Unrolling Cycle to Decide Trigger Termination

Sin Yeung LEE & Tok Wang LING
School of Computing,
National University of Singapore,
Lower Kent Ridge, Singapore 0511, Singapore.
email : jlee@comp.nus.edu.sg, lingtw@comp.nus.edu.sg

Abstract

Active databases have gained a substantial interest in recent years in enforcing database integrity, however, its current implementations suffer many problems such as running into an infinite loop. While deciding termination is an undecidable task, several works have been proposed to prove termination under certain situations. However, most of these algorithms cannot conclude termination if a cyclic execution actually presents during run-time. This is rather limited. The trigger system can still terminate if these cycles can only be executed a finite number of times. Adopting the trigger graph approach, we propose a method to detect if some cycles can only be executed finitely. We then present a cycle-unrolling algorithm to remove those cycles that can only be executed finitely from a trigger graph. Similarly, we present the concept of finitely-updatable predicate to further improve most existing detection methods. Finally, we conclude with an algorithm to detect if a given trigger system will terminate.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

1 Introduction

Recently, there is an increasing interest in providing rule processing to database systems so that they are capable of automatic updating as well as enforcing database integrity. One of the most popular approaches is to make use of the ECA (Event-Condition-Action) rules [6, 7, 9, 8, 14]. In this active database model, whenever an event occurs, trigger rules with the matching event specifications are triggered, and their associated conditions are checked. For each trigger rule, if its condition is satisfied, then the associated actions of that trigger rule will be executed. These actions may in turn trigger other rules.

While active database systems are very powerful, the way to specify rules is unstructured and its processing is difficult to predict. This is especially true when the action of an ECA rule can falsify the condition of another ECA rule and thus deactivates the second trigger rule. Clearly, the final database state can be highly dependent on the execution order of the set of trigger rules. Furthermore, the set of rules may be triggering each other indefinitely, thus preventing the system from terminating. To illustrate, suppose rule 1 is to increase the salary of an employee if his allowance is increased; while rule 2 is to increase the allowance of an employee from the accounts department if his salary is increased. Intuitively, this pair of rules can reactivate each other ad infinitum if we update the salary of an employee from the accounts department. Such an execution will not terminate.

This non-termination problem makes developing even a small application system a difficult task. Hence, the rule programmer must perform some analysis on the set of trigger rules to predict its behaviour in advance. Although it is undecidable whether the execution of any given set of trigger rules will finally terminate, it is beneficial to have tests to detect the subset of trigger rules which are terminating and those which may not terminate. This can assist the rule programmers so that they need only to verify a smaller subset of the trigger rules.

There are recent works on this termination problem. [1] proposed a method in the context of the Starburst Rule System to detect some definite terminating conditions. This approach makes use of a graph called directed triggering graph proposed in [5]. If the triggering graph constructed has no cycle, then the trigger system will always terminate. This approach is simple but rejects too many terminating situations. In particular, it does not make use of the condition-parts of the trigger rules. As pointed out by later works, many obviously terminating situations cannot be detected.

In [17], the authors suggested some sufficient conditions for trigger termination. However, the method only works for a rather restricted trigger system. For each ECA rule, the condition is a simple query and the action is a simple and single attribute assignment. Deletions and insertions are not considered. This is one of the major drawbacks. [10] proposed a rather different approach. The Event-Condition-Action rules are first reduced to term rewriting systems, then some known analysis techniques for termination are applied. However, it is unclear whether a general trigger database system can always be expressed as a term rewriting system. [4] used an algebraic approach to attack this termination problem for expert database systems. Although the method offers a much stronger solution than [1], the analysis does not involve much on trigger conditions. Later works such as [15, 2, 16, 11] make us of the trigger conditions. To decide if one rule *r* can actually trigger another rule *s*, [11] constructs a conjunct based on the trigger conditions of the two rules. If the conjunct is not satisfiable, then the edge between the two rules is removed. On the other hand, [2, 3] augments a trigger graph with an activation graph. An edge is removed unless it is in a cycle and can be re-activated after a self-deactivation [2, 3]. All these analysis are still based on edges in the trigger graph, [13] remedied this problem by considering trigger paths instead of trigger edges so that more terminating situations can now be detected.

However, all these methods suffer from one common drawback — if the trigger system, indeed, has a cyclic execution, none of the previous methods can conclude termination. This is obviously limited. For example, consider the following trigger rules:

Rule 1 : ON insert $a(X,Y)$ IF $(X=1)$
DO insert $a(Y,0)$

Clearly, it is possible that during run-time, rule 1 successfully activates itself several times. Upon closer examination, we see that within this trigger session, after the first insert of the tuple $a(X,Y)$, the system may trigger an insertion of $a(Y,0)$. Thereafter, the insertion of $a(0,0)$ may be triggered. Upon the insertion of $a(0,0)$, the trigger condition is definitely unsatisfied, hence, this trigger session can at most trigger the same rule for three times only. In other words, although the rule causes cyclic execution, the system can still be proven to terminate.

In this paper, we will discuss in more detail the termination problem and propose a method to detect more situations where a given trigger system can be proven to terminate. In our approach, we first translate a trigger system into a graph called *trigger graph* as proposed in [5]. We then present an algorithm to construct a condition to verify if a cycle can only be cycled at most a given finite number. Finally, a graph unrolling method is proposed to remove a given cycle from a given trigger graph. If our method answers affirmatively that the given system always terminates, then the given system can indeed terminate.

2 Trigger Architecture

We assume a general abstract architecture of the underlying active databases that does not depend on any particular architecture. The underlying database can be an active OODB, or just a simple RDB. Each trigger rule takes the following form:

$rule\_name :: event$ IF $condition$
DO $action_1, \ldots, action_n$

where *rule_name* is the name of the trigger rule. The *event* and $action_i$ in the trigger rules are abstracted to take the following form,

$event\_name(variable\_list)$

event_name can refer a simple update such as *incr_salary*, or a complex action such as *cascade_delete*. In this paper, we shall use the names $e1$, $e2$, etc in most cases to refer to general events.

condition is a conjunction of positive literals, negative literals and/or evaluable predicates. Any variable which appears in the variable list of any $action_i$ must either appear in the *event* or in the *condition*. Furthermore, in order that the evaluation of *condition* is safe, any variable that appears in any negative literal or an evaluable predicate in *condition* should also appear in either *event* or in a positive literal in *condition*. We shall call those variables that appear in *condition* but not in *event* as *local variables*. Throughout this paper, we will also refer to *event* as *rule_name.*EVENT and *condition* as *rule_name.*COND .

Example 2.1 The following specifies that an increase in an employee's salary should also trigger an increase in the salary of his/her manager by the same amount:

$$\begin{aligned}&incr\_salary\_rule :: incr\_salary(EmpID, IncrAmt)\\ &\quad \text{IF } emp(EmpID, MgrID, Salary) \text{ DO}\\ &\qquad incr\_salary(MgrID, IncrAmt)\end{aligned} \qquad \square$$

As our proposed method is independent of the rule execution order, we do not need to assume any specific model of execution. Our method is equally applicable to trigger systems with rule priority and to those with deferred execution.

3 Activation Formula

[11] introduces the trigger formula, which is a more refined condition to decide if a rule r_1 can trigger another rule r_2. The trigger formula is constructed by conjuncting the trigger conditions of trigger rules r_1 and r_2 in a selective way. If the trigger formula is unsatisfiable, then the corresponding trigger edge between r_1 and r_2 in the trigger graph is removed. If the final graph is acyclic, then the trigger system will terminate. For example, given the following two trigger rules:

$\mathtt{r1} :: e1(X) \text{ IF } (X>1) \text{ DO } e2(X)$
$\mathtt{r2} :: e2(X) \text{ IF } (X<1) \text{ DO } e1(X)$

[11] constructs the trigger formulae as,

$(X>1) \wedge (X<1)$

This formula is unsatisfiable, therefore, the edge $\ll \mathtt{r1}, \mathtt{r2} \gg$ is removed from the trigger graph. Since the resultant graph is acyclic, this trigger system will terminate.

A direct generalization is to consider more than one edge by conjuncting more conditions together. If the resultant condition is a contradiction, then the sequence of rules cannot be executed at all. This guarantees termination. For example,

$\mathtt{r1} :: e1(X) \text{ IF } (X>1) \text{ DO } e2(X)$
$\mathtt{r2} :: e2(X) \text{ IF } (X>5) \text{ DO } e3(X)$
$\mathtt{r3} :: e3(X) \text{ IF } (X<1) \text{ DO } e1(X)$

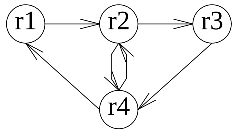
The conjunction of conditions of the rules from the execution sequence $\ll \mathtt{r1}, \mathtt{r2}, \mathtt{r3} \gg$ is

$(X>1) \wedge (X>5) \wedge (X<1)$

which is clearly unsatisfiable. Hence, r1 cannot trigger r3 via r2 during run time.

However, this generalization of conjuncting conditions is far from trivial, there are two major considerations to be handled:

1. Conflict of variables.
   The scope of the variables used in a trigger condition is confined to the rule itself. When the conditions from several trigger rules are conjuncted together, conflict of variables may occur. For example, the variable X of the following two rules belongs to different scope:
   $\mathtt{r1} :: e1(X) \text{ IF } (X>1) \wedge p(X,Y) \text{ DO } e2(Y)$
   $\mathtt{r2} :: e2(X) \text{ IF } (X<1) \text{ DO } e1(X)$
   Hence, we cannot directly conjunct the trigger conditions of these two rules together. The solution adopted by [11] requires that each variable used in each trigger rule has a different name. This solution has an obvious problem. The variable conflict still occurs if the condition from the same rule appears at least twice in the conjunction. For example, consider another trigger rule:
   $\mathtt{r3} :: e3(X,Y) \text{ IF } (X>Y) \text{ DO } e3(Y,X)$
   A simple conjunction of r3 with r3 gives a wrong formula:
   $(X>Y) \wedge (X>Y)$
2. Process to eliminate trigger edges.
   Even if we can prove that a sequence of rules cannot be executed, it is possible that no edge can be removed from the trigger graph. For example, consider the following graph, even if we prove that rule r1 cannot trigger r4 via r2, no edge can be immediately removed from the trigger graph without destroying other cycle. In Section 4, we propose a method to eliminate a path instead of an edge from a trigger graph.



3.1 Predicate Selection Procedure

Our method investigates the conjunction of the trigger conditions in each trigger rule in a given execution sequence. However, as shown in [11], we cannot include every predicate in the trigger conditions, we need a *predicate selection procedure* to select the correct predicates to be included in the conjunction of the trigger conditions. We propose two possible predicate selection procedures:

1. Non-Updatable predicate selection procedure
2. Finitely updatable predicate selection procedure

The first selection procedure is a simplified version of the second selection procedure, and is used for the ease of discussion. The second procedure — finitely updatable predicate selection procedure, will be used in the final termination decision algorithm. Section 5 will discuss the algorithm making use of the latter selection procedure. The two selection procedures are further elaborated as follows:

3.1.1 Non-Updatable predicate procedure

This selection procedure only selects predicates that cannot be updated by any trigger execution. Non-updatable predicates fall into the following categories:

1. Evaluable function
2. Predicate/attribute that is not modified, whether directly or indirectly, by any action of some trigger rules.

Example 3.1 Given a bank database which contains the following relations:

1. $acc(Acc\#, Owner, Bal)$
   Acc# is the account number owned by *Owner* with amount *Bal*.
2. $bankcard(Card\#, Acc\#)$
   The bankcard *Card#* is associated with the account *Acc#*. One account can have many bankcards, but each bankcard can only be associated with one account.

In this database, two trigger rules are specified,

```
// If a bankcard is lost,
// debit into the owner's account
// a service charge of 10 dollars.
    r1 :: replace_lost_card(Card#)
           IF bankcard(Card#,Acc#)
               DO debit(Acc#,10)
// If an account has insufficient fund
// during debit, alert the owner,
// but allow overdraft.
    r2 :: debit(Acc#,Amt) IF acc(Acc#,Owner,Bal)
          ∧ (Bal < Amt) DO
              alert(Owner,Acc#,'Overdraft')
```

The trigger events update the databases in the following ways:

1. The event *replace_lost_card* will issue a new bankcard to its owner. It will not update the relations *acc* and *bankcard*.
2. The event *debit* is to update the *Bal* of *Owner*'s account in the relation *acc*.
3. The event *alert* does not update the database.

Note that all these information can be easily extracted from the SQL implementation.

Now consider the following formula,

$$bankcard(Card\#, Acc\#) \wedge acc(Acc\#, Owner, Bal) \wedge (Bal < 10)$$

the predicate $(Bal < 10)$, is a non-updatable predicate because it is an evaluable function. The predicate $bankcard(Card\#, Acc\#)$ is another non-updatable predicate as no trigger action updates the relation *bankcard*. On the other hand, the predicate $acc(Acc\#, Owner, Bal)$ is an updatable predicate as it can be updated by the action *debit* in trigger rule r1. Therefore, using our non-updatable predicate selection procedure, the formula

$$bankcard(Card\#, Acc\#) \wedge acc(Acc\#, Owner, Bal) \wedge (Bal < 10)$$

is modified to be,

$$bankcard(Card\#, Acc\#) \wedge (Bal < 10)$$

As the variable *Bal* appears once in the above condition, $Bal < 10$ is trivially satisfiable. Now, the final condition is simplified to,

$$bankcard(Card\#, Acc\#)$$ □

3.1.2 Finitely Updatable predicate procedure

In practice, it is unlikely to include many predicates in the conjunction of the trigger conditions, as relations/objects are often targets of update. The *finitely-updatable predicate selection* is therefore an improvement on it. Instead of including only predicates which are not updated by any trigger action, this predicate selection procedure includes predicates that are not updated indefinitely by any trigger action. However, to decide which predicate is updated only finitely is as hard as the original termination problem. In this case, we need to incorporate a much more complex incremental algorithm for termination detection. We will discuss this further in Section 5. Meanwhile, to clarify the basic concept, we employ the non-updatable predicate selection procedure as our default predicate selection procedure.

3.2 Construction of Activation Formula

Definition 3.1 Given an execution sequence $\ll r_1, \ldots, r_n \gg$, an *activation formula* $F_{act}(\ll r_1, \ldots, r_n \gg)$ is a necessary condition for rule r_1 to eventually trigger rule r_n via rules $r_2, \ldots, r_{n-1}$. □

Example 3.2 Consider the following trigger rules,

r1 :: $e1(X,Y)$ IF $X > 3$ DO $e2(X,Y)$
r2 :: $e2(X,Y)$ IF $X < 1$ DO $e1(X,Y)$

An activation formula of path $\ll$r1,r2$\gg$ is $(X > 3) \wedge (X < 1)$. Events such as $e1(4,2)$ cannot trigger another event $e1$ via the trigger sequence r1,r2. Note that since an activation formula gives only the necessary condition, it is not necessarily unique. Another weaker activation formula is $(X < 1)$. □

The following algorithm computes an activation formula for a given execution sequence $\ll r_1, \ldots, r_n \gg$:

Algorithm 3.1 Given an execution sequence $\ll r_1, \ldots, r_n \gg$, and a predicate selection procedure *PSP*, we compute an activation formula, $F_{act}(\ll r_1, \ldots, r_n \gg)$ as follows,

1. We first compute an intermediate condition C as follows, This C will be transformed into

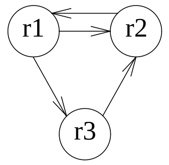
$F_{act}(\ll r_1, \ldots, r_n \gg)$ at the final step.

2. When $n = 1$, C is set to $r_1.\textsc{Cond}$
3. Otherwise, let C be $F_{act}(\ll r_2, \ldots, r_n \gg)$ subject to the selection procedure *PSP*, and let σ be the substitution unifier between the event of rule r_2 and the triggering action of rule r_1. We perform the following steps,
   i) Rename any local variable in C that also appears in r_1 to another name to avoid name conflict.
   ii) C is set to $r_1.\textsc{Cond} \wedge C\,\sigma$
4. To compute the final $F_{act}(\ll r_1, \ldots, r_n \gg)$, apply the predicate selection procedure *PSP* to remove any unwanted predicates in the formula C. □

Example 3.3 Consider the following three trigger rules,

r1 :: $e1(X,Y)$ IF $(X>1)$ DO $\{e2(X,0), e3(1,Y)\}$
r2 :: $e2(X,Y)$ IF True DO $e1(Y,X)$
r3 :: $e3(X,Y)$ IF $X \neq Y$ DO $e2(X,Y)$

The trigger graph of these rules is as follows,



Subject to the non-updatable predicate selection procedure, the activation formula for $F_{act}(\ll r_1, r_2, r_1 \gg)$ can be computed as follows,

1. First, we compute $F_{act}(\ll r_2, r_1 \gg)$, which requires to compute $F_{act}(\ll r_1 \gg)$.
2. Now, $F_{act}(\ll r_1 \gg)$ is actually the condition of rule r1 after removing any updatable predicate. This gives the formula $(X>1)$.
3. To compute $F_{act}(\ll r_2, r_1 \gg)$, we first observe that $\{X/Y, Y/X\}$ is the substitution σ between the event of r_1, $e1(X,Y)$, and the matching action of r_2, $e1(Y,X)$. Applying the algorithm, we have
   $\textsc{True} \wedge (X>1)\{X/Y, Y/X\}$
   After simplification and elimination of any updatable predicate, we have $(Y>1)$.
4. Finally, to compute $F_{act}(\ll r_1, r_2, r_1 \gg)$, $\{X/X, Y/0\}$ is the substitution σ between the $e2(X,Y)$, and the matching action $e2(X,0)$. We compute C to be,
   $(X>1) \wedge ((Y>1)\{X/X, Y/0\})$
   It can be simplified to False. In other words, the activation formula of the execution sequence from r_1, to r_2, and then back to r_1 is not possible. □

Definition 3.2 An execution sequence $\ll r_1, \ldots, r_n \gg$ where r_i's are trigger rules which are not necessarily distinct, is an *activable path* subject to a predicate selection procedure *PSP* if the activation formulae $F_{act}(\ll r_1, \ldots, r_n \gg)$ subject to *PSP* is satisfiable. Otherwise, the sequence is a *non-activable path*. □

Example 3.4 From Example 3.3, the path $\ll$r2,r1$\gg$ is an activable path. The path $\ll$r1,r2,r1$\gg$ is a non-activable path. □

Definition 3.3 A *cycle* is denoted by $[\![r_1, \ldots, r_n]\!]$ where r_i's are not necessarily distinct. It indicates the cyclic execution that rule r_1 triggers r_2, which then triggers r_3 and so on until r_n, which triggers back r_1. □

Example 3.5 In Example 3.3, $[\![$r1,r2$]\!]$, $[\![$r1,r3,r2$]\!]$ and $[\![$r1,r2,r1,r3,r2$]\!]$ are three different cycles. □

Definition 3.4 The *k*-execution sequence ($k \geq 1$) of a cycle $[\![r_1, \ldots, r_n]\!]$ is denoted as $[\![r_1, \ldots, r_n]\!]^k$. It is the execution sequence if the cycle loops itself for *k* times. It can be represented by an execution sequence which is defined recursively as follows,

1. $[\![r_1, \ldots, r_n]\!]^1$ is the execution sequence $\ll r_1, \ldots, r_n, r_1 \gg$.
2. $[\![r_1, \ldots, r_n]\!]^k$ is the concatenation of the execution sequence $\ll r_1, \ldots, r_n \gg$ with $[\![r_1, \ldots, r_n]\!]^{k-1}$. □

Example 3.6 $[\![r_1]\!]^1$ represents the execution sequence $\ll r_1, r_1 \gg$. $[\![r_1, r_2]\!]^2$ represents the execution sequence $\ll r_1, r_2, r_1, r_2, r_1 \gg$. □

Definition 3.5 A cycle $[\![r_1, \ldots, r_m]\!]$ contains another cycle $[\![s_1, \ldots, s_n]\!]$ if and only if the execution sequence $[\![r_1, \ldots, r_m]\!]^1$ contains the execution sequence $[\![s_1, \ldots, s_n]\!]^1$. □

Example 3.7 The cycle $[\![r_1, r_2, r_1, r_3]\!]$ contains the cycle $[\![r_1, r_2]\!]$. However, the cycle $[\![r_1, r_2]\!]$ does not contain the cycle $[\![r_1]\!]$ as the execution sequence $\ll r_1, r_2, r_1 \gg$ does not contain the execution sequence $\ll r_1, r_1 \gg$. □

Definition 3.6 A cycle in a graph is a *prime cycle* if it does not contain any other cycle. □

Example 3.8 The cycle $[\![r_1, r_2, r_1, r_3]\!]$ contains the cycles $[\![r_1, r_2]\!]$ as well as $[\![r_1, r_3]\!]$, hence, it is not a prime cycle. On the other hand, both $[\![r_1, r_2]\!]$ and $[\![r_1, r_3]\!]$ are prime cycles. □

Definition 3.7 A cycle $[\![r_1, \ldots, r_n]\!]$ is a *k*-cycle ($k \geq 0$) if

1. $[\![r_1, \ldots, r_n]\!]^{k+1}$ is not an activable path, and
2. if $k \geq 1$, then $[\![r_1, \ldots, r_n]\!]^k$ is an activable path. □

Note that if $[\![r_1, \ldots, r_n]\!]$ is a 0-cycle, it simply means that $\ll r_1, \ldots, r_n, r_1 \gg$ is a non-activable path. If a cycle is a *k*-cycle, then it can only be repeated at most *k* times.

Using this definition, the following algorithm can decide if a given cycle Γ is a k-cycle.

Algorithm 3.2 Given a cycle Γ in a trigger graph G, and an integer k, we decide if Γ is a k-cycle by the following steps:

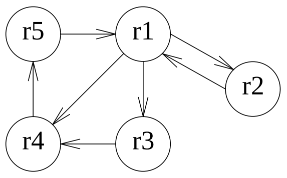
1. Construct the k-execution sequence Γ^k of the cycle Γ and the $(k+1)$-execution sequence $\Gamma^{(k+1)}$ of the cycle Γ.
2. Apply Algorithm 3.1 to construct $F_{act}(\Gamma^k)$ and $F_{act}(\Gamma^{(k+1)})$.
3. If $F_{act}(\Gamma^k)$ is satisfiable but $F_{act}(\Gamma^{(k+1)})$ is not satisfiable, then return "Γ is a k-cycle."
4. Otherwise, return that "Γ is not a k-cycle." □

Theorem 3.1 Given a trigger graph G, and if each prime cycle is a 0-cycle, then the trigger system can terminate. □

Example 3.9 Consider the following five rules:

r1 :: $e1(X,Y)$ IF $X>1$ DO $\{e2(X,Y), e3(X,Y)\}$
r2 :: $e2(X,Y)$ IF $Y<1$ DO $e1(Y,X)$
r3 :: $e2(X,Y)$ IF $X<2$ DO $e3(X,Y)$
r4 :: $e3(X,Y)$ IF $X<Y$ DO $e4(Y+1,X)$
r5 :: $e4(X,Y)$ IF $X<Y$ DO $e1(X,Y)$

Its trigger graph can be described as:



From the graph, we can see that there are three prime cycles: Γ_1:⟦r1,r2⟧, Γ_2:⟦r1,r4,r5⟧ and Γ_3:⟦r1,r3,r4,r5⟧. Note that there are many different cycles such as ⟦r1,r2,r1,r2,r1,r4,r5⟧, but it is not a prime-cycle. It contains three prime cycles: Γ_1,Γ_1 and Γ_2.

Each of the three prime cycles contains at least one non-activable path. For instance, the cycle ⟦r1,r2⟧ contains a path ≪r1,r2,r1≫. This path is non-activable because its activation formula F_{act}(≪r1,r2,r1≫), which can be computed by Algorithm 3.1 to be

$$(X>1)\wedge(Y<1)\wedge(Y>1)$$

is clearly not satisfiable. Since each of the three prime cycles contains at least one non-activable path, all the three cycles are 0-cycle. From Theorem 3.1, this trigger system can always terminate. □

Theorem 3.1 covers many terminating cases detected by previous methods. A natural extension is to ask if every prime cycle is a k-cycle for some finite k, does it imply that the system can always terminate? Unfortunately, the answer is only partially true, as stated in the followings:

Definition 3.8 Given a trigger graph G, its cycles can be *partitioned hierarchically*, if and only if for any two different cycles Γ_1 and Γ_2 in G, whenever Γ_1 can reach Γ_2, it implies that Γ_2 cannot reach Γ_1. □

Theorem 3.2 Given a trigger system, if its prime cycles can be partitioned hierarchically, and all of its prime cycle is a k-cycle for some finite k, then the trigger system can terminate. □

In general, however, the above theorem is false if cycles cannot be partitioned hierarchically. Every prime cycle is some k-cycle for some finite k alone does not imply that the system can always terminate, even if k is as small as 1. This can be illustrated by the following example:

Example 3.10 Consider the following trigger system:

r1 :: $e1(X,Y)$ IF $X<Y$ DO $\{e1(Y,X), e2(Y,X)\}$
r2 :: $e2(X,Y)$ IF $X>Y$ DO $e1(Y,X)$

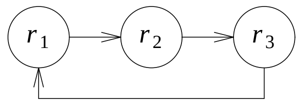
There are two prime cycles — ⟦r1⟧ and ⟦r1,r2⟧. Note that these two cycles cannot be partitioned hierarchically. The first cycle can reach the second cycle, and the second cycle can also reach the first cycle. Now, both prime cycles are only 1-cycles, however, the trigger system may not necessarily terminate. The non-prime cycle ⟦r1,r1,r2⟧ can be executed without termination. □

To handle this problem, we propose a new method which attempts to remove any k-cycle found in the trigger graph. If every cycle can be removed, then the trigger system can terminate. This differs from all the previous works [1, 17, 2, 11, 18, 3, 13] in that they are aiming to prove that no cycle actually exists in the trigger graph. Our method, however, aims to prove that every cycle in the trigger graph is only finitely executable.

4 Cycle removal by cycle-unrolling method

To prove that every cycle in the trigger graph is only finitely executable, we propose to remove each finitely executable cycle from a given trigger graph. If all finitely executable cycles can be removed from the graph, then we can conclude that the trigger system can terminate.

In order to remove a finitely executable cycle from a trigger graph, we propose a *cycle unrolling* method. The main idea of this method is to unroll the given cycle by multiplicating the nodes inside the given cycle and break the cycle. For example, if a cycle ⟦r_1,r_2,r_3⟧ is proven to be a 2-cycle, then the very definition of a 2-cycle implies that the path P, ≪$r_1,r_2,r_3,r_1,r_2,r_3,r_1,r_2,r_3,r_1$≫ is not an activable path. In this case, we can unroll the cycle by replacing it with a path that does not contain the non-activable path P. This idea is summarized in the following diagram:



The above 2-cycle ⟦r1,r2,r3⟧ is unrolled to

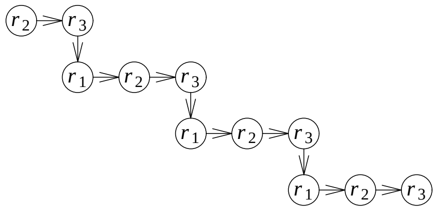


Figure 4.1

There are some points about figure 4.1 that we would like to mention:

1. There are extra duplicated copies of r_2 and r_3 at the front. This is so because the execution sequence $\ll r_2, r_3, r_1, r_2, r_3, r_1, r_2, r_3, r_1, r_2, r_3 \gg$ may be an activable path as it does not contain the non-activable path *P*. As no activable path should be lost during the unroll operation, we need to include an extra copy of the cyclic nodes except the first node (r_1).
2. After the unroll operation, the cycle $[\![r_1, r_2, r_3]\!]$ is allowed to loop twice as the execution sequence $\ll r_1, r_2, r_3, r_1, r_2, r_3, r_1 \gg$ is in the modified graph. Furthermore, this cycle is correctly denied to loop thrice as the non-activable path *P* is missing from the modified graph.

We call the above mentioned operation, which replaces a *k*-cycle Γ in a trigger graph *G* with a non-cyclic path, the unroll operation $unroll(G, \Gamma, k)$. This operation can be defined formally as follows,

Definition 4.1 The operation $unroll(G, \Gamma, k)$ where *G* is a trigger graph, Γ is a cycle in *G* and *k* is a positive integer generates a new graph G' such that

1. The *k*-execution sequence Γ^k can be represented in the new graph G', but
2. Γ^{k+1} cannot be represented in the new graph G', and
3. all other paths in *G* which do not contain the sequence Γ^{k+1} are represented in G'. □

The following gives a formal algorithm on how the operation $unroll(G, \Gamma, k)$ can be implemented:

Algorithm 4.1 *(Cycle unroll algorithm)* Consider a trigger graph *G* and a *k*-cycle $\Gamma [\![r_1, \ldots, r_n]\!]$ where $n \geq 1$ The operation $unroll(G, \Gamma, k)$ is done as follows,

1. (*Duplicate the cyclic path*) Duplicate the path $\ll r_1, \ldots, r_n \gg$ for $k+1$ times, we denote $r_i^{(0)}$ as the original node r_i, and we denote $r_i^{(j)}$ $(1 \leq j \leq k+1)$ as the j^{th} copy of the node r_i. Furthermore, for each *j* from 0 to *k*, add the edge $\ll r_n^{(j)}, r_1^{(j+1)} \gg$.
2. (*Allow any iteration of the cycle to exit.*) For each outgoing edge $\ll r_i^{(0)}, t \gg$ of node $r_i^{(0)}$, except the edge $\ll r_n^{(0)}, r_1^{(0)} \gg$ and the edges $\ll r_i^{(0)}, r_{i+1}^{(0)} \gg$ where $1 \leq i \leq n-1$, insert the edge $\ll r_i^{(j)}, t \gg$ in the graph for each *j* from 1 to $k+1$.
3. (*Allow any incoming edge to go into the cycle.*) For each in-coming edge $\ll t, r_1^{(0)} \gg$ of node $r_1^{(0)}$, insert the edge $\ll t, r_1^{(1)} \gg$ in the graph.
4. (*remove the node $r_1^{(0)}$ to break the cycle.*) Remove the node $r_1^{(0)}$ from the graph, and its attached edges from the graph. □

Example 4.1 Given the trigger graph in Figure 4.2, assume that we have proven that the cycle ⟦r1,r2⟧ is a 1-cycle, in another words, the execution sequence ≪r1,r2,r1≫ is activable, but the sequence ≪r1,r2,r1,r2,r1≫ is not. We can remove this cycle by our unroll algorithm, Algorithm 4.1.

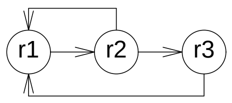


Figure 4.2

In the first step of the algorithm, we duplicate the path ≪r1,r2≫: $\ll r1^{(1)}, r2^{(1)} \gg$ and $\ll r1^{(2)}, r2^{(2)} \gg$. We also denote the original r1 as $r1^{(0)}$ and original r2 as $r2^{(0)}$. Furthermore, the algorithm connects the last node of each cyclic path duplication to the first node of the next duplication. In this example, we add the edge $\ll r2^{(0)}, r1^{(1)} \gg$ and $\ll r2^{(1)}, r1^{(2)} \gg$. The resultant graph is in figure 4.3(i).

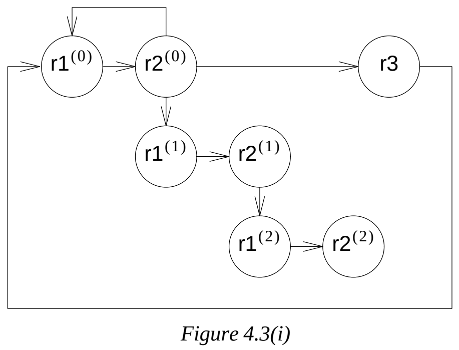


Figure 4.3(i)

In step 2, we add all the outgoing edges to provide exit from the cycle. In this example, since ≪r2,r3≫ was an edge in the original graph, we insert two new edges: $\ll r2^{(1)}, r3 \gg$ and $\ll r2^{(2)}, r3 \gg$. The resultant graph will be in Figure 4.3(ii).

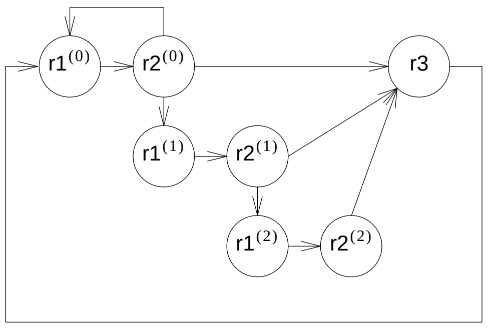


Figure 4.3(ii)

In step 3, all the incoming edges which were originally to r1$^{(0)}$ are now duplicated to point also to r1$^{(1)}$. In this example, since ≪r3,r1$^{(0)}$≫ is the only incoming edge to r1$^{(0)}$, we add the edge ≪r3,r1$^{(1)}$≫ to the graph. This gives Figure 4.3(iii).

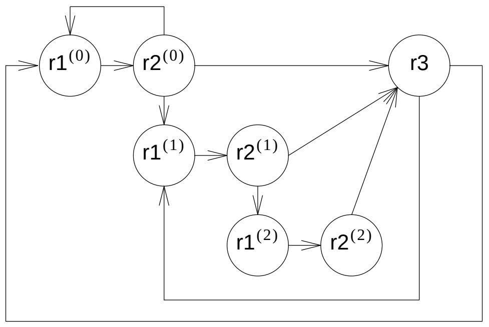


Figure 4.3(iii)

The final step of the algorithm requires us to remove the node r1$^{(0)}$. The resultant graph is as in Figure 4.3(iv).

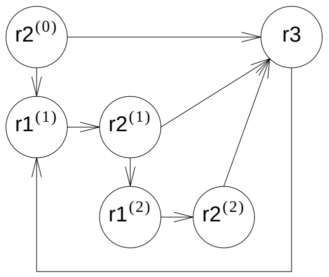


Figure 4.3(iv)

Now there remains only two prime cycles — ⟦r1$^{(1)}$,r2$^{(1)}$,r3⟧ and ⟦r1$^{(1)}$,r2$^{(1)}$,r1$^{(2)}$,r2$^{(2)}$,r3⟧. The original cycle ⟦r1,r2⟧ has been successfully removed. □

Repeatedly of applying Algorithm 4.1 to remove cycles from a trigger graph, if it terminates, can decide trigger termination. However, this repetition process, per se, does not necessarily terminate for two different reasons:

1. After application of Algorithm 4.1, we can generate new prime cycle of longer cyclic length. In the previous example, when the prime cycle ⟦r1,r2⟧ of length 2 is removed, a new prime cycle ⟦r1$^{(1)}$,r2$^{(1)}$,r1$^{(2)}$,r2$^{(2)}$,r3⟧ of length 5 is introduced. In theory, this process may not terminate. Longer and longer cycle can be found when some smaller cycles are merged. In the previous example, the node r2 of the 1-cycle ⟦r1,r2⟧ is duplicated and merged into the cycle ⟦r1,r2,r3⟧ to generate a new longer cycle ⟦r1$^{(1)}$,r2$^{(1)}$,r1$^{(2)}$,r2$^{(2)}$,r3⟧. In practical systems, this merging process is unlikely to be too complex. For a terminating system, it is unlikely that a longer prime cycle is still an activable path. Hence, we propose to set up a threshold value, D_{MAX}, which denotes the maximum number of times a node can be duplicated. If termination cannot be proven after a node has participated in Algorithm 4.1 for D_{MAX} times, then we stop without drawing any conclusion.
2. To decide if a cycle is a k-cycle for a finite value of k is itself also undecidable. To make our algorithm terminate, we need another threshold, K_{MAX}, to limit the value of k.

With these two thresholds in mind, we can now apply Algorithm 4.1 to construct a new termination decision algorithm as follows,

Algorithm 4.2 Given a trigger system *TS*, and two threshold values D_{MAX} and K_{MAX}, we decide if *TS* terminates by the following steps,

1. Construct a trigger graph *G* for *TS*.
2. Remove any node which is not inside any cycle in the current trigger graph.
3. Remove any edge ≪s,t≫ if its activation formula is unsatisfiable.
4. If the graph is acyclic, then conclude "Terminate" and exit the algorithm.
5. Pick up one of the smallest prime cycles in the current trigger graph. If this cycle contains a node that has been duplicated for D_{MAX} times, then return "May not terminate" and exit the algorithm.
6. Apply Algorithm 3.2 to decide if the chosen cycle is a k-cycle where $0 \le k \le K_{MAX}$. If not, then return "May not terminate" and exit the algorithm.
7. Otherwise, remove this cycle according to Algorithm 4.1. The new graph is now the current trigger graph. Repeat step 2. □

Example 4.2 Given the following trigger sets,

r1 :: $e1(X,Y)$ IF $(X>Y)$ DO $e2(Y,1)$
r2 :: $e2(X,Y)$ IF $b(X,Y)$
    DO $\{e3(X,Y), e1(X,Y)\}$
r3 :: $e3(X,Y)$ IF $X<1$ DO $e1(X,Y)$

We assume no event updates the relation *b*. In addition, we assume K_{MAX} to be 4, and D_{MAX} to be 4. We now apply Algorithm 4.2 to decide if the set of rules will terminate. The first step is to construct a trigger graph as in Figure 4.4(i).

In step 2, we eliminate any node which is not inside any cycle. No such node is found from the current graph. Similarly, no edge can be removed at step 3.

In step 5 of Algorithm 4.2, we pick one of the smallest cycles — $[\![\text{r1},\text{r2}]\!]$ in the graph. It is not a 0-cycle. The activation formula of the path $\ll\text{r1},\text{r2},\text{r1}\gg$ is

$$(X>Y)\wedge b(Y,1)\wedge(1>Y)$$

It is satisfiable. However, it is a 1-cycle. The activation formula of the path $\ll\text{r1},\text{r2},\text{r1},\text{r2},\text{r1}\gg$ is

$$(X>Y)\wedge b(Y,1)\wedge(Y>1)\wedge b(1,1)\wedge(1>1)$$

and it is unsatisfiable. Therefore, we have proven that the cycle $[\![\text{r1},\text{r2}]\!]$ is a 1-cycle. From step 7, we apply Algorithm 4.1 to remove this cycle. The new graph is shown in figure 4.4(ii):

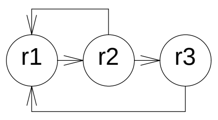


Figure 4.4(i)

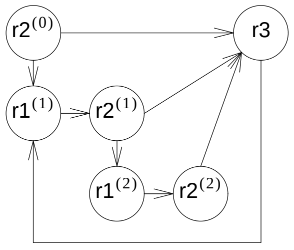


Figure 4.4(ii)

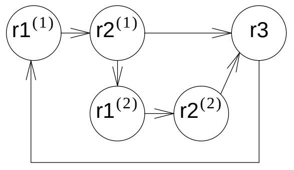


Figure 4.4(iii)

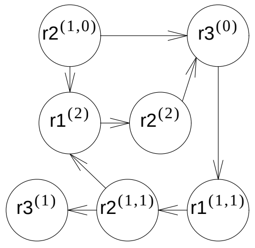


Figure 4.4(iv)

We now repeat step 2. Since the node $\text{r2}^{(0)}$ is not inside any cycle in the current trigger graph, it can be removed. The new graph is shown in Figure 4.4(iii). Next, we consider the cycle $[\![\text{r1}^{(1)},\text{r2}^{(1)},\text{r3}]\!]$. Without much difficulty, we can prove that it is a 0-cycle, for its activation formula is

$$(X>Y)\wedge b(Y,1)\wedge(Y<1)\wedge(Y>1)$$

is unsatisfiable. This cycle is unroll. The new graph is shown in Figure 4.4(iv). Clearly, $\text{r3}^{(1)}$ and $\text{r2}^{(1,0)}$ can be removed. Now, there remains only one last cycle: $[\![\text{r1}^{(1,1)},\text{r2}^{(1,1)},\text{r1}^{(2)},\text{r2}^{(2)},\text{r3}^{(0)}]\!]$. This cycle is another 0-cycle that can be removed after unrolling. Since there is no more cycle in the transformed graph, we have proven that this system can always terminate. Note that existing works [2, 11, 13] cannot draw the same conclusion. □

5 Complexity of Algorithm 4.2

In this section, we will discuss the complexity of our method.

Theorem 5.1 For any trigger system, if its termination can be detected using existing methods such as [11], then Algorithm 4.2 can also detect termination with the same time complexity at step 4 without any cycle unrolling. □

Lemma 5.1 For a *k*-cycle Γ in a trigger graph *G*, assume it shares nodes with *p* other cycles, then the number of cycles after $unroll(\Gamma,G,k)$ increases at most by $p(k+2)-1$. □

Putting *p* to be zero, we arrive the following lemma,

Lemma 5.2 For any trigger system such that the cycles in the corresponding trigger graph can be partitioned hierarchically, i.e., for any two different cycles Γ_1 and Γ_2 in the corresponding trigger graph, if Γ_1 can reach Γ_2 implies that Γ_2 cannot reach Γ_1, each *unroll* operation will decrease the number of cycles in the trigger graph by one. □

Theorem 5.2 Given a trigger system of *n* nodes that can be partitioned hierarchically, Algorithm 4.2 determines its termination with only $O(nK_{MAX})$ evaluations of trigger conditions. □

Theorem 5.3 Given a trigger system such that all its cycle is a 0-cycle, the unroll operation is similar to the path-splitting operation in [13]. Hence, our method can cover the cases detected by [13] with the same time complexity. Note that as stated in [13], considering path (and thus cycle) instead of edge to detect termination is NP-complete. □

Finally, we have an analysis for the general cases as follows,

Theorem 5.4 Given a trigger graph *G* with *p* prime cycles, the number of times that operation *unroll* needed to be done before its termination is at most

$$(K_{MAX}+2)^{D_{MAX}}p \qquad \square$$

Although our method can decide much more terminating cases which cannot be detected using existing methods, the worst case run-time complexity of this method is exponential. Despite that, our method, in practice, is still an effective method for the following reasons:

1. Any termination case that can be detected by methods [1, 11] can also be detected within the first

four steps of Algorithm 4.1 with the same time complexity.
2. For any method that detects all our termination cases, it is unavoidable that it must be at least NP-complete as stated in [13].
3. Worst case performance only appears in some artificially designed cases. In practice, our method still performs reasonably well.

6 Finitely-updatable predicate

One of the limitations of using non-updatable predicates is that usually very few predicates can be used in an activation formula. To include more predicates in the activation formula, and thus detect more termination conditions, we propose to use a more powerful predicate selection procedure during the computation of activation formula. The idea is to take not only non-updatable predicate, but also finitely updatable predicate. As described in Section 3.1.2, a finitely-updatable predicate is a predicate which can be updated only finitely during a trigger session. Since to decide exactly the set of finitely-updatable predicates is as hard as to prove trigger termination, we can only approximate this set by maintaining a set of predicates which are definitely finitely-updatable, but we have no conclusion on those predicates that are not in the set. The following algorithm describes how to incorporate finitely-updatable predicates in our termination decision algorithm, as well as other existing methods.

Algorithm 6.1 Given a trigger graph, we detect termination by the following steps,

1. Initialize the set of finitely updatable predicates *S* to be the set of predicates that are not updated by any action of any trigger rule.
2. Treat every element in *S* as if it is not updated by the trigger system, apply any existing termination detection method to decide termination.
3. If termination is proven, report "termination" and exit.
4. Otherwise, we mark the following rules as possibly infinitely execution rules:
   i) Rules that are in any unresolved cycle, i.e., cycle which has not been proven to terminate, and
   ii) rules that are reachable from some unresolved cycles in the trigger graph.

   Let *T* be the set of predicates that are updated by some actions of some possibly infinitely execution rules. We update *S* to contain all other predicates used in the databases except those in *T*.
5. If any new element is introduced into *S*, then repeat the process from step 2.
6. Otherwise, *S* is unchanged. Exit the process and report "no conclusion". □

The following example demonstrates how [11] can be improved with the finitely-updatable predicate concept:

Example 6.1 Given the following three trigger rules in an OODB environment:

r1 :: $incr\_salary(EMP1)$ IF $EMP1 \neq me$
DO $incr\_salary(me)$

r2 :: $e2(EMP2)$ IF $EMP2.salary > 1000$
DO $e3(EMP2)$

r3 :: $e3(EMP3)$ IF $EMP3.salary < 1000$
DO $e2(EMP3)$

Since the *salary* property of an EMPLOYEE class can be updated by rule r1, [11] will not make use of any predicate that used the attribute "salary". The "qualified connecting formula" of the edge ≪r2,r3≫ is therefore a simple TRUE. Hence, [11] can only conclude that the rules set may not terminate.

On the other hand, our concept of finitely-updatable predicate improves this situation. Using the activation formula, ≪r1,r1≫ is not an activable path, and hence it can only be finitely executed. The salary is only updated finitely. After the set of finitely updatable set is incrementally refined, we return to step 2 of Algorithm 6.1 and apply [11]. We improve [11] to construct a more useful formula,

$$(EMP2.salary > 1000) \wedge (EMP2 = EMP3) \wedge (EMP3.salary < 1000)$$

As this formula is unsatisfiable, termination therefore is detected. □

7 Conclusion

We have proposed a new method to detect more termination situations for an ECA-rule active database model. Although our algorithm cannot find the exact answer for all cases, it gives much stronger sufficient conditions than previous works. Our methods can isolate rules that might give rise to non-terminating execution. This means that our methods can be incorporated into an interactive tool for the specification of active database rules.

In comparison with other existing methods, we have shown that we can cover most of the termination situations which other existing methods have covered. For example, if recent works such as [11] conclude that there is no cycle in a trigger graph, then our methods can also conclude the same result with the same efficiency.

Furthermore, our methods detect far more terminating cases. Once a cyclic execution actually exists in a trigger system, existing methods will conclude that an infinite loop may occur. No other attempt is done to further examine the cycle. Our method, however, introduces the concept of *k*-cycle and allows a finite looping of the given set of trigger rules.

In future, we would also like to investigate any other possible means to detect more terminating situations. So far, our method does not assume any execution model. We would therefore like to extend the analysis techniques to incorporate additional features of active databases such as rule priorities, different execution mode of the actions as well as more complex events augmented with temporal elements.

REFERENCE

[1] A.Aiken, J.Widom and J.M.Hellerstein, "Behaviour of database production rules: Termination, confluence, and observable determinism", *Proc ACM SIGMOD International Conf on the Management of Data*, 59-68, 1992.

[2] E.Baralis, S.Ceri and S.Paraboschi, "Run-Time Detection of Non-Terminating Active Rule Systems", *DOOD*, 38-54, 1995.

[3] E.Baralis, S.Ceri and S.Paraboschi, "Compile-time and runtime analysis of Active Behaviors", *IEEE Transactions on Knowledge and Data Engineering*, Vol 10, No. 3, May/June 1998, pp 353-370.

[4] E.Baralis and J.Widom, "An Algebraic Approach to Rule Analysis in Expert Database Systems", *20th VLDB Conf*, 475-486, Sept 12-15, 1994.

[5] S.Ceri and J.Widom, "Deriving Production Rules for Constraint Maintenance", *Proc 16th VLDB Conf*, 566-577, Brisbane, 1990.

[6] U.Dayal, "Active Database Systems", *Proc 3rd International Conf on Data and Knowledge Bases*, Jerusalem Israel, June 1988.

[7] Dennis R.McCarthy and U.Dayal, "The Architecture of An Active Database Management System", *Proc ACM SIGMOD International Conf on the Management of Data* Vol 18, Number 2, 215-224, June 1989.

[8] O.Diaz, N.Paton and P.Gray, "Rule management in object-oriented databases: A uniform approach", *Proc 17th International Conf on VLDB,* Barcelona, Spain, September 1991.

[9] S.Gatzin, A.Geppert and K.R.Dittrich, "Integrating Active Concepts into an Object-Oriented Database System", *Proc 3rd International Workshop on database programming languages,* Nafplion, 1991.

[10] A.P.Karadimce and S.D.Urban, "Conditional term rewriting as a formal basis for analysis of active database rules", *4th International Workshop on Research Issues in Data Engineering (RIDE-ADS'94)*, February 1994.

[11] A.P.Karadimce and S.D.Urban, "Refined Trigger Graphs: A Logic-Based Approach to Termination Analysis in an Active Object-Oriented Database", *ICDE'96*, 384-391.

[12] T.W.Ling, "The Prolog Not-Predicate and Negation as Failure Rule", *New Generation Computing*, 8, 5-31, 1990.

[13] S.Y.Lee and T.W.Ling, "A Path Removing Technique for Detecting Trigger Termination", *EDBT*, 341-355, 1998.

[14] M.Stonebraker and G.Kemnitz, "The POSTGRES Next-Generation Database Management System", *CACM*, 34(10), 78-93, Oct 1991.

[15] H.Tsai and A.M.K.Cheng, "Termination Analysis of OPS5 Expert Systems", *Proc of the AAAI National Conference on Artificial Intelligence*, Seattle, Washington, 1994.

[16] T.Weik and A.Heuer. "An algorithm for the analysis of Termination of Large Trigger Sets in an OODBMS", *Proceedings of the International Workshop on Active and Real-Time Databases Systems,* Skovde, Sweden, June 1995.

[17] L.Voort and A.Siebes, "Termination and confluence of rule execution", *Proc 2nd International Conf on Information and Knowledge Management*, Nov 1993.

[18] D.Zimmer, A.Meckenstock and Rainer Unland, "Using Petri Nets for Rule Termination Analysis", *Proc of Workshop on Databases: Active and Real-Time,* Rockville, Maryland, November 1996.