Investigating Termination in Active Database Systems with Expressive Rule Languages

Anca Vaduva, Stella Gatziu, Klaus R. Dittrich

Technical Report 97.03, April 97

Institut für Informatik, Universität Zürich
{vaduva, gatziu, dittrich}@ifi.unizh.ch

Abstract

The powerful functionality that active mechanisms add to database management systems presents, besides many advantages, a number of problems related to the control of their behavior. This paper deals with one of these problems: the termination of rule execution. We explain the termination aspect and the aim of termination analysis. Then, we present our approach for investigating the termination of rule execution. In contrast to others, this approach also addresses expressive rule languages as they have been proposed for various recent active database management system prototypes.

1 Introduction

Many applications from areas like workflow management, network management, finance, medicine and transportation show some sort of active behavior and are thus well-suited for support by active database management systems (active DBMS). Accordingly, much research has been done in the area so far, e.g., towards the integration of active mechanisms into relational and object-oriented database management systems, the support of language constructs for the modeling of reactive behavior, the development of concepts for rule execution, or the investigation of architectural issues for the construction of an active DBMS. A number of prototypes have been built, including Starburst [19], POSTGRES [16], ACCOD [5], REACH [6], Sentinel [7], NAOS [8], Ode [11], Ariel [13], TriGS [14], Chimera [19], and SAMOS [10].

Although the advantages of active DBMS are nowadays well-known, they are still not widely used in practice. This is even true for simple active mechanisms like triggers in relational systems which are commercially available since several years. One main problem is that especially for large rule sets, defined by different people at different points in time, rule behavior and potential conflicts and dependencies between rules are hard to predict. One critical aspect is how to guarantee the termination of rule execution, i.e., how to avoid the faulty behavior when rules trigger each other indefinitely ("nontermination").

There are two ways to approach this problem. One way is to handle termination just during rule execution. This means that the system would stop rule execution if it assumes that an infinitely cascading triggering is taking place (e.g., if a specific upper bound of triggered rules has been reached). The other way to handle termination for a given set of rules is through termination analysis, which may be static or dynamic. Dynamic methods [4] examine rule behavior at runtime. The main drawback is that termination cannot be guaranteed before the actual use of active DBMSs. Static analysis investigates rule definitions at compile time in order to determine subsets consisting of rules that may trigger (directly or indirectly) each other. If such a rule subset exists, the rule designer has to modify the specification of some rules and investigate termination again. Otherwise, when no rule subsets are found, termination of rule execution is guaranteed for the given rule set.

Some work on static termination analysis of rule execution has been reported elsewhere [1, 2, 3, 18]. The existing approaches deal, however, only with "simple" rule languages, that means languages supporting only a limited set of constructs for specifying active behavior. Expressive rule languages like those as supported in, for example, SAMOS [10], ACCOD [5], REACH [6], Sentinel [7], NAOS [8], Ode [11], TriGS [14] and Chimera [19] offer more elaborate features (e.g. composite events, time events, complex actions, coupling modes). They generally allow the specification of more application semantics in the form of rules. However, the price to be paid is that rule analysis is more complex. Conflicts between rules are harder to foresee. One of the reasons is that there are far more elements the triggering of a rule depends on. For example, a rule defined on a complex event is triggered only when all component events have occurred. Therefore, compared to a simple rule language, it is more difficult to decide when rules may generate an infinite loop during execution.

The aim of the paper is twofold. First, we present the aspect of termination of rule execution in active DBMS in general, we give a definition for it and discuss the semantics of nontermination for rules specified using expressive rule languages. Second, based on the conclusions from the first part, we present our approach for supporting termination analysis covering these complex cases. It focuses on the influence of complex events on the analysis. Our ideas have been implemented in a tool that detects and visualizes potential loops that could occur during the execution of an application. This information helps the rule designer to decide whether the rule set has to be improved. If so, termination analysis must be repeated.

The remainder of this paper is organized as follows. The next section gives an overview of expressive rule languages. Section 3 discusses the exact meaning of termination of rule execution, Section 4 presents our approach for the investigation of termination, Section 5 considers related work, and Section 6 concludes the paper.

2 Expressive Rule Languages

This section introduces the most important concepts of a rule language, and emphasizes the particular elements we would expect to see in an expressive one. A rule definition language is used for the specification of **E**vent/**C**ondition/**A**ction (ECA)-rules [9]. Each rule consists of an event, a condition and an action. If R is the set of defined ECA-rules, we will use the notation $r_i(e_i, c_i, a_i)$ for a rule $r_i \in R$. For the sake of simplicity, if a part of a rule has no relevance in a specific context of discussion, it will be left out in the notation, e.g., $r_i(e_i, a_i)$.

An *event* is an occurrence the active database system must react to. Simple rule languages allow only single events (called *primitive events*) to be specified, whereas expressive rule languages also support the specification of *composite events*. Examples of primitive events are:

- *data modification* (for relational database systems) or *method events* (for object-oriented database systems) occur at the beginning or at the end of a data modification operation or of a method execution,
- *transaction events* occur before or after a transaction operation (begin, abort, commit),
- *time events* occur at a specific point in time (absolute time event) or periodically.

Composite events are specified by applying operators to component events which can be primitive or other composite events. For example, the *disjunction* is signalled when at least one component event occurs. The *relative time* (<event> ± <time unit>) specifies an event occurring at a certain time interval distance to another event. The *conjunction* and the *sequence* are signalled when all component events occur, regardless of their order or considering their order, respectively. We distinguish between two groups of operators, depending on the number of event occurrences that the resulting composite event requires in order to occur: (a) “unary” operators (e.g., disjunction, relative time) require only one component event to occur, (b) “binary” operators (e.g., conjunction, sequence) require more than one component event to occur.

For composite events, *event parameter restrictions* may be defined, that establish e.g., whether all component events are signalled for the same object, transaction or user.

Primitive events that occur as the direct consequence of the execution of an application program, of an interactive transaction, or that are time events are called *outside events*. A primitive event that occurs as the consequence of a rule execution is called an *inside* or *rule-signalled event.* Moreover, it is possible that during the execution of an application, the same event definition has occurrences as both, inside and outside events.

When an event occurs, all respective rules are *triggered*. If the *condition* of a triggered rule is fullfilled then its *action* is executed. A condition may be a predicate on the database state or a database query. If the result of evaluation is true or non-empty, respectively, the condition is satisfied. The *action* specifies the reactive behavior of the rule. It may contain data modification and retrieval (in relational DBMS), method invocation (in object-oriented DBMS), transaction operations like commit or abort, etc.

The language used for the specification of conditions and actions is called *host* language. It may be anything from a database language (e.g., SQL for Starburst [19], POSTQUEL for

POSTGRES [16] and Ariel [13]) to a general-purpose programming language (e.g., O++ for Ode[11], Smalltalk for HIPAC [19] or a persistent extension of C++ for SAMOS[10]). Obviously, a rule language improves its expressive power if the used host language tends to a general purpose programming language. That means, not only sequence and selection are supported, but also the programming constructs abstraction and iteration. In this case, a condition may be, besides predicate or query, any procedure or method invocation that returns a boolean value. An action may contain any executable program, including any specific database operations, rule operations (like deletion, modification and enabling/disabling of rules), and so on. Therefore, conditions and actions are more complex than those specified using simple rule languages based on database languages.

Some rule languages also allow the specification of execution constraints (like priority and coupling modes). The priority specifies the execution order of a rule related to other rules triggered simultaneously. Coupling modes are pairs of values (`x`, `y`), where `x`, `y` $\in$ {`immediate`, `deferred`, `decoupled`}. `x` specifies when the condition is evaluated with respect to the transaction in which the event has occurred, and `y` determines when the action is executed with respect to the condition evaluation. For example, `x` = `immediate` means that the rule is processed immediately after the triggering event has been detected, `x` = `deferred` postpones the processing to the end of the triggering transaction (before it is committed), and so on. In order to simplify the explanations, we will use the term “immediate execution” only when both coupling modes are immediate, and “deferred execution” when at least one coupling mode is deferred.

To summarize, the elements that distinguish an expressive rule language from a simple rule language are on one hand constructs like composite operators, rule execution constraints, event parameter restrictions, rule operations, and on the other hand complex conditions and actions. We will see at the end of Section 3 to which extent these elements influence rule execution and its termination.

3 Termination of Rule Execution

Let R be the set of defined ECA-rules. At runtime application programs are sequentially executed until an event is signalled. Control then moves to rule processing. Appropriate rules of R are triggered and inserted into a set, called *conflict set*, containing all triggered and not yet executed rules (see Figure 1, activity ①).

The rule execution component selects one rule at a time (activity ②) from this set in the order required by the *conflict resolution policy.* This strategy for solving the conflict depends on the used active DBMS. If the condition of the selected rule is satisfied, its action is executed. If the execution of the action signals further events, these events are inside ones and the execution is *nested* (activity ③). In this case, further rules are inserted into the set of triggered rules. When the set of triggered rules is empty, control moves back to the application program. Note that we refer to sequential execution only, i.e., at any given time only one rule is chosen and executed. We assume also that all participating rules have immediate execution. However, the basic idea also applies for deferred and decoupled execution.It may happen that the set of triggered rules never empties, for the following reasons:

- concurrent applications cause the steady arrival of outside events and triggered rules are accumulating in the set because rule processing is not fast enough to keep up with demand,
- there are rules in R which trigger each other indefinitely and their execution creates an infinite loop (between activities 1, 2 and 3).

The aim of termination analysis is precisely to avoid the latter case, i.e., infinite nested rule execution. The former case, nontermination caused by constant occurrence of outside events, is irrelevant for termination analysis because it cannot be considered as faulty rule behavior.

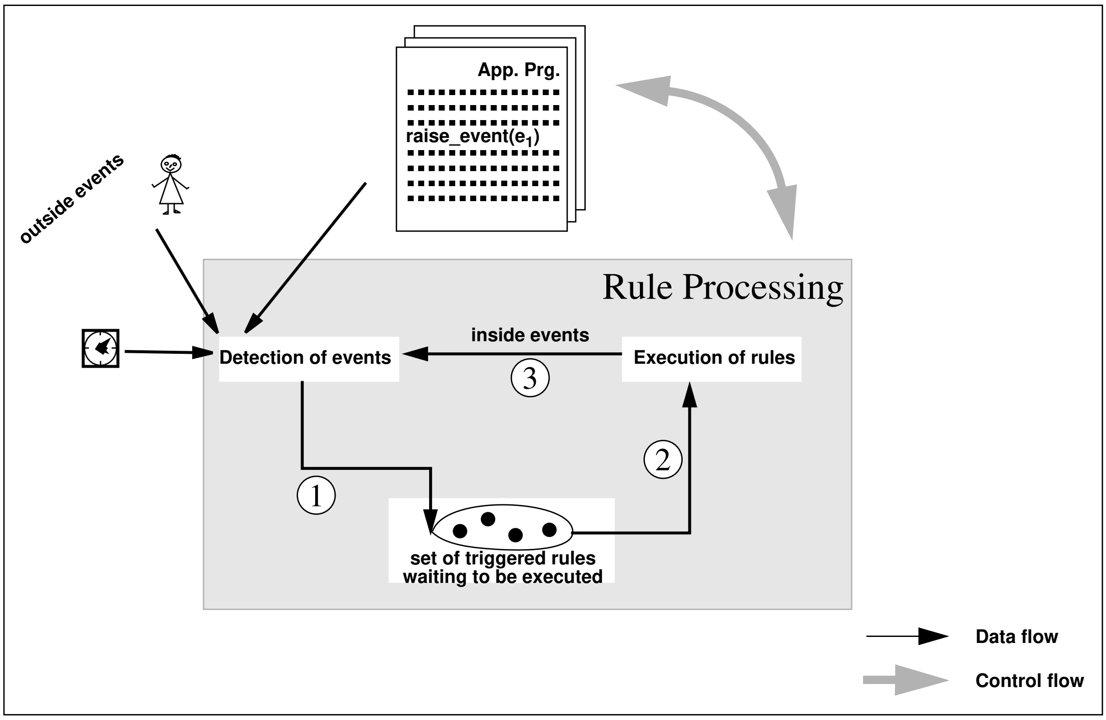

Figure 1: Rule processing in active database systems

Related to the infinite nested rule execution, we have to differentiate between two cases:

1. all involved rules are executed one after the other, without interruption
2. “niches” exist during the infinite nested rule execution that allow also other rules than those involved in the loop to be considered and executed in the meantime.

Note that the conflict resolution strategy plays an important role in the classification above. For example, a *recursive* conflict resolution (i.e., requiring rules triggered by other rules to be first selected for execution) is a precondition for the first case. The second case may be provided by a conflict resolution where rules are triggered in the chronological order of their triggering only. Also rules with different coupling modes may cause "niches" to appear during the infinite nested execution. We will see in the next section their influence on termination.

3.1 Definition of Termination

We call a given finite set of outside events and an initial state of the database an *initialization frame*. The rule execution started by a certain initialization frame is called a *rule execution thread*. We then define termination of rule execution as follows:

> **Definition 1** A rule execution thread *terminates* as soon as the set of triggered rules is either empty or contains only rules triggered by outside events. Generally, *termination of rule execution* means that *every possible* rule execution thread terminates.

Termination of rule execution reached without external intervention is called *normal*. Rule execution can also be enforced to terminate *abnormally*, when the rule execution is explicitly terminated by an application program or by the active DBMS. For example, the active DBMS may support the definition of an upper limit that determines how many (nested) rules are allowed to be executed during rule processing. If during rule execution this limit is reached, rule execution is aborted. Another way to support abnormal termination is through the rules themselves. This is only possible when "niches" exist during the nested rule execution. In this case the execution of further rules than those involved in the loop may lead to an explicit termination. Such rules should for example assure the correct termination and empty the set of triggered rules, ensure the rollback of started transactions, and so on. However, abnormal termination is only an emergency solution that does not guarantee a correct modeling of the active behavior.

In this paper, we focus on termination analysis which must guarantee that rule execution terminates normally. The following section provides an informal basis for our approach. It discusses the aspect of nested rule execution, stressing the role played by events.

3.2 Nested Rule Execution

In order to better understand the causes for infinite nested execution, let us first analyze the finite case.

After completing their execution, rules are inserted in chronological order into the *history of executed rules*. An interval of the history is called a *rule sequence*. It describes the execution of rules over a certain finite time interval.

If r_1, r_2, $r_3 \in \mathsf{R}$ are three rule definitions, let $\rho = (r_1{}^1, r_2{}^1, r_3{}^1, \ldots r_1{}^N, r_2{}^N, r_3{}^N)$ be a rule sequence where $r_i{}^j$ is the j-th execution of rule r_i. If the triggering of one rule depends on the execution of the other two rules (e.g., r_1 triggers r_2, r_2 triggers r_3 and r_3 triggers r_1), then ρ reflects a *nested rule execution*.

Let us now see how the kind of events (primitive or composite) can influence the behavior of a nested execution:

- assume that r_1, r_2 and r_3 are defined only on primitive events and r_1 triggers r_2, r_2 triggers r_3, and r_3 triggers r_1. In this case, the triggering of each rule depends completely on the execution of the previous rule. That means, only r_1^1 is triggered by an outside event in ρ. Thereafter, all rules are triggered by inside events (see Figure 2). Thus, *each rule* triggers (through intermediaries) itself again.

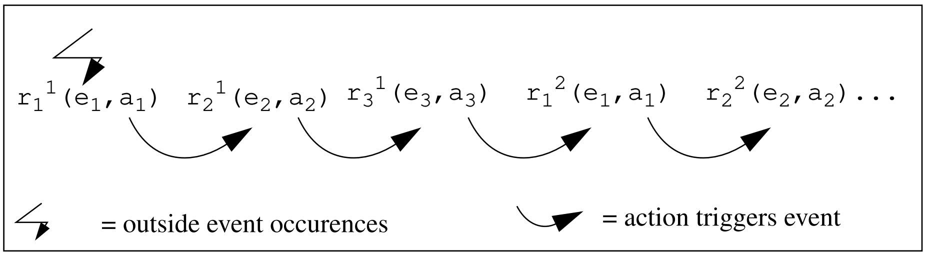

Figure 2: Triggering of primitive events in nested rule execution

- if at least one of the three rules is specified using a composite event, its triggering depends on the occurrence of several primitive events. These primitive events may be generated not only by the immediately preceding rule, but also by some other predecessor rules in the nested execution. In this case, the start of nested execution may depend on several outside events. For example, let

r_1: ON e_1	r_2: ON conj(e_{21}^e_{22})	r_3: ON e_3
IF true	IF true	IF true
DO...	DO...	DO ...
raise_event(e_{21})	raise_event(e_3)	raise_event(e_1)
...	...	raise_event(e_{22})

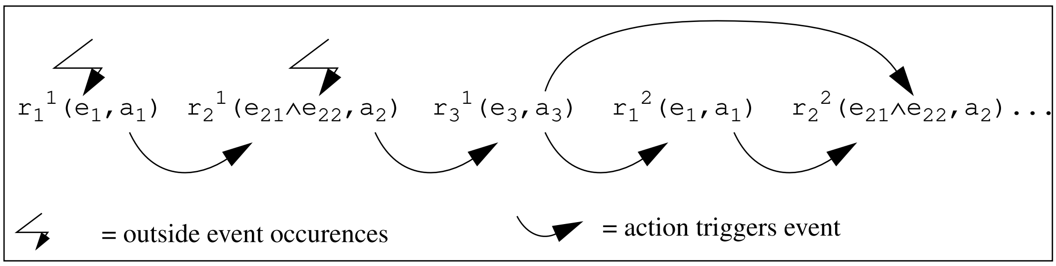

Figure 3: Triggering of composite events in a nested rule execution

Obviously, r_1 and r_3 each trigger a component event of r_2, r_2 triggers r_3 and r_3 triggers r_1. Figure 3 depicts a rule execution scenario. The nested rule execution is started by two outside events, e_1 and e_{22}, that occur in the first execution of the three rules (r_1^1, r_2^1, r_3^1). Beginning with the second execution, $(r_1{}^2, r_2{}^2, ...)$, no other outside occur. All rules are triggered by inside events raised by previous rules. Since each rule of the subset $\{r_1, r_2, r_3\}$ is triggered only if one or some rules in the subset execute, we can say that *the subset* triggers itself and leads to a nested rule execution.

Summarizing, a nested rule execution is determined either by one rule or by a rule subset that triggers itself directly or indirectly. We will use this result in our approach and discuss in Section 4.3 how it can be statically detected if a rule subset *may* trigger itself.

3.3 Termination Analysis for Expressive Languages

In the previous section, we have discussed how rule specification using composite events influences the behavior of nested rule execution. Let us now summarize to what extent this and other specific features of expressive languages influence termination analysis:

- composite events: termination analysis determines whether a rule defined on a composite event may participate in an infinite nested rule execution. In this case, it is important to check whether the composite event is entirely the consequence of the execution of an action, whether it depends on the execution of a certain rule sequence, or whether it depends on one or more occurrences of outside events, and so on.
- complex conditions and actions: termination analysis uses information about actions that may trigger events and that may change the evaluation result of conditions. Providing this information for complex actions and conditions is more difficult, because
  - all possible operations that are used within conditions/actions must be taken into account
  - not only actions but also conditions may trigger events
  - methods are invoked not only directly, but by nested calls as well
  - language constructs like conditional statements, loops and late binding make the decision of the triggering of events (respectively the dependence of condition evaluation on action execution) statically undecidable. In order to perform static analysis hard approximations are required.

  Additional problems arise when the host language has no formal specification (like e.g., C++). In this case, the semantic analysis can be done only superficially.
- time events: they cannot be signalled by rules. Thus, they cannot participate in any infinite nested rule execution. The only exception are relative time events. They depend on a basis event which may be signalled by other rules.

In the following section, we present the underlying concepts of our termination analysis for expressive rule languages. We will focus on how composite events influence the analysis and we will refer only briefly to condition and action analysis.

4 Termination Analysis: Our Approach

Termination analysis determines subsets of R that may lead to potentially nonterminating rule execution, i.e., infinite nested rule execution. To this end, we establish dependencies between rule definitions (called *relationships*) and build a graph that shows possible rule triggering at runtime. Cycles in this graph indicate potential infinite nested rule execution. In order to make the analysis more precise, we extend the analysis with methods to find out whether each cycle in the graph may lead to nontermination or not.

Our notation uses standard logical connectives ($\wedge$ $\vee$, $\Leftrightarrow$, etc.), universal quantifiers ($\exists$, $\forall$), and set operations ($\in$, $\subset$, $\cup$, etc.) with their standard semantics.

4.1 Relationships Between Rules

Termination analysis strongly relies on the recognition of rule *relationships*. By analyzing rule relationships, we can foresee the termination of rule execution. The accuracy of prediction depends on the granularity of the analysis of rule definitions. For example, the more precisely we can decide when one rule may trigger another, the more accurately we can decide if a nested rule execution will really occur at runtime. Rule relationships are defined as follows:

Definition 2 $r_i(e_i, c_i, a_i)$ *is related to* $r_j(e_j, c_j, a_j)$ if one of the following cases applies:

(1) **total AC/E-relationship**: The execution of a_i or the evaluation of c_i potentially signal e_j. ("r_i *may totally trigger* r_j").

(2) **partial AC/E-relationship**: The execution of a_i or the evaluation of c_i potentially signal at least one, but not all component events of e_j. ("r_i *may partially trigger* r_j").

(3) **A/C-relationship**: The execution of a_i potentially modify the truth value of the evaluation result of c_j.

(4) **A/R-relationship**: The execution of a_i enables/disables rule r_j.

We mention that the use of "potentially" is necessary because sometimes it is uncertain if an action or a condition signals an event. For example, it is in general statically undecidable if a method executed within an action and an appropriate method event refer to the same object. It is also uncertain that an event will be signalled if it is raised within a conditional statement.

Let us now discuss the connection between rule relationships and termination analysis. Recall the discussion of Section 3. If each rule in a subset is triggered by the execution of other rules in the subset, then this rule subset leads to a nested execution at runtime. Therefore, it is necessary to find out which rules may totally or partially trigger other rules, i.e., which rules from the rule set are in AC/E-relationships.

For this aim we search information about actions or conditions that may signal primitive events by applying syntax analysis methods to rule definitions. Then we assign to each event e the set $\mathsf{PE}(e)=\{pe_k, k=1,...,n\}$ of primitive events that must occur in order to signal e. In this case, $r_i(e_i,a_i)$ may partially trigger $r_j(e_j,a_j)$, if a_i or c_i may signal at least one element of $\mathsf{PE}(e_j)$, but not all of them. $r_i(e_i,a_i)$ may totally trigger $r_j(e_j,a_j)$ if all the primitive events in the set $\mathsf{PE}(ej)$ may be signalled by a_i or c_i. If e is a primitive event or if it has been built using "unary" composite operators, then $|\mathsf{PE}(e_j)|=1$. If e is a composite event, the computation of $\mathsf{PE}(e_j)$ depends on the semantics of the composite operators. Transformation rules are provided such that each composite event is rewritten in a disjunctive normal form. Then PE is recursively computed by applying the following two laws:

$e_j = e_{j1} \wedge e_{j2} \Rightarrow \mathsf{PE}(e_j) = \mathsf{PE}(e_{j1}) \cup \mathsf{PE}(e_{j2})$

$e_j = e_{j1} \vee e_{j2} \Rightarrow \mathsf{PE}(e_j) = \mathsf{PE}(e_{j1}) \vee \mathsf{PE}(e_{j2})$.

How can A/R- and A/C-relationships influence termination? A nested rule execution is terminated, if either a rule has been disabled by the execution of another rule, or a condition is not satisfied any more as a consequence of the execution of a rule. Let us refer again to the example of Figure 2:

- assume $\exists\ r_i, r_j \in \{r_1, r_2, r_3\}$ in an A/R-relationship such that r_i disables r_j and no other rule exists which enables r_j. Then the three rules are executed only once. Afterwards, the nested rule execution is terminated because r_j cannot be executed any more.
- assume $\exists\ r_i(c_i,a_i), r_j(c_j,a_j) \in \{r_1, r_2, r_3\}$ in an A/C-relationship such that the execution of action a_i sets the condition c_j to false after the rules have been executed N times. For example, a_i deletes objects from a set with N objects and c_j controls if the set is empty. Then, r_j is triggered but cannot be executed the $(N+1)$-st time because its condition is false, and the nested execution is terminated.

  An A/C-relationship between the two rules r_i and r_j is signalled if a_i includes an operation that may change the value of an attribute appearing in c_j. For example, in object-oriented DBMS this implies the need to examine method implementations in order to find out the ones that change attributes either directly or indirectly by nested method execution. However, this technique for detecting A/C-relationships is based on syntactical methods only and is too weak to determine if the execution of a rule action changes the truth value of another rule condition. The rule designer himself has to decide if A/C-relationships can break a nested rule execution.

4.2 The Relationship Graph

Almost all work on termination analysis [1, 2, 3, 18] approaches the termination problem using a directed *triggering graph* for rules specified using simple rule languages. The nodes of the graph represent rules in the rule set. An edge from rule r_i to rule r_j, denoted (r_i, r_j), exists, if r_i may trigger r_j. If the triggering graph has no cycles, the termination of rule execution is guaranteed [1]. A cycle in the graph means that at least one rule may trigger itself and therefore corresponds to a possible nested rule execution at runtime, which in the worst case, may be infinite. In other words, a cycle in the graph indicates possible nontermination of rule execution.

Our approach refines the concept of a triggering graph with a so-called *relationship graph*, a directed, labelled graph (R, Γ) whose nodes represent the rules and whose edges symbolize the AC/E-relationships between the rules. The relationship graph provides two types of edges, (drawn with solid $r_i \longrightarrow r_j$ or dashed $r_i \dashrightarrow r_j$ lines), called total or partial, depending on the sort of relationship they represent. Similar to the classical way to study termination, if the relationship graph has no cycles, the termination of rules in R is *guaranteed.*

We denote with $\Sigma = \{\rho \subseteq (R, \Gamma)$, where $\rho = (r_1, \ldots, r_k, r_1)$ is a cycle$\}$ the set of cycles of a relationship graph. Σ may also contain "false" cycles that must be detected during termination analysis.

Definition 3 A cycle in the relationship graph is called *false* if the subset of rules participating in it can never lead to infinite nested execution.

Recalling the discussion in Section 4.1, a cycle ρ is false if two rules $r_i(e_i, a_i)$, $r_j(e_j, a_j)$ exist such that:

- r_i disables r_j and no other rule in ρ enables r_j again
- r_i is in A/C-relationship with r_j and the rule designer finds arguments to prove that r_i eventually changes the truth value of r_j to false and no other A/C-relationship exists that could again change the truth value of the evaluation of c_j. A/C-relationships are searched not only for rules participating in ρ, but also for rules that may be triggered by the execution of rules from ρ. The reason is that the execution of these rules outside ρ could change the database state and influence the evaluation of conditions in ρ.

Let us now focus on another reason why a cycle $\rho \in \Sigma$ may be "false". Recall from Section 3.2 that a nested execution is determined by a set of rules that triggers itself. Then, a detected cycle ρ is "false" if the set of rules participating in ρ cannot trigger itself.

Definition 4 Let a set $C \subseteq R$ and a rule $r(e, a)$ such that $PE(e) = \{pe_j, j=1,\ldots,k\}$.

- C *may directly trigger* $r(e, a)$ if $\exists\{r_j(e_j, a_j)\}_{j=1,\ldots,k} \subseteq C$, such that $\forall j$, a_j or c_i may signal pe_j
- C *may indirectly trigger* $r(e, a)$ if $\exists\{r_j(e_j, a_j)\}_{j=1,\ldots,k} \subseteq R$, such that $\forall j$, C *may trigger* $r_j(e_j, a_j)$ and a_j may signal pe_j

Obviously, a set C may also trigger r if a single $r_j(e_j, a_j) \in C$ exists such that a_i or c_i may signal e.

Definition 5 Let a set $C \subseteq R$ and a rule `r(e,a)`.

- C *may trigger* `r(e,a)` $\Leftrightarrow$ C *may directly or indirectly trigger* `r(e,a)`
- C *may trigger itself* $\Leftrightarrow \forall$ `r` $\in C$, C *may trigger* `r`.

We focus in the following on the detection of cycles that cannot trigger themselves and therefore cannot lead to a nested rule execution.

4.3 Detecting a Rule Subset not Triggering Itself

Let $\rho \in \Sigma$ be a cycle in a relationship graph and C the set of rules in ρ. If a rule $r_j \in C$ can be found such that C cannot trigger r_j, then ρ is a false cycle. In order to find such a rule, we investigate the rules in C which are terminal nodes for partial edges. Let $r_j(e_j, a_j)$ be a rule such that (r_{j-1}, r_j) is a partial edge in ρ. This means that the execution of $r_{j-1} \in C$ may only signal one or some primitive events $(pe_i)_{i=1,\dots,l} \in PE(e_j)$. We must investigate whether the other primitive events in $PE(e_j) \setminus \{(pe_i)_{i=1,\dots,l}\}$ can be signalled directly or indirectly by rules of C. This can be done by traversing the relationship graph in a way similar to a "depth-first" search, but against the edges' orientation, beginning with r_j as root. The algorithm terminates when the search reaches either nodes from C, or the event of the last visited rule/node is composed of at least one outside event, i.e an event that cannot be signalled by any other rule. In the latter case, the conclusion is that C cannot trigger the rule r_i; that means C cannot trigger itself and the cycle is proven to be false. If, on the contrary, all rules in C may be triggered by C, then nested execution cannot be precluded.

The Algorithm

In the following we sketch the algorithm for investigating whether a set of rules C may trigger a rule r_j in C.

```
MayTrigger(C, rj(ej, aj)): BOOLEAN{
IF rj ∈ C THEN RETURN(TRUE)
FOREACH pei ∈ PE(ej){
    IF pei is outside event THEN RETURN(FALSE)
    result = TRUE
    FOREACH rj-1(ej-1, aj-1) such that aj-1 may signal pei{
        flag=RETURN(MayTrigger(C, rj-1(ej-1, aj-1))
        result = result ∧ flag
        }
    }
RETURN(result)
}
```

If C may trigger itself and all rules in C are defined for primitive events (i.e., $\forall\, r_i\ (e_i, a_i)$, $|PE(e_i)| = 1$), then only rules in C determine the nested rule execution. If there are rules in

C defined on composite events, then there exist at least another cycle in the graph which has common nodes and edges with C.

4.4 A Tool for Termination Analysis

We are now putting together all pieces introduced so far and summarize our approach to termination analysis (Figure 4). We assume that a set of rules specified using an expressive language is given. We propose a termination analysis tool that investigates rule definitions (at compile time). The rule designer uses the results and decides if the rule set must be changed.

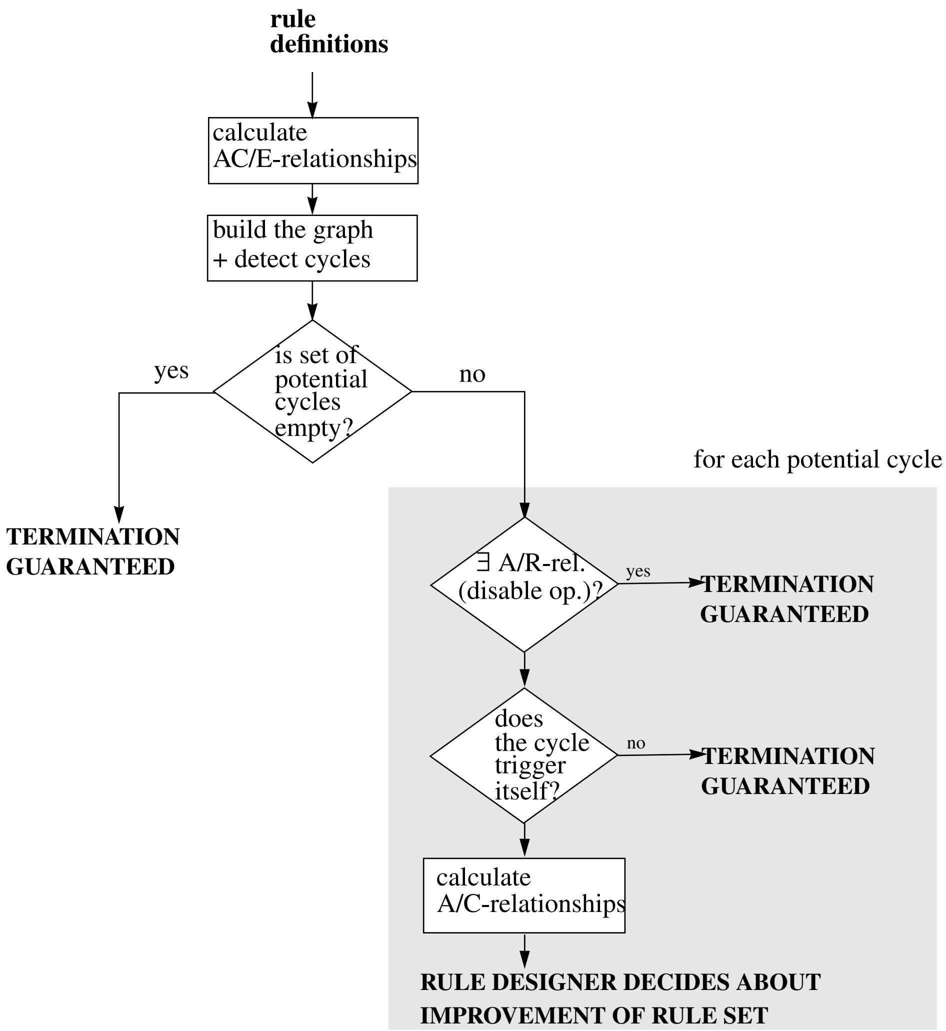

Figure 4: An overview of termination analysis

The tool has been implemented for the active database system SAMOS. It works as follows. First of all, the AC/E-relationships must be computed. Then the relationship graph is built. For each detected potential cycle, the tool searches for reasons that could guarantee that the nested rule execution cannot be infinite. Therefore, the cycles are first investigated to determine if they contain rule pairs that can disable each other (A/R-relationships). Then, for each cycle it is investigated if the composite events of the rules in the subset allow the subset to trigger itself. The A/C-relationships between rules are calculated for each of the remaining cycles.

Finally, the tool visualizes potential cycles which could not be proven to be false and displays the A/C-relationships between rules participating in these cycles. Based on this information, the rule designer checks whether eventually the action of one rule in the cycle may change the truth value of the condition of another rule to false. To achieve this, he has to inspect the code and completely understand the semantics of the involved rules. For those cycles that cannot be proven to be false, he must change the definitions of the rules and repeat the termination analysis. This process is iterated until no further cycles are found.

4.5 Example

To illustrate our approach we use the well known "IFIP case" [17] that deals with the organization of conferences. Figure 5 describes a part of a workflow which models administrative tasks performed with regard to paper submission, distribution of papers to the referees, deciding the admission of papers, etc. Some of the operations must be performed by human beings (author, conference chairman, referee).

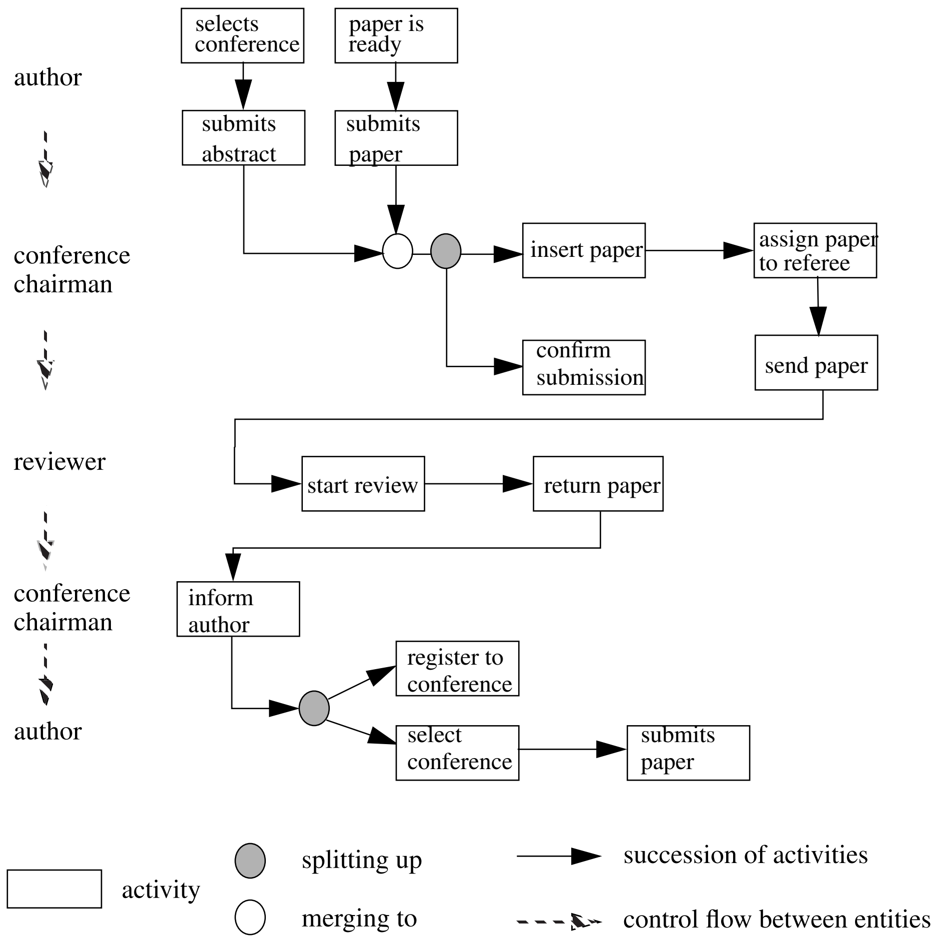

Figure 5: Workflow of the “IFIP case”

When authors of a paper select a conference, and its deadline has not already passed, an abstract of the paper is submitted to this conference. When the authors finish the paper and the number of words is less than 6000, the entire paper is submitted. Some time later the submission is confirmed to the first author of the paper.

When the abstract and the paper have been submitted, and the topic of the paper matches the topics of the conference, the paper is inserted into the list of submitted papers. Each submitted paper must be assigned to certain referees that must review it. One of the conditions is that no referee is author of a paper he should review. When referees are established for a paper, the paper is send to them. If referees do not return the paper in a certain time interval, they will be reminded. When all referees of a paper return their review, their responses are evaluated and the authors are informed. If the paper is accepted, its authors may register to the conference. When a paper is rejected, the authors will probably select another conference and send the paper to this new conference.

There are four entities (paper, author, conference chairman, referee) that take part in the workflow. The left side of Figure 5 illustrates which entity performs certain activities. Information about the four entities is stored in a database system. The `paper` is characterized by a `name`, `number_of_words`, `topic`, a set of `authors`, a set of `assigned_referees` that must review the paper and a `status` (i.e., accepted or rejected). The `conference` includes as attributes a list of `topics`, a boolean `deadline_occurred`, and a list of `submitted_papers`. Instances of `conference` are saved in `all_Conf`. The `author` and the `referee` are specializations of the type `person`. `Referee` contains also an attribute specifying whether the reviewed papers have been returned to the organizers of the conference or not.

Obviously, if an active database system is used, the workflow activities may be controlled and coordinated using ECA-rules. For the specification of events, composite operators are used. We use the syntax of the SAMOS rule language. A sequence is denoted as $(e_1; e_2)$, a relative time as `e + t`, where `t` is the time unit and `e` the basis event. An event defined with a *monitoring interval* `e IN [a, b]` is signalled only if `e` occurs in the interval `[a, b]`. The *negative event* `NOT e IN [a, b]` is signalled at the end of the interval if `e` has not occurred.

The actions of the rules contain various operations which may again signal events. The simplified notation `RAISE(operation)` reflects both the execution of an operation and the signalling of the event defined on it.

```
ON    new conference                                    r0
DO    insert_conference_in all_Conf

ON    selects_conference                                r1
IF    deadline did not occur
```

```
DO    RAISE (submits_abstract)


ON    paper_is_ready                                       r2
IF    number_of_words < 6000
DO    RAISE (submits_paper)

ON    (submits_abstract; submits_paper) + t                r3
DO    retrieve_first_author_of_paper
      confirms_submission

ON    (submits_abstract; submits-paper)                    r4
IF    topic_of_paper ∈ topics of conference
DO    inserts paper into set_of_submitted_papers
      RAISE (assigns_paper_to_referees)

ON    assigns_paper_to_referees                            r5
IF    authors ∩ assigned referees ≠ 0
DO    RAISE (assigns_paper_to_referees)

ON    assigns_paper_to_referees                            r6
IF    authors ∩ assigned referees = 0
DO    foreach referee of paper
      RAISE (send_paper_to_referee)

ON    send_paper_to_referee                                r7
DO    RAISE (start_review)

ON    start_review + t                                     r8
IF    review ready
DO    RAISE (return_paper)

ON    NOT return_paper IN [send_paper, send_paper + t]     r9
DO    warning message

ON    return_paper                                         r10
IF    all referees returned paper
DO    RAISE (inform_authors)

ON    inform_authors                                       r11
IF    paper.status = rejected
DO    authors select conference
      RAISE (submits_paper)


ON    inform_authors                                       r12
IF    paper.status = accepted
DO    author_register_to_conference
```

Consider the set of rules $\mathsf{R} = \{r_0, ..., r_{12}\}$. Let us see how the termination tool works and where the rule designer must intervene. First of all, the relationship graph (see Figure 6) is built.

There are 3 cycles in the relationship graph, caused by the following subsets of rules:

R_1={r_1, r_4, r_6, r_7, r_8, r_{10}, r_{11}}

R_2={r_4, r_6, r_7, r_8, r_{10}, r_{11}}

R_3={r_5}

In the next step, the termination tool considers each cycle separately. Cycle R_1 does not contain any A/R-relationships. The question is whether the rule subset R_1 may trigger itself. The analysis begins with the terminal node r_4 for the partial edge (r_1, r_4). The set $\mathsf{PE}(e_4)$ must be computed, $\mathsf{PE}(e_4)$ = {`submits_abstract`, `submits_paper`}. All elements of $\mathsf{PE}(e_4)$ are determined by the execution of rules in R_1 (r_1 in R_1 may signal `submits_abstract` and r_{11} in R_1 may signal `submits_paper`). That means, r_4 is triggered by rules in R_1. Because there are no other partial edges in the cycle, it is concluded that R_1 may trigger itself. The cycle could not be proven to be false. The same reasoning is valid for the second cycle, R_2. That means, the set $\mathsf{R}_1 \cup \mathsf{R}_2$ may lead to a nested rule execution.

Even if A/C-relationships are signalled by the tool (see Figure 6), we cannot show that nested rule execution does not take place. Therefore, some rule definitions must be changed. One possibility is to change the event that rule r_4 is defined on:

```
ON    ((submits_abstract; submits_paper); deadline)
```

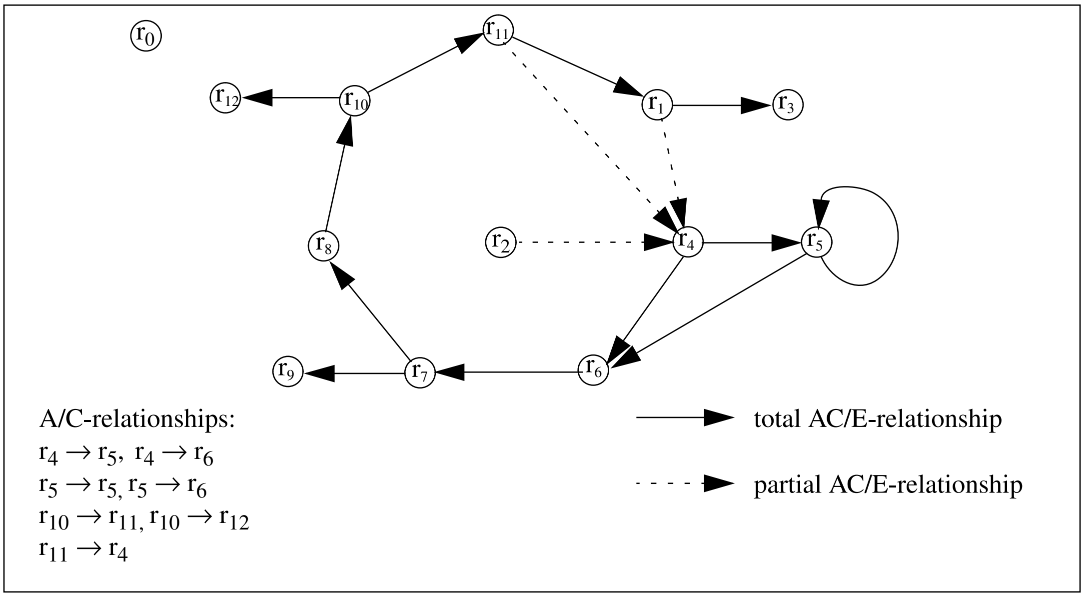

Figure 6: Relationship Graph

The new semantics of e_4 requires another primitive event, `deadline` to occur in order to trigger r_4. After this modification the analysis is started again. `Deadline` is an outside event. Applying again the algorithm for determining if R_1 (respectively R_2) may trigger itself, the answer is negative. Not all events in $\mathsf{PE}(e_4)$ = {`submits_abstract`, `submits_paper`, `dead-line`} are signalled by rules in R_1 (respectively R_2). In other words, by changing the rule r_4, a nested execution will never take place, in the worst case only a cyclic rule execution may occur (i.e., rules are repeatedly in the same order executed without being triggered by other rules). If the rule designer decides that even a cyclic rule execution should not be allowed, he can modify the rule set again, by changing the condition of r_{11}:

```
IF    paper.status less than 4 times rejected
```

and by adding a new rule

```
ON    inform_acceptance_decision
IF    paper.status 5 times rejected
DO    advise to write a new paper or to give up
```

In this case, it is ensured that if a paper has been rejected 5 times, the cyclic execution is terminated.

Let us now discuss the cycle determined by r_5. The rule designer considers the A/C-relationships signalled by the tool that include r_5. r_5 is in A/C-relationship with itself. r_5 imposes that each referee of a paper is not among the authors of this paper. The execution of the rule should be repeated as long as its condition is fulfilled. The rule designer then has to decide whether the self-triggering of r_5 will eventually terminate when the condition is not fulfilled anymore. For example, if there are enough referees then it is sure that the operation `assign_paper_to_referee` will find a set of `assigned referees` for each paper such that `authors ∩ assigned referees = 0`.

5 Discussion

Much work has been done in the area of termination investigation, covering static methods analyzing the specification of rules [1, 2, 3, 18, 20] (at compile time) and dynamic methods analyzing rule execution [4] (at runtime). However, no method handle termination analysis for rules specified using composite events, although nowadays most advanced active database systems [5, 6, 7, 8, 10, 11, 14, 19] supports an event algebra. One main contribution of our approach is the consideration of composite events. The basic idea of using a graph for termination analysis with rules as nodes is the same as in [1, 2, 3, 20]. We have extended it to make it applicable for rules defined for composite events as well.

Another contribution is that we presented the whole process of termination analysis and considered all necessary steps (computing AC/E-relationships, building the relationship graph, eliminating false cycles, computing A/C-relationships) in order to obtain a reliable termination analysis. Some other work only refers to parts of the process: [1] focuses on our so called AC/E-relationships and the building of a relationship graph, while [2] focuses on the semantic information contained in the actions and conditions (our A/C-relationships). [3] and [20] consider both, AC/E- and A/C-relationships. We mention that in [3] A/C-relationships are computed for all rules in the rule set. We computed them in our approach only for some rules (for each cycle it is sufficient to consider only the rules in the cycle and the rules that may be triggered by the execution of the rules in the cycle).

All known static methods are conservative, that means they aim at fulfilling sufficient (but not necessary) conditions. Our approach also performs approximations that are always on the safe side with regard to guaranteeing termination:

- When no safe assertions can be made, AC/E-relationships between rules are assumed to exist.
- Termination of rule execution is guaranteed only in the cases where rigorous conditions are fulfilled (e.g., the relationship graph has no cycles, the cycle does not trigger itself). Otherwise nontermination cannot be automatically precluded.

However, the precision of termination analysis directly depends on the host DML used for the specification of actions and conditions:

- If the host language supports conditional statements and while statements, AC/E-relationships cannot be accurately provided because it cannot be definitely decided if an event appearing within such a statement will be raised at runtime. Additional problems occur in systems where the host language supports late binding and pointers. We encountered these problems in our implementation for SAMOS which has as host language a persistent C++.
- If the host language supports the explicit specification of the type of operations that the events and the actions are defined upon (e.g., insert, delete, increase, decrease, retrieve, update), computing AC/E-and A/C-relationships is more precise. This applies for [20]. Due to the language of the OSCAR Trigger System, the analysis can avoid cases when set of operations or set of objects are disjoint.
- If the host language provides a formal language definition, it may be more precisely established whether an action may change the truth value of a condition. For example, [2] assumes that conditions and actions are both represented by relational algebra expressions. In this case, formal considerations are applied and the approach may accurately detect A/C-relationships. Unfortunately, these ideas cannot be directly used for host languages that are not based on relational algebra or relational calculus. We used in our implementation a coarse syntactical method for computing A/C-relationships because C++ does not provide formal language definition. However, we plan to investigate if, using an object-oriented algebra [12], the results of [2] may be adjusted to our case as well.

For runtime analysis, [4] proposes a technique to recognize that a given situation (i.e., database state and rule to be triggered) has already occurred and therefore will occur an infinite number of times in the future. However, for general rule sets, if the rule execution does not terminate, this does not necessarily mean that situations should repeat. Therefore, this method works only for rule sets which do not generate new values in the database or which do not include rules that are nondeterministically selected by the rule processing.

6 Summary

This paper introduces concepts needed for a better understanding of the problem of rule termination in active DBMS, and describes our approach for handling termination analysis for expressive rule languages. The expressiveness of a rule language depends on:

1. the number of constructs provide by the rule language (composite operators, coupling modes, etc.)
2. the host language used for the specification of actions and conditions. If the host language tends to a general-purpose language, the flexibility of specification increases.

We referred in this paper mainly to the aspect of composite operators and proposed a solution for deciding if rules defined for composite events may lead to nontermination. Termination of rule execution is guaranteed if *no rule subset* may trigger itself directly or indirectly. A rule subset may trigger itself if *all* the primitive component events of the rules may be triggered by the execution of other rules in the subset. Using these ideas, we extended classical conservative approaches [1, 3] to be also applicable for the termination analysis of expressive rule languages.

We also discussed problems raised by host languages that tend to provide the same expressive power as general-purpose languages. They are hard to analyze statically and the rule analysis lacks precision.

Summarizing, more expressiveness for rule specification must be paid for with a more conservative termination analysis. However, if the rule designer chooses the use of an expressive language, the basic idea of our approach regarding composite events is indispensable for the investigation of termination. To our best knowledge, it is the only approach so far that eliminates the signalling of "false" cases of nonterminating rule execution due to composite events.

7 References

1 Aiken A., Widom J., Hellerstein J.M.; Behaviour of Database Production Rules: Termination, Confluence and Observable Determinism; *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, San Diego, June 92.

2 Baralis E., Widom J.; An Algebraic Approach to Rule Analysis in Expert Database Systems; *Proc. of the 20th VLDB Conf.*, Santiago, Chile 94.

3 Baralis E., Ceri S., Paraboschi S.; Improved Rule Analysis by Means of Triggering and Activation Graphs; *Proc. of 2nd Intl. Workshop on Rules in Database Systems*, RIDS '95, Athens, Greece, September 95.

4 Baralis E., Ceri S., Paraboschi S.; Run-Time Detection of Non-Terminating Active Rule Systems; *Proc. of the 4th Intl. Conf. on Deductive and Object-Oriented Databases*, DOOD'95, Singapore,

December 95.

5 Berndtsson M., Lings B.; On Developing Reactive Object-Oriented Databases; *IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Database Research*, January 93.

6. Buchmann A.P., Blakeley J., Zimmermann J.A., Wells D.L.: Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions; *Proc. of the 11th Intl. Conf. on Data Engineering*, Taipei, Taiwan, March 95.

7. Chakravarthy S., Krishnaprasad V., Tamizuddin Z., Badani R.H.; ECA Rule Integration into an OODBMS: Architecture and Implementation; *Proc. of the 11th Intl. Conf. on Data Engineering*, Taipei, Taiwan, March 95.

8 Collet C., Coupaye T., Svensen T.; NAOS: Efficient and Modular Reactive Capabilities in an Object-Oriented Database System; *Proc. of the 20th VLDB Conf.*, Santiago, Chile, September 94.

9 Dayal U.; Active Database Management Systems; *Proc. of the 3rd Intl. Conf. on Data and Knowledge Bases*, Jerusalem, 88.

10 Gatziu S., Dittrich K.R; Events in an Active Object-Oriented Database System; *Proc. of the 1st Intl. Workshop on Rules in Database Systems*, Edinburgh, August 93.

11 Gehani N.H., Jagadish H.V.; Ode as an Active Database: Constraints and Triggers; *Proc. 17th Intl. Conf. on Very Large Data Bases*, Barcelona, September 91.

12 Geppert A., Dittrich K.R., Goebel V., Scherrer S.; The NO^2 Data Model; *Technical report 93.09*, Institut für Informatik, Universität Zürich, April 93.

13 Hanson E.; The Design and Implementation of the Ariel Active Database Rule System; *IEEE Transactions on Knowledge and Data Engineering*, 8(1), February 96.

14 Kappel G., Rausch-Schott S., Retschitzegger W., Viewweg S.; TrigGS: Making a Passive Object-Oriented Database System Active; *Journal of Object-Oriented Programming* 7:4, July 94.

15 Simon E., Kotz-Dittrich A.: Promises and Realities of Active Database Systems; *Proc. of the 21th VLDB Conf.*, Zuerich, Switzerland, September 95.

16 Stonebraker M.L., Kemnithz G.; The POSTGRES Next-Generation Database Management System; *Communications of the ACM,* 34(10), October 91.

17 Van Assche F., Moulin B., Rolland C.; Object Oriented Approach in Information Systems; *Proc. of the IFIP TC8/WG8.1 Conf. on the Object Oriented Approach in Information Systems*, Quebec City, Canada, October 91.

18 Van der Voort L., Sibes A.; Termination and Confluence of Rule Execution; *Proc. of the 2nd Intl. Conf. on Information and Knowledge Management*, Washington DC, November 93.

19 Widom J., Ceri S.; *Active Database Systems: Triggers and Rules for Advanced Database Processing*; Morgan-Kaufmann, San Francisco, California, 96.

20 Weik, T., Heuer, A.; An Algorithm for the Analysis of Termination of Large Trigger Sets in an OODBMS; *Proceedings of the 1st Intl. Workshop on Active and Real-Time Database Systems,* Skoevde, Sweden, June 95.