Rulebase Evolution in Active Object-Oriented Database Systems: Adapting the Past to Future Needs

Andreas Geppert, Stella Gatziu, Klaus R. Dittrich

Institut für Informatik, Universität Zürich[1]

Technical Report 95.13

Abstract

Active database management systems (ADBMSs) support the specification of reactive behavior in the form of event-condition-action rules. The capability to modify the definition of reactive behavior is a crucial part of the ADBMS-functionality. Rulebase evolution is however a complex problem in systems that support composite events, particularly when event occurrences should be taken into account that have occurred prior to the modification. We describe which kinds of rulebase modifications are meaningful and should thus be supported by an ADBMS. We investigate syntactic and semantic aspects of rulebase evolution, and present an algorithm for rulebase evolution for the object-oriented ADBMS SAMOS.

Keywords:
active database systems, ECA-rules, rulebase evolution, object-oriented database systems

1 Introduction and Motivation

Active database management systems (ADBMSs) allow the specification and implementation of reactive behavior in the form of event-condition-action rules (ECA-rules). Example ADBMSs include SAMOS [8], REACH [4], Sentinel [6], ODE [11] (see [16] for a survey).

The collection of ECA-rules defined at a given point in time forms the *rulebase*. It is not feasible to assume that the rulebase remains fixed once it has been defined. It may be necessary to define new rules, to delete old ones, or to modify existing ones. An ADBMS thus has to support *rulebase evolution*. Rulebase evolution is necessary in at least two practical cases:

- the real-world behavior to be modeled changes. The ability to cope with changes in the real world is of particular importance in some potential application domains, such as software process modeling and enactment [2], and workflow management [12].

1. Authors' address: Institut fuer Informatik, Winterthurerstr. 190, CH-8057 Zurich, Switzerland. Email: {geppert|gatziu|dittrich}@ifi.unizh.ch. Fax: +49-1-363 0035.

- the modification of rules is required during debugging.

In the current state of the art, rulebase evolution is an urgent yet open problem, especially when composite event types are affected. A composite event is an event that happens only when other events (its components) have previously occurred. The occurrences of these components may span a large time interval, and hence the ADBMS has to monitor and store information about them. What makes rulebase evolution a complex problem is the need to adapt such "old" occurrences to "new" rules or event definitions.

A brute-force solution to ECA-rule modification would be to delete the rule to be modified (including its event, condition, and action parts) and to create the new desired one. As a side effect, occurrences of the deleted event would also be erased. This might or might not be correct, depending on the intended semantics. For instance, assume that rules model a process, and the process state is monitored through composite events. If component occurrences are deleted in such a case, information on the process state is lost, and it may be impossible to continue and complete the process correctly.

In this paper, we show how the problem of rulebase evolution can be approached. Particularly, we investigate the modification of event definitions and the adaptation of old occurrences to the new rulebase. This is a general problem in the area of ADBMSs and their applications, and we show how the problem can be solved independent from any concrete system, provided that the following prerequisites are fulfilled: previously occurred events are stored persistently and old primitive event occurrences can be re-constructed from the event history. We also show how the problem is solved algorithmically using the Petri net-based approach [10] of SAMOS [8] to event modeling and composite event detection.

1.1 Related Work

Rulebase modification as addressed in this paper is a challenge in any ADBMS that supports composite events. Other systems that support composite events include REACH [4], Sentinel [5], and ODE [11]. They of course support the definition of new rules, but to the best of our knowledge do not yet fully support rulebase evolution.

In several respects, rulebase evolution is equivalent to *schema evolution* in (passive) database systems. For schema evolution in relational systems, see [7]. Two schema evolution approaches for object-oriented DBMSs have been described for ORION [3] and O_2 [18]. The problems are quite similar to the topic of this paper: updates on the type level must be propagated to instances. Propagations can happen immediately upon schema modification, or stepwise when instances are actually used. Furthermore, the semantics of schema evolution must be well-defined (e.g., [3] checks soundness and completeness, and [18] considers structural and behavioral consistency). Similarly, updates of event definitions must be propagated to existing occurrences, and the set of supported update operations must be sound and complete.

1.2 Structure of the Paper

In the next section, we review the most important concepts of SAMOS as far as necessary for understanding the rest of the paper. Section 3 presents a solution to the rulebase evolution problem, and section 4 investigates syntactic and semantic aspects of rulebase evolution. Section 5 describes an implementation of event type modification based on Petri Nets, and section 6 concludes the paper.

2 Reactive Behavior in SAMOS

2.1 Rule Definition

SAMOS is an object-oriented ADBMS [8]. In addition to the data definition language, SAMOS provides a *rule definition language* as a means to specify ECA-rules [9]. It consists of constructors for the definition of events, conditions, and actions. Events can also be defined separately and afterwards used by name in rule definitions. Rule and event definitions in SAMOS have the following form:

```
DEFINE RULE      rule_name
ON               event_clause [WITHIN time_interval]
IF               condition
DO               action
COUPLING MODE    "(" coupling, coupling")"
PRIORITIES       ((BEFORE | AFTER) rule_name)*

DEFINE EVENT     event_name event_def
```

Specification 1. Top Level of Rule Definition Syntax in SAMOS

Rule or event definitions specify *event types*. Instances of event types are the actually happening events; they are called *event occurrences*. Subsequently, we simply refer to "events" if it is clear whether event *types* or event *occurrences* are meant. We also say sometimes that an event type `E` occurs when it is clear that an arbitrary instance of type `E` is meant to occur.

Events can be primitive or composite. Primitive events include time events, message sending events, transaction events, or abstract events (i.e., events that are explicitly signalled by applications). Composite events can be composed out of primitive or other composite ones using one of the event type constructors:

- *conjunction*: both components have to occur, regardless of order,
- *sequence*: like conjunction, but requires a specific order of the components,
- *disjunction*: one of the two components must occur,
- *negation*: the component event must not occur within a specific time interval,
- *times*: the component event must occur a specific number of times,
- *closure*: the component event is signalled at most once within a specified time interval.

For event definitions, a time interval can be specified to determine when an event has to occur in order to be considered as relevant (*monitoring intervals*). A monitoring interval is defined by two points in time, the *start time* and *end time*. Both can also be defined implicitly, i.e., through the occurrence of some event.

By default, every event occurrence pertaining to a component event type is considered for event composition (e.g., regardless of the transaction in which the event occurred). Event parameters (e.g., the identifier of the transaction in which an event occurred) can be used to restrict composite events to actually interesting ones. For instance, it can be required by such restrictions that all component events of a composite one have to occur within the same transaction (`same transaction` restriction).

Conditions and actions can be any executable program fragment including DBMS-operations, with the restriction that conditions always return a boolean value. If the event of a rule has occurred, the condition is evaluated. If the condition evaluation returns `true`, the action is executed. The transaction that contains an event occurrence is called the *triggering transaction* of the event. In SAMOS, it can be specified when to evaluate conditions with respect to the triggering transaction, and it can be defined when to execute the action with respect to the condition evaluation (*coupling modes*):

- *immediate*: directly after the triggering event has been detected/directly after the condition evaluation,

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

More details on rule specification can be found in [9], details of event definition and detection can be found in [9, 10]. The implementation of SAMOS on top of a commercial object-oriented database management system (OODBMS) is described in [14].

Subsequently, we introduce a simple running example. We consider the curriculum at a university. Assume that a student has to register for an intermediate exam. If he/she fails this exam, he/she has a second chance. After a second failure, however, the student will be relegated (Example 1). Let students be represented as instances of the class `Student`, which has methods `registerExam`, `failedExam`, `notify`, and `relegate` defined.

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

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

Example 1. Example SAMOS Rule with Composite Event

3 Rulebase Modification

Rulebase modification includes the deletion of defined rules, the creation of new rules, or the modification of existing rules. Deletion of rules means the deletion of the corresponding event, condition and action. Accordingly, creation of rules means the creation of events, condition and actions. A rule is modified whenever its event, condition, action, coupling modes, or priorities are modified.

Recall that events can be defined explicitly and that further event or rule definitions can refer to them by their name. Rules can also define an event type implicitly; in this case the event type is defined in-line in the event part of the rule definition. Modifying an explicitly defined event type thus affects all the rules and composite events that use this event type. The modification of the event part of a rule only affects that rule, but not others where the event type (even in the case that it is an explicitly defined event) is also used.

What makes the rulebase modification a complex problem is the modification of the set of event types and the migration of the sequence of occurred events to the new rulebase. These complex modifications of the rulebase are the major focus of this paper.

3.1 Terminology

Before we elaborate on rulebase modification we introduce some necessary terminology.

A *composite event type* is constructed out of primitive or other composite event types using the constructors described above. The event types participating in such a composition are called the *component events*. We say that two event types E_1 and E_2 *overlap in* E if E_1 and E_2 are different and both have E as a component (E *is in the overlap of* E_1 and E_2). An event occurrence e is said to be *used up for E'* in a sequence s of event occurrences if e is an occurrence of event type E in s, E is a component of E′, e is a component of e′, and e′ is an occurrence of E′ in s. Event occurrence e is said to be *completely used up* in s if it is used up in s for all E (the meaning of *completely used up* expresses the fact that an occurrence cannot be used as a component for any further composite event occurrence).

Finally, the *event history at time* t is a sequence s of time ordered event occurrences that have occurred until t and that are not yet completely used up in s. The event history at t+1 will be the old one, plus the new event occurrences (which have occurred at time t+1), minus those ones that are now completely used up.

The consumption mode underlying the definition of *used up* is *chronicle* [6]. This means that upon event composition the oldest existing occurrences of the component events are used. Furthermore, each component event occurrence can be used for at most one occurrence of a specific composite event type. Thus, once such an “oldest” occurrence has been used as a component for a composite event (say, E), then it is used up for E. Once it has been used for occurrences of all composite events its type participates in, then it is completely used up.

Consider our running example again. A sample event history might look like as follows (Figure 1). Let “r” stand for Student.registerExam and “f” for Student.failedExam, and let “Lucy” (L) and “Snoopy” (S) be the names of example students. Some time after the rule and event definition, both register for the exams. Lucy then fails twice while Snoopy fails only once. As soon as Lucy fails the second time, the sequence < L->r, L->f, L->f > is an occurrence of the event type RELEGATION. These three primitive events are thus completely used up and therefore are removed from the event history.

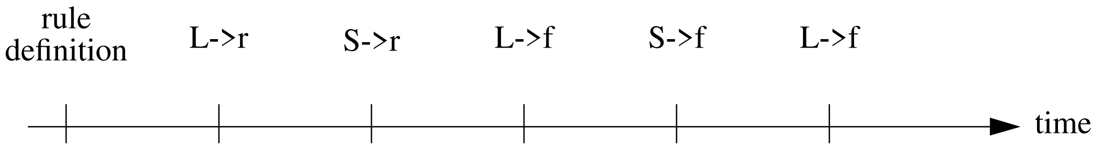

Figure 1. A Sample Event History with Rule Modification

3.2 Rule Modifications

In this section, we describe possible kinds of rulebase modifications.

The following operations for a rule are possible:

- it can be defined,
- it can be deleted,
- the event can be changed,
- the condition can be changed,
- the action can be changed,
- priorities can be added, deleted, or changed,
- coupling modes can be changed.

The creation of a new rule includes checks of the syntactic and semantic correctness of its definition. If the definition is correct, the responsible components are informed about the new rule (i.e., event detectors and rule execution component) and the rule definition is stored in the rulebase.

If an existing rule is deleted, it is removed from the rulebase. The only problems in the context of creations and deletions are the effects on existing event types; these problems are investigated separately in the subsequent section.

Conditions and actions are implemented using the DML of the underlying OODBMS. Thus, modifying either of them means to exchange the old code by the new one and to recompile the new condition or action. Obviously, the modification is allowed only if the new code fragments for the condition/action can be successfully compiled, i.e., syntactic and semantic analysis done by the DML-compiler succeeds.

Priorities are used to constrain the order of rule executions that take place upon the same event occurrences. In the case of adding or deleting priorities, this partial order changes and must subsequently be considered by the rule execution component. An addition of priorities is allowed only if the new set of priorities forms a partial order. Whenever a modification would establish a cycle in the set of relative priorities (e.g., `R1` before `R2` and `R2` before `R1`), the modification request is rejected.

Changing coupling modes means that the point in time when a condition (action) is executed with respect to the event occurrence (condition evaluation) changes. The only requirement for legitimate modifications of coupling mode is that the new coupling mode is one of the supported ones (`immediate`, `deferred`, or `decoupled`). Note that as a side effect, priorities can become meaningless when coupling modes are changed, since priorities define the partial order of rules with the same coupling mode (`immediate` or `deferred`).

All these changes are simple to implement from a system point of view. Nevertheless, we also require the rulebase to meet some semantic properties (such as termination), which can be affected by any of the modifications and thus should be checked after rule modification. We discuss this issue in section 4.3.

Finally, note that all the discussed modifications affect rule execution in some way. They however do not depend on the current event history, nor do they require changes of the existing event history. This is different with event type modifications, to be discussed in the following section.

3.3 Event Type Modifications

In this section, we investigate the possible kinds of event type modifications:

- creation of new event types,
- deletion of existing event types,
- modification of existing event types.

For some of the modifications there are different options whether (and how) the event history is migrated and used for the occurrence of new composite event types. The possible relationships with the event history are the following:

- old event occurrences are *not* taken into account,
- old event occurrences *are* considered, in which case we can distinguish two subcases:
  — the ADBS tries to use them for the new event types(s), or
  — the modification results in a new event type, the old one remains in the rulebase, and some of the old and new event occurrences are used for the old version.

It depends on the application semantics which case should apply for a specific event type modification. The ADBMS hence has to support all the options and leave the choice of the desired semantics to the rule designer.

Subsequently, all these cases are motivated by examples and discussed in more detail. We thereby assume that the modification (i.e., the new event type in the case of modifications or creations) is syntactically and semantically correct with respect to the rule definition language. This property is assured by the corresponding language compiler.

3.3.1 Insertions

The first type of rulebase evolution with respect to event types is *insertion* of a new event type. It depends on the application semantics whether past occurrences of existing event types are taken into account or not.

In the first case, assume that either the new event type or some components of it have already been components of some other existing event type. The corresponding occurrences of these events can be used for the new event type (default case). It is therefore possible that the new rule affects the past (i.e., the current behavior is determined by new rules and events which have happened in the past). Moreover, depending on the concrete event history at this point in time, it can happen that upon event type or rule definition, newly defined events occur immediately and that attached rules fire.

In the second case, the rule designer does not want old occurrences to be taken into account (for a concrete modification in question). He/she excludes old occurrences for the new event in that he/she specifies a monitoring interval for the new event type. The start time of the monitoring interval would then be the current date, and thus old event occurrences would not be considered.

In our running example, consider the following new rule that triggers the sending of a bill upon registration for the exam:

```
DEFINE RULE BILL_REGISTRATION
ON      Student.registerExam
DO      ... //send a bill for the registration
```

Example 2. New Rule Using old Event Occurrences

The example event history contains an old occurrence of `Student.registerExam` (for the student Snoopy). Hence, Snoopy will receive a bill after the definition of the new rule. Alternatively, if this case is not desired, one would specify ON Student.registerExam WITHIN [today[2] - ∞]. The monitoring interval then would ensure that only future registrations are billed.

3.3.2 Deletions

The second case of rulebase modification with respect to event types is *deletion*. Those parts of the event type that are not used as components by other event types or rules are removed from the rulebase. Those that still are components of other events or rules remain. Deletion of event types is propagated to the occurrences of the type: if the corresponding event type is removed from the rulebase (since it is no longer needed for other events or rules), its occurrences will be removed from the event history. Otherwise, if the event type remains in the rulebase, its occurrences remain in the event history.

3.3.3 Updates

The third kind of event type modification is *update*. Updating an event type refers to modifying its definition and adapting previous event occurrences if necessary. It again depends on the desired application semantics whether the new event type is valid for past (component) events, or whether old occurrences (plus new ones) are still considered under the old definition. Likewise, are old event occurrences taken into account for new composite event occurrences or not?

In discussing these questions, let us assume that there are two versions of an event type: the old, unmodified one, and the new, modified one (whereas we do not intend to say that these versions actually coexist in the rulebase). In principle, there are three possibilities for relating event occurrences to versions of the event type:

1. occurrences of the *old* event type are signalled even after the modification if component occurrences already exist,
2. occurrences of the *new* event type are signalled after the modification, possibly with old occurrences as components,
3. occurrences of the *new* event type are signalled and only component occurrences after the modification are taken into account.

2. this corresponds to the point in time when the rule modification happens

Each of the three cases can make sense depending on the application semantics. The first two cases have in common that they adapt the existing event history. This is important in environments that use the ADBMS as an engine for controlling the dynamic behavior of the applications. Examples of such applications are workflow management and process-centered software development environments. In one of our projects, we use an ADBMS for controlling and guiding *processes* [15]. The ADBMS stores the internal state of a process as a sequence of events. Occasionally, updating event types might be required due to changed application semantics (e.g., changed guidelines in a design environment, or new laws to be enforced by a workflow management system). If now the ADBMS "forgets" old occurrences, then information on process states gets lost, and it is difficult, if not impossible, to successfully complete the process. Especially in long-lasting design processes, this might have catastrophic consequences.

The three cases can be expressed syntactically by two operations: modification (including modification/addition of monitoring intervals), and insertion of new event types and rules. Case 1 is a modification followed by an insertion, and cases 2 and 3 are pure modifications. Subsequently, we discuss the semantics for each of the three cases in more detail, illustrated by our running example.

Detecting Old and New Event Types

In this case, the application semantics requires that "old" rules are still valid for past occurrences, while the new rule is valid for new occurrences. Consider a modification of the university curriculum such that students have a third chance in an oral exam. However, this is only valid for those students who will register for the exam in the future. For those who have already registered, rules (and therefore event types) remain unchanged.

It is not appropriate to simply define a new event type in addition to the old one, since then some occurrences could be used for *both* types. The old type cannot be deleted, since then the information on the old occurrences is also lost, and the new event type is not valid for these occurrences. Thus, actually *two versions* of the same event type are required. The two versions differ in one or more component events whose occurrence time identifies the relevant event type version.

We refer to this specific kind of component events as *determining events*. If a determining event has occurred in the past, then the old event type is still valid regardless of the occurrence time of other component events. The new event type, on the other hand, can only have occurrences whose component determining events occur posterior to the modification.

It depends on the application semantics which component events are "determining". The rule designer has to identify them and assigns monitoring intervals to them which serve to distinguish between old and new versions of the composite event. The monitoring interval for a determining event of the old version is [∞ - today] and that for the new version is [today - ∞]. In other words, the old version is valid for those composite events that can be constructed out of determining events that have occurred until today (and possibly other events). The new version is valid for composite events that can be constructed from determining events that occur after today (and possibly other events).

"Determining events" are specified implicitly through monitoring intervals by the rule designer. They are thus not a SAMOS language construct, but are introduced solely to clarify the discussion. Additionally, note that in general any component event can act as a determining event.

In our running example, the resulting event types would look like as follows (`Student.registerExam` is the determining event). The old event is the result of a modification, the second one is the result of an insertion.

old: (Student.registerExam **WITHIN [∞ - today]**;
TIMES(Student.failedExam, 2): same object): same object
new: ((Student.registerExam **WITHIN [today - ∞]**; TIMES(Student.failedExam,2):
same object) : same object; Student.FailedOralExam): same object

Example 3. Event Type Modification Resulting in Two Event Type Versions

Detecting Modified Event Types with Old Occurrences

Here we assume a modification that should be valid immediately (e.g., from now on, all who have failed twice get a third chance). The event type is modified, but old occurrences of components must still be taken into account.

This case is handled as follows. The old event type will be "updated in place". The event definition will be modified according to the modification request. As a side effect, component events that are no longer part of the (modified) type will be removed from the rulebase, unless they are also components of other event types. Afterwards, previous primitive event occurrences will be considered and the event history will be adapted.

Detecting Modified Event Types and Forgetting Old Occurrences

In this case, assume that a new rule is defined whose event has already been a component of an existing event (i.e., the event type is modified in that new rules are attached to it). The ADBMS might then have old occurrences stored, but it is not desired to take them into account for the modified event type. As an example, assume a new rule that bills each future registration with a certain amount (Example 2). We thus have to prevent that old occurrences are used for the modified event. This is done in that the user specifies monitoring intervals for component events.

3.4 Multiple Rulebase Modifications

Consider now the case that multiple event types are modified in subsequent rulebase updates. Multiple event type modifications are not necessarily confluent; i.e., the final result may depend on the order in which the modifications are executed. Especially, different execution orders of multiple modifications can lead to different sets of existing event types whose definition and occurrences can be preserved, as illustrated by the following example. Assume that the rulebase consists only of the following rules:

Rule 1:		Rule 2:	
ON	`(E1,E2)`	ON	`(E3;E4)`
IF	<condition1>	IF	<condition2>
DO	<action1>	DO	<action2>

Example 4. Sample Rulebase for Multiple Rulebase Modifications

Consider the following event type modifications:

A. the event of `Rule1` is modified to `(E1,E3)`,

B. the event of `Rule2` is modified to `(E5;E4)`.

If the modification `A` is performed first, then the event history can be preserved for `E3`, since it is a component of the second old event. Note however that there is no overlap of `(E3;E4)` and the event part of another rule. Thus, if modification `B` is performed first, then `E3` would first be deleted and inserted again (in `A`). Although the final results are equal as far as event *types* are concerned, there are differences with respect to the part of the event history that can be adapted to the new types, and consequently in the resulting event history after the modifications. If the conjunction is modified first, then instances of `E3` would be preserved, while otherwise `E3` is first deleted and then created again, and thus its occurrences would be lost.

We therefore group event type modifications into transactions and remove event types only at the end of such a transaction. In this way, “old” event types removed from one composition can still be preserved for new ones, as long as both modifications are performed within the same transaction.

3.5 Rule Execution upon Event Type Modification

Consider a modified event type whose components have already been in the old rulebase, and let the (old) event history contain corresponding occurrences. As a result, the new event type will have occurrences at modification time. Furthermore, let one or more rules be attached to this event type. Obviously, these rules would fire directly upon rulebase modification! Depending on the intention of the rule designer, this can be desired or not.

On the one hand, rules cannot simply be executed upon rulebase modification by default. For instance, consider a rule that should be executed, and that has the coupling mode `immediate` or `deferred`. In this case, the rule would have to be executed as a subtransaction of the triggering transaction, which however has already committed. Even worse, the action part of such a rule may issue an abort of the triggering transaction. Therefore, in this case it is intuitive not to execute rules at all *upon rule modification*. On the other hand, the rule execution may be quite important for the application. Assume, for instance, that the monitoring interval of a negative event has been shortened, that the end point is already reached, and that the action is a notification of a user. Apparently, one would expect *this rule* to be executed.

This dilemma is resolved in a meaningful way as follows. First, rules can be executed as subtransactions of the rulebase modification transaction. All rules with coupling modes `immediate` or `deferred` are executed at the end of this transaction. Second, such rules are not allowed to abort the triggering transaction. By default, rules are executed whenever they are triggered due to rulebase modification. However, the rule designer can specify for each rulebase modification that rule execution is actually not desired on a per-rule basis (see the option NO_EXEC_RULE in Specification 2 below).

4 Syntax and Properties of Rulebase Evolution

This section describes how the rule definition language of SAMOS supports rulebase evolution (the syntactic aspect). Focus then turns towards properties of event type modification and towards effects on rulebase semantics (the semantic aspect).

4.1 Specifying Rulebase Modifications in SAMOS

The syntax of rulebase modification is as follows (for the sake of comprehensiveness, the syntax includes modifications of conditions and actions as well). Note that there are two commands for the modification of event types: the first one only alters the definition of the event part of a single rule, while the second modifies an explicitly defined, named event type.

```
MODIFY EVENT IN          rule_name TO new_event_def option
MODIFY CONDITION IN      rule_name TO new_condition
MODIFY ACTION IN         rule_name TO new_action
MODIFY COUPLING IN       rule_name TO coupling, coupling
ADD PRIORITY IN          rule_name (AFTER | BEFORE) rule_name
DELETE PRIORITY IN       rule_name (AFTER | BEFORE) rule_name
option ::= [NO_EXEC_RULE]

MODIFY EVENT event_name TO new_event_def [NO_EXEC_RULE rule_name*]

DELETE RULE      rule_name
DELETE EVENT     event_name
```

Specification 2. Top-Level Syntax of Rule Modification

According to the modified curriculum, a student that failed twice has a third chance in an oral examination (see Figure 2). Only if he/she fails in this exam as well, relegation will be the consequence.

```
MODIFY EVENT RELEGATION TO
((Student.registerExams; TIMES(Student.failedExam,2): same object) : same object;
Student.FailedOralExam): same object
```

Example 5. Sample Event Type Modification

Note that no monitoring intervals are specified in this example. Thus, the modified event type overrides the old one, and old occurrences are taken into account (the default behavior). This might not be the case that users expect; some might assume that by default old occurrences are not considered (and consequently, that rules do not fire as a result of rulebase modification). However, we argue that active databases — unlike passive DBMSs — always have a

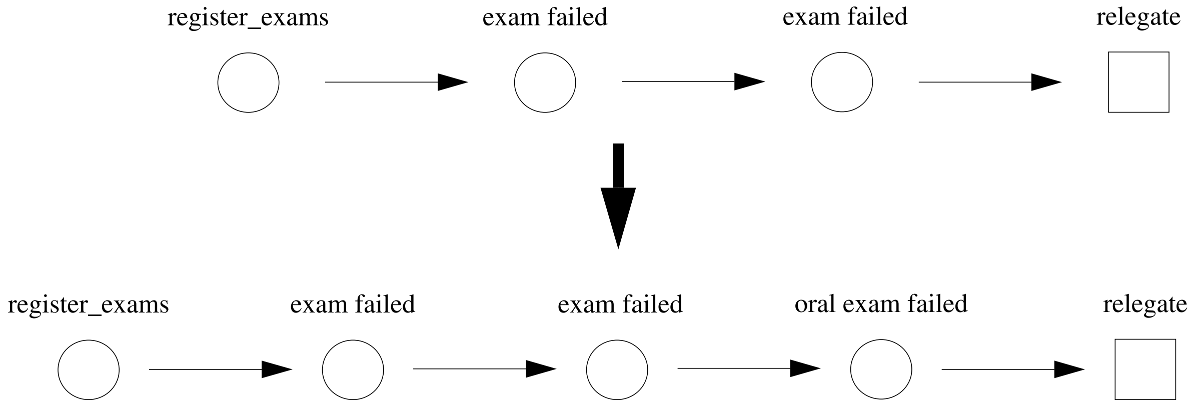

Figure 2. Sample Event Type Modification

recorded history, and that users should be aware that their application is dynamic in nature. It goes without saying that ADBMSs must support them in understanding the impacts of rulebase evolution, e.g., by appropriately visualizing the rulebase and the event history.

4.2 Properties of Event Type Modifications

Event type modification should have two properties:

- the resulting event history *conforms* to the rulebase (the event history contains only those occurrences that are instances of event types defined in the rulebase),
- the resulting rulebase is *consistent* with the event history (the event types of the rulebase have adapted occurrences in the new event history).

Both properties are formal notions and therefore must be proven. We give a verbal rationale why they hold for our approach.

Since the event history is adapted to the new rulebase, event occurrences are removed if they cannot be used for future composite events or if they are completely used up under the new rulebase. Thus, the new event history conforms to the modified rulebase.

After rulebase modification, the event history contains all those (old) primitive event occurrences that are not yet used up. Composite events are composed out of the primitive ones according to the new event type definitions. Thus, the new rulebase is consistent with the new event history after rulebase modification.

4.3 Effects of Rulebase Evolution on Rulebase Semantics

In this section, we consider the effects of rulebase evolution on desired semantic properties of rulebases. A rulebase is said to be [1]:

- *terminating* if rule execution terminates for any possible database state,
- *confluent* if the final database state after rule execution does not depend on the execution order of rules (as far as the execution order is not pre-defined by priorities), and
- *observably deterministic* if its rules produce a unique stream of actions visible in the environment (i.e., which are observable by users or applications).

SAMOS guarantees confluence and observable determinism for rules with `immediate` and `deferred` modes out of the following reasons:

1. the order of rule execution is always the same for the rules trigger by a specific event. This order is defined by the order of rule definitions and the specified priorities. In other words, there is never a real choice which rule out of a set of rules is to be executed next.
2. in the current prototype version, SAMOS executes rules sequentially in the order described above.

Out of the three properties for rulebases, termination remains to be proven. Moreover, termination might be violated after any of the modifications of the rulebase has been performed. For instance, a set of rules which has been known to be terminating might not be so after the action of a rule has been modified, namely if the new action raises an event that establishes a cycle in the triggering graph. We thus require that for a terminating rulebase the modified rulebase is terminating as well.

Rulebase analysis is a complex task, especially in systems such as OODBMSs that use computationally complete languages for specifying conditions and actions. Proving the termination property therefore requires a large degree of user interaction. SAMOS users are currently assisted by the SAMOS rule analyzer [17] in proving termination, which then can also assist in proving termination for modified rulebases.

Elaborate interactions between rulebase evolution and rulebase analysis have not been considered so far and are not (yet) provided by SAMOS. To the best of our knowledge, approaches for this task also have not yet been found elsewhere; only initial ideas on advanced rulebase analysis in relational systems (e.g., incremental analysis) can be found in the conclusion of [1].

5 Implementation

In this section, we show how event type modification is implemented in SAMOS using the composite event detector of SAMOS and its management of the event history. For the sake of comprehensiveness, we first review the SAMOS approach to composite event detection.

5.1 Implementation of the Event History and Composite Event Detection

The SAMOS Petri Net (S-PN) [10] is responsible for composite event detection and also stores the event history. Each kind of composite event constructor defines a corresponding S-PN structure as a set of places, transitions and connections between them. Event types are modeled as places which are marked with (possibly multiple) tokens. Generally, tokens represent occurrences; they are in turn objects and their attributes store parameters of the corresponding occurrence (e.g., identifier of the triggering transaction). Components of a composite event type are the input places of the corresponding S-PN structure. Whenever the input places are marked, the corresponding transition fires. As a result, the output places of the transition are marked (possibly followed by the firing of further transitions, and so on recursively). This procedure is called the *token game*. Tokens of output places generally represent composite event occurrences. The fact that a composite event type is in turn a component of another one is reflected in S-PN by output places being also input places. The event history in SAMOS is represented by the current state of the S-PN component (i.e., as the set of tokens of input and output places).

Figure 3 shows the S-PN structure for our running example.

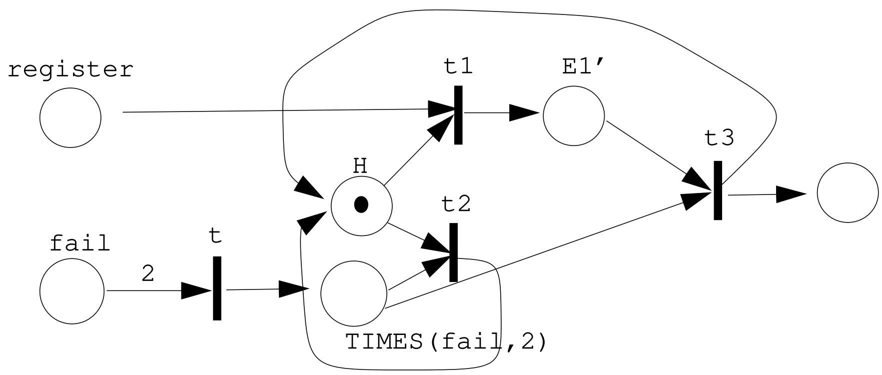

Figure 3. Petri Net Structures for the Example Composite Events

5.2 Implementation of Event Type Modification Using SAMOS Petri Nets

In this section, we show how event type modification is performed operationally.

The initial phase of event type modification determines all affected (composite) event types. Even if only one event type is modified, several others may be affected, namely if there are non-empty overlaps with other unchanged event types. Second, we need information on the primitive components of the composite event to be modified, and on occurrences of its component events (i.e., the event history). This information is only available if the ADBMS supports a model (like automata or Petri nets) which allows the representation of ordered sequences of primitive events and the actual state of detection processes in the system. In other cases, this information might be hidden in programs which implement the event detection.

In terms of S-PN, the process for composite event definitions is as follows:

1. the places affected by the event type modification are determined and then their tokens are used to identify primitive event occurrences,
2. the Petri Net *structure* is modified,
3. the token game is re-played.

In its usual operation mode, SAMOS performs the token game in order to *compose* composite events out of primitive or other composite ones. However, in the first step of event modification, composite event occurrences are *decomposed* into their component events. The structure of S-PN parts is used to determine the *types* of components (e.g., a method event, or in turn a specific composite event type). Furthermore, concrete tokens and their attributes are used to determine component occurrences. First, the presence of a “composite event token” determines that there must be component tokens (depending on the applied composite event constructor). Second, the attributes of the composite event token are used to determine the attributes of the respective component tokens. This procedure is applied transitively, until only primitive event occurrences remain.

Whenever an occurrence of a primitive event type is obtained during this step, the corresponding token is added to the place that represents the event type. Since one place may contain several tokens which are ordered according to their occurrence time, the new token is inserted into the list of tokens so that the ordering of tokens still holds. Note that in contrast to the normal mode of operation the token game is not played upon primitive event signalling during this step.

In the second step, the structure of the S-PN is modified, so that it reflects the new definition of the composite event. Places that represent modified composite event types, as well as eventual auxiliary places are removed. For primitive events, however, two cases are distinguished: primitive event types that are a component in the new definition as well, and such that are not. In the former case, the corresponding place including its tokens must be preserved for the new Petri net structure. In the latter case, the place representing that event type is removed as well.

In the second (sub)part of this step, the new structure of the S-PN is built. Primitive events that have already been a component of *any* composite event in the rulebase are reused. Furthermore, the S-PN is built according to the modified definition. Note that in this way, any modification can be performed. The important question is just how many of the old primitive events are also components of the new one. In the extreme case, the old and the new composite event have nothing in common, with the effect that no primitive event occurrences can be taken over from the old event history.

The third step plays the token game. Recall that as a result of this step, events may occur that trigger rules (section 3.5).

The appendix shows this algorithm in pseudo-code.

6 Conclusion and Future Work

In this paper, we have introduced an approach to rulebase evolution. We have mainly considered the modification of composite event types, which is a major requirement in some ADBMS-applications. Concretely, we have encountered this requirement in a project where we use SAMOS as the implementation platform for cooperative process-centered environments [15] such as workflow management systems and process-centered software development environments. Although we illustrated the rulebase evolution approach with our own system SAMOS, the approach and the results are applicable for other systems provided that three prerequisites hold:

- the system stores previous event occurrences,
- primitive event occurrences can be re-constructed from the event history, and
- the rule definition language supports monitoring intervals.

The first two prerequisites are indispensable, while the third one allows to express some cases such as versioning of event types without the need to extend the rule definition language.

Three issues are subject to future work. First, our approach requires the ADBMS to be quiescent when event type modification is performed, and changes are propagated to the event history immediately. Ideally, both restrictions would be dropped; it would then be possible to

- perform event type modification even if simultaneously new events are occurring,
- propagate changes to the event history on demand.

Second, our approach guarantees certain properties of event type modifications (consistency of the rulebase with respect to the event history and conformance of the event history to the rulebase). As far as the rulebase itself is concerned, the rule designer has to check termination for the entire rulebase after a modification. An incremental approach is a subject of future work.

Finally, we have so far aimed at functionality rather than performance. We expect that rulebase evolution occurs rather seldomly. Thus, performance of rulebase evolution is not as important as that of other tasks of an active DBMS. We however plan to enhance the BEAST benchmark we have developed for ADBMS performance measurement [13] with tests for rule and event definition and modification.

7 References

1. A. Aiken, J.M. Hellerstein, J. Widom: *Static Analysis Techniques for Predicting the Behavior of Active Database Rules*. ACM Trans. on Database Systems 20:1, March 1995.
2. P. Armenise, S. Bandinelli, C. Ghezzi, A. Morzenti: *A Survey and Assessment of Software Process Representation Formalisms*. International Journal of Software Engineering and Knowledge Engineering. December 1993.
3. J. Banerjee, W. Kim: *Semantics and Implementation of Schema Evolution in Object-Oriented Database Systems*. Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, San Francisco, CA, May 1987.
4. A.P. Buchmann, J.Blakeley J.A. Zimmermann, D.L. Wells: *Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions*. Proc. of the 11th Intl. Conf. on Data Engineering, Taipei, Taiwan, March 1995.
5. S. Chakravarthy, D. Mishra: *Snoop: An Expressive Event Specification Language for Active Databases*. Data & Knowledge Engineering 14:1, November 1994.
6. S. Chakravarthy, V. Krishnaprasad, E. Anwar, S.-K. Kim: *Composite Events for Active Databases: Semantics, Contexts, and Detection*. Proc. 20th Intl. Conf. on Very Large Data Bases, Santiago, Chile, September 1994.
7. R. Elmasri, S.B. Navathe: *Fundamentals of Database Systems*. 2nd ed., Benjamin/Cummings, 1994.
8. S. Gatziu, A. Geppert, K.R. Dittrich: *Integrating Active Mechanisms into an Object-Oriented Database System*. In P.C. Kanellakis, J.W. Schmidt (eds): Proc. 3rd Intl. Workshop on Database Programming Languages, Nafplion, Greece, August 1991.
9. S. Gatziu, K.R. Dittrich: *Events in an Active Object-Oriented Database System*. In N.W.

Paton, H.W. Williams (eds): Rules in Database Systems. Workshops in Computing, Springer-Verlag, 1994.

10. S. Gatziu, K.R. Dittrich: *Detecting Composite Events in an Active Database Systems Using Petri Nets*. In J. Widom, S. Chakravarthy (eds): Proc. 4th Intl. Workshop on Research Issues in Data Engineering: Active Database Systems, Houston, February 1994.
11. N.H. Gehani, H.V. Jagadish, O. Shmueli: *Composite Event Specification in Active Databases: Model & Implementation*. Proc. of the 18th Conf. on Very Large Data Bases (VLDB), Vancouver, Canada, August 1992.
12. D. Georgakopoulos, M. Hornick, A. Sheth: *An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure*. Distributed and Parallel Databases, Kluwer Academic Publishers, September 1994.
13. A. Geppert, S. Gatziu, K.R. Dittrich: *Performance Evaluation of an Active Object-Oriented Database Management System: 007 Meets the Beast*. In T. Sellis (ed): Proc. 2nd Intl. Workshop on Rules in Database Systems, Athens, Greece, September 1995. LNCS 985, Springer 1995.
14. A. Geppert, S. Gatziu, K.R. Dittrich, A. Vaduva, H. Fritschi: *Architecture and Implementation of the Active Object-Oriented Database Management System SAMOS*. Technical Report, Institut fuer Informatik, Universitaet Zuerich, 1995.
15. A. Geppert, M. Kradolfer, D. Tombros: *Realization of Cooperative Agents Using an Active Object-Oriented Database Management System*. In T. Sellis (ed): Proc. 2nd Intl. Workshop on Rules in Database Systems, Athens, Greece, September 1995. LNCS 985, Springer 1995.
16. N.W. Paton, O. Diaz: *Active Database Systems*. Technical Report, Heriot-Watt University, Edinburgh, Scotland, November 1994.
17. A. Vaduva, S. Gatziu: *Rule Analysis in SAMOS*. Technical Report, Database Technology Research Group, University of Zurich, 1995.
18. R. Zicari: *A Framework for Schema Updates in an Object-Oriented Database System*. Proc. 7th Intl. Conf. on Data Engineering, Kobe, Japan, April 1991.

Appendix: The Event Type Modification Algorithm

```
algorithm EventTypeModification
    in modifications: set (tuple (old: EventType, new: EventDefintion) )
    /* set of modifications. “old” is NULL in case of insertions, “new” is empty in case of deletions*/

isComponent(ev:EventType): BOOLEAN
/* checks whether an event type is a component of another composite event type*/

isComposite(ev:EventType): BOOLEAN
/* checks whether an event type is a composite one*/

isOld (ev: EventDefinition): BOOLEAN
/* checks whether an event type has already existed before the modification */

composites(ev:EventType): set(EventType)
/* determines for an event type all composite events types in which it participates */

components(ev:EventType): set(EventType)
/* determines for a composite event type all its components*/
proc decompose(finEvs: set(EventType))
```

/* composite event occurrences (tokens) are decomposed into their component events */

```
proc determineFinalEvs (start: EventType, visited: set(EventType) ): set(EventType)
/* determines for a particular event type to be modified the complete set of event types that (1) are not
components themselves and (2) have to be decomposed in step1 of the modification algorithm. The set
visited is used to prevent cycles in case of overlaps*/
        result: set(EventType)
        if (isComponent(start) ) then
            foreach ev in composites(start)
                if (ev not in visited) then result += determineFinalEvs(ev, visited + {start})
                /*the current event is a component -> its composites have
                to be decomposed as well*/
        else result += {start}
        if (isComposite(start) then
            foreach ev in components(start)
                if (ev not in visited) then result += determineFinalEvs(ev, visited + {start})
                /*check components as well, since there might be overlaps leading to
                further events to be decomposed*/
        return (result)

proc modifyEventType (newEvDef: EventDefinition)
/* the structure of the Petri net is modified, such that it afterwards reflects the new definition of the composite
event */
        if (isComposite(newEvDef) )
            foreach evDef in newEvDef
                compList += modifyEventType(evDef)
                /*make event types for the components*/
            newET := makeSPNstructure (newEvDef, compList)
            /*make a new component event type based on its component event types*/
        else if (isOld(newEvDef) then
            /*newDef defines a primitive event -> check if it can be preserved*/
                newET = copy(newEvDef)
                /*the event type can be preserved, do not create a new type*/
            else newET = makePET (newEvDef)
        return(newET)

main program
    mod: tuple (old: EventType, new: EventDefintion)
    affectedEvents, visited: set(EventType) = { }

    foreach mod in modifications                 // determine event types to be decomposed
        affectedEvents += determineFinalEvs(mod.old, visited)
    decompose(affectedEvents)                                         // decompose them
    foreach mod in modifications              // create the new event type out of the definition
        modifyEventType (mod.new)
    playTokenGame                                                    //re-play the token game
```