file. When you install it as described by this installation procedure, you will place approximately 25 megabytes of SAMOS software into respectively below the directory you choose. In order to execute SAMOS applications, the object-oriented database management system ObjectStore™ has to been installed on your system. Furthermore, you need OpenWindows™ 3 to be able to execute the graphical components of SAMOS.

You unload the SAMOS distribution file *samos1.tar.Z*. The installation procedure will create a hierarchy of directories below the target directory, wherein the SAMOS files are placed. This arrangement is mandatory and must not be changed.

Once the software is accessible to a given host, you must configure it to run SAMOS.

## 2.1 Installing SAMOS

1. Create the root directory for the SAMOS installation, e.g.,
   **mkdir samos**

   We recommend you to set now the environment variable SAMOSROOT to this directory, e.g.,
   **setenv SAMOSROOT ~/samos**
   Note that the definition of this variabe is not mandatory at this stage but it is convient due to following installing.

2. Move the SAMOS distribution file *samos1.tar.Z* into this directory.
   **cd samos**
   **cp samos1.tar.Z .**

3. Uncompress the distribution file *samos1.tar.Z*
   e.g.
   **uncompress -v samos1.tar.Z**

4. Expand the uncompressed file *samos1.tar*
   e.g.
   **tar -xvf samos1.tar**

5. Install the required ObjectStore application schemas, which are placed in the subdirectory *$SAMOSROOT/os_schemas* and patch the delivered tools that they reference the installed schemas. Installing the schemas means that they are to be placed in the directory where the application schema databases are to be found according to your ObjectStore policy. We reference this directory as *schema directory.*
   You can perform the whole task by calling the the shellscript *setup_schema*: Go into the subdirectory *os_schemas* and call the shellscript *setup_schema* as follows:
   **cd os_schemas**
   **setup_schema <schema directory> $SAMOSROOT**

   There are four application schemas delivered:
   \- *SamosCompiler.app_schema*
   \- *SamosEt.app_schema*
   \- *SamosTools.appschema*
   \- *samoslib.libschema*

   Afterwards the applications placed in the *$SAMOSROOT/bin* directory are to be patched to look for their application schema in the schema directory. This can be achieved by the ObjectStore command *ossetasp* (cf. ObjectStore manuals). The applications and their respective aplication schemas are:

   \- *ECA* < -> *SamosCompiler.app_schema*
   \- *SamosEt* <-> *SamosEt.app_schema*
   \- *createRuleDB, rmRuleDB, samosshell* <-> *SamosTools.appschema*

   The SAMOS library has not to be patched to the library schema *samoslib.libschema.* The whole task (placing the schemas and patching the applications).

## 2.2 Setting the environment

The most important part is the SAMOS library, so that the programs that you will write have access to SAMOS. You need to set up some environment variables for SAMOS. The best way to do this, is to include it in your *.cshrc* file. We assume that environment variable for ObjectStore are already included in your *.cshrc*. Otherwise please see the ObjectStore Manuals. Here are the environment variables necessary for SAMOS:

- OSUSER defines an ObjectStore user and points to the directory which includes his/her databases :

**setenv OSUSER <schema directory>**

- SAMOSROOT specifies the root directory, where SAMOS is located at:

  **setenv SAMOSROOT <path>**

- OS_LIB_SCHEMA specifies the library schema path; add the path to the directory where the library schema of the SAMOS library is situated:

**setenv OS_LIB_SCHEMA_PATH <schema directory>:$LD_LIBRARY_PATH**

- The path LD_LIBRARY_PATH must be extented as follows:

  **setenv LD_LIBRARY_PATH $SAMOSROOT/lib:$LD_LIBRARY_PATH**

# 3 User Guide

SAMOS is an active object-oriented database management system prototype. It 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. In this chapter we describe the main features of SAMOS and we show briefly how SAMOS, can be used for implementing active applications. Further details on the implementation of active applications are given in tutorial chapter. The aim of this chapter is to give users the background for understanding the tutorial chapter.

## 3.1 What is SAMOS?

SAMOS is an active object-oriented database management system whose development started in 1991 at the University of Zurich. SAMOS is now operational and thus is among the first demonstrable 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 prototype implementation of SAMOS uses ObjectStore 3.1 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™.

SAMOS is an acronym for Swiss Active Mechanism Based Object-Oriented Database Systems. In reality, SAMOS is a Greek island known from the philosopher and mathematician Pythagoras who lived there b.c. From that time there is the ruin of a pillar of an ancient temple of Hera. We selected this pillar as the sign of the project because, today after its renovation, the pillar consists of several layers that can be interpreted as the several layers of an active database system built based on a layered architecture.

### 3.1.1 ECA-Rules in SAMOS

In addition to the data definition language of ObjectStore, SAMOS provides a rule definition language as a means to specify 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>)
```

A rule definition specifies an event, a condition, an action and execution constraints (coupling modes). Rules have a unique name in the context of a database. The definition of events as a part of the rule definition is not always meaningful as events are often used in multiple rules. For reusability and convenience, it is often undesirable to describe the same event (especially a composite one) in each rule. Therefore, events may be named and defined separately outside the rule definition:

```
DEFINE EVENT event_name event_clause
```

Corresponding rules can thus refer to the event via its name:

```
DEFINE RULE rule_name
    ON event_name IF condition_clause DO action_clause...
```

Similarly, conditions and actions may be named and defined separately outside the rule definition. Rules can then refer to them via their names.

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.

The rule language of SAMOS is discussed in detail in [5] and in [3] chapters 3 and 4. However, SAMOS Release 1.0 does not support the whole functionality described in [3]. There are some special cases which are not included in the current release. In the following section we introduce the rule language of SAMOS Release 1.0. The syntax of the rule language is given in the Appendix. We recommend the user to read the description in [3, 5] as well.

#### 3.1.1.1 Events in SAMOS

In SAMOS we distinguish between *event descriptions* and *event occurrences*. 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.

The event specification part of the rule language of SAMOS is characterized by built-in support for:

- primitive events: An event can be specified by an occurrence within the database (*method* events), in the database management system (*transaction* events), or in the database environment (*time* and *abstract* events),
- composite events, e.g., “when the last of a set of events has occurred” or “when an event has not occurred within a predefined time interval” that can be constructed out of primitive or other composite events.
- the integration of time aspects into event definitions, e.g., “consider as relevant only those events that occur between 8:00 and 16:00” and
- parameterizing of events.

Following the guideline of “providing complex event specifications only if you are able to define their semantics” we have attempted to keep the language simple but nevertheless powerful. Our aim is to provide an easy-to-use language.

| *name* | *signature* | *composite event is signalled when* |
|---|---|---|
| `sequence` | `E1;E2` | `E1 and then E2 occur` |
| `conjunction` | `E1,E2` | `E1 and E2 occur regardless of order` |
| `disjunction` | `E1/E2` | `E1 or E2 occurs` |
| `reduction` | `*E` | `E1 occurs for the first time` |
| `reduction` | `TIMES(n,E)` | `E1 occurs n times` |
| `negation` | `NOT E IN I` | `E1 does not occur in a given time interval I` |

*Table 1: Event Constructors in SAMOS*

Time events are signalled as soon as the appropriate point in time is reached. For the specification of a time event, an absolute time specification, e.g., `ON 01.06.94`, or a repeated absolute time specification, e.g., `ON EVERY 5 DAY 20:00 WITHIN [05.01-05.31]`(which is signalled only every fifth day in May), or a relative time specification can be used. In the latter case, SAMOS provide*s relative time events (t+x)*, where *t* is an implicit point in time and *x* is a time unit called relative time (e.g., `1 min`).

In an object-oriented environment, users manipulate and access objects by sending messages to them. Each message sent to an object invokes the execution of a method. Thus, the execution of the method gives rise to a method event. More precisely, as an event has to be mapped at a point in time and the execution of the method will itself last for a period of time, we have to indicate whether the event is the point in time immediately before the object begins the execution of the method (specified as `BEFORE`) or the point in time immediately after the completion of it (specified as AFTER). In the release version of SAMOS, a method event is always related to a class, so it is signalled before or after the execution of the appropriate method on any object of this class.

Transaction events are defined by the start time (the begin), the termination time (corresponding to transaction commit) or the transaction abort. They are defined using the keywords BOT or EOT or ABORT.

Method, time and abstract events express specific semantics known by SAMOS so their occurrence can be detected by the system itself. However, users and applications may need to define other events according to their specific semantics as well (*abstract* events). They are defined and named by users using the following syntax:

```
DEFINE EVENT event_name
```

and can be used in rules like any other event.

Abstract events are not detected by SAMOS, but users/applications notify the system about their occurrence by issuing the explicit operation:

```
raiseEvent event_name
```

SAMOS supports a variety of composite event constructors (see Table 1). For more details, please see [3] chapter 4 and [6]. The event history in SAMOS consists of all occurrences of the defined event descriptions. 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. The signalling of composite events is based on the event history; only events included in the event history may be used for the signalling of composite events. In SAMOS it is, thus, 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 description (the last three in the table). Time intervals are defined through two points in time, a *start_time* and an *end_time*. *Start_time* and *end_time* can be defined as absolute, *e.g.,* `[96.01.15,08:00-96.12.15,18:00]`, or implicit points. In the example `[93.05.20,00:00-X]`, *start_time* is an absolute point in time and *end_time* is an implicit one defined according to a system activity `X` (in SAMOS this corresponds to the occurrence of an event). The first instance of the system activity `X` after the point in time `93.05.20,00:00` closes the interval. Future occurrences of the system activity `X` are ignored. In addition, *end_time* can be defined as a relative point in time in relation to the *start_point, e.g.*, `[X-(X+01:10)]`. Then, each time (say `ti`) the system activity `X` happens, a time interval begins and ends at the point in time `ti+01:10`. Furthermore, a monitoring interval can be specified as an absolute time interval that reappears periodically `EVERY WEEK [MO,18:00-FR,24:00]`.

For all event constructors, except negation, time intervals are optional. When specified, a time intervals defines when the component events must occur to let the composite event occur.

Furthermore, SAMOS supports predefined event parameters. The set of event parameters is fixed, i.e., users cannot define event parameters during the definition of an event. Time events have no parameters because they occur in the database system environment and thus have no connection to the data. In all other cases, we distinguish between *environment* parameters and parameters which depend of the event type. Environment parameters are:

- *occ_point*(`E`): the point in time of the occurrence of an event `E`,
- *occ_tid*(`E`): the identifier of the transaction during which event `E` has occurred (*occurring transaction),*
- *user_id*(`E`): the identifier of the user who has started the occurring transaction of event `E`.

Further parameters depend on the event type. Method events have the object identifier (*oid*) of the object executing the method as a parameter. Then, it is possible to refer to the value of an object through its object identifier. This means that the condition can refer to the actual state during the condition evaluation of the object. In ObjectStore, the object identifier corresponds to a pointer to the object. Transaction and abstract events have only environment parameters.

Before we discuss the parameters for composite events, we introduce the keyword *same.* The definition of composite events can be extended with the keyword *same* to denote that the component events have the same value of a particular parameter. We can define the *same* keyword in relation to the environment parameter *occ_tid* and *user_id*. For example, the sequence `(E1;E2):same transaction` is signalled if `E1` and after that `E2` have occurred within the same transaction (the occurring transaction of `E1`). In case of method events in any composition, if we require that the appropriate methods are executed on the same object, then we use the *same* keyword in relation with the oid, e.g., `TIMES(n,E):same object`.

Now back to the parameters of the composite events. In general they take as parameter only the *occ_point*. However, composite events defined with the keyword *same transaction*, *same user* or *same object* have as parameters the *occ_tid*, the *user_id* or *oid* respectively.

#### 3.1.1.2 The Condition, Action and Execution Constraints in SAMOS

Both conditions and actions must be programmed in the data manipulation language (DML) of ObjectStore (i.e., extended C++). A condition is an expression written in the query language of ObjectStore. An action can be any executable program. Furthermore, it may signal the occurrence of an abstract event by calling the function *raiseEvent*.

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. The first coupling mode of a rule specifies when the condition part of the rule is 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 two 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.

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. The release version of SAMOS does not support priorities and the execution order is arbitrary (system-determined).

### 3.1.2 The SAMOS Prototype

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.

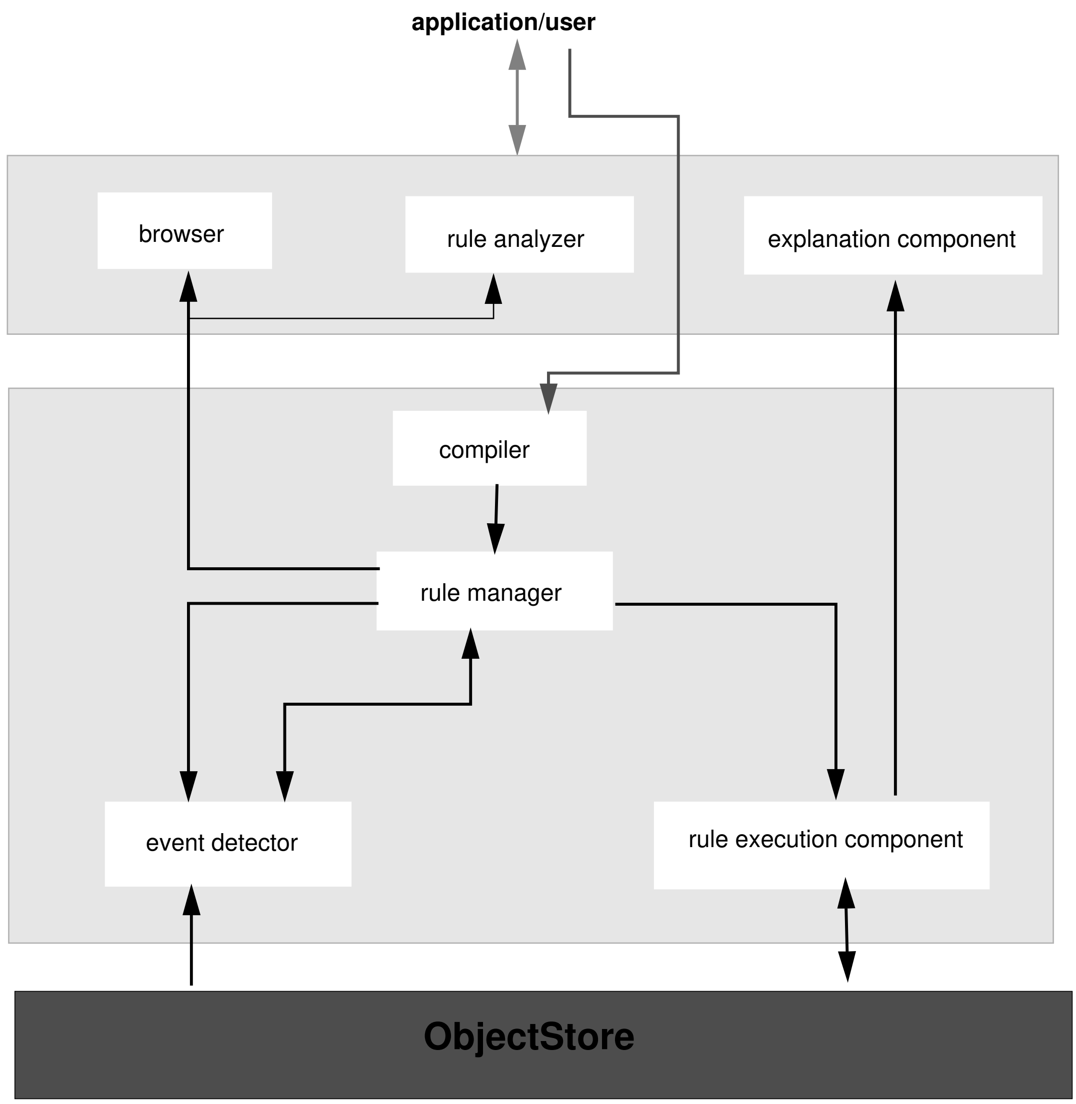



*Figure 1 The SAMOS Architecture*

The top-most SAMOS layer consists of tools, such as a rule analyzer, a rule browser and a rule explanation component. The intermediate layer consists of a number of runtime components (Figure 1):

- a rule compiler which takes ECA-rules defined in the syntax given above as input, checks their syntactic and semantic correctness, and updates the rule base appropriately.
- a rule manager responsible for managing the rulebase and thus for the retrieval of information about event and rule definitions,
- a detector for composite events, and
- a rule execution component responsible for condition evaluation and action execution.

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

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

#### 3.1.2.1 Mapping of Rule Definitions onto ObjectStore Data Structures

The information about rule and event definitions has to be a part of the rulebase in order to be persistent. 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. 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.

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

#### 3.1.2.2 SAMOS Initialization for Event Detection

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

In order to detect method events, the body of the appropriate method has to include a call of the operation *raise event* at the beginning in the case of BEFORE or at the end in the case of AFTER. This means that the program which contains the method body has to be modified and then recompiled. The nature of ObjectStore prevents the creation of a method wrapper, a new method which contains a call of the operation *raise event* and a call of the original method, which can be compiled and replace the old method at runtime. In SAMOS release 1.0 users themselves must modify the appropriate method as described in section 4.5.1 and start the compilation.

Time events are detected by monitoring the system clock. For this we use the UNIX system process *cron* which executes commands, in our case calls of the operation *raise event*, at specified dates and times. Commands have to be inserted into a *cron*- or into an *atfile*. The specification of a time in a *cron*- or in an *atfile* is similar to the specification of time events in SAMOS. Thus, it is reasonable to use the cron process for the detection of time events. The entry in the *cron*- or *atfile* is a time specification which corresponds to the time event description. The command is the invocation of the operation *raiseEvent <event_name>*. After *cron* signals the arrival of a specified point in time, the appropriate time event is signalled. Note that the command, the invocation of the *raiseEvent* operation, is executed in a specially created process.

The detection of transaction events is done by overwriting the transaction operations so that the operation *raise event* is invoked. In ObjectStore, each transaction is an instance of the class *os_transaction*. Operations on transactions are implemented as methods of this class (e.g., *os_transaction::begin* starts a new transaction). To implement this overwriting we defined a new class *samTransaction* (Figure 2). Each instance of *samTransaction* (we call it *SAMOS transaction*) contains a reference to an ObjectStore transaction. The variable *list_of_rules*, which implements the rule register, stores rules triggered in *deferred* mode. Furthermore, each element of this list contains a flag indicating whether the condition has already been evaluated. This class defines methods for the transaction operations *start*, *commit*, and *abort*. Within each such operation, the operation *raise event* is invoked for the signalling of transaction events and the corresponding operations of the class *os_transaction* are called.

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

*Figure 2 : Definition of class samTransaction*

Composite events are detected stepwise (or incremental).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 *raiseEvent* is called) as soon as the last element of the appropriate sequence order gets marked. Stepwise 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].

#### 3.1.2.3 Activities Upon Primitive Event Detection

The occurrence of a primitive event is signalled in that the message *raiseEvent* is sent to the rule manager. The rule manager then retrieves the ObjectStore object representing the signalled event. 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 which stores the composite event(s) in which the appropriate event participates. 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.

#### 3.1.2.4 Rule Execution

The rule execution component is responsible for rule execution and is called with a 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. More details are given in the tutorial in section 4.3.

### 3.1.3 The Tools

A set of tools supports the user during the development and the maintenance of active applications.

#### 3.1.3.1 The Rule Browser

The browser retrieves the rulebase by performing retrieval queries over the rulebase. It shows individual rules, events, actions and conditions, and allows the selection of items which meet various desired criteria. It contains four parts: a rule-, an event-, a condition- and an action browser. The structure of each browser is mainly the same. It contains a list with names of items (e.g., a list of rule names) from which the user can select a desired one. For each selected item its body or description is shown. For actions, conditions or events al- the list of rules they belog to is also visible. In addition, for a primitive event the event browser indicates the list of composite events it is participating in. Navigation through the rulebase is managed by the rule browser, which has links to the other browsers. The rule browser shows the list of all rules and for each selected rule the appropriate rule body. If an event, condition or action is selected from the rule body description, the appropriate browser is activated.

## 3.2 Using SAMOS for the Implementation of Active Applications

The aim of this section is to provide an overview of how SAMOS can be used for the implementation of active applications. Details can be found in the tutorial in chapter 4. Before we continue with the discussion of the components of a SAMOS application, let us consider again the way SAMOS is implemented. As mentioned in section 3.1.2, SAMOS is implemented on top of ObjectStore using the ObjectStore interface like any other ObjectStore application. The result is a set of ObjectStore application programs and a set of class definitions which implement the active parts of SAMOS. We call them here the active components of SAMOS. Thus, when we talk about SAMOS we mean ObjectStore plus these active components. The source of the active components build a library which must be linked with any SAMOS application.

### 3.2.1 Active vs. Passive Applications

Compared with the development of applications with a passive DBS (e.g., ObjectStore), users of an active DBS must deal with aspects concerning the active behavior. Additionally, SAMOS users must take into account the C++ environment of SAMOS.

Implementing an active application means the implementation of a set of

- classes using the facilities offered by ObjectStore (the passive part),
- ECA-rules using the rule definition of SAMOS (the active part), and
- application programs.

Thus, the development of a SAMOS application includes the following tasks:

- the rule definition: rules are defined either using the interactive interface of SAMOS or using the SAMOS compiler.
- class definition and implementation: based on the database schema of the application which determines the structure and the behavior of the application miniworld, a set of classes are defined.
- application programs: they deal with application data, e.g., they manipulate the application data, as in the case of a passive DBS, but also they include functionality which is specific for an active DBS such as signalling the occurrence of an event in case of

an abstract event. In the latter case, the application programs use some functions which are implemented as an extension of ObjectStore specifically for SAMOS. All these functions offered by SAMOS must be included in the corresponding application programs.

### 3.2.2 The Database

Let us now examine the information managed by SAMOS, i.e., the contents of the database. On the one hand, we talk about an application database which consists of the application data, the structure of which is described in the database schema. On the other hand, we talk about a rulebase which stores information about the defined rules and events and all other data (like information about event occurrences) required for SAMOS. Rule and event definitions are stored as objects in the rulebase. Whenever SAMOS needs information about a rule or an event, the appropriate object is accessed. The schema of the rule database is predefined in SAMOS whereas users must determine the schema of the application database.

Note that in this SAMOS release we assume that the application database and the rulebase are physically one ObjectStore database. This was necessary to have several possibilities for defining actions (e.g., in case of time events to operate on the application database)

### 3.2.3 Handling Conditions and Actions

Conditions and actions can be any executable program implemented in the DML of ObjectStore, an extension of C++. This code fragment must be executed each time the appropriate rule is triggered. Thus, when a condition or an action is defined, SAMOS compiles the appropriate source code and links it with the rest of the application object code. Therefore, two files exist for each SAMOS application: the condition and the action file which include the code fragments for all defined condition and actions.

Upon rule execution, the executable code fragments for conditions and actions must be found and executed. The solution of SAMOS is as follows (see Figure 3).

A function type including a fixed set of formal parameters is defined for conditions and actions, respectively. For each condition and action specified in a rule, a C++-function is generated whose signature conforms to that function type and whose body implements the condition (action). The source code for these C++-functions is stored in two files. References to the functions are stored in an array of function pointers. The index of a function pointer in this array is recorded in an attribute value of the corresponding rule object. In order to execute a function, its index has to be found through the corresponding rule object, the function pointer has to be retrieved from the function pointer array, and the pointer finally has to be de-referenced.

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

- extension and compilation of the condition and action files,

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

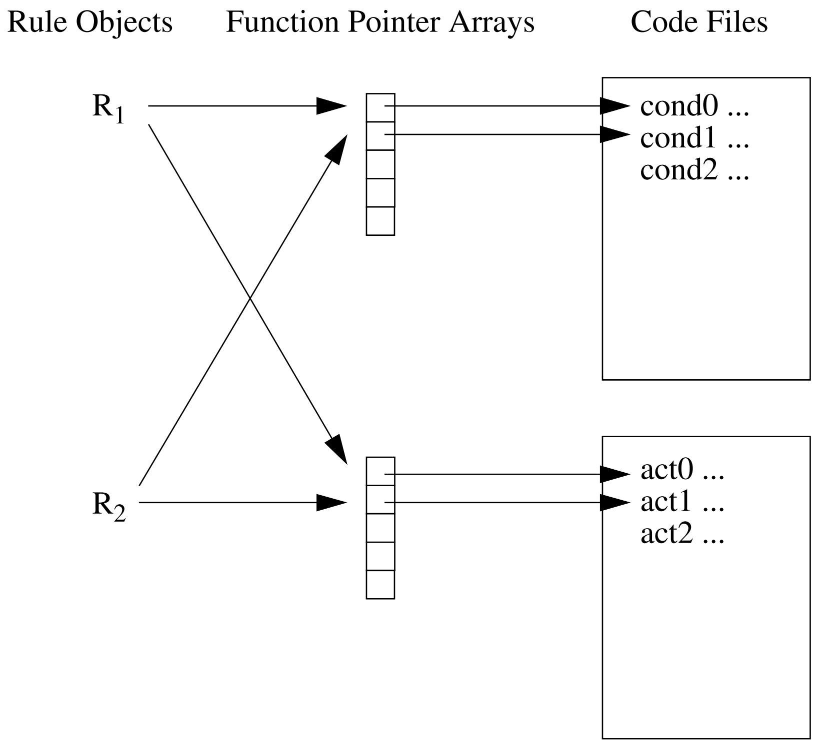



*Figure 3 Management of Code Fragments for Conditions and Actions [11]*

### 3.2.4 Rule Modification

Rulebase modification is necessary because the real-world behaviour modeled as ECA-rules changes. Furthermore, it may be necessary to modify rules during the debugging of the active application. Rulebase modification includes the deletion of defined rules, events, conditions or actions; the creation of new rules, events, conditions or actions; or the modification of existing definitions.

#### 3.2.4.1 The Create Operation

Whenever necessary the user can extend the rulebase by defining new rules, events, conditions and actions. Existing events, conditions and actions can be used in the new rule definitions. Furthermore, composite events can be defined on existing events. Note that the signalling of the new defined composite events takes place only because of event occurrences that happen after the new definitions. This means that relevant event occurrences in the past are not considered.

#### 3.2.4.2 The Delete Operation

A rule determines the connection between an event condition and action. Thus, the deletion of a rule means the deletion of this connection. If the event, condition or action of a deleted rule are used only in this rule, they are deleted as well. Otherwise, the deletion of the rule does not affect these definitions.

On the other hand, the deletion of an event means that all rules defined on this event must be deleted as well. If the deleted event is part of a composite event then the composite event must also be deleted. Event deletion is a relatively complex procedure because the following aspects must be taken into account:

- As a side effect, existing occurrences of the deleted event would also be forgotten. This means that the event history must be updated which requires that the appropriate tokens must be removed from the Petri net.
- Event deletion means not only the deletion of the appropriate event object, like in case of the rule deletion, but also the removal of the appropriate event detector. For example, in the case of a composite event, the part of the Petri net which models the deleted event must be removed as well.

SAMOS release 1.0 does not support the event deletion. A user can delete a rule but only the connection between the event, condition and action is deleted. Sophisticated event modification is discussed in [9].

Furthermore, we support the deletion of a condition or an action. Note that the deletion of a condition or an action affects the deletion of all rules defined on it.

#### 3.2.4.3 The Modify Operation

A rule is modified whenever its event, condition or action is modified. SAMOS does not explicitly supports the modify operation. If a modification is necessary, the user must delete the old rule and define the new ones.

# 4 Creating Applications for SAMOS: A Tutorial

The tutorial assumes that ObjectStore and SAMOS have already been installed at your site and that an ObjectStore server has been started. For installing ObjectStore please see the ObjectStore manuals and for installing SAMOS please see chapter 2. Our aim in this chapter is to sequentially introduce what must be done to implement an active application with SAMOS. As an example, we have implemented a financial application called FAME and all our explanations in this tutorial are based on FAME. In particular, the tutorial includes:

- instructions for starting and working with FAME (section 4.2)
- presentation of files necessary for an active application (section 4.3)
- instructions for developing your own active application (section 4.4)
- discussion of the differences between the implementation of an active application using SAMOS and the implementation of a passive application using ObjectStore you must taken into account (section 4.5)
- instructions for handling the application database which now includes data needed for SAMOS, as well as the application data (section 4.6)
- instructions for defining ECA-rules in SAMOS (section 4.7)
- instructions for compiling your active application (section 4.8)
- instructions for handling time events (section 4.10) and alerts (section 4.9) in SAMOS
- instructions for maintaining the application (Section 4.11)

Note that as a style convention for file names we have chosen to distinguish between files which remain the same in each active application and those which belong specifically to the FAME application. The former are written in italics (e.g., *include_file.hh*) and the latter ones are written in a memospace typewriter format (e.g., `accounts.hh`).

## 4.1 FAME: a Financial Application

Let us first give a short description of FAME. FAME is a financial application. A bank with information about its customers and their accounts is modeled. A customer can have five different types of bank accounts:

- A *salary* account, where the customers make all of their deposits and withdrawals.
- A *savings* account, where the customer gets a higher interest rate.
- An *income* account, which is an investment account with a low risk profile and low profitability.
- A *medium* account, which is also an investment account with a medium risk profile and moderate profitability.
- A *growth* account, which is the third investment account with a higher risk profile but also higher profits.

The three investment accounts are managed by the bank as a combination of two different funds. The income account consists of 70% bonds and 30% stocks, the medium account consists of 50% bonds and 50% stocks, and finally the growth account consists of 30% bonds and 70% stocks. Since these are funds, it is irrelevant for customers to know exactly what kind of stocks and bonds they have. It is the responsibility of the bank to make sure that the profits will be as high as possible.

Customers can perform deposits and withdrawals into their salary accounts. They also define a specific limit for the salary account. Whenever their salary account exceeds this limit, the difference should automatically be transferred to their savings account. Customers also decide how they want to have their savings split up. For example, customers can specify that they want 50% of all their savings in the savings account, 25% in the income account, 10% in the medium account and 15% in the growth account. This “investment policy” is totally up to the customer. They decide how it is split up; sometimes the bank gives them advice according to the actual market situation, but customers do not have to follow it.

The management of the accounts is based on a set of business rules. In FAME, the following rules were identified:

1. If after a withdrawal from the salary account the balance drops below zero, then the salary account balance must be updated so that it is at its limit.

2. If after a deposit into the salary account the balance is above the limit, then the difference “balance-limit” must be deposited into the savings account.

3. From time to time is checked whether the ratio setting of the customer investment policy still holds. For example, this can be checked after three consecutive deposits into the savings account occurred.

4. When customer change their investment policy, a recalculation of the different balances in the four saving accounts is necessary.

5. If there were no salary transaction within a month then the bank should be alerted.

6. Customers should be alerted if they are changing their investment policy more than twice within one month.

7. The bank should be notified about active and knowledgeable customers like customers who perform three deposits in their income accounts after they have changed their investment policy.

8. If the interest rate for the growth account exceed 15% then the customer should be advised to invest more in the growth account.

9. Customers should be informed when they have more than 65% of their investments in the two most riskiest accounts with a very high interest rate.

10. A daily report with customer information must be generated every day at 17:35.

## 4.2 Starting FAME

All files needed for FAME are included in the directory

**$SAMOSROOT/tutorial**

You can copy this directory into your home directory. This directory includes a number of general files which are necessary for each SAMOS application and must always be included in the application directory, as well a number of files implementing the functionality of FAME. The meaning of each file is discussed in section 4.3.
If you want to experiment with it, you need to create a database. But before, you should the compile the program to create the application database:

**make createAccountsDB**

You can now create the application database by calling the program for creating an application database with the desired database name as an argument:

**createAccountsDB $OSUSER/AccountsRuleDB -i**

Then, the database **$OSUSER/AccountsRuleDB** is created with some sample data for two customers in FAME. Application data and rules are stored in the same application database. Details about handling the application database are given in section 4.6. In order to compile the FAME rules given in the file `accounts_rule_def` you must call the SAMOS compiler as follows:

**$SAMOSROOT/bin/ECA accounts_rule_def $OSUSER/AccountsRuleDB**

The SAMOS compiler *ECA* takes as arguments the name of the file containing the rule definitions and the name of the application database used for the storage of rule and event definitions.

The next step is the compilation of the application by calling

**make all**

Now you can start to work with FAME by calling

**accounts $OSUSER/AccountsRuleDB**

Using the menu interface you can execute predefined operations like a deposit or a withdrawal on a salary account. These operations lead to the signalling of events and to the execution of one or more rules. All steps done by SAMOS after the signalling of an event are reported in the file *samosOutfile* (for details see section 4.11.2).

Furthermore, each time the database status is changed, maybe because of the execution of a rule with action including database operations, the actual database status is shown in a file called *displayFile*. With the UNIX command **tail -f filename** you can keep these files opened during your session.

## 4.3 Files in FAME

In this section our aim is to discuss each file of the example application FAME to give you an in-depth view of an active application implemented with SAMOS.

### 4.3.1 Files Necessary for all Active Applications

The first group of files in FAME are necessary for each SAMOS application. They are files containing information about the defined conditions and actions:

- *condArrayDecFile.hh* — External declarations of the functions containing the condition body.
- *conditionArrayFile.cc* — Includes an array of function pointers to the conditions (corresponds to the function pointer arrays shown in Figure 3 in section 3.2.3.)
- *conditionFile.cc* — The code of all defined conditions is listed here (corresponds to the code files shown in Figure 3 in section 3.2.3.).
- *actArrayDecFile.hh* — External declarations of the functions containing the action body.
- *actionArrayFile.cc* — Includes an array of function pointers to the actions.

- *actionFile.cc* The code of all defined actions is listed here.

Recall that conditions and actions in SAMOS are programs in the DML of ObjectStore, a persistent C++. Note that each time the user defines a condition or an action, SAMOS extends the *conditionFile.cc* and the *actionFile.cc* with the appropriate program code automatically. Furthermore, these files must be compiled and linked together with the rest of the application object code. Details are given in section 4.8.

In addition, there are three further files which are necessary for the rule execution and event detection:

- *AppGlobal.hh* This header file (shown in Figure 4) contains the declaration of all global variables of the application programs, that SAMOS must know during rule execution.
- *include_file.hh* All include files necessary for the application are listed here. They are all the SAMOS header files necessary for rule execution, such as the header file of SAMOS' own transaction class. These definitions are predefined and must not be deleted. On the other hand, the user can add further necessary include files here. An example is the class definition file of the application. To be able to operate inside the conditions and actions on objects the class definition file must be included here (in FAME the file `accounts.hh`).
- *deal_time_events.cc* This program contains functions necessary for the detection of time events.

```
#ifndef APPGLOBALS_HH
#define APPGLOBALS_HH
/**************************************************************************
                    *DON'T DELETE THIS FILE!!!!!!!!!!*
**************************************************************************/


//////////////////////////////////////////////////////////////////////////////
// Don't delete the following lines!! They are vital for executing actions!!
#include "SamosGlobals.h"
/////////////////////////////////////////////////////////////////////////////;/

/**************************************************************************
*                   Declare your global variables below*
**************************************************************************/

#endif
```

*Figure 4 : The file AppGlobals.hh*

### 4.3.2 Files Particular for FAME

Implementing FAME on SAMOS means the implementation of the passive part (a set of classes that form the schema definition), of the active part (a set of rules), and of some applications programs. A sample schema of FAME is given in Figure 5. For each class the attributes are shown. For the class customer the defined methods are shown as well.

The following files implement the passive part of the application:

- `accounts.hh` Headerfile, with class definitions. Figure 5 illustrates a part of the schema definition. For the class `customer` some representative methods are also given. A part of `accounts.hh` is shown in Figure 6.
- `accounts.cc` Class implementation file.

The active part of FAME includes the rule and event definition file used by the compiler see section 4.7.1 `accounts_rule_def`.

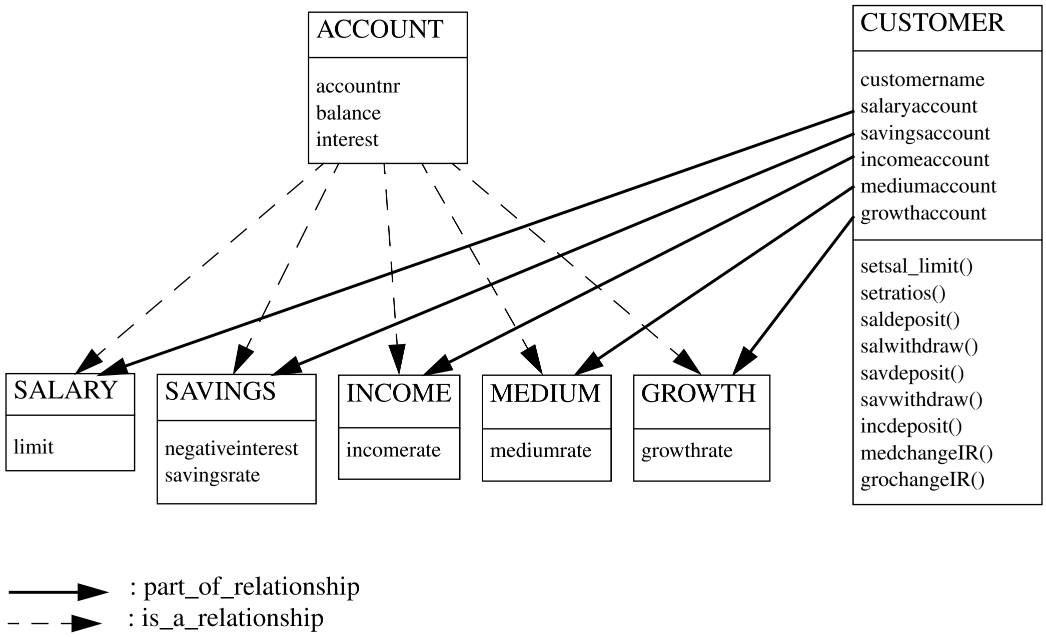



*Figure 5 Sample schema of FAME*

Finally, FAME includes the following application programs:

- `accountsMain.cc` Main program of FAME which implements the user interface to FAME. We provide here a very simple user interface. Customers can perform deposits and withdrawals into their salary accounts and change their investment policy. The bank can change the interest ratio for medium and growth accounts.
- `createAccountsDB.cc` Program to create the application database used for the storage of the application data and the rule and event definitions. An option determines whether the database should be filled with some sample application data or not. Note that if the database you want to create already

exists (i.e., it has the same name as an already existing one), then the old one is removed and a new one is created. This means that all of your old rule definitions and the event history get lost.

## 4.4 Developing Your Own Application

### 4.4.1 Starting from Scratch

The development of an active application using SAMOS includes the following steps:

1. preparing your environment
2. implementing the passive part, the application programs and defining the rules
3. compiling the application

**Step 1:**

First of all you must prepare the environment for your concrete application. You create a directory which will include all files implementing your application. In the following we refer to it as the *application directory*. Note that we assume that all application files are in one directory.

As we have seen in the implementation of our concrete active application FAME, there are a number of files which are necessary for each SAMOS application. They must always be in the application directory. The copying of those files can be done using the shell script *prepare_appl*:

**$SAMOSROOT/etc/prepare_appl <application directory>**

It will copy the files necessary for rule execution into the specified directory. In particular, the three files necessary for the condition evaluation (*condArrayDecFile.hh*, *conditionArrayFile.cc*, *conditionFile.cc)* and the three files necessary for the action execution (*actArrayDecFile.hh, actionArrayFile.hh*, *actionFile.cc*) will be copied. Each time the user defines a condition or an action in the DML of ObjectStore, SAMOS extends these files with the appropriate program code automatically. Furthermore, the following three files are copied: *AppGlobal.hh, include_file.hh, deal_time_events.cc*. Explanations of these files can be found in Section 4.3.1.

**Step 2:**

A passive application consists of a set of classes defining the schema and application programs. An active application includes additionally a set of ECA-rules implementing application semantics which otherwise would be hidden in the class implementation as part of the methods or in the application programs. Therefore, before starting the implementation of your active application, you should decide which parts of the application must be modelled as ECA-rules.

You can now start with the definition and implementation of your classes. In the next step you should implement and compile the application program which creates the application database (see section 4.6). Then, you should create the application database.

After that, you can go on with the rule definitions (see section 4.7). Remember that in the case of method events you must modify the method body as well (explanations are given in section 4.5.1).

Then, you can implement some application programs.

**Step 3:**

Finally, you must compile the application.

### 4.4.2 Modifying Your Application

During the lifetime of your application it may happen that modifications are necessary. Modifications may concern rule definitions, class definitions, class implementations and the application programs. Modification of the class implementations and of the application programms requires, like in a passive application, a recompilation.

Regarding rule modifications, SAMOS Release 1.0, as discussed in section 3.2.4, supports the deletion of conditions, actions and rules by deleting only the connection between an event, condition and action. This can be done either using the compiler (see the syntax in the appendix) or using the samosshell.

In the case of class definition modifications which modify the schema, you should check whether they influence the rule definitions. Furthermore, if an application database exists before the schema modification and you want to delete it later, you must delete it manually using, for example the UNIX command `rm <database_name>`.

## 4.5 Characteristics of an Active Application

To implement your own active application you must understand the differences between the implementation of a passive application using ObjectStore and the implementation of an active application using SAMOS. We discuss the differences based on our example application FAME. Thus, this section also gives you explanations about the implementation of FAME. Obviously, we discuss only the passive part of FAME and the application programs.

### 4.5.1 The Passive Part

The use of SAMOS also influences the implementation of the passive part of the application, i.e, in case of FAME the files `accounts.hh`, `accounts.cc`:

- The file *samTrans.hh* must be included in the headerfile for class definitions (for FAME in `accounts.hh`) (see Figure 6)
- The files *eventclass.hh* and *samTrans.hh* must be included in the class implementation file (for FAME in `accounts.cc`)

```
#ifndef ACCOUNTS_HH
#define ACCOUNTS_HH

#include "samTrans.hh"

class CUSTOMER;

class CUSTOMER
{       public:             persistent<db> os_Set<CUSTOMER*> *extent;
                            SALARY* salaryacc inverse_member salowner;
                            ...
                            CUSTOMER(char* name);
                            ~CUSTOMER();
                            void salwithdraw(long bal);
                            ...
                             char*custName;
};
```

*Figure 6 : A part of the file accounts.hh*

- The method body of those methods on which method events are defined must contain the function call *raiseEvent* (see section 3.1.2.2). In the case of BEFORE method events, this call must be inserted at the beginning of the method body, while in case of AFTER method events, at the end of the method body. The call of the function has the following form:

```
raiseEvent("<method_name>", current_samTrans, this)
```

For example, in FAME we defined the method event `SalWithdraw` as `AFTER.customer.salwithdraw`. The appropriate method must be as follows:

```
void CUSTOMER::salwithdraw(long bal)
{
     salaryacc->balance -= bal;
     raiseEvent("SalWithdraw", current_samTrans, this);
}
```

Furthermore, the function *raiseEvent* must be declared. This means the following line must be inserted into the class implementation file. For example in FAME the `accounts.cc` file includes the declaration:

```
extern void raiseEvent
         (char*, samTransaction*, os_reference_protected);
```

### 4.5.2 The Application Programs

The SAMOS user who implements an application program must be aware of the following points:

- The files *AppGlobals.hh* and *RuleManager.hh* must be included in any application program.
- The class *RuleManager* must be declared in each application program as follows:

```
CRuleManager RuleManager;
```

- An application program which signals abstract events must contain a declaration of the function *raiseEvent*, as discussed above.
- Opening or creating of the application database must be done using the predefined methods *ColdStart* and *HotStart*. For details see section 4.6.
- The pointer to the database must be called *db*.
- In order to be able to signal transaction events and to support the deferred execution of rules, users must use SAMOS' own transaction operations for starting, committing or aborting of transactions. For example, a transaction must be started and committed as follows:

```
samTransaction* stx = samTransaction::begin();
samTransaction::commit(stx);
```

Recall that we defined a new class *samTransaction*. Each instance of *samTransaction* contains a reference to an ObjectStore transaction and further information required for the rule execution. This class defines methods for the transaction operations *start*, *commit*, and *abort* which must be used by the SAMOS user.

## 4.6 Handling the Application Database

Each active application has only *one* database which includes the application data, the rule and event definitions, and further information, such as the event history or the Petri net . Two methods, *ColdStart* and *HotStart*, of the class *RuleManager* have been implemented to support the SAMOS user for the management of the application database. They should be called as follows:

**RuleManager.ColdStart(<db_name>, <replace_option>)**
**RuleManager.HotStart(<db_name>)**

Both *ColdStart* and *HotStart* initialize ObjectStore, *os-collections* and some persistent structures for the rulebase. Additionally, *HotStart* opens a database which must already be created. Depending on the value of the *replace_option*, *ColdStart* can be used for creating an application database or for opening an existing database. If you call *ColdStart* with a new database name then a new database is created and the environment for the detection of time events is prepared. However, if you use an existing database name then you can determine with the value of the *replace_option* whether the existing database is deleted and a new one is created (*replace_option*=1) or whether the exsiting database is only opened (*replace_option*=0). It is important to know that in the first case all rule, event, condition and, action definitions, all SAMOS specific entries in the *crontab* and *atfile*, and the event history are removed. In the second case only the entries in the *atfile* are removed. This is also the difference between a *HotStart* and a *ColdStart* with *replace_option* 0.

The application program responsible for the creation of the application database (in FAME the file `createAccountsDB.cc`) should include a call of the *ColdStart* method with 1 as the value of the *replace_option*. On the other hand, in applications programs where only the database must be opened you should use the *HotStart* method (e.g. in FAME the application program `accountsMain.cc`).

## 4.7 Defining Rules in SAMOS

Recall that before you begin with the definition of rules and events you should first create the application database. There are two different approaches for the definition of rules in SAMOS: interactively and by file processing. We discuss both of them in the following sections. The definition of new rules may take place at any point in time during the lifetime of your application. Note that the definition of new rules may require a recompilation of your appplication. For more information, see section 4.8.

### 4.7.1 Defining Rules Using the Compiler

In order to help speed up the process of defining events, conditions, actions and rules, we have implemented a compiler which is able to process a subset of the SAMOS rule language (see section 3.1.1). The syntax of the rule language of SAMOS on which the compiler is based is given in the appendix. This is a part of the rule language presented in [3] since some features of SAMOS are not yet implemented in the prototype.

Using the SAMOS compiler the idea is that all rule and event definitions are written in a file, such as *accounts_rule_def* for FAME. The next step is to call the compiler. The compiler checks all rule and event definitions syntactically and semantically and stores the information about the new definitions in the previously created application database. The command would look like this:

**$SAMOSROOT/bin/ECA accounts_rule_def $OSUSER/AccountsRuleDB**

To be able to define rules yourself, please study the syntax of the rule language of SAMOS. Note that in general it depends on you if you define events, conditions and actions as part of a rule (then they are relevant only to this rule) or outside a rule. Both cases are possible using the compiler .

As part of the definition of a method event the names of the files in which the definition and the implementation of the appropriate method are defined must be also given.

There is one point concerning the definition of a condition which we have to discuss. Let us consider the condition `AdjustAccounts` used in the rule `AdjustAccounts`. This rule checks if the ratio setting of the customers investment policy still holds. It has to be checked when there have been more than three deposits into the savings account. The condition is defined in the following way:

```
DEFINE CONDITION AdjustAccounts
  @ CUSTOMER *oid=(CUSTOMER*)parameters->objectPointer;
     long sum = oid->savingsacc->balance + oid->incomeacc->balance
     + oid->mediumacc->balance + oid->growthacc->balance;
     long s_am = sum * (oid->savingsacc->savingsrate * 0.01)
     - oid->savingsacc->balance;
     long i_am = sum * (oid->incomeacc->incomerate * 0.01)
     - oid->incomeacc->balance;
     long b_am = sum * (oid->mediumacc->mediumrate * 0.01)
     - oid->mediumacc->balance;
     long g_am = sum * (oid->growthacc->growthrate * 0.01)
     - oid->growthacc->balance;
     if (!(((s_am == 0) && (i_am == 0)) && (b_am == 0)))
          return 1;
     else
          return 0;@
```

First, notice the @ signs. They embrace the body of the condition or an action. Of importance is also the line `CUSTOMER *oid=(CUSTOMER*)parameters->objectPointer;`. Whenever you want to access data of some of the class members, you will need to cast a pointer of the object id to the respective class.

### 4.7.2 Defining Rules Using the samosshell

Interactive definition of rules is provided by the samosshell. The samosshell also includes a very simple browsing function as discussed in section 4.11.3. If we compare the compiler with the samosshell regarding the definition of rules, we see that the advantage of the samosshell is that you need not know the exact syntax of the rule language because you are guided through the whole definition process by the samosshell. On the other hand, if you are familiar with the rule language of SAMOS, you should use the compiler because you can define your whole set of rules at once.

There will be one rule, the *BelowLimit* Rule, explained using the samosshell as a means to define active behavior for an application. This rule checks whether after a withdraw in the salary account, the balance drops below zero. It should then update the salary balance so that it is at its limit.

Using the syntax of the rule language of SAMOS, the following definitions must be made:

```
DEFINE EVENT SalWithdraw AFTER.customer.salwithdraw

DEFINE CONDITION BelowLimit
     @ CUSTOMER *oid=(CUSTOMER*)parameters->objectPointer;
     if (oid->salaryacc->balance <= 0)
          return 1;
     else
          return 0;@

DEFINE ACTION BelowLimit
     @ CUSTOMER *oid=(CUSTOMER*)parameters->objectPointer;
     long am = oid->salaryacc->limit - oid->salaryacc->balance;
     oid->savwithdraw(am, action_tx);
     oid->saldeposit(am, action_tx);@
```

```
DEFINE RULE LowLimit
     ON SalWithdraw
     IF Below_Limit
     DO Below_Limit
     COUPLING(immediate, deferred)
```

It is not possible to define a condition or an action as part of a rule definition using the samosshell. They must always be named and defined outside rule definitions. This is a restriction of the shell compared to the compiler.

You start the samosshell by typing:

**$SAMOSROOT/bin/samosshell $OSUSER/AccountsRuleDB**

Then a menu appears. First you will choose *Define event* by typing

**10**

You are presented with another menu. Here you choose to define a method event.

Next you will be prompted to type in the name of the event:

**SalWithdraw**

The event body will be typed as follows:

**AFTER.CUSTOMER.salwithdraw**

When asked where the definition and the implementation of the class is at, you type:

**accounts.hh** *and* **accounts.cc**

This completes the definition of the event.

Now you will have to define the condition *BelowLimit*. For this just choose 20 from the main menu, and then give the condition name and condition body. The condition body must be typed as follows:

```
if (oid->salaryacc->balance <= 0)
        return 1;
else
        return 0;@
```

You must complete the definition of an action or a condition with a @. In contrary with the compiler you must not begin the definition with a @.

In the same way you can define the action by choosing 30 from the main menu. Now the action name is BelowLimit and the action body is:

```
long am = oid->salaryacc->limit - oid->salaryacc->balance;
oid->savwithdraw(am, action_tx);
oid->saldeposit(am, action_tx);@
```

Finally, you can define the rule `BelowLimit` by choosing 40 from the main menu. This concludes the definition of the rule `BelowLimit`. You define all the other rules in the same way: first comes the definition of the event, then the condition, then the action and the definition of the rule at the end.

## 4.8 Compiling the Application

When you have completed the programing of the passive part as well as have defined all or some of the rules, you can go on with compiling your active application. We want to take a closer look of how the makefile has to be set up.

First of all, you need to create a makefile for the application which looks like the makefile of FAME shown in Figure 7. To emphasize elements of the makefile which are relevant only for FAME we write them in bold.

The *$(OSUSER)/portfolio.comp_schema* and *$(OSUSER)/portfolio.app_schema* provide names for databases that will be created automatically during compilation and that will contain information about the classes that your application uses. Aside from specifying these pathnames, you don‘t need to worry about these databases. If you want to know more about this, see ObjectStore Administration and Development Tools [ObjDes93].

You need to specify the include directive of the SAMOS makefile. This is done with the line:

*include $(SAMOSROOT)/etc/samos.mk*

You also need to mention, where the SAMOS library is at. For this look at the line:

*LDLIBS = -loscol -los $(SAMOSROOT)/lib/samoslib.a*

The parts SOURCES, OBJECTS and EXECUTABLES are self-explanatory. Furthermore, there are sections that specify the compiling and linking of the different applications programs, (in FAME `accounts` and `createAccountsDB)`. Note that the program *deal_time_event.cc* which is provided by SAMOS must be also compiled and linked together with the application program (in FAME together with `accounts`)

You need a recompilation of your application program (here called `accounts`) each time a rule with a new condition or new action is defined. For this just call

**make all**

If you define a new rule with a new event and with an already existing condition or action (which has been defined outside a rule and has been named) then you need no recompilation. Actually, the recompilation concerns the compilation of the modified files *conditionFile.cc* and *actionFile.cc* which include the condition and action bodies

Note that errors in the condition and action body are recognized by the C++ compiler and not by the SAMOS compiler. Therefore, if you have errors in the files *conditionFile.cc* and *actionFile.cc* after the compilation of the application, means that your definitions are wrong. In this case, you should modify your definitions, compile them (using the SAMOS compiler) and compile the whole application (using `make accounts`).

```
include $(OS_ROOTDIR)/etc/ostore.mk
OS_COMPILATION_SCHEMA_DB_PATH= $(OSUSER)/portfolio.comp_schema
_OS_APPLICATION_SCHEMA_DB_PATH= $(OSUSER)/portfolio.app_schema

include $(SAMOSROOT)/etc/samos.mk

CPPFLAGS = -gx -I$(SAMOSINCL)
LDFLAGS = -g
LDLIBS = -loscol -los  $(SAMOSROOT)/lib/samoslib.a

SOURCES = conditionFile.cc actionFile.cc accounts.cc
accountsMain.cc createAccountsDB.cc  deal_time_event.cc

OBJECTS =${SOURCES:.cc=.o}# name object files like the source files  but with .o extension

EXECUTABLES = deal_time_event accounts createAccountsDB
all: ${EXECUTABLES}

#-----------------------------------------------------------------------
# Program to deal with time events
#-----------------------------------------------------------------------
deal_time_event: .depend $(SAMOSROOT)/lib/samoslib.a \
                          deal_time_event.o accounts.o $(SAMOS_LOCAL_RULEXE_OBJ)
        ${LINK.cc} -o  deal_time_event \
                          $(SAMOS_LOCAL_RULEXE_OBJ) deal_time_event.o accounts.o ${LDLIBS}

#---------------------------------------------------------------------------
# Accounts Application program
#---------------------------------------------------------------------------
accounts: .depend $(SAMOSROOT)/lib/samoslib.a \
                          $(SAMOS_LOCAL_RULEXE_OBJ) accounts.o accountsMain.o
        ${LINK.cc} -o  accounts \
                          $(SAMOS_LOCAL_RULEXE_OBJ) accounts.o accountsMain.o ${LDLIBS}

#----------------------------------------------------------------------------
# program which creates a DB for the Accounts application
#----------------------------------------------------------------------------
createAccountsDB: .depend createAccountsDB.o accounts.o  $(SAMOS_LOCAL_RULEXE_OBJ)
                       ${LINK.cc} -o createAccountsDB  \
                       $(SAMOS_LOCAL_RULEXE_OBJ) createAccountsDB.o accounts.o${LDLIBS}


#####################################################
# remove all executables, all object files and the compilation schema
#####################################################
clean:
        rm -f ${EXECUTABLES} ${OBJECTS}
        osrm -f ${OS_COMPILATION_SCHEMA_DB_PATH}

.depend: $(AUTOSOURCES)
        osmakedep .depend $(CPPFLAGS) -I$(SAMOSINCL) -files $(SOURCES)

depend:
        osmakedep .depend $(CPPFLAGS) -I$(SAMOSINCL) -files $(SOURCES)

include .depend
```

*Figure 7 : The Makefile*

You should not forget to start a compilation of your application program each time a method event is defined and you have modify the appropriate method body due to the initialization of the event detection.

## 4.9 Alerts in SAMOS

Rules with actions including only user notification are called *alerts*. There are several possibilities how the user can be notified. The message can be written in a file or a dialog box displaying this message can be shown or a mail can be send to the user. In SAMOS we support the first case. Thus, the user is not directly informed but the notification is written in a file called *alertFile*. The alert file is created when the first alerter of the application is executed. Once the *alertFile* is created all notifications are added in it sequentially. The user can any time delete the *alertFile* and when the next alerter is executed the file is created again. This means that the user is responsible for the management of the *alertFile* and he decides when old notifications must be deleted.

For defining alerts there is a predefined function `alerter(alerttext)` which can be used as follows:

```
char* alerttext = new char[800];
strcpy(alerttext, "\n message \n");
alerter(alerttext);
delete alerttext;
```

## 4.10 Handling Time Events in SAMOS

As mentioned in chapter 3.1.2, time events are detected using the UNIX cron process. Here we give some technical details about the using of the cron process in SAMOS. Each time a time event is defined, SAMOS inserts the appropriate entry in the *atfile*, in the case of absolute time events, or in the *cronfile*, in the case of periodic events.

The *cronfile* is user and machine dependent; this means one user may have a different *cronfile* on each machine. For SAMOS, all entries for time events are inserted in the *cronfile* of the machine on which the ObjectStore server runs. The file *cronfile* with all entries about periodic time events, is included in the application directory. Each time after listing the *cronfile*, the user can ebe informed about the existing definitions. Note that the user may have own definitions in the *cronfile* independent to SAMOS.

During the execution of the *ColdStart* operation (as mentioned in section 4.6), the *cronfile* is extended if it exists already for this user on the concrete machine or otherwise, a new *cronfile* is created. A time entry is then inserted for checking every hour at five to hour whether a periodical event must be signalled in the next time unit. This is necessary for cases of periodic events which can not be directly converted into the *cronfile* format (like EVERY 5 HOUR). For further information about event detection, please see [3] in chapter 9. For the SAMOS-user, however, it is important to know that the program *deal_time_events.cc*, which handles some of these special cases, must be linked together with the concrete application. Therefore, it must be included in the application directory.

## 4.11 Maintaining the Application

### 4.11.1 The Browser

Using the browser it is possible to look through the defined rules. The argument `AccountsRuleDB` in the example below refers to the database where the rules are stored:

**$SAMOSROOT/bin/SamosEt $OSUSER/AccountsRuleDB**

The browser consists of four parts: a rule-, an event-, a condition- and an action browser. Each browser can be started by double clicking on its button (See Fig. 8). The rule browser allows access to the other three and viceversa: each of the event-, condition- or action browsers provides for a return to the rule browser.

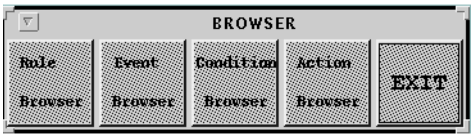



*Figure 8 : Starting the browser*

The representation of the four browsers is mainly the same: the list of the items is shown in the upper part and the description of a selected item in the lower part of the window. Initially, only the list of items in the upper part is visible.

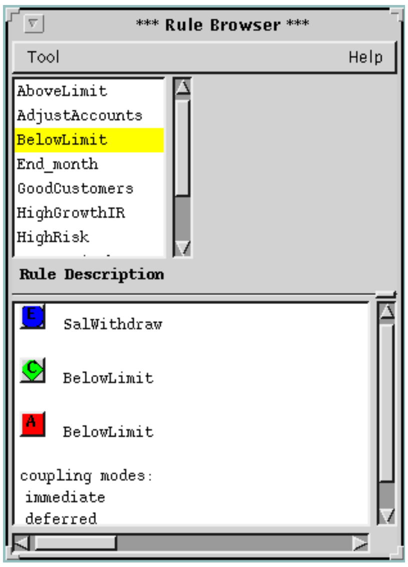



*Figure 9 : Rule browser*

We illustrate the functionality of the tool using the rule *BelowLimit* mentioned before. The rule browser displays the list of all rules. Selecting a rule (in our case *BelowLimit*) causes its description to be shown (See Fig. 9). The description contains the names of the appropriate event, condition, action and the types of coupling modes. By clicking on the buttons on the left, the user can access the other browsers to gain more information about individual elements of the rule. For example, if the button for the event *SalWithdraw* is selected, the control flows to the event browser, and the information available for *SalWithdraw* is shown (See Fig. 10).

The event browser displays the list of composite events the event is participating in (the panel called “participates in”) for a selected item and the list of rules where this event appears (in the panel “rules”). Obviously, by clicking one of the events belonging to the list of composite events, the event description in the bottom part of the window will be updated accordingly, showing the selected item.

If a rule is selected from the list of rules on the right side, the control flows to the rule browser and the description of the appropriate rule is displayed.

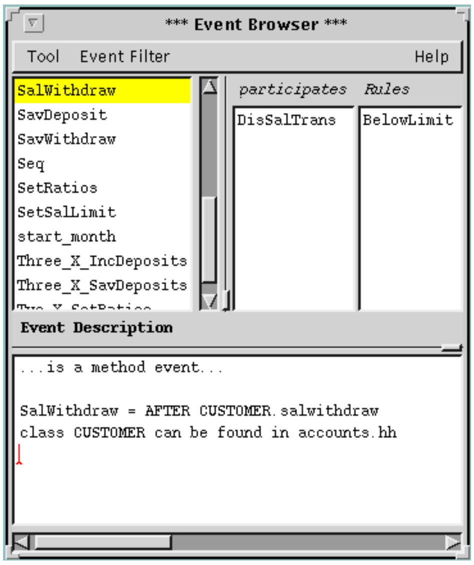



*Figure 10 : Event browser*

The user also has the possibility to filter (Menu *Event Filter*) the list of all displayed events and only view a subset of them, for example, only the composite events or only the time events. The event description contains, depending on the event type, information of interest to the user. For a method event, it will contain the name of the called method, which determines the occurrence of the event, the file name where the method is defined, and the specification BEFORE or AFTER, depending on when the event is signalled: before or after the method was executed. For a time event, the only information available in its description is the point in time (absolute, implicit or relative) when the event occurs. Abstract events have no description because they are characterized only by their names. For a composite event, its definition and the names and descriptions of its components are recursively shown.

The condition browser shows the list of all conditions , the list of rules the selected condition belongs to, and also the body of any selected item, i.e., the corresponding source code in the underlying data manipulation language. The same holds true with respect to the action browser (Figure 11).

By selecting one of the rules of the list to the right (in our case the only existing rule is

(*BelowLimit*), the control returns to the rule browser which manages the entire navigation through the rulebase.


*Figure 11 : Condition and action browser*

## 4.11.2 The Explanation Component: The samosOutfile

As mentioned in section 3.1.3, the explanation component is responsible for providing:

- all information about the current status of the application running, e.g., rule(s) executed, event(s) signalled, new database states
- information about activities in the future, e.g., composite events where a part of their component events already occurred
- information about activities in the past, e.g., the event history.

In this SAMOS release we provide a simple explanation component. Information about the control flow of SAMOS is stored in a file called *samosOutfile*. The management of *samosOutfile* is the same as that of *alertFile*. The file is generated after initial use of the application. All further activities are subsequentelly reported in it. Since the file can grow very large, we recommend replacing it with an emprty file frequently.

The *samosOutfile* reports all tasks done by SAMOS after a primitive event is signalled, i.e., it reports which rules are retrieved and executed or which further events are signalled. To be able to see which activities have been performed after one user operation (e.g., a salary withdraw in FAME)

In FAME there is another file, the *displayFile*, which shows the actual database status. Each time the database status is changed (maybe because of the execution of a rule with action including database operations), the actual database status is shown in the *displayFile*.

### 4.11.3 The samosshell

Samosshell is an interactively user interface which provides:

- the definition of rules, events, conditions and actions as discussed in section 4.7.2
- the deletion of rules as discussed in section 4.8
- the navigation through rule, event, condition and action definitions.

The first two services are also supported by the compiler and the third service is supported in a more elegant way from the browser, see section 4.11.1.

In addition, samosshell shows information about the Petri net. A Petri net consists of a set of places, transitions and connections between them. You have the possibility to diplay the adjacency matrix of the Petri net which shows the connections between places and transitions. Rows of the adjacency matrix are places and columns are transitions. If a connection between a place `i` and a collumn `j` exist then the matrix element `(i,j)` has one of the following forms:

'=>' : in case of a connection from place to transition
'<=' : in case of a connection from transition to place
'<=>': in case of a connection in both directions

If no connection exists, the matrix element is a '-' .

Furthermore, samosshell gives you information about the actual state of the Petri net. A list of all places with their tokens can be displayed. Obviously, these facilities of the samoshell are very primitive and require a very good knowledge of the way Petri nets are used in SAMOS. We work now on a visualisation of the Petri net component as part of the sophisticated explanation component.

You will start the samosshell by typing:

**$SAMOSROOT/bin/samosshell <database_name>**

### 4.11.4 SetUp

Whenever you want to test out some new events, conditions, actions or rules, it makes sense to have some shell scripts handy for your work to speed up this process. For making a complete refresh of the application, i.e., for emptying the application database, the samosOutfile and the alertfile, you need to have a shellfile setup like the one shown in Figure 12 (in FAME called *init-accounts*)

```
/bin/rm samosOutfile                                              [1]
/bin/rm alertFile                                                 [2]
/bin/rm displayFile                                               [3]
touch samosOutfile                                                [4]
touch alertFile                                                   [5]
touch displayFile                                                 [6]
createAccountsDB $OSUSER/AccountsRuleDB -i                        [7]
$SAMOSROOT/bin/ECA accounts_rule_def $OSUSER/AccountsRuleDB       [8]
make al                                                           [9]
```

*Figure 12 : init-accounts: a shellfile example*

Here is an explanation of the shellfile *init-accounts*:

4. *samosOutfile* will be removed.
5. *alertFile* will be removed.
6. *displayFile will be removed.*
7. A new empty *samosOutfile* will be created.
8. A new empty *alertFile* will be created.
9. A new empty *displayFile* will be created.
10. A new empty database will be created.
11. The rule definitions will be compiled
12. an executable version of the application will be created.

## 4.12 Things to Remember

At the end of the tutorial we give a list of some importants aspects must be taken into account during the development and maintenance of an active application. Most of them have been already mentioned or detailed discussed in the previous sections and thus, no comments are given here. Otherwise, a short explanation is given:

*For opening or creating the application database the predefined ColdStart and HotStart must be used* (see section 4.6)

*After defining new conditions or new actions remember to recompile your application* (see section 4.8).

*Before you begin with rule and event definitions you should create your application database* (see sections, 4.6 and 4.7).
*For transaction management in your application programs use the SAMOS'own transaction operations* (see section 4.5.2)

*You should always define method events outside rule definitions.*
The SAMOS compiler supports the event definition outside the rule definition by giving explicit name to events but also as part of the rule definition. In the later case, SAMOS is responsible for giving system generated names to events. In this SAMOS release 1.0 users are responsible for initializing the event detector for method events as discussed in 4.5.1. For this they must know the event name.

# 5 The Syntax of the SAMOS Rule Language

| | | |
|---|---|---|
| samos_statement | : | eca_def \| event_def \| condition_def \| action_def<br>\|<br>rule_del \| condition_del \| action_del |
| eca_def | : | DEFINE RULE rule_name ON event_decl [condition] DO action_decl coupl_def ';' |
| event_def | : | DEFINE EVENT event_name [IN time_interval] ';'<br>\|<br>DEFINE EVENT event_name event_clause [IN time_interval] ';' |
| condition_def | : | DEFINE CONDITION condition_name query_statement ';' |
| action_def | : | DEFINE ACTION action_name dml_statement ';' |
| rule_del | : | DELETE RULE rule_name ';' |
| condition_del | : | DELETE CONDITION condition_name ';' |
| action_del | : | DELETE ACTION action_name ';' |
| event_decl | : | event_name [IN time_interval] \| event_clause [IN time_interval] |
| condition | : | IF condition_name \| IF query_statement |
| action_decl | : | action_name \| dml_statement |
| coupl_def | : | COUPLING '(' coupling_mode ',' coupling_mode ')' |
| event_clause | : | prim_event \| comp_event |
| coupling_mode | : | IMMEDIATE \| DEFERRED |
| time_interval | : | (OVERLAP \| EXTENT) '(' '[' time_point TO time_point ']' ','<br>'[' time_point TO time_point ']' ')'<br>\|<br>'[' time_point TO time_point ']' |
| prim_event | : | method_event \| time_point \| periodic_event \| transaction_event |
| comp_event | : | '(' event_decl OR event_decl ')'<br>\|<br>'(' event_decl AND event_decl ')' [ ':' SAME '(' same_parameter ')' ]<br>\|<br>'(' event_decl SEQ event_decl ')' [ ':' SAME '(' same_parameter ')' ]<br>\|<br>NOT '(' event_decl ')' IN time_interval<br>\|<br>CLOSURE '(' event_decl ')' [ ':' SAME '(' same_parameter ')' ]<br>\| |

```
                        TIMES '(' repetitions ',' event_decl ')' [ ':' SAME '(' same_parameter ')' ]

same_parameter       :  USER | TRANSACTION | PARAMETER | OBJECT

time_point           :  absolute_time_point | relative_time_point | implicit_time_point

periodic_event       :  yearly_event | monthly_event | weekly_event | daily_event | hourly_event

method_event         :  (BEFORE | AFTER) class_name '(' header_file ')' '.' method_name
                        '(' source_file ')'

transaction_event    :  (BOT | EOT | ABORT)

absolute_time_point  :  year '.' month '.' day ',' hour ':' minute

relative_time_point  :  implicit_time_point '+' relative_time

implicit_time_point  :  OCCPOINT event_name

relative_time        :  ryears '.' rmonths '.' rdays ',' rhours ':' rminutes

yearly_event         :  EVERY [ frequency ] YEAR month [ '.' integer ] [ ',' hour ':' minute ]

monthly_event        :  EVERY [ frequency ] MONTH day [ ',' hour ':' minute ]

weekly_event         :  EVERY [ frequency ] WEEK weekday [ ',' hour ':' minute ]

daily_event          :  EVERY [ frequency ] DAY [ ',' hour ':' minute ]

hourly_event         :  EVERY [ frequency ] HOUR [':' minute]

year                 :  integer
month                :  1 .. 12
weekday              :  1 .. 7
day                  :  1 .. 31
hour                 :  0 .. 23
minute               :  0 .. 59
frequency            :  integer
ryears               :  integer
rmonths              :  integer
rdays                :  integer
rhours               :  integer
rminutes             :  integer

repetitions          :  integer

header_file          :  file_name
source_file          :  file_name

transaction_name     :  name
class_name           :  name
method_name          :  name
event_name           :  name
condition_name       :  name
action_name          :  name
rule_name            :  name

query_statement      :  str_const
dml_statement        :  str_const

str_const            :  '@' string '@'
file_name            :  <path> '/' <filename>
name                 :  letter (letter | digit )*
```

## Publications


1 K.R. Dittrich, S. Gatziu: Time Issues in Active Database Systems. *Proc. Intl. Workshop on Infrastructure for Temporal Databases*, June 1993.

2 K.R. Dittrich, S. Gatziu, A. Geppert (eds): The Active Database Management System Manifesto: A Rulebase of ADBMS Features. T. Sellis (ed): Proc. 2nd Intl. Workshop on Rules in Database Systems (RIDS), Athens, Greece, September 1995. Lecture Notes in Computer Science, Springer.

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

4 S. Gatziu, K.R. Dittrich: SAMOS: An Active, Object-Oriented Database System. *Bulletin of the TC on Data Engineering 15:1-4*, 1992.

5 S. Gatziu, K.R. Dittrich: Events in an Active Object-Oriented Database System. *Proc. 1st Workshop on Rules in Database Systems*, Edinbourgh, September 1993.

6 S. Gatziu, K.R. Dittrich: Detecting Composite Events in an Active Database Systems Using Petri Nets. *Proc. 4th Intl. Workshop on Research Issues in Data Engineering: Active Database Systems*, February 1994.

7 S. Gatziu, A. Geppert, K.R. Dittrich: Integrating Active Mechanisms into an Object-Oriented Database System. *Proc. 3rd Intl. Workshop on Database Programming Languages*, Nafplion, Greece, August 1991.

8 S. Gatziu, A. Geppert, K.R. Dittrich: The SAMOS Active DBMS Prototype. Technical Report 94.16, Computer Science Department, University of Zurich, October 1994.

9 A. Geppert, S. Gatziu, K.R. Dittrich: Rulebase Evolution in Active Object-Oriented Database Systems: Adapting the Past to Future Needs. Technical Report 95.13, Computer Science Department, University of Zurich, October 1995.

10 A. Geppert, S. Gatziu, K.R. Dittrich: A Designer's Benchmark for Active Database Management Systems: 007 Meets the BEAST. *Proc. 2st Workshop on Rules in Database Systems*, Athens, September 1995.

11 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, Computer Science Department, University of Zurich, November 1995.

12 A. Vaduva, S. Gatziu: Supporting Development and Maintenance of Rules in Active Database Systems. Working paper, Computer Science Department, University of Zu-

rich, February 1996.

13 A. Vaduva, S. Gatziu: Rule Termination in SAMOS. Working paper, Computer Science Department, University of Zurich, February 1996.

More information on SAMOS is available in the World Wide Web:
http: //www.ifi.unizh.ch/groups/dbtg/
Postsript files of almost papers can be also found on our www server.