Supporting Distributed Event-Condition-Action Rules in a Multidatabase Environment

Vasiliki Kantere
NTUA
Athens, Greece
verena@dblab.ece.ntua.gr

Iluju Kiringa
University of Ottawa
Ottawa, Canada
kiringa@site.uottawa.edu

John Mylopoulos
University of Toronto
Toronto, Canada
jm@cs.toronto.edu

Abstract

We describe a mechanism based on distributed Event-Condition-Action (ECA) rules that supports data coordination in a multidatabase setting. The proposed mechanism includes an ECA rule language and a rule execution engine that transforms rules when they are first posted, and then coordinates their execution. Like traditional ECA rules, our ECA rule language has three parts: an event language, a condition language, and an action language. The event language provides a set of operators with a formal semantics for a multidatabase environment, and which allows a wide variety of composite events. The condition language provides Boolean algebra operators that take as operands either composite or simple conditions. The action language provides a conjunction of simple or composite actions. The execution model partitions rules to more easily manageable forms, distributes them to relevant databases, monitors their execution and composes their evaluations. The mechanism has been designed in a manner that minimizes the number of messages that need to be exchanged over the network. We have also conducted an experimental evaluation to compare the implementation with a naïve centralized execution model. The paper also presents a prototype implementation as well as experimental results on its performance. This work is part of an on-going project intended to develop data coordination techniques for data sharing settings.

1. Introduction

In the last fifteen years, relational and object-oriented database management systems have been augmented with an assortment of mechanisms for expressing active behavior to yield active database management systems (ADBMSs). Usually, ADBMSs employ Event-Condition-Action (ECA) rules as the standard mechanism for capturing active behavior. Using ECA rules, an ADBMS performs database definition and manipulation operations automatically, without any user intervention. However, ECA functionality has been studied mostly in centralized environments [Paton99]. Attempts were made to integrate ECA rules in distributed database systems in [TC97], [CL01], [CGMW94], [ABD+93], [KLW93], and [KRS99]. These approaches, however, generally elaborate on particular aspects of the problem, but do not present a complete solution. An approach presents a complete solution if it considers the distribution of all the components of an ECA rule, that is, the event, condition, and action parts. A notable exception is the vision for ECA functionality in a distributed setting discussed in [BKLW99]. In any case, to our knowledge, there is no published account of the details of an implementation of distributed ECA rules to the extent of [CPM96].

Many data sharing scenarios justify our interest in distributed ECA rule mechanisms. For example, in a health care application, hospitals, family doctors, pharmacists, and pharmaceutical companies maintain databases about patients and would like to participate in a data sharing network in order to establish acquaintances, exchange information about patient histories, medication, symptoms, treatments and the like. Likewise, there are many genomic data sources all over the world, from widely known ones, like

GenBank and GDB, to individual and independent lab repositories [KAM02]. It would be useful for scientists to be able to establish acquaintances and share genomic data on an as-needed basis, rather than through a global, static integration solution. E-commerce is another potential application domain, where buyers and sellers with the same business interests would like to share information and build *ad hoc* communities. As well, the transportation industry -- including travel agencies, air, rail and bus companies – could use facilities that help them build on-the-fly *ad hoc* database networks. The same can be said for the news industry -- e.g., TV channels, radio stations, and press could set up partnership communities for sharing information around specific and common topics. In all these applications, the type of rules investigated in this paper can be used.

In this paper, we present a prototype distributed rule mechanism for a multidatabase environment that can subsequently be used in other networked environment for data sharing. We assume that a multidatabase system with the following characteristics: (1) the system consists of participating autonomous databases that reside on different nodes of a network and exchange information without complying with any central administration through direct acquaintances; (2) the participants have partial knowledge of other participant's schemas and data; (3) there is a user interface on each participant for locally querying and updating the participant's data; and (4) changes are forwarded to acquainted participants via the ECA rule mechanism. In this setting, we propose ECA rules as a means for databases to coordinate their contents (e.g., by propagating updates). The main result of the paper is an ECA rule language that includes an expressive event language. We have also designed an execution model that minimizes the number of communication messages among databases involved in the execution of a rule. The proposed evaluation procedure is fully distributed, and involves partial evaluations of rule components in different databases, together with the composition of partial results into a global evaluation. We have also developed a prototype implementation of the proposed mechanism and have performed experiments comparing it with a centralized rule execution model. A preliminary and abridged version of this work was presented in [KMK03].

Next, we give a specific scenario that illustrates the usefulness of the active functionality in a multidatabase environment.

1.1 Motivating Example: A Healthcare Application

Consider a health care domain (adapted from [BGK+02]). Assume that there is a multidatabase system where the involved databases belong to family doctors, hospitals and pharmacists. Assume that Dr. Davis is a family doctor whose database, DavisDB, is acquainted with the database of a pharmacist, AllenDB and with the database of the Toronto General Hospital, TGHDB. Also, AllenDB is acquainted with TGHDB. Figure 1 depicts the network of the induced system with three multidatabases, on top of which resides a rule management system.

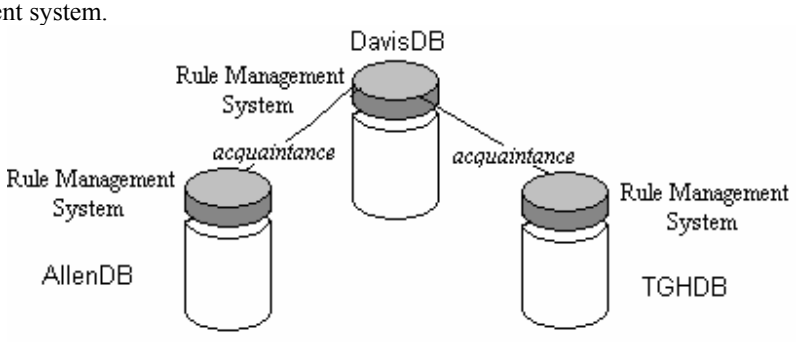


Figure 1: Acquainted databases.

The three databases contain the following tables:

DavisDB:
Visit (OHIP#, Date, Time, Symptom, Diagnosis)

Prescription (OHIP#, DrugID, Date, Dose, Quantity)
DiseaseOccur (OHIP#, Date, DiseaseDescr)
Treatments(DiseaseDescr, TreatmRecord, DrugsRecord)
Allergies (OHIP#, Allergy)

AllenDB:
Presciption (Prescr#, CustName, CustPhone#, DrugID, Dose, Repeats, DiseaseDescr)
Allergies (DrugID, Allergy)

TGHDB:
DiseaseOccur (Date, OHIP#, DiseaseDescr)
Treatment(TreatID, TGH#, Date, DiseaseDescr, TreatDescr, PhysID)
Admission (AdmID, OHIP#, AdmDate, ProblemDesc, PhysID, DisDate)
MedRec (OHIP#, Date, Time, Record)
Medication (OHIP#, Date, Time, DrugID, Dose)

Note that these databases would be heterogeneous in the general case, i.e. they would store the same kind of information in different formats. For simplicity we assume here that the databases participating in a multidatabase system are homogeneous. That is, the data values used in all these databases constitute one single universe of discourse.

Suppose that Dr. Davis wants to keep track of treatment methods and medications for a number of specific diseases. Thus, he wants to get information about treatments in the hospital and the medication that Mr. Allen is selling after prescription for this disease. An appropriate rule to capture this situation is:

Rule 1:
when TGHDB.('insert',(Treatment,(_, _, _,
<DiseaseDescr_value>, <TreatDescr_value>, _)))
OR AllenDB.('insert', (Prescription,(_,_,_,
<DrugID_value>, _, _, <DiseaseDescr_value>))))

if <DiseaseDescr_value> in DiseaseDescrSet
(*where* DavisDB.('retrieve',15, {DiseaseDescrSet}))

then DavisDB.('update',(Treatments, (<DiseaseDescr_value>,<TreatmRecord_update>,
<DrugsRecord_update>)))

Query 15:
Select DiseaseDescr
From Treatments

Rule 1 above is triggered by a composite event which follows the keyword *when*: either an insertion in TGHDB about a treatment for a disease or an insertion in AllenDB about a prescription for a disease. When such an insertion occurs the condition part of the rule, introduced by the keyword *if*, checks if the disease is among a set of disease for which Dr. Davis is particularly interested. The *where* clause in the condition denotes that we get the set of values of the *DiseaseDescr* attribute of the *Treatments* relation by performing Query 15 on DavisDB. The action part of the rule, introduced by the keyword then, states that the tuple concerning the disease DiseaseDescr_value in Treatments relation of DavisDB is updated with new values for attributes TreatmRecord and DrugsRecord that incorporate the newly obtained information from TVHDB or AllenDB.

Suppose that a patient visits Dr. Davis and the latter suggests that she be admitted to the Toronto General Hospital for examination. When she is admitted to the hospital, information on the diagnosis of Dr. Davis must be transferred to the TGHDB. For such a transfer to take place, the following rule may be used:

Rule 2:

when DavisDB.('insert', (Visit, (<OHIP#_value1>,
<Date_value>,<Time_value>, _,<Diagnosis_value>))) «
TGHDB.('insert', (Admission, (_, <OHIP#_value2 >, _, _, _, _)))

if <OHIP#_value1> = <OHIP#_value2>

then TGHDB.('insert', (MedRec, (<OHIP#_value2>,
<Date_value>, <Time_value>, <Diagnosis_value>)))

The symbol '«' denotes a 'sequence' (of events) and is introduced explicitly in the next section. Rule 2 says that when there is an insertion in the *Visit* relation of DavisDB for a patient, and afterwards there is an insertion in the *Admission* relation of TGHDB for another (possibly different) patient, if the two patients are the same person according to their OHIP#, then insert to the *MedRec* relation of TGHDB a tuple with information about the diagnosis of Dr. Davis for that patient.

In this work, we propose an ECA rule language appropriate for expressing composite events that may involve more than one database. The language includes several useful event operators, each one accompanied by a specific time interval, which is enabled across multiple databases. The rule language also provides means for expressing composite conditions and actions involving many databases. Furthermore, we propose a rule execution algorithm that decomposes the ECA rules into segments that can be separately evaluated in a single database. Decomposition and partial local evaluation reduces significantly the number of exchanged messages among the databases involved in a rule execution. Finally, partial local evaluations are gathered in one database, which performs additional checks on the validity of the rule.

1.2 Outline

The core of this paper is structured as follows. Section 2 presents the main features of our ECA rule language; it describes the main parts of an ECA rule. Section 3 describes the main algorithms needed to coordinate the distributed execution of the language. This section also discusses the issues of how instances of events are consumed by the execution mechanism. Section 4 describes our prototype implementation, and reports on experimental results. We discuss related work in Section 5. Finally, Section 6 concludes the paper, and raises some issues that need further work.

2. Distributed ECA Rule Language

The ECA rules have the following general form:

R: *when <event>,*
[if <condition >,]
then <action>

where the brackets denote an optional part. In the rest of this section we define the languages that we use to declare each one of the event, condition, and action parts.

2.1 Event Language

The event language that we propose provides a set of operators with clear semantics for a multidatabase environment. By a clear semantics we mean one that is formal and uses a first order logical language. Our semantics allow a wide variety of composite events. We believe that the high level of expressiveness of the

event language makes it suitable for many types of multidatabase systems. A simple or primitive event SE in a database DB, denoted by DB.SE, can be either a database or a time event.

A time event can be an absolute, a relevant or a periodic time event. A database event is one of the following four types of primitive database operations: *retrieve*, *update*, *insert*, and *delete*. A composite event is an expression that is formed by applying the operators of the event algebra on simple or composite events. An event instance is an occurrence in time of the respective event. Generally, an event instance starts at a time point t_s and ends at a time point t_e. Our convention is that an event instance is an instant occurrence and happens at the time point t_e.

The operators of the proposed event algebra are shown in Figure 2. The general form of an operator is $OpType_{ti}$, where *OpType* is the type of operator and *ti* is the time interval within which an instance of the composite event specified by the operator should be completed. More specifically, for every instance of a composite event, the earliest simple event instance and the latest one should have a time distance equal or shorter than the time interval denoted by the value of *ti*. The *ti* parameter of the operators is the main difference between our event algebra and the ones used for the declaration of composite events in centralized database environments. The time interval of an operator provides reference time points (either relative or absolute) that allow the performance of partial asynchronous distributed evaluations in several sites, which produce partial results (i.e. local event instances) that could be used for the detection of an event instance of a global rule.

A set of composite event operators is minimal if it is a core from which all the other composite events can be defined by Zimmer and Unland in [ZU99]. The later identify the loose sequence and the negation as being such a minimal set. Notice that the set of operators in Figure 2 is not minimal in this sense.

Operator	Type	Function	Syntax
∧	Binary	Logical AND	<event1>∧<event2>
∨	Binary	Logical OR	<event1>∨<event2>
!	Unary	Logical NOT	!<event>
«	Binary	Loose Sequence	<event1> » <event2>
*	Unary	Zero or more occurrences	*<event>
+	Unary	One or more occurrences	+<event>
#	Binary	Exact number of occurrences	<number of occurrences>#<event>
&	Binary	Maximum number of occurrences	<number of occurrences>&<event>
$	Binary	Minimum number of occurrences	<number of occurrences>$<event>
>	Binary	Strict sequence	<loose sequence expression> > <not expression>

Figure 2: Event Operators

Let CE(t) denote a composite event instance occurring at the time point t. Also, let E denote an event expression (either simple or composite). Then the following defines the event operators of our language (See Figure 2).

1. Conjunction: ∧

$CE(t) = (E1 \wedge_{ti} E2)(t) =_{df} E1(t_1)$ *AND* $E2(t_2)$, where $t_2 = t$ if $t_1 \leq t_2$, or $t_1 = t$ if $t_2 < t_1$, and $t_1, t_2 \in ti$. Thus, the conjunction of E1 and E2 given ti means that both E1 and E2 occur during ti.

2. Disjunction: ∨

$CE(t) = (E1 \vee_{ti} E2)(t) =_{df} E1(t_1)$ *OR* $E2(t_2)$, where $t_2 = t$ if $t_1 \leq t_2$ or there is no t_1, or $t_1 = t$ if $t_2 < t_1$ or there is no t_2, and $t_1, t_2 \in$ ti. Thus, the disjunction of E1 and E2 given ti means that either E1 or E2, or both E1 and E2 occur during ti.

3. Negation: !
$CE(t) = (!_{ti} E)(t) =_{df}$ NOT $(\exists t' \in ti : E(t'))$. Thus, the negation of E given *ti* means that E does not occur during ti. Also, t is the end point of ti.

4. Loose sequence: «
$CE(t) = (E1 \ll_{ti} E2)(t) =_{df} E1(t_1)$ *AND* $E2(t)$, where $t_1 \leq t$ and $t_1, t \in$ ti. Thus, loose sequence of E1 and E2 given ti means that E1 occurs before E2 during ti.

5. Strict sequence: >
$CE(t) = ((E1 \gg_{ti} E2) >_{ti} (!_{ti} E3))(t) =_{df} E1(t_1)$ *AND* $E2(t)$ *AND (NOT* $E3(t_3)$*)*, where $t_1 \leq t_3 \leq t$ and $t_1, t_3, t \in$ ti. Thus, the strict sequence of E1 and E2 given ti and E3 means that E1 occurs before E2 without E3 occurring between them during ti.

6. Zero or more occurrences: *
$CE(t) = (*_{ti} E)(t) =_{df} (E(t_1)$ *AND* $E(t_2)$ *AND …* *AND* $E(t_{m-1})$ *AND* $E(t_m))$ *OR true*, where $t_1 \leq t_2 \leq \ldots \leq t_{m-1} \leq t_m$ and $t_m = t$ and $t1, t \in$ ti and $1 \leq m$ and $m \in$ N. Thus, 'star' of E given ti means that E either does not occur or occurs one or more times during ti. Note that the goal of the * operator, according to the definition above, is not to search for the validity of the expression $*_{ti}E$, but to keep track of the instances of the event expression E(t), if there are any. The role of the 'star' operator is associative. It is not used in the declaration of original event expressions, but in the construction of sub-events (see Section 3).

7. One or more occurrences: +
$CE(t) = (+_{ti} E)(t) =_{df} E(t_1)$ *AND* $E(t_2)$ *AND …* *AND* $E(t_{m-1})$ *AND* $E(t_m)$, where $t_1 \leq t_2 \leq \ldots \leq t_{m-1} \leq t_m$ and $t_m = t$ and $t_1, t \in$ ti and $1 \leq m$ and $m \in$ N. Thus, 'plus' of E given ti means that E occurs one or more times during ti.

8. Exact number of occurrences: #
$CE(t) = (_m\#_{ti} E)(t) =_{df} E(t_1)$ *AND* $E(t_2)$ *AND …AND* $E(t_{m-1})$ *AND* $E(t_m)$, where $t_1 \leq t_2 \leq \ldots \leq t_{m-1} \leq t_m$ and $t_m = t$ and $t_1, t \in$ ti and $m \in$ N. Thus, given m and ti E occurs exactly m times during ti.

9. Maximum number of occurrences: &
$CE(t) = (_{m'}\&_{ti} E)(t) =_{df} E(t_1)$ *AND* $E(t_2)$ *AND…AND* $E(t_{m-1})$ *AND* $E(t_m)$, where $t_1 \leq t_2 \leq \ldots \leq t_{m-1} \leq t_m$ and $t_m = t$ and $t_1, t \in$ ti and $m' \geq m$ and $m', m \in$ N. Thus, given m' and ti E occurs at most m' times during ti.

10. Minimum number of occurrences: $
$CE(t) = (_{m'}\$_{st} E)(t) =_{df} E(t_1)$ *AND* $E(t_2)$ *AND …* *AND* $E(t_{m-1})$ *AND* $E(t_m)$, where $t_1 \leq t_2 \leq \ldots \leq t_{m-1} \leq t_m$ and $t_m = t$ and $t_1, t \in$ ti and $m' \leq m$ and $m', m \in$ N. Thus, given m' and ti E occurs at least m' times during ti.
In the definitions above and the following ones the symbol N represents the set of natural numbers.

As we can observe, the NOT operator has no practical meaning when the start or the end of the ti interval is not bound to a specific time point. For example, $CE(t) = !_{ti} E(t)$ is a composite event, which, although it is declared correctly, will never be evaluated, thus an instance of it will never be recognized. This happens because the ti time interval has no specific start or end point. In order for ti to start being in effect 'something' should occur. In case of all the other operators except NOT this 'something' is the earliest

event instance of one of the simple events that are included in the respective composite event. In the case of the ! operator, however, there is nothing to bind the start/end point of ti to a time point. Consequently, the ! operator is unique in that the binding of ti precedes the occurrence of the earliest involved simple event instance.

The ti interval denotes the following string of parameters: (second, minute, hour, day, weekday, month, year). Blanks can be used instead of a parameter value to denote a lack of interest in the value of the parameter.

Examples of the usage of event operators

1. If we want to signal that a specific patient in the hospital has 2 or more epileptic crises in a day, the appropriate event is:

$_2\$_{(0,\ 0,\ 0,\ 1,-,\ 0,\ 0)}$ TGHDB.('insert',('MedRec', (OHIP#_value, _, _, 'epileptic crisis')))

2. If we want to signal that a patient in the hospital has taken a specific drug twice a day, then the appropriate event is:

$_2\#_{(0,\ 0,\ 0,\ 1,-,\ 0,\ 0)}$ TGHDB.('insert',('Medication', (OHIP#_value, _, _, DrugID_value,_)))

3. If we want to know how many times a patient in the hospital has been treated in a period of one month for a specific disease, then we form the following event:

$*_{(0,\ 0,\ 0,\ 0,-,\ 1,\ 0)}$ TGHDB.('insert',('Treatment', (_, TGH#_value, _, DiseaseDescr_value,_, _)))

Note that this event is not standing alone, meaning that it can be monitored only as a subevent in combination with another event (for example a time event that signals the start of the monitoring period).

4. Suppose that we want to signal that a patient of Dr Davis is admitted twice in the hospital without visiting Dr Davis in the meantime and we set the monitoring time to be one year. Then the appropriate event is:

(TGHDB. ('insert', ('Admission' (_, OHIP#_value, _, _, _, _))) $\ll_{(0,\ 0,\ 0,\ 0,-,\ 0,\ 1)}$ TGHDB.('insert', ('Admission', (_, OHIP#_value, _, _, _, _)))) $>_{(0,\ 0,\ 0,\ 0,-,\ 0,\ 1)}$ (!DavisDB.('insert',('Visit',(OHIP#_value,_, _, _, _))))

2.2 Condition Language

The composite condition of the ECA rule is a Boolean expression using the operators of the Boolean algebra, i.e., AND, OR, and NOT. These operators take as operands either composite or simple conditions. A simple condition is one of the following:

1. a mathematical expression of the form: $f(X_1, X_2,\ldots, X_n)\ mop\ Y$, where *mop* (which stands for mathematical operator) is either $=, \neq, >$ or $<$; $X_1, X_2, \ldots, X_n, Y \in \Re$; $n \in N$; and f is any kind of mathematical function.
2. a logical expression of the form $X == Y$ *or* $X \neq Y$, where *X, Y* are strings.
3. an expression of the form $X \in \{ X_1, X_2,\ldots, X_n\}$, where X, X_1, X_2,…, X_n are strings, or X, X_1, X_2,…, $X_n \in \Re$.

In the definition above, the symbol $\Re$ represents the set of real numbers. For all three cases, X_i, for $i = 1,..,n$, *X* or *Y* are either variables or constants. If they are variables, they take values either directly from parameters of the simple event instances of the event part of the rule or indirectly from results of queries that are performed in order to evaluate the condition. The definition of such queries is parametrical and can use parameters from the event part of the rule.

Example of a composite condition

Suppose that for a patient of Dr Davis that is currently in the hospital, we want to check if he has ever been found with a specific disease. The appropriate condition is:
(DiseaseSet1 = ∅) AND (DiseaseSet2 = ∅) AND

(*where* DavisDB.('retrieve',10,{OHIP#_value, DiseaseDescr_value, DiseaseSet1})
and TGHDB.('retrieve',11,{OHIP#_value, DiseaseDescr_value, DiseaseSet2}))

Query 10:
Select *
From DiseaseOccur
Where OHIP# = OHIP#_value AND
DiseaseDescr = DiseaseDescr_value

Query 11:
Select *
From DiseaseOccur s
Where OHIP# = OHIP#_value AND
DiseaseDescr = DiseaseDescr_value

Queries 10, 11 are exactly the same but 10 is performed on DavisDb and 11 on TGHDB. The results of queries 10 and 11 are stored in the variables DiseaseSet1 and DiseaseSet2 respectively.

2.3. Action Language

The composite action of an ECA rule is a conjunction of simple or composite actions. A simple action is of the form: SA (DBi, Tr). Here, the expression DB_i, for $\forall i \in N$ and $1 \le i \le n$, is one of the n-1 databases that are acquainted to the database on which the rule that contains this specific action is to be installed. The variable Tr comprises the necessary information in order to perform the predefined transaction that is actually the simple action that we want to perform on DBi.

2.4 A Further Rule Example

We present a sample rule (from the health care domain of Section 1) emphasizing on the condition part. Suppose that we want to monitor a specific patient in the hospital that has 2 or more epileptic crises in a day. If she is taking a specific drug, <DrugID_value1>, and the dose is more than <constant>, if she is not allergic to drug <DrugID_value2>, then change the medication to the latter. We assume that the family doctor of the patient is Dr. Davis. The corresponding rule is:

Rule 3:
when $_2\$_{(0,\,0,\,0,\,1,-,\,0,\,0)}$ TGHDB.('insert',('MedRec', (OHIP#_value, _, _, 'epileptic crisis')))

if (<DrugID_value> == <DrugID_value1>) AND (<Dose_value> > <constant>)
AND ({<Allergy_value1> ∈ AllergySet1} ≠ {<Allergy_value2> ∈ AllergySet2})

(*where* TGHDB.('retrieve', 15, {OHIP#_value, DrugID_value, Dose_value})
and DavisDB.('retrieve',45,{OHIP#_value, AllergySet1})
and AllenDB.('retrieve',17,{DrugID_value2, AllergySet2}))

then TGHDB.('insert', ('Medication', (OHIP#_value,
<currentDate>, <currentTime>, <DrugID_value2>, _)))

Query 15:
Select DrugID, Dose *From* Medication
Where OHIP# = OHIP#_value, Date = <current date>, Time = <most recent time>

Query 45:
Select Allergy *From* Allergies
Where OHIP# = OHIP#_value

Query 17:
Select Allergy *From* Allergies
Where DrugID = DrugID_value2

In this example, the value (0,0,0,1,_,0,0) denotes the period of one day. The 'where' clause in the condition means that we have to perform some queries in order to get the values of the attributes that we want to compare. Thus, we perform query 15 in TGHDB in order to retrieve the drug name and the dose that the patient is receiving currently; we perform the query 45 in DavisDB and the query 17 in AllenDB to retrieve the allergies of the patient and the allergies that the drug <DrugID_value2> can provoke, respectively. The results of queries 45 and 17 are stored in the variables AllergySet1 and AllergySet2 respectively.

3. Processing ECA Rules

When a new rule is created in a database, this database is responsible for the coordination of the global evaluation of the rule. However, the evaluation of the rule involves other databases with respect to the event, condition and action expressions. This section describes the general evaluation procedure as well as the two basic algorithms that construct the input of the partial evaluations that take place in databases involved in the rule.

3.1 Algorithm for global evaluation

Figure 3 presents a pseudo-code of the general algorithm for the global distributed evaluation procedure. The evaluation procedure is as follows: The new rule, which we refer to as global/original rule, is decomposed into sub-rules that can be evaluated separately in each one of the databases that are involved

```
NewRule ()
{    while (creation of a new rule)
      {  -- decompose the rule into sub-rules and send
        them to the respective databases
           -- run the evaluation procedure for this rule
           -- run the garbage collection for this rule
}}
EvaluationProcedure (Rule newRule)
{    while (true)
      {  -- wait until a sub-rule instance of this rule is received
           -- boolean valid = evaluate the original rule with all the
      accumulated sub-rule instances
         if (valid)
        {if (additional info for the condition is
                needed)
           { -- request additional info for the
                    evaluation of the condition
                     -- boolean newruleinstance = evaluate the condition
           with all the
                           accumulated info
                     if (newruleinstance)
                     { --  execute the action part of the global rule
                        --  throw  away  the  sub-rule  instances  and  the
             additional  condition  information  used  to  form  this  rule
             instance
                 }}
          else
     {  -- execute the action part
                -- throw away the sub-rule instances
                    used to form this rule instance
}}}}
GarbageCollection (Rule newRule)
{ periodically remove the obsolete sub-rule instances}
```

Figure 3. Evaluation Procedure

in the original rule. We produce one such 'sub-rule' for each database involved in the event part. For those databases that appear both in the event and the condition parts, we add the appropriate condition expression to their corresponding sub-rule.

The condition expression of a sub-rule is a sub-expression of the condition part of the global rule. Each sub-rule is sent to the database it was made for and is evaluated there. The action part of these sub-rules is to send the sub-rule instance to the database responsible for the global evaluation of the rule. The global evaluation of the original event and the original condition expression starts when the local evaluation of one of the sub-rules created produces an instance that could contribute to the production of an instance of the original-global rule. When such a sub-rule instance is received in the database where the global rule was created, the procedure tries to find a valid match using the sub-rule instances that are stored and not used by other rule instances. If a valid match is found, additional information about the condition is requested, if needed, and gathered. If finally the condition instance is true, the execution of the action part takes place. The goal of the whole global evaluation procedure is to minimize the number of messages exchanged among the multidatabases.

The algorithm in Figure 3 guarantees that the maximum number of messages (that are exchanged among the databases involved in the processing of a rule until the event and the condition expressions are valid and the action part is executed) is the lowest possible. That number is Max # messages = n+2k+r, where n, k and r is the number of databases involved in the event expression, in the condition part and the action part, respectively (for details, see the Appendix).

3.2 Transformation Algorithm

Before the event expression is decomposed into parts, it is transformed in order to have a form more convenient for decomposition. The goal of the following algorithm is to push the 'not' operators down to the leaves of the tree of the event expression. The evaluation of the 'not' operators is hard and by pushing them down in the event tree, we may be able to eliminate some of them, or reduce the number of databases involved in the 'not' operators, or even push the evaluation of the 'not' operators to a single database. Moreover, it is possible to give to the event a simpler expression that is more convenient for the evaluation procedure. However, the decomposition of an event expression is possible even without running the transformation algorithm. The steps of the algorithm are executed multiple times until there are no transformations left to be done. Note that, in case that the operand of the 'not' operator is a composite event, it is required that the time interval of the 'not' operator has to be the same as the time interval of the operator of its operand.

```
While (there are transformations to be done) do
{   1. eliminate pairs of consecutive ! operators
    2. replace the pattern ! ti1 (E1 ∧ ti1 E2) with (! ti1 E1) ∨ ti2 (! ti1 E2)
    3. replace the pattern ! ti1 (E1 ∨ ti1 E2) with (! ti1 E1) ∧ ti2 (! ti1 E2)
    4. replace the pattern ! ti1 (E1 « ti1 E2) with (! ti1 E1) ∨ ti2 (! ti1 E2) ∨
       ti2 (E2 « ti1 E1)
    5. replace the pattern ! ti1 ((E1 « ti1 E2) < ti1 (! ti1 E3)) with (! ti1 E1)
       ∨ ti2 (! ti1 E2) ∨ ti2 (E2 « ti1 E1) ∨ ti2 (E1 « ti1' E3 « ti1'' E2), where
       ti1' + ti1'' = ti1;
    6. replace the pattern: ! ti1 (+ ti1 E) with ! ti1 E
    7. replace the pattern ! ti1 (m $ ti1 E) with m-1 & ti1 E
    8. replace the pattern ! ti1 (m & ti1 E) with m+1 $ ti1 E
    9. replace the pattern ! ti1 (m # ti1 E) with (m-1 & ti1 E) ∨ ti2 (m+1 $ ti1 E)
    10. change the time interval of ! operators that have as operand a
        composite event to be equal to the time interval of the operator of
        their operand.
}
```

Figure 4. Transformation Algorithm

In steps 1, 2, 3, 4, 5, 9 of the algorithm above the time interval ti2 of the transformed expressions is of zero length: |ti2| = 0. Also, note that the ∨ (disjunction) operator in step 9 serves as an exclusive OR. Step 10 is associative and adapts the time interval of the ! operators to the interval in which the non-occurrence of their operand should be evaluated. After the transformation the 'not' operators have as operands only simple events.

The initial event expression is semantically equivalent to the transformed expression. For details about the semantic equivalence of the expressions before and after the transformations see the Appendix.

Note that it is meaningful to have composite events of the ! operator with composite operands of a different (smaller) time interval that the first. For example: $!_{ti1}(E1 \wedge_{ti2} E2)$ where ti2 < ti1 is meaningful and means: "we do not want in a time interval equal to ti1 both E1 and E2 to occur with a time difference equal to ti2". However, pushing down the ! in this event would produce the transformation: $(!_{ti1} E1) \vee (!_{ti1} E2) \vee (E1 \wedge_{ti3} E2)$, where ti3 should be a special time interval that would denote a lower as well as an upper bound. The lower bound of ti3 should be: |ti2| < |ti3|. Moreover, it is obvious that |ti3| < |ti1|. The meaning of the latter is: "we want either, both E1 or E2 to occur in a time interval equal to ti1 with a time difference bigger than ti2, or, at least one of them should not occur." However, our definition of the ti does not allow for such a lower bound in the time distance of operands. Also, a series of such transformations might expand the event expression a lot. Furthermore, we believe that such granularity of detailed expression is not necessary for real-life situations of complex events. Nevertheless, for completeness we note that even cases of events like the described: $!_{ti1}(E1 \wedge_{ti2} E2)$ where ti2 < ti1, can be rewritten as: $(!_{ti1} E1) \vee (!_{ti1} E2) \vee (Te \wedge_{ti1} (!_{ti}(E1) \vee_{ti2} !_{ti'}(E2)))$ where Te is a time event (relative: it will be bound to the start of the ti1 time interval) and ti, ti' is the time interval of the operator of E1, E2, respectively. The latter form expresses precisely the initial event. Moreover, this form is compliant with the semantics of our event language. However, this rewriting is hard to be evaluated and will probably need associative messages between involved databases in order to instantiate the variable Te (see section 4.4). Note that, depending on the implementation, the user may be able to choose if and how low to push the 'not' operators.

3.3 Event Decomposition Algorithm

The transformed event expression is broken into sub-rules, one for each database involved in the event expression. Each one of them is sent and installed in the appropriate database. The algorithm in Figure 5 describes the generation of event expressions of these sub-rules:
A decomposed sub-event concerning one database matches at least the same combinations of event instances as those matched by the original event expression. Transforming and decomposing the condition of the ECA rule is done in the obvious way of first order logic.

Each one of the sub-events produced by the decomposition algorithm is able to detect the event instances as the original event expression for the specific database for which it was created. In other words, the evaluations of event instances according to the sub-event or the original event expression of this database are the same. However, because of the 'not' operator we need to preserve in a sub-event time variables for the time occurrences of events in other databases.

Even though this transformation is correct with respect to equivalence, in practice it might create a problem. In order to obtain values for these variables, the databases need to exchange extra messages, which add to the communication load. The goal of the evaluation of the event part is to keep the lower bound of the exchanged messages that can produce an event instance of the original rule equal to the number of involved databases minus one (one less because the master[1] does not send a message to another database), thus, to have one such message from each involved database beyond the master one. In order to maintain this number, we have to eliminate the messages with information on time variables; thus, we have to eliminate time variables.

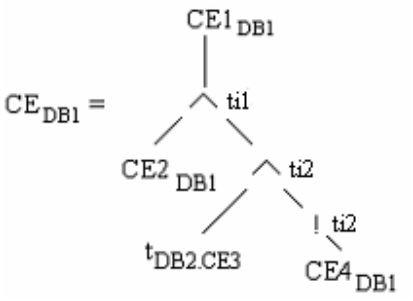
[1] Here and in the following we use the term 'master' database to denote the database where a rule is declared, and which has the responsibility of the coordination of the evaluations of the respective sub-rules.

For (each database involved in the event expression)
{ -- In the event expression replace the simple events that do not belong to this database with time variables that represent their time of occurrence.
-- Simplify the event expression according to the following rules:
1. Eliminate the nodes of all the unary operators that are left with no operand (i.e. their operand is a time variable).
2. Eliminate the nodes of all the binary operators except $\vee$ (disjunction) that are left with one operand, that is not a composite event of the ! (not) operator, and all the binary operators that are left with no operands. For the latter cases, substitute the time variable with 'null'.
3. For all operators that have operands changed because of elimination, except the $\vee$ ones, and except the « ones with elimination under their right operand, change their time interval, ti, to the sum of their initial ti and the ti intervals of the 'underlying' eliminated operators, (for one exception, see proof).
4. If there is a < (strict sequence) operator eliminate the part that has no information relevant to this database. Eliminations are performed according to steps a, b, c.
a. If the left operand, thus the « (loose sequence) operator is left with two operands, but the right, thus the ! operator is null, then we substitute the strict sequence with its loose sequence operand.
b. If the left operand, thus the « (loose sequence) operator is left with no operands, but the right, thus the ! operator, has a not null operand, then we substitute the < with a $\wedge$ (conjunction). The one operand of the latter is the time occurrence of the right operand of the « and the other the ! part of the <. We change the time interval of the ! to [t_{lsleft}, $t_{lsright}$], where t_{lsleft} is the occurrence time of the left operand of the « of the < and $t_{lsright}$ the right.
c. If the left operand, thus the « operator is left with one operand, and the right, thus the ! operator, has a not null operand, then we substitute the with a new « operator with one operand being the operand of the « of the < and the other being the ! operator of the <. If the not null operand of the « of the < is the left one, then it is also the left operand in the new «. If it is the right, then the opposite holds. The time interval of the new « operator is equal to that of the <.
5. Change the « operators that have binary operator eliminated in their right operand to a $\wedge$ operator.
6. Change the ! operators to * ones with the same time interval.
}

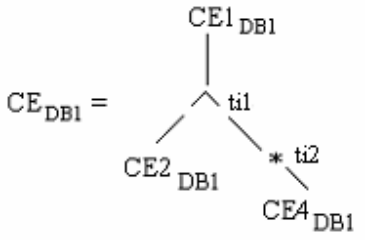
Figure 5. Decomposition Algorithm

From the rules of step 4.c in the event decomposition algorithm, it is obvious that in a sub-event time variables occur only as operands of binary operators where the other operand is a composite event of the ! operator. Actually, these time variables have been preserved in order to help in the correct evaluation of the ! operator: their role is to offer a reference point in time, and as we have seen, we cannot evaluate a ! operator without it. But, what if we could postpone the evaluation of the ! for later, and perform it in the master database using the initial event expression of the rule? That would eliminate the need to wait and accumulate information in the database of the respective sub-event about event instances in other databases. Of course, some information about the operand of the ! operator should be captured by the sub-event and sent to the master database, because otherwise the latter will not have any information to rely on about the evaluation of the !. What we can do is gather some instances of the operand of the !, and send them without attempting to evaluate them. For that, we can use the * operator, whose role is exactly this: to

report instances of its operand, if there are any. Suppose we have the following sub-event to be evaluated in DB1 (for representation explanation see the Appendix):

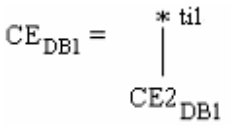


Instead of keeping a time variable for the occurrence time of DB2.CE3 in the sub-event of DB1, we could just keep track of the instances of $CE4_{DB1}$ (i.e., the part of CE4 that concerns DB1) in a time interval big enough to gather the information needed for the evaluation of all the possible cases of !CE4. It is obvious that !CE4 could occur in a time distance as long as ti1 after CE1, or at a distance as long as ti1+ti3 before CE1. Thus, we need to keep the instances of CE4, if there are any, in this period. The sub-event for DB1 for this example is formed as:



In this way, we can eliminate all the time variables. However, there is one case where it might be wise to keep the time variable. Consider the following event expression:

In this case, according to the practical transformation proposed in this section the sub-expression for DB1 should be:



If the master database is DB2, a message would be sent from DB1 to DB2 whenever the system checks for an occurrence of CE_{DB1}. This message consists of the occurrences of $CE2_{DB1}$ in the previous ti1 time units. So, even though we have eliminated the time event $t_{DB2.CE1}$, we actually need another time event from the system which will trigger the check for instances of CE_{DB1}. In this way, even though we avoid the one message with the value of $t_{DB2.CE1}$ sent to DB1 by DB2, we may end up with more messages sent from DB1 to DB2 with useless information. In cases like this it is probably better to keep the time event of another

database in a sub-rule. Nevertheless, the implementation of the * operator depends on the application and is decided by the designer of the latter.

3.4 Decomposition of condition and action expressions

In contrast with the event, the condition and the action expressions of a rule are very easy to decompose to equivalent sub-expressions for each involved database. The reason is that operands that form the composite expressions from the simple ones are only the Boolean operators in the case of the condition expression, and only the AND Boolean operator in the case of an action. Thus, the decomposition of both of them is straightforward. For condition expressions the steps of the decomposition procedure are:

Decomposition of the condition expression
For each database involved do:

1. *Replace the operands or parts of them that concern other databases with null.*
2. *Eliminate the NOT operators of which the operand is null.*
3. *Eliminate the AND operators that are left only with one operand (the other being null)*
4. *Eliminate the AND and OR operators that are left with no operands (both of them are null).*

In step 1 of the above procedure we are talking about 'operands or parts of them' that should be set to null. We remind the reader that even a simple condition can involve variables from more than one database. For example a simple condition could be:

DB1.string1 == DB2.string2

Thus, even part of a simple condition can be set to null. In this example if we were constructing the condition sub-expression for DB1 we would have:

DB1.string1 == null

Note that this new simple condition is not considered to be null. Thus, if it is the operand of a NOT or an AND operator, the latter cannot be eliminated (according to steps 2 and 3).

As we can observe from the decomposition procedure the OR operators that are left with one operand are not eliminated. The reason is the same as for the respective strategy in the decomposition of the event expression: In these cases the OR operator is necessary in order to report instances of its remaining operand, which may be useful for the production of global instances. (See proof of rule 3 of the decomposition algorithm in the Appendix).

Decomposition of the action expression
The decomposition of the action expression is even more trivial. We just keep the actions that concern the database for which we are building the sub-expression. Note that the decomposition of the action is performed after the complete evaluation of the event and the condition expression. Thus, all the variables in the action expression are bound to values and there is no case of insufficient information in the decomposed action sub-expressions.

3.5 A Rule Decomposition Example

We consider a sample rule from the health care domain of Section 1. Suppose that until the end of the second day of a patient's admission to the hospital we want a trigger when she has not had either a prescription for a specific drug by Dr. Davis in the past 12 days or less than 2 epileptic crises. Then, if the patient has not been under some other medication in the past 12 days, we would like to give this specific drug to the patient. The ECA rule is:

Rule 4:

when TGHDB.('insert', (Admission, (_,<OHIP#_value >, _, _, _, _))) $«_{(0,0,0,2,-,0,0)}$ $(!_{(0,0,0,12,-,0,0)}$
$((_2 \&_{(0,0,0,2,-,0,0)}$ TGHDB.('insert',('MedRec', (OHIP#_value, _, _, 'epileptic crisis'))))
$\vee_{(0,0,0,12,-,0,0)}$ DavisDB.('insert', (Prescription, (<OHIP#_value>, <DrugID_value>, _,_,_)))

if (DrugSet1 = ∅) AND (DrugSet2 = ∅)
(*where* TGHDB.('retrieve', 31, {OHIP#_value, DrugSet1}) *and*
DavisDB.('retrieve', 25,{OHIP#_value, DrugSet2})

then TGHDB.('insert', ('Medication', (OHIP#_value,
<currentDate>, <currentTime>, <DrugID_value>, _)))

Query 31:
Select DrugID, *From* Medication
Where OHIP#=OHIP#_value

Query 25:
Select DrugID, *From* Prescription
Where OHIP# = OHIP#_value, Date > <12 days ago>

Here, the information retrieved by query 31 and query 25 is stored in DrugSet1 and DrugSet2, respectively. The generated sub-rules for TGHDB and DavisDB are:

TGHDB Sub-rule:
when TGHDB.('insert', (Admission, (<OHIP#_value >, _, _, _, _))) $\wedge_{(0,\,0,\,0,\,2,-,\,0,\,0)}$
$(_2\ \$\ _{(0,\,0,\,0,\,2,-,\,0,\,0)}$ TGHDB.('insert','MedRec', (OHIP#_value, _, _, 'epileptic crisis'))))
if (DrugSet1 = ∅) where TGHDB.('retrieve', 31, {OHIP#_value, DrugSet1})
then *propagate the sub-rule instance*

DavisDB Sub-rule:
when $*_{(0,\,0,\,0,\,12,-,\,0,\,0)}$ DavisDB.('insert', (Prescription, (<OHIP#_value>, <DrugID_value>, _,_,_)))
if (DrugSet2 = ∅) where DavisDB.('retrieve', 25, {OHIP#_value, DrugSet2})
then *propagate the sub-rule instance*

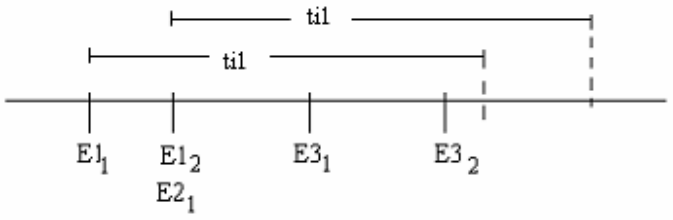
3.6 Event Instance Consumption

An important issue in all the rule mechanisms is how the event instances are consumed by the event expressions of the rules. In rule mechanisms that are designed for a centralized database, the choice of the event consumption policy is a matter of which composite event instances would be more convenient according to specific characteristics of the information in the database and the goal of the rule mechanism. However, in our case the distributed nature of the event evaluation adds to the importance of the event instance consumption matter. This is because the consumption mode affects the correctness of the composite event instances that are produced.

Imagine the case of the following event expression:

$$CE = (DB1.E1 \wedge_{(0,0,0,0,\_,0,0)} DB2.E2) \ll_{(0,0,0,\_,7,0,0,0)} DB1.E3$$

which describes that E3 should occur after a simultaneous occurrence of E1 and E2. The sub-event for DB1 is: CE_{DB1} = DB1.E1 $\ll_{(0,0,0,\_,7,0,0,0)}$ DB1.E3 and the sub-event for DB2 is: CE_{DB2} = DB2.E2
Suppose that the sequence of event instances is the following:



where ti1 = (0,0,0,_,7,0,0,0). The above figure shows two instances of E1: $E1_1$ and $E1_2$, which occur before two instances of E3: $E3_1$ and $E3_2$, and one instance of E2, $E2_1$, that co-occurs with the second instance of E1. If the consumption policy is such that an instance is used only once in a specific event (or sub-event) expression, then there would be two instances of the sub-rule of DB1:

a. $Inst1(CE_{DB1}) = DB1.E1_1 \ll_{(0,0,0,\_,7,0,0,0)} DB1.E3_1$
b. $Inst2(CE_{DB1}) = DB1.E1_2 \ll_{(0,0,0,\_,7,0,0,0)} DB1.E3_2$

and one instance of the sub-rule of DB2:

$$Inst(CE_{DB2}) = DB3.E2_1$$

The evaluation of the initial event expression would first try to match $Inst1(CE_{DB1})$ and $Inst(CE_{DB2})$. This matching would not be successful because $DB1.E1_1$ does not co-occur with $DB2.E2_1$. Then the evaluation procedure would try to match $Inst2(CE_{DB1})$ with $Inst(CE_{DB2})$. This matching would be successful because $DB1.E1_2$ and $DB2.E2_1$ co-occur. Thus, an instance of CE would be produced that would be comprised of $DB1.E1_2$, $DB1.E3_2$ and $DB2.E2_1$:

$$Inst(CE) = (E1_2 \wedge_{(0,0,0,0,\_,0,0)} E2_1) \ll_{(0,0,0,\_,7,0,0,0)} E3_2$$

Now assume that the events E1, E2, E3 belong to the same database; thus, there are no sub-rules and the evaluation procedure tries to match instances of simple events directly in the initial event expression. It is obvious that in this case, according to the same sequence of instances, $E1_2$ and $E2_1$ would match in CE (because they co-ccur) and then, when $E3_1$ would occur, it would also match the initial expression. Thus the instance produced would be:

$$Inst'(CE) = (E1_2 \wedge_{(0,0,0,0,\_,0,0)} E2_1) \ll_{(0,0,0,\_,7,0,0,0)} E3_1$$

As we can observe Inst(CE) and Inst'(CE) are not the same: the first contains $E3_2$ and the second $E3_1$. The goal of the distributed evaluation procedure of an event expression (and furthermore of a whole rule) is to minimize the communication load, but also to produce results that would be the same as the ones produced by a centralized procedure, as the evaluation procedures of rule mechanisms of centralized databases. Thus, we want our way of event processing to 'behave' as if all the event instances were in the same database. Hence, the difference between Inst(CE) and Inst'(CE) is totally undesired. In order to solve this problem, we have to change the consumption policy in the processing of event instances for the evaluation of sub-rules (only). The most straightforward solution is to keep all the matches of simple event instances in the time interval of the operator as operands of which they are matched. According to this policy, in the above example we would have the following instances of the sub-rule of DB1:

a. $Inst1(CE_{DB1}) = DB1.E1_1 \ll_{(0,0,0,\_,7,0,0,0)} DB1.E3_1$
b. $Inst2(CE_{DB1}) = DB1.E1_1 \ll_{(0,0,0,\_,7,0,0,0)} DB1.E3_2$
c. $Inst3(CE_{DB1}) = DB1.E1_2 \ll_{(0,0,0,\_,7,0,0,0)} DB1.E3_1$
d. $Inst4(CE_{DB1}) = DB1.E1_2 \ll_{(0,0,0,\_,7,0,0,0)} DB1.E3_2$

The evaluation procedure of the initial event expression would first try to match $Inst1(CE_{DB1})$ and $Inst(CE_{DB2})$ unsuccessfully, then $Inst2(CE_{DB1})$ and $Inst(CE_{DB2})$ again unsuccessfully and finally $Inst3(CE_{DB1})$ and $Inst(CE_{DB2})$ successfully. So the instance of CE produced would be the same as if E1, E2, E3 were defined in the same database. Note that if the time distance between $E1_1$ and $E3_2$ was longer than ti1 = (0,0,0,_,7,0,0,0), the instance $Inst2(CE_{DB1})$ would not exist.

Lets consider another example. Suppose that the event expression of a rule is:

$$CE = (_3 \&_{ti1} (DB1.E1 \wedge_{ti2} DB2.E2)) \wedge_{ti3} DB2.E3$$

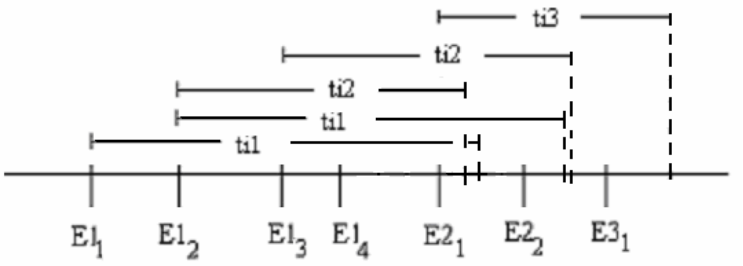
The produced sub-events are:
The sub-event for DB1 is:

$$CE_{DB1} = {}_3 \,\&\, {}_{ti1+\,ti2}\ DB1.E1$$

And the sub-event for DB2 is:

$$CE_{DB2} = ({}_3 \,\&\, {}_{ti1+\,ti2}\ DB2.E2) \wedge {}_{ti3+\,ti2}\ DB2.E3$$

Also suppose that the global sequence of simple event instances is as in the following figure:



Because CE_{DB1} is instantiated even if there is only one instance of E1, in DB1 the following instances of CE_{DB1} would occur:

a. $Inst1(CE_{DB1}) = {}_3 \,\&\, {}_{ti1\,+\,ti2}\ \{E1_1\}$
b. $Inst2(CE_{DB1}) = {}_3 \,\&\, {}_{ti1\,+\,ti2}\ \{E1_2\}$
c. $Inst3(CE_{DB1}) = {}_3 \,\&\, {}_{ti1\,+\,ti2}\ \{E1_3\}$
d. $Inst4(CE_{DB1}) = {}_3 \,\&\, {}_{ti1\,+\,ti2}\ \{E1_4\}$

Also in DB2 one instance of CE_{DB2} would occur:

$$Inst(CE_{DB2}) = ({}_3 \,\&\, {}_{ti1\,+\,\,ti2}\ \{E2_1, E2_2\}) \wedge {}_{ti3+\,ti2}\ E3_1$$

Thus, the evaluation of the initial event expression would match $Inst1(CE_{DB1})$ with $Inst(CE_{DB2})$. However, the instance $E2_2$ of $Inst(CE_{DB2})$ would have to be thrown away because it cannot be matched with an instance of E1. The instance of CE would be formed as follows:

$$Inst(CE) = ({}_3 \,\&\, {}_{ti1}\ \ \{E1_1 \wedge {}_{ti2}\ E2_1\}) \wedge {}_{ti3}\ E3_1$$

However, assuming that all the instances of the above figure occur in the same database, the instance of CE that would occur is:

$$Inst'_{CE} = ({}_3 \,\&\, {}_{ti1}\ \ \{(E1_2 \wedge {}_{ti2}\ E2_1), (E1_3 \wedge {}_{ti2}\ E2_2)\}) \wedge {}_{ti3}\ E3_1$$

As we can see Inst(CE) and Inst'(CE) are not the same. The solution, as we have seen in the previous case, is to 'consume' the simple event instances in the evaluation of the sub-events more than once following the time restrictions of the time intervals of the operators. Thus, in DB1 the instances of CE_{DB1} that should be produced are:

a. $Inst1(CE_{DB1}) = {}_3 \,\&\, {}_{ti1\,+\,ti2}\ \{E1_1\}$
b. $Inst2(CE_{DB1}) = {}_3 \,\&\, {}_{ti1\,+\,ti2}\ \{E1_1, E1_2\}$
c. $Inst3(CE_{DB1}) = {}_3 \,\&\, {}_{ti1+\,ti2}\ \{E1_1, E1_2, E1_3\}$
d. $Inst4(CE_{DB1}) = {}_3 \,\&\, {}_{ti1+\,ti2}\ \{E1_2\}$
e. $Inst5(CE_{DB1}) = {}_3 \,\&\, {}_{ti1+\,ti2}\ \{E1_2, E1_3\}$
f. $Inst6(CE_{DB1}) = {}_3 \,\&\, {}_{ti1+\,ti2}\ \{E1_2, E1_3, E1_4\}$
g. $Inst7(CE_{DB1}) = {}_3 \,\&\, {}_{ti1+\,ti2}\ \{E1_3\}$
h. $Inst8(CE_{DB1}) = {}_3 \,\&\, {}_{ti1+\,ti2}\ \{E1_3, E1_4\}$
i. $Inst9(CE_{DB1}) = {}_3 \,\&\, {}_{ti1+\,ti2}\ \{E1_4\}$

From the instances of CE_{DB1}, the evaluation of the initial rule would match the earliest and biggest one that can be matched with one of the instances of CE_{DB2}.

The above policy of the evaluation of the sub-events seems to have a big disadvantage: It overloads the communication between the master database of a rule and the rest of the databases involved in the event expression of this rule with useless messages: according to the communication algorithm of section 3.1 a message should be sent for every instance of a sub-rule (thus, disregarding the condition part, for every instance of a sub-event). For example, in the situation described above, if DB2 is the master database, nine messages should be sent, even though the information of only one of them would be used. In order to solve this problem, we could group the information in fewer messages, according to the time interval of the 'root' operator or/and specific characteristics of the latter. In the above case we could group the instances according to the time interval ti1: thus we would send all nine instances of CE_{DB1} in one message; or according to the maximum number of the & operator: thus, we would send (a), (b), (c) in one message, (d), (e), (f) in a second one, (g), (h) in a third one and (i) in a fourth one. Moreover, additional techniques can be used in order to further reduce the number and size of exchanged messages. For example, instead of sending all (a), (b) and (c) instances only the last one could be transmitted. Nevertheless, how event instances are transmitted depends on the implementation and is out of the scope of this paper.

Note that, up to this point, we have assumed that the evaluation procedure in the master database of a rule is simple, in the way that it tries to match only one instance of each sub-event in the initial event expression. However, an alternative solution to the event consumption problem could be a more sophisticated matching of the sub-event instances. For example in the first case described in this section, the evaluation procedure in the master database could try to use parts of the information available in the instances: $Inst1(CE_{DB1}) = DB1.E1_1 \ll_{(0,0,0,\_,7,0,0,0)} DB1.E3_1$, $Inst4(CE_{DB1}) = DB1.E1_2 \ll_{(0,0,0,\_,7,0,0,0)} DB1.E3_2$ keeping the earliest simple event instances. Also, in the second case, the evaluation procedure could try to combine the information of: $Inst1(CE_{DB1}) = {}_3\,\&_{ti1+ti2}\,\{E1_1\}$, $Inst2(CE_{DB1}) = {}_3\,\&_{ti1+ti2}\,\{E1_2\}$, $Inst3(CE_{DB1}) = {}_3\,\&_{ti1+ti2}\,\{E1_3\}$. Nevertheless, this solution is harder to be designed in detail. A 'clever', complete and correct in any case matching algorithm is not so trivial to invent. Moreover, the sequence of messages expected and the waiting time needed in every case by the master database should be studied thoroughly.

3.7 Trade-offs of Rule Decomposition

As we have seen, a distributed rule that involves a number of databases in the event and the condition parts can always be decomposed into sub-rules, one for each database, that can be evaluated locally. The main achievement of decomposition is the minimization of the number of exchanged messages among the databases. The obvious trade-off of the rule decomposition is that the master database affords the cost of the rule decomposition. Moreover, the master endures the cost of composing the sub-rule instances into instances of the original rule. Sub-rule instance composition includes (a) keeping track of non-obsolete sub-rule instances and, (b) extracting the simple event instances of all available sub-rule instances and composing them into one or more instances of the initial rule. Beyond the processing trade-off on the master, the rest of the involved databases must support techniques that group event instances into packages to be sent to the master, in order to minimize the number of messages. As one may expect in a distributed computation, we encounter the usual trade-off between the number of exchanged messages and local processing costs.

4. Implementation

The rule mechanism presented in Sections 2 and 3 is partially implemented. The platform used for the implementation is Java™ 2 SDK, Standard Edition Version 1.2.2. The application is intended to run on top of a relational DBMS. We choose the layered architecture instead of building the rule language in a DBMS

for several reasons[2]. Thus, an application running on top of the DBMS is easier to install without making any changes that affect the kernel of an existing DBMS. In addition, this way is more secure in that authorization and user-privilege problems are avoided. Of course, this approach also has disadvantages. In particular, we can expect sub-optimal performance because of communication overhead between the application and the DBMS. Also, there is unavailability of some important features, such as concurrency control and authorization for rules, as well as coupling modes, which need to access certain elements of the DBMS implementation. However, our implementation is not complete and moreover, as it is, aims to test the feasibility of the presented distributed rule evaluation mechanism. Hence, we do not deal with problems concerning the layered architecture. We choose the latter for rapid prototyping.

The layered architecture ensures a separation from the ECA rule module from the underlying query processor, so that the rule module only interfaces with the query processor in order to check conditions and execute actions of ECA rules. This separation minimizes the impact of rule processing on query optimization in each participating database system. The optimization done at the ECA rule processing level deals with the decomposition of events, condition, and action expressions, separately from the optimization done at the level of participating database systems.

In order to make the implementation simpler and to facilitate the creation of testing scenarios we actually did not use a real relational database but a simulation of it: files that store series of simple event instances and others that store simple condition instances are used for this purpose. Also, log files are used to record the rule instances. In testing, several instances of the application are used, where each one is supposed to reside on top of a separate database. These instances establish communication among them and exchange information. Additionally, an associative program is used in order to provide the appropriate environment for the execution of the rule mechanism.

4.1 Event Detectors

Event detectors recognize simple and/or composite event instances. A local event detector recognizes the respective instances of simple/composite sub-events that belong to sub-rules handled by a specific database, (actually, instance of the application). The global event detector collects instances sent from local event detectors and composes event instances of the original rules. Figure 6 presents a general architecture of the rule mechanism. The ellipses named ‘clients’, ‘servers’ and ‘remote database’ represent instances of the rule mechanism playing different roles in an execution.

4.1.1 Local Event Detector

For each unique sub-event[3] the local event detector keeps track of the instances of every unique simple event that is of interest (i.e., contained in the sub-event). When a new instance of a simple event is received, it is stored in an appropriate vector, and triggers the local evaluation procedure. The latter first checks if the received instance has a time distance with the earliest occurred instance still held bigger that the possible maximum time distance[4]. If it does, it sends the sub-event instances accumulated till that time to the master database of the original rule. Also, it finds the obsolete instances and throws them away. The obsolete instances are the ones that because of their old time of occurrence they cannot be combined with newly received instances. The criterion for this is the result of the comparison of the difference of the time

[2] Three types of architecture exist in active database systems: the layered, where all the active components reside ‘on top’ of the database; the built-in, where the database system is modified in order to include the active components; and the compiled, where no run-time activity is required and the active features are added to database application at the compile time.

[3] Two sub-rules can have the same event part but different condition parts; thus, for both of them the system has to recognize the same sub-event, but finish the partial evaluation differently (i.e. initialize the condition part differently).

[4] The possible maximum distance is calculated by traversing in depth the sub-event tree and adding the ti time intervals of the operators following the path that gives the biggest result.

occurrences of the last occurred instance and the examined for obsolesce instance, with the possible maximum time distance of the simple events of this specific sub-event expression. After that, it makes all the combinations of the simple event instances that have not been realized in a previous execution of the evaluation procedure, and adds the valid ones in a vector where the sub-event instances are stored. However, simple event instances may occur rarely, (for example in the experiments the number of simple event instances is limited; thus, they stop occurring at some point). Because of this, the transmission of the sub-event instances to the master database may be delayed, (or in the case of the experiments, the last group of instances will never be sent). To solve this problem, we run an additional procedure that checks regularly if the vector with the sub-event instances is not empty and in this case it sends it to the master database.

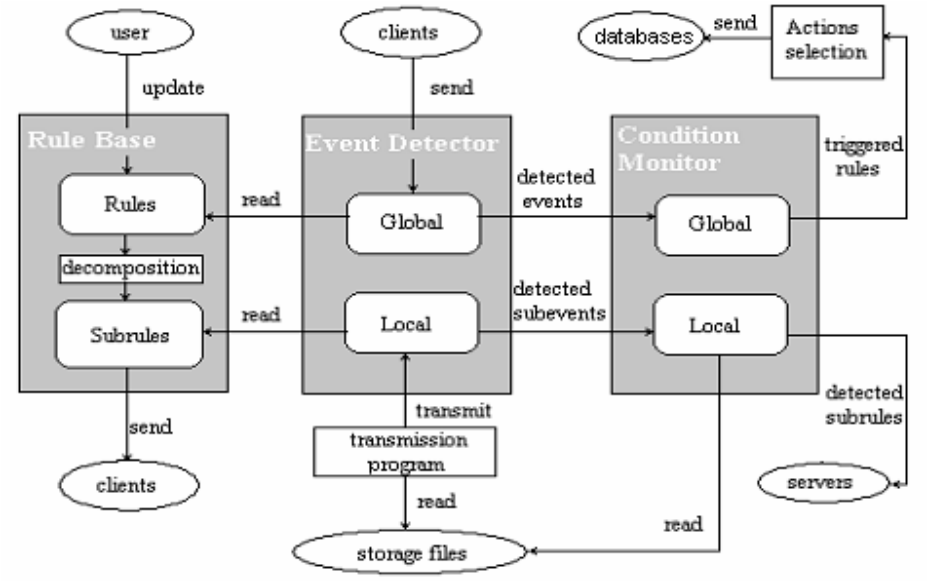


Figure 6: General architecture of the rule mechanism.

4.1.2 Global Event Detector

The global event detector of an event of a rule receives vectors[5] of sub-event instances of the involved databases. The evaluation procedure stores these instances together with others received previously that have not yet been used. The procedure keeps also track of the simple event instances that have already been used to form valid event instances of the original rule. Thus, when the vector of the new sub-events is received, each one of these instances is checked in order to determine if it contains one of the used simple event instances. If it does not, it is added to the available sub-event instances of the appropriate database. After this the procedure tries to find valid combinations of sub-event instances of all (or some, in case there is a disjunction operator in the event expression of the rule) the involved databases. When a valid event instance is formed, all the simple event instances involved are added to the group of used simple event instances. Then the sub-event instances that are left are checked in order to determine if they contain one of

[5] The vectors also contain condition instances. This is common for all sub-event instances sent as a group. The condition is checked and instantiated right before the sub-event instances are sent to the master database.

these used instances and if they do, they are thrown away. The event detector first tries to match a sub-event instance as it is in the event expression. If this is not possible, it breaks it in the simple event instances it is composed of and tries to match them separately.

4.2 Experimental Setup

Our algorithms have been implemented to compare our distributed rule mechanism with a naïve implementation in terms of communication messages among participating databases. In the naïve implementation, all the simple event instances needed for the evaluation of a specific rule are sent individually, as soon as they occur, to the database that is responsible for the global evaluation of that rule. Also, when the event of the rule is instantiated, information about the condition is requested by the responsible database from the appropriate databases, and the condition instances are sent from the latter ones to the former. Hence, the naïve case has no partial evaluations of sub-rules: the database where the global rule resides collects all the necessary information.

We experiment on four rules that involve two databases DB1 and DB2; the rules are declared in DB1 (master database):

Rule r1:
when DB1.E1 $\wedge_{ti}$ DB2.E2
if *null*
then DB1.A1 AND DB2.A2

Rule r2:
when DB1.E1 $\wedge_{ti}$ DB2.E2
if DB2.C1
then DB1.A1 AND DB2.A2

Rule r3:
when DB1.E1 $\wedge_{ti1}$ (DB2.E2 $\wedge_{ti2}$ DB2.E3)
if *null*
then DB1.A1 AND DB2.A2

Rule r4:
when DB1.E1 $\wedge_{ti1}$ (DB2.E2 $\wedge_{ti2}$ DB2.E3)
if DB2.C1
then DB1.A1 AND DB2.A2

In the rules above, DB1.E1 is a simple event in DB1 and DB2.E2, DB2.E3 are simple events in DB2. Also, DB2.C1 is a simple condition that involves only DB2. Finally, DB1.A1 and DB2.A2 are simple actions to be executed in DB1 and DB2, respectively.

As we can observe, the first and third rules have an empty condition part. With these rules we want to test how the number of event instances influences the number of communication messages between DB1 and DB2. In Rule r1 the event is composed by a simple event from DB1 and a simple event from DB2. Thus, for every event instance of DB2.E2 a new message with this information is sent by DB2 to DB1. In rule r3 there is a simple event from DB1 and a composite (composed of two simple events) event of DB2. Thus, in the case of our rule mechanism, again one message is sent by DB2 to DB1 for every instance of the composite event of DB2. However, in the naïve case one message is sent for every event instance of DB2.E2 or DB2.E3 and the composite event instances of DB2 are formed and validated in DB1. The rules r2 and r4 are similar to rules r1 and r3, respectively, with a condition part referring to DB2. For these rules, the naïve case has additional messages for the request and the accumulation of condition information.

In the experiments, we measure the number of transmitted messages among the databases versus the maximum number of possible sub-rule instances. As discussed in section 4.4, the number of exchanged messages plays the most significant role in the processing cost and it totally depends on the number of sub-rules instances that are transmitted to the master database. Furthermore, the number of sub-rules instances depends on the length of the ti intervals and the number of total simple events as well as if the conditions evaluate to 'true'. If all conditions are always true and all the simple events are suitable to form sub-rule instances, then we get the maximum possible number of sub-rule instances. The parameters of the experiments are the In the experiments we test the two mechanisms for a total of 10, 100, 200, 300 and 400 maximum possible sub-rule instances. For example, in the first test, for rules r1 and r2 we have 10 event instances 5 instances of DB1.E1 and 5 more of DB2.E2. Thus we have exactly 10 sub-rule instances in total for rule r1 and r2. In fact the actual number of sub-rule instances for r1 and r2 is the maximum one. (However, r2 would have a smaller number if we took into account the form of the condition in order not

to communicate sub-rule instances with an invalid condition). As for rules r3 and r4, the test comprises 15 event instances: 5 of each one of DB1.E1, DB2.E2 and DB2.E3. Nevertheless, the maximum number of possible sub-rule instances is 10 and depends on the time interval ti2. Consequently, for rules r1 and r2 the maximum number of possible sub-rule instances coincides with the total number of primitive event instances of DB1 and DB2, whereas, for r3 and r4, given a maximum number of sub-rule instances d, the total number of event instances involved is 3/2d (d/2 DB1 and d DB2 primitive event instances). The reason why we count the exchanged messages versus the maximum number of possible sub-rule instances and not versus the number of primitive event instances, is because the number of rule instances depends totally on the first, but only partially on the latter. For example, for a total of 100 primitive event instances we could have at most 50 rule instances of rule r1 but only 33 of rule r3, (note that we assume that there are 100/2 instances of events DB1.E1 and DB2.E2 in the first case and 100/3 instances of events DB1.E1, DB2.E2 and DB2.E3, in the second).

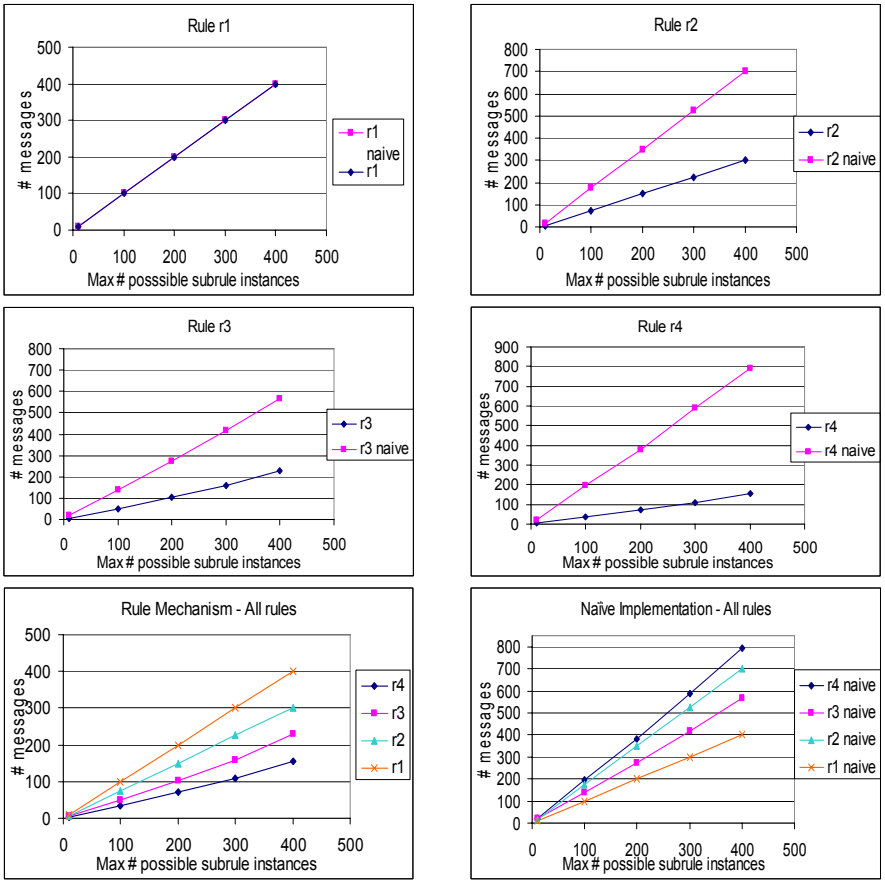


Figure 7: Experimental results

For rules r1 and r2 the time parameter ti of the conjunction operator is set to a value big enough so that none of the possible composite event instances is rejected by the evaluation procedure because of a difference in the occurrence time of the component simple event instances bigger than ti. The same holds for the ti1 parameter in rules r3 and r4. Yet, in these rules we play with the parameter ti2 of the composite sub-event of DB2 in order to observe how the partial evaluations benefit our rule mechanism (in terms of number of messages) versus the naïve implementation. Therefore, we perform the experiments for 3 values

of ti2: 0, 5, 15 seconds. When ti2 equals 0, only the instances of DB2.E2 and DB2.E3 that occur simultaneously form sub-events of DB2 used by the evaluation procedure for rules r3 and r4. Also, when ti2 equals 15 almost all the instances of DB2.E2 and DB2.E3 participate in sub-events of DB2 that are used to form instances of rules r3 and r4. Additionally, for rules r2 and r4 we play with the percentage of valid condition instances of DB2.C1 and perform tests for 100%, 60%, 40%, and 0% of valid condition instances. The number of the latter influences the number of global rule instances.

4.3 Experimental Results

The graphs in Figure 7 show the average number of messages exchanged between DB1 and DB2 versus the number of maximum possible sub-rule instances for the experiments on rules r1, r2, r3, r4 performed using the implementation of the proposed rule mechanism and the naïve one. For each one of the four rules, for a specific maximum number of sub-rule instances, we average the number of messages we get from testing all the possible combinations of the values of the parameters ti2 and the percentage of valid condition instances. Thus, each point in the graphs is: for rule r1, the result of one test; for rule 2, the average of the results of 4 tests; for rule r3, the average of the results of 3 tests; for rule r4, the average of the results of 12 tests.

The first four graphs of Figure 7 compare for each one of the rules the results of the proposed mechanism and the naïve implementation. As we can observe, the two rule mechanisms have exactly the same number of exchanged messages for r1, but have great differences for the rest of the rules. Moreover the difference in the number of exchanged messages is big for rule r3, bigger for r2 and even bigger for r4. A closer observation shows that this occurs because from experiment r1 to r2 or r3 the number of exchanged messages increases for the naïve case whereas it decreases for the proposed mechanism! The reason for this is that the proposed rule mechanism takes advantage of the limitations imposed by the form of the rule in order to diminish the number of messages during the evaluation. Specifically, it takes into consideration that for r2 a sub-rule of DB2 should involve a condition instance. Thus, the partial evaluation procedure in DB2 forms and sends sub-rules (that involve both a primitive event instance and a condition instance), whereas the naïve implementation needs three messages in order to complete the exchange of the appropriate information for a sub-rule instance of DB2 (1 for sending the primitive event instance and 2 for requesting and sending the condition instance). Similarly, in case of the r3 the rule mechanism sends sub-rules of DB2 in which the event involves an instance of DB2.E2 and an instance of DB2.E3, whereas the naïve implementation sends all the primitive event instances of DB2 individually. Moreover, in many cases, the rule mechanism communicates groups of DB2 sub-rule instances instead of sending them one by one. In case of rule r4, where there are restrictions both in the event and the condition part, the difference in the number of exchanged messages becomes even bigger.

The last two graphs compare the results of all the experiments for the proposed rule mechanism and the naïve implementation. We can certify by them the previous conclusion: for the naïve case the number of exchanged messages augments as the number of primitive event instances increases and when there is a condition part in the rule (rules r2 and r4), whereas for the proposed mechanism it decreases when there is a condition part or/ and there is a composite event of DB2.

As we can observe, the experimental results on all four rules are linear, which means that the number of messages is proportional to the maximum number of sub-rule instances. For the first rule the explanation of the linearity is straightforward: all the sub-rule instances of DB2 are sent individually to DB1 and all of them are involved in instances of r1 (thus, for each one of them there is a message containing a DB2 action). In addition, for rule r2, the number of valid condition instances depends on the number of event instances, because we test the system on fixed percentages of valid condition instances. Hence, the number of messages containing DB2 actions depends on the number of sub-rule instances generated from r2. For rule r3, the average number of DB2 sub-rule instances is proportional to the maximum number of sub-rule instances because we kept fixed values for the ti2 (while the frequency of occurrence of the event instances was kept reasonable). Rule r4 is a combination of r2 and r3. From the discussion above we can infer that the linearity of the results would disappear in case of a real workload where the number of valid condition instances and the number of composite sub-event instances would not depend on the number of primitive

event instances. Furthermore, in environments where participant databases come and go, these results would be random.

4.4 Discussion on cost issues

The experimental study made above focuses on the number of exchanged messages among the databases involved in a distributed rule. We have made a common assumption in distributed computation that local computations are more efficient that any message passing. With this assumption, the complexity of the rule (i.e. the complexity of the event and the condition) does not have a significant role in the overall cost. Specifically, the processing time is not influenced in a noteworthy way by the complexity of sub-events and sub-conditions that can be evaluated in a single database. This is why the experimental study does not extend to rules with very complex sub-events and sub-conditions.

Of course, in case of some kind of malfunction in a local processing of a sub-rule (or the processing of sub-rules in the master database) considerable delays in message transmissions can be caused. Such delays can also be caused by extremely complex sub-events or by sub-events that involve very long time intervals. Inevitably, these situations would delay message transmission and further rule processing and, thus, augment the overall processing cost (i.e. time). However, this kind of delays is not due to the distributed nature of the proposed rule processing and would influence to a similar extent the naïve implementation, too. A thorough study of the local bottlenecks in processing cost is out of the scope of this paper.

5. Related Work

Generally, ECA rule functionality has been considered mostly in centralized database environments. However, there have been attempts to integrate ECA rules in distributed environments such as [HSL92], [CW92], [TC97], [CL01], [CGMW94], [ABD+93], [BKLW99], and [KRS99].

To the best of our knowledge, Hsu *et al.* [HSL92] were the first to investigate the use of ECA rules in a distributed environment. They show how to decompose rules by algebraic manipulation, to distribute the resulting smaller rules, and to evaluate the latter. Rule processing however remains centralized in the framework of Hsu *et al.*.

Ceri and Widom [CW92] also investigate the use of ECA rules in distributed environments. Their rules reference data located at multiple sites. A locking-based scheme is used by the rule execution engine to coordinate the participating sites.

In [VCR00], a component-based architecture for an active mechanism is described for federated database systems, where the database involved use a common universe of discourse. The central components of this architecture offer services for event detection and rule processing. The ECA rules executed by the system are global to all the databases involved in the federation, and their conditions and actions may involve subparts destined to different databases. Like their ECA rules, our distributed triggers are global; however, the globality of our triggers is limited to acquainted databases, since we do not permit a participant to the multidatabase system to define triggers that involve participant databases that are not acquainted with the former.

In [BKLW99], a framework for characterizing active functionality in terms of dimensions of a distributed ECA rule language and its execution model is introduced. The authors of [BKLW99] envision the future computer systems as a network of active heterogeneous and autonomous databases with various services. In such systems, the role of ECA rules will be substantial. The problem of incorporating ECA rules in a distributed environment is properly defined and broken into parts in order to discover how distribution affects the ECA rules and their execution model. The authors try to reconsider the techniques used in centralized databases and decide if and how they can be adapted in order to fit the distributed dimension of the problem.

We would like to underline how some of the dimensions of distributed active functionality given in [BKLW99] are realized in our rule mechanism:

Event semantics: It is difficult to define a point in time in the context of a multidatabase system where each participating database has its own local time. To solve this problem, our event operators have a subscripted time interval that can be used by any of the database systems in the multidatabase system to timeout the local detection of events.

Event detection: The rule manager of the database on which a global rule is declared acts as a global rule manager (in the sense of [TC97]) with respect to that rule. Therefore it is responsible for detecting composite events of that rule by composing smaller events that occur on acquainted databases.
Condition evaluation: From the sample scenarios given in Section 1, one sees that conditions are explicitly tied to acquainted databases. Therefore, any such condition is evaluated in the database to which it is tied. In fact, this is not new since one finds this mechanism in multidatabase languages [LMR90], [LSS01]. Also, we perform asynchronous partial condition evaluations.
Execution model: According to the architecture of the proposed mechanism, primitive event instances produced in a site are sent to the local event manager. In a similar way, sub-rule instances are propagated to rule managers that are responsible for the evaluation of global rules. Thus, the system uses the reactive method for event detection and communication of sub-rule instances. Each site owns a global rule manager that maintains a local set of global rules (i.e. rules that involve events, conditions and actions from many sites), which are visible only locally. Finally, the condition evaluation and action execution is achieved by

Event Signaling	Reactive Detection Method Distributed Event Manager
Rule Triggering	Reactive Communication Method Distributed Rule Manager
Scheduling	Local Rule Managers Local Sets of Global Rules Transparent Rule Distribution
Condition Evaluation and Action Execution	Global State Access Multidatabase Environment Coordination with Global Transactions

Figure 8. Features of the execution model.

performing transactions in one or more sites of the multidatabase system. The dimensions of the execution model are summarized in Figure 8.

[KRS99] proposes an approach for managing consistency of interdependent data in a multidatabase environment. In particular, they define a framework for the specification of relationships and consistency requirements between data objects and they design a mechanism in order to maintain mutual consistency. They introduce the 'data dependency descriptors' (D^3s) which are objects distributed among the databases that describe the consistency constraints among data and the restoration transactions. The authors consider loose consistency between one or several source data objects and one target object. This means that they allow consistency to be violated up to a certain specified point (for example for a number of changes in the source objects, or for an amount of time). When a consistency violation has reached its limits, they provoke restoration transactions at the target site. The system, called Aeolos, is built on top of local existing DBMSs and provides procedures for the declaration of D^3s ensuring correctness and monitors for detection of events that occur on interdependent data. Contrary to the D^3s mechanism, our approach is more general by providing distributed ECA rules as a global means of expressing data coordination, including various degrees of consistency enforcement. In the near future, we plan to complete the implementation of an application showing how to enforce consistency constraints similar to those enforced by the D^3s mechanism.

Interesting work is also presented in [CL01], where local and global events are detected asynchronously in a distributed environment and are distributed together with their parameters to sites interested in them. This work is based on the active database system Sentinel and its event specification language Snoop. Both are extended in order to accommodate composite events from various sources and an appropriate execution model. They propose a Global Event Manager (GEM), which receives requests for the detection of global events on behalf of client sites. Local Event Managers (LEMs) reside on the client sites and detect local events that are propagated to the GEM in order to participate in the detection of global events. When a

global event instance is detected the interested clients are notified. In GEM a composite global event tree is composed by various global events that are requested from the clients. This work is similar to ours in its mechanism for distributed event detection. However, it differs from our work in two ways. First, recall that the database on which a newly created ECA rule resides coordinates the global evaluation process of the new rule. The event and rule managers on this database act as global event and rule managers. Therefore, we do not need a dedicated Global Event Manager in the sense of [CL01] which could cause a bottleneck in the network. Second, our execution model distributes entire rules to acquainted databases, not only sub-events. In fact, the evaluation of the condition together with the respective rule event is meaningful in order to decide if the event instance is worth of propagation. In [CLD98] the authors present a prior work related to this one.

Furthermore, a noticeable old work is presented in [ABD+93], where the authors propose a specification for inter-database dependencies in a federated database system and an execution model. The work is based on the quasi-transaction model and on rules from the active databases field. More specifically, the activation of rules in the system provokes the execution of quasi-transactions and transactions. Events that are generated during the execution of quasi-transactions or at the end of the execution of transactions can trigger other rules. Each of the component local database managers of the federated system owns a rule manager and is able to communicate with the quasi-transaction manager and the transaction manager, which are shared elements. We share the motivation and main goals of this work that provides an early distributed ECA framework. However, the level of details of the ECA language described in [ABD+93] does not permit a precise comparison with our language. This work tackles the issue of processing ECA rules within the context of transaction, whereas we do not yet.

The authors in [TC97] present a method that uses ECA rules in order to maintain global integrity in a federated database system. When a global rule that contains events from more than one local databases is declared, local rules are produced in order to propagate primitive relevant events that occur in the member databases. The global rules are stored in the global database that contains the meta information for the federation layer. Furthermore, the local rules produced from the global ones are sent and installed in the local databases of the federation. This approach is similar to that of [CL01], but with a different mechanism for propagating events across acquainted databases: [CL01] uses requests while [TC97] uses propagation rules. Like our approach, [TC97] involves member databases of the federation in the process of event detection. Unlike our approach, [TC97] uses a global database of meta information. Our approach deals with such meta information in a distributed fashion. Moreover, in [TC97] only primitive event instances without any processing are propagated. The system in [TC97] is similar to the naïve implementation we used for comparison in the experiments of Section 4.

Furthermore, in [CVG00] the authors propose open and distributed active services. Specifically, they describe a framework that supports meta-models for event definition and event management, as well as for rule definition, execution and cooperation. The goal of the framework is to provide event and rule behaviors that are adaptable to the specific characteristics of the target application.

Some approaches to workflow systems also provide support for ECA-like rules in distributed environments [CBB03a, CBB03b, JH99]. For example, [JH99] gives an ECA-rule based semantics of the execution behavior of a workflow task for supporting flexible workflows in distributed environments. In this framework, workflows are made of user-defined tasks associated with task behaviors. A task is a graph made of several component nodes that are linked by control flow dependencies as well as by dataflow relationships. The semantics of the control flow dependencies is given in terms of statecharts and ECA rules. In this context, an ECA rule is typically used as follows: a task has a provision of several built-in operations, some of which are transition operations. For every such operation, there is an associated ECA rule. The transition defines the action part of the rule. The task has an event for triggering the ECA rule operation and a guard (i.e. a condition) for that transition to be executed. The execution behavior of the whole task, together with its associated ECA rules, is given in terms of statecharts [Harel87, Harel et al.90].

The use of ECA rules in [JH99] is similar to our distributed ECA rules since task components may be located on different nodes of a network. For example, one component may enable another component located on a remote node of the network. However, the ECA rules used in [Joeris99] are not distributed per

se: the ECA rule associated with the enabling component is completely processed at the node of that enabling component; only an appropriate enabling signal is sent to the node of the enabled component.

ECA rules have been directly associated with statecharts [Harel et al.90]. The specification formalism of statecharts expresses the behavior of a system in terms of the specification of the control flow between the components of the system. A statechart is a finite state machine where each transition is associated with an ECA rule. The condition of the ECA rule acts as a guard for the transition. The effect of the rule is that the start state of the transition is left, and the end state of the trasition is reached whereby the action of the ECA rule is executed. The statechart that is associated with an ECA rule can be considered as it formal semantics. However, statecharts constitute a more elaborate mechanism for modeling reactive systems than database ECA rules because they include the notions of state and superstate. Moreover, to our knowledge statecharts have not been used in a distributed setting, such as the one proposed in this paper.

On the industry side, a few companies, from startups like iSpheres (www.ispheres.com) and KnowNow (www.knownow.com) to middleware giants such as IBM (www.ibm.com/developerworks/library/ws-bpel) have recognized the importance of event-driven applications in the context of the execution of business processes. These kind of applications are seen as increasingly important because the huge size of corporate information that is daily being updated makes it practically impossible for corporate executives to reach rapid decisions based on a quick monitoring of the huge quantity of data being daily updated without appropriate representational models which are, at the same time, tractable.

iSpheres, whose initial research on event processing was conducted in the context of control systems for the US department of defense, is reusing its technology in a vast array of applications such as portfolio management, fraud detection in computer networks, risk management, crisis management, inventory management. All these applications have in common the addition of event processing to the enterprise architecture. iSpheres offers two main products, a rule based language, called the Event Processing Language (EPL), and a server, the EPL server, used to process applications written in EPL. The language EPL is an high level ECA rule language intended to free developers from the burden of writing low level programs. Like in any other ECA rule language, an EPL rule has an event, a condition, and an action part. Events are streams of complex happenings monitored by the system. Actions are application specific responses to events that occurred under well defined conditions. The language EPL is viewed by iSpheres as a complement to IBM's Business Process Execution Language (BPEL), which is an industry wide standardization effort towards a language for programming the automation of the processing of typical business processes.

Applications written in EPL are run on an iSpheres EPL Server. The latter actively monitors (possibly) distributed information sources, detects events specified in users' EPL programs, and responds to the events by either alerting appropriate instances or invoking (possibly) remote business processes.

Like EPL, our ECA rules are distributed in the sense that the database that raises events might be different than the one on which the condition must be evaluated or the one where the actions of the ECA rule must be executed. Moreover, the event detectors in our databases also monitor distributed information sources. However, despite being a commercial product, iSpheres' EPL still has no way of solving the semantic discrepancies that realistically may arise between information sources. Ours is a mechanism that aims at dealing with these discrepancies in the near future for heterogeneous environments that need the type of rules studied in this paper.

6. Conclusions

We have presented a novel distributed ECA rule mechanism for coordinating data in a multidatabase environment. The mechanism consists of a rule language and an execution model that transforms rules to more easily manageable forms, distributes them to relevant databases, monitors their execution and composes their evaluations. The mechanism has been designed in a manner that minimizes the number of messages that need to be exchanged over the network. We have also conducted an experimental evaluation to compare the implementation with a naïve centralized execution model.

Our approach appeals to many parameters, which would make the usability of the language difficult from the average user point of view. However, the focus of this paper was to provide a versatile and

expressive language for distributed active behaviour, and also to provide an execution model for the latter. Ours is a language whose parameters are made explicit for subsequent execution. One may envision an end-user friendly ECA rule language (similar to the iSphere's EPL) which is translatable to the one given in this paper.

Our objective is to later use this mechanism to support coordination among databases in a peer-to-peer (P2P) network [AKK+02], [BGK+02]. Unlike conventional multidatabase systems, the set of participating databases in such a setting is open and keeps changing. So are also the rules that define the degree of coordination among the databases. Moreover, we assume that there is no global schema, control, coordination or optimization among peer databases. Also, access to peer databases is strictly local. These assumptions violate many of the premises of traditional (multi)database systems, where a global schema is taken for granted, optimization techniques are founded on global control, and database solutions are arrived at design time and remain static during run time.

The notion of coupling modes (i.e. the timing of event detection, condition evaluation, and action execution) plays an important role in active rules [Paton99]. The execution model that we presented uses the decoupled coupling mode for both pairs of event-condition and condition-action in order to achieve asynchronous event and condition evaluation, and action execution. However, we intend to follow the idea in [TC97] that tackles the issue of distributed ECA functionality in the context of advanced transactions by systematically studying coupling modes in a distributed context. It is important to notice that it is not realistic to express the active behaviours encompassed in our ECA rules in a procedural transactional program with ACID properties. The reason is that the triggering chain of the ECA rules may be very long and thus induce a very long running ACID transaction. The atomicity of transactions would make the length of transactions unbearable. This is why the so-called advanced transactions [JK97] that relax the ACID properties seem to be the natural way to follow.

For a problem such as the support for distributed rules in a multidatabase environment, mere simulation is not sufficient. A full implementation on a real platform is necessary. Such an implementation would help to do experiments and overhead measurements in the real world. Work is underway to implement the rule mechanism on top of a real, experimental platform for data sharing which is demonstrated in [RGJ+05]. At the time of the writing of this paper, our underlying platform for data sharing did not exist yet. The recent surge of interest in peer-to-peer (hereafter P2P) architectures has served as impetus for renewed interest in distributed ECA rule mechanisms. Peer databases [RGJ+05] are stand-alone databases which have been independently developed and are linked to each other through run-time acquaintances. Unlike multidatabase systems, the acquaintances between peer databases are transient and developed on the fly. Moreover, data instances of peer databases are heterogeneous in general, thus requiring a mechanism for bridging the heterogeneity. Though independent from each other, peer databases need a mechanism for exchanging data among them and coordinating their activities such as querying, and updating data. We envision the use of a mechanism similar to the ECA rules described in this paper for this task, but one that must take heterogeneity into consideration. With such ECA rules, owners of peer databases may write active behaviours for coordinating exchange of data between them.

In practice, we want to design and implement a language that ultimately supports P2P interoperability along the lines of SchemaSQL [LSS01], and at the same time, incorporates distributed active functionality. While multidatabase system languages such as SchemaSQL offer to the user the possibility of explicitly querying and manipulating both data and schemas across the multidatabases, locality of user interaction forces P2P multidatabase languages to hide most of these facilities. This can only be possible if we automate [ERS99] these facilities. One step in that direction is made by the work reported in [KAM02] where the problem of semantic heterogeneity in a homogeneous context is solved by using mapping tables.

Acknowledgements

This research has been partly supported by the European Union - European Social Fund & National Resources under the PENED/EPAn program (Greek Secretariat of Research and Technology).

References

[ABD+93] R. Arizio, B. Bomitali, M.L. Demarie, A. Limongiello, and P.L. Mussa. Managing inter-database dependencies with rules + quasi-transactions. In *3rd Intern.Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, pages 34--41, Vienna, April 1993.

[AKK+02] Arenas M., Kantere V., Kementsietsidis A., Kiringa I., Miller R., Mylopoulos J.. Research Issues in Peer-to-Peer Multidatabase Systems. Tech. Report, Dept. Comp.Science, University of Toronto, 2003.

[BGK+02] P. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini. Data Management for Peer-to-Peer Computing: A Vision. In Proceedings of the *2nd Workshop on Databases and the Web*, 2002.

[BKLW99] G. von Bültzingslöwen, A. Koschel, P. C. Lockemann, and H. Walter. ECA Functionality in a Distributed Environment. In [Paton99], pages 147--175.

[CBB03a] M. Cilia , C. Bornhövd , A. P. Buchmann, Moving Active Functionality from Centralized to Open Distributed Heterogeneous Environments,. In *Proceedings of the 9th International Conference on Cooperative Information Systems*, p.195-210, September 05-07, 2001

[CBB03b] M. Cilia, C. Bornhövd, A. Buchmann, CREAM: An Infrastructure for Distributed, Heterogeneous Event-based Applications. In *Proceedings of COOPIS'03*, volume 2888 of LNCS, p. 482-502, Catania, Italy, November 2003. Springer.

[CL01] S. Chakravarthy and H. Liao. Asynchronous monitoring of events for distributed cooperative environments. In *Intern. Symposium on Cooperative Database Systems for Advanced Applications,* 2001.

[CDL98] S. Chakravarthy, R. Le, R. Dasari. ECA Rule Processing in Distributed and Heterogeneous Environments. *In Proc. of the 14th Intern. Conference on Data Engineering*, Florida, USA, February 1998.

[CGMW94] S. Chawathe, H. Garcia-Molina, and J. Widom. Flexible constraint management for autonomous distributed databases. In *Bulletin of the IEEE Technical Committee on Data Engineering*, 17(2):23--27, 1994.

[CPM96] R. Cochrane, H. Pirahesh, and N. Matos, Integrating Triggers and Declarative Constraints in SQL Database Systems. In Proceedings of the 22nd VLDB Conference, pages 567--578.

[CVG00] C. Collet, G. Vargas-Solar, H. Grazziotin-Ribeiro Open Active Services for Data-Intensive Distributed Applications. In Proc. of the Intern. Database Engineering and Applications Symposium IDEAS), 2000.

[CW92] S. Ceri and J. Widom, Production rules in parallel and distributed database environments. In Proceedings of VLDB, pages 339--351, 1992.

[ERS99] Elmagarmid A., Rusinkiewicz M., and Sheth A., *Management of Heterogeneous and Autonomous Database Systems*, Morgan Kaufmann Publishers, 1999.

[GZ02] F. Giunchiglia, I. Zaihrayeu. Making Peer Databases Interact - a Vision for an Architecture Supporting Data Coordination. In Proceedings of the *Conference on Information Agents*, Madrid, 2002.

[Harel87] D. Harel: Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming* 8(3): 231-274 (1987).

[Harel et al.90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. B. Trakhtenbrot, STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Trans. Software Eng.* 16(4): 403-414 (1990).

[HSL92] I.M. Hsu, M. Singhal, and M.T. Liu, Distributed rule processing in active databases. In Proceedings of ICDE, pages 106--113, 1992.

[JH99] G. Joeris and O. Herzog, Towards Flexible and High-Level Modeling and Enacting of Processes. *Proceedings of CAISE'99*, p. 88-102.

[JK97] S. Jajodia, L. Kerschberg, *Advanced Transaction Models and Architectures* Kluwer 1997.

[K02] Kantere V., A Rule Mechanism for P2P Data Management, Technical Report, University of Toronto, 2002.

[KRS99]. G. Karabatis, M. Rusinkiewicz, A.Sheth. Aeolos: A System for the Management of Interdependent Data. In A. Elmagarmid, M. Rusinkiewicz, A.Sheth (Eds.), *Management of Heterogeneous and Autonomous Database Systems*, Chapter 8, Morgan Kaufmann 1999.

[KAM02] A. Kementsietsidis, M. Arenas, and R. Miller. Data Mapping in Peer-to-Peer Systems. Technical Report CSRG-456, University of Toronto, July 2002.

[KAM03] Anastasios Kementsietsidis, Marcelo Arenas, Renée J. Miller. Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 2003.*

[KMC99] K. Kulkarni, N. Mattos, R. Cochrane. Active database features in SQL3. In [Paton99], 197--219.

[KMK03] V. Kantere, J. Mylopoulos, I. Kiringa. A Distributed Rule Mechanism for Multidatabase Systems. In Proceedings of the *Intern. Conference on Cooperative Information Systems*, November 2003.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of Multiple Autonomous Databases. *ACM Comp Surveys*, 22(3): 267-293, 1990.

[LSS01] L. Lakshmanan, F. Sadri, and S. Subramanian. SchemaSQL: an Extension to SQL for Multidatabase Interoperability. *ACM Transactions on Database Systems*, 26(4), 2001.

[NOLZ02] W.S. Ng , B.C. Ooi, K.L. Tan, and A.Y. Zhou. PeerDB: a P2P-based System for Distributed Data Sharing. Tech. Report, University of Singapore, 2002.

[Paton99] N.W. Paton. *Active Rules in Database Systems*. Springer Verlag, New York, 1999.

[RGJ+05] Rodriguez-Gianolli P., Garzetti M., Jiang L., Kementsietsidis A., Kiringa I., Masud M., Miller R., and Mylopoulos J. [2005], Data Sharing in the Hyperion Peer Database System. In the *Proceedings of the International Conference on Very Large Databases (VLDB'05)*, p. 1291-1294.

[SGMB02] Serafini L., Giunchiglia L., Mylopoulos J., and Bernstein P., The Local relational Model: Model and Proof Theory. Tech. Report DIT-02-0009, Univ. of Trento, 2002.

[TC97] C. Turker and S. Conrad. Towards maintaining integrity in federated databases. In *3rd Basque Intern. Workshop on Information Technology* Biarritz, 1997.

[VCR00] G. Vargas-Solar, C. Collet, and H.G. Ribeiro, Active Services for Federated Databases. In the Proceedings of the ACM Symposium on Applied computing, pages 356--360, Como, Italy, 2000.

[ZU99] Zimmer D. and Unland R., On the Semantics of Complex Events in Active Database Management Systems. In the *Proceedings of the International Conference on Data Engineering (ICDE'99)*, 1999, p. 392-399.

Appendix

Claim 1:

$$\text{Lower bound of max \# messages} = n+2k+r \qquad \text{(A1)}$$

where n, k and r is the number of databases involved in the event expression, in the condition part and the action part, respectively.

Proof:
Suppose that n is the number of databases involved in the event expression of a rule. In the worst case the master database where the evaluation takes place is not involved in the event expression. Thus, in the worst case the master database will need n messages, one from each one of the n databases involved in the event, to produce a valid event instance. Note that we are considering only the messages that contain the instances of the event sub-expression that will form an instance of the original event. Of course there could be other messages sent to the master database that are rejected. Also, remember that one instance of a sub-event is used to form at most one instance of the respective event. Again if none of the databases involved in the condition expression is also involved in the event, in the worst case the master database needs information from all the databases involved in the condition expression in order to instantiate it. Thus, it has to send k messages, one to each one of the k databases involved in the condition expression, and wait for k answers: 2k messages are needed to gather the information for the condition. Finally, r messages are needed to send the actions to the r databases involved in the action part. (A1) is the lowest limit of the maximum number of messages exchanged for the evaluation of a rule: for each one of the event, condition and action part, the number of exchanged messages: n, 2k, r, respectively, is the minimum possible in the worst case. This minimum occurs when each one of the instances of sub-expressions received by the master database is used to form an instance of the original respective rule. If this is not the case, it is obvious that more that n messages with sub-event instances and more than 2k messages with requests and answers for sub-condition instances are needed for the formulation of a rule instance. ■

Claim 2:
The minimum number of messages exchanged is:

$$\text{Min \# messages} = n+r-2 \qquad \text{(A2)}$$

Proof:
Suppose that n is the number of databases involved in the event expression of a rule. In the best case the master database where the evaluation takes place is involved in the event expression, so it is one of the n databases. Thus, the master database will need n-1 messages, one from each one of the rest n-1 databases involved, to produce a valid instance of the original event expression. For the evaluation of the condition expression, no more information is needed from the databases acquainted to the master one. In the best case all the information needed from these databases has been brought to the master with the n-1 messages of the sub-rules instances. Finally, for the execution of the action part the master database has to send r messages, one for each one of the r databases involved in the action expression. If one of the actions is local, the master has to send out r-1 messages. Note, that in the best case all the sub-rule instances that come to the master are used to form a rule instance.■

Transformation Algorithm

The following proofs ensure the semantic equivalence of an event expression before and after the transformations for the respective step of the transformation algorithm.

Proofs:

1. CE(t) = (! $_{ti}$ (! $_{ti}$ E))(t) := NOT ($\exists t' \in ti$: NOT ($\exists t'' \in ti : E(t'')$)) = ($\forall t \in ti$: NOT (NOT($\exists t'' \in ti : E(t'')$))
   = $\forall t \in ti : \exists t'' \in ti : E(t'') = \exists t'' \in ti : E(t'') = \exists t : E(t)$ =: E(t)
   Note that the ti intervals of both the NOT operators are bound to the same start and end points. Contrary to all other operators the ti is essential for the ! rather than associative: the evaluation of the composite event formed by ! starts when the respective ti starts counting and not when one of the comprised simple events occurs (as is the case of evaluation for all the other operators). Thus, the inner NOT is totally depended on the outer one and, consequently, ti is equal to the time interval of the operator right 'above' the 'not' operators, we can eliminate it. (See comment 4 in section 2.4.1.3). Also, because ti is a time interval that has no absolute start or end point, we can infer that $\exists t' \in st : E(t)$ is equal to $\exists t : E(t)$.

2. CE(t) = (! $_{ti}$ (E1 $\wedge_{ti}$ E2))(t) := NOT ($\exists t' \in ti$:(E1(t_1) AND E2(t_2), where t_2 = t' if $t_1 \le t_2$, or t_1 = t' if $t_2 < t_1$, and $t_1, t_2 \in$ ti1)) and t is the end point of ti1.
   = NOT ($\exists t_1 \in ti$:(E1(t_1)))(t_3) OR NOT ($\exists t_2 \in ti$:(E2(t_2)))(t_4). Because $t_3 = t_4 = t$, the time interval of the new OR should be 0. Thus: CE(t) = ((! $_{ti}$ E1) $\vee_{ti'}$ (! $_{ti}$ E2)), with |ti'|=0.

3. CE(t) = (!$_{ti}$ (E1 $\vee_{ti}$ E2))(t) := NOT ($\exists t' \in ti$:(E1(t_1) OR E2(t_2), where t_2 = t' if $t_1 \le t_2$ or there is no t_1, or t_1 = t' if $t_2 < t_1$ or there is no t_2, and $t_1, t_2 \in$ ti)) and t is the end point of ti
   = NOT ($\exists t_1 \in ti$:(E1(t_1))(t_3) AND NOT ($\exists t_2 \in ti$:(E2(t_2))(t_4)), where $t_3 = t_4 = t$, or $t_3 = t$ if there is no t_4, or $t_4 = t$ if there is no t_3. Also, because $t_3 = t_4 = t$, the time interval of the new AND should be 0. Thus: CE(t) = ((! $_{ti}$ E1) $\wedge_{ti'}$ (! $_{ti}$ E2))(t), with |ti'|=0.

4. CE(t) = (! $_{ti}$ (E1 « $_{ti}$ E2))(t) := NOT ($\exists t' \in ti$:(E1(t_1) AND E2(t'), where $t_1 \le t'$ and t_1, t' $\in$ ti)) and t is the end point of ti = (NOT ($\exists t_1 \in ti$:(E1(t_1)))(t_3) OR NOT ($\exists t_2 \in ti$:(E2(t_2)))(t_4) where, as in the previous proofs, where $t_3 = t_4 = t$, or $t_3 = t$ if there is no t_4, or $t_4 = t$ if there is no t_3. Also, because $t_3 = t_4 = t$, the time interval of the new OR should be 0) OR ((E2(t_3) AND E1(t'), where $t_3 < t'$ and t_3, t' $\in$ ti) =: (((! $_{ti}$ E1) $\wedge_{ti'}$ (! $_{ti}$ E2)) $\wedge_{ti''}$ (E2 « $_{ti}$ E1))(t) , where |ti'|=0.
   Note that the time interval of the second 'disjunction' is set to zero, too. This is correct, since (E2 « $_{st2}$ E1) cannot co-occur with the other operand of the disjunction. Also, note that for the special case where E2 and E1 occur simultaneously, the part (E2 « $_{ti}$ E1) should not be instantiated (because E1 « $_{ti}$ E2 is valid, too). This should be taken care by the implementation of the « operator. Finally, ti'' is redundant because the respective $\wedge$ is actually a XOR.

5. CE(t) = (! $_{ti}$ ((E1 « $_{ti}$ E2) < $_{ti}$ (! $_{ti}$ E3)))(t) := NOT ($\exists t' \in ti$: E1(t_1) AND E2(t') AND NOT E3(t_3), where $t_1 \le t_3 \le t'$ and t_1, t_3, t' $\in$ ti) and t is the end point of ti = (NOT ($\exists t_1 \in ti$:E1(t_1))(t_3) OR NOT($\exists t_2 \in ti$:E2(t_2))(t_4), where, as in previous proofs, $t_3 = t_4 = t$, or $t_3 = t$ if there is no t_4, or $t_4 = t$ if there is no t_3. Also, because $t_3 = t_4 = t$, the time interval of the new OR should be 0) OR (E1(t_5) AND E3(t_6) AND E2(t) where $t_5 \le t_6 \le t$ and t_5, t_6, t $\in$ ti) =: ((((! $_{ti}$ E1) $\vee_{ti'}$ (! $_{ti}$ E2)) $\vee_{ti''}$ (E2 « $_{ti}$ E1)) $\vee_{ti'''}$ (E1 « $_{ti2}$ E3 « $_{ti3}$ E2))(t), where |ti'| = 0. Also ti'' and ti''' are redundant because the respective $\wedge$ operators are used as XOR. Also, ti2+ti3=ti, i.e. |ti2|+|ti3| = |ti|, and the start point of ti is the same as the start of ti2, and the end point of ti3 is the same as the end of ti.

6. CE(t) = (! $_{ti}$ (+ $_{ti}$ E))(t) := NOT ($\exists t' \in ti$:E(t_1) AND E(t_2) AND … AND E (t_{m-1}) AND E(t_m), where $t_1 \le t_2 \le \ldots \le t_{m-1} \le t_m$ and $t_m = t'$ and t_1, t' $\in$ ti and $1 \le m$ and $m \in$ N). However, m cannot be greater

than 1, because since E occurs once – and m becomes 1 – CE does not have a chance to occur. Thus CE(t) = NOT ($\exists t' \in ti$: E(t_1), $t_m = t$ ', m = 1) = NOT($\exists t \in ti$: E(t)) =:(! $_{ti}$ E)(t).

7. CE(t) = (! $_{ti}$ ($_{m'}$ \$ $_{ti}$ E))(t) := NOT ($\exists t' \in ti$: E(t_1) AND E(t_2) AND … AND E(t_{m-1}) AND E(t_m), where $t_1 \le t_2 \le ... \le t_{m-1} \le t_m$ and $t_m = t'$ and t_1, t' ∈ ti and $m' \le m$ and $m', m \in$ N) = NOT($\exists t' \in ti$: E(t_1) AND E(t_2) AND … AND E($t_{m'}$) … AND E(t_m), where $t_1 \le t_2 \le ... \le t_{m'} \le ... \le t_m$ and $t_m = t'$ and t_1, t' ∈ ti and $m' \le m$ and $m', m \in$ N) = E(t_1) AND E(t_2) AND … AND E(t_m), where $t_1 \le t_2 \le ... \le t_m$ and $t_m = t$ and t_1, t ∈ ti and m ∈ N, and NOT($\exists t_{m''} \in ti$: E($t_{m''}$) and $t_{m''} \le t_m$ and $m' \le m''$ and $m', m'' \in$ N) = E(t_1) AND E(t_2) AND … AND E(t_{m-1}) AND E(t_m), where $t_1 \le t_2 \le ... \le t_{m-1} \le t_m$ and $t_m = t$ and t_1, t ∈ ti and $m'-1 \ge m$ and $m', m \in$ N := ($_{m'-1}$ & $_{ti}$ E)(t) .

8. CE(t) = (! $_{ti}$ ($_{m'}$ & $_{ti}$ E))(t) := NOT ($\exists t' \in st1$: E(t_1) AND E(t_2) AND … AND E(t_{m-1}) AND E(t_m), where $t_1 \le t_2 \le ... \le t_{m-1} \le t_m$ and $t_m = t'$ and t_1, t' ∈ ti and $m' \ge m$ and $m', m \in$ N) = E(t_1) AND E(t_2) AND … AND E(t_m), where $t_1 \le t_2 \le ... \le t_m$ and $t_m = t$ and t_1, t ∈ ti and NOT($m' \ge m$) and $m', m \in$ N = E(t_1) AND E(t_2) AND … AND E(t_{m-1}) AND E(t_m), where $t_1 \le t_2 \le ... \le t_{m-1} \le t_m$ and $t_m = t$ and t_1, t ∈ ti and $m'+1 \le m$ and $m', m \in$ N := ($_{m'+1}$ \$ $_{ti}$ E)(t).

9. CE(t) = (! $_{ti}$ ($_{m}$ # $_{ti}$ E))(t) := NOT ($\exists t' \in st1$: E(t_1) AND E(t_2) AND … AND E(t_{m-1}) AND E(t_m), where $t_1 \le t_2 \le ... \le t_{m-1} \le t_m$ and $t_m = t'$ and t_1, t' ∈ ti and $m \in$ N) = E(t_1) AND E(t_2) AND … AND E($t_{m'}$), where $t_1 \le t_2 \le ... \le t_{m'}$ and $t_{m'} = t'$ and t_1, t' ∈ ti and $m' \in$ N AND m' ≠ m, = (E(t_1) AND E(t_2) AND … AND E(t_{m-1}) AND E(t_m), where $t_1 \le t_2 \le ... \le t_{m-1} \le t_m$ and $t_m = t$ and t_1, t ∈ ti and $m'-1 \ge m$ and $m', m \in$ N) OR (E(t_1) AND E(t_2) AND … AND E(t_{m-1}) AND E(t_m), where $t_1 \le t_2 \le ... \le t_{m-1} \le t_m$ and $t_m = t$ and t_1, t ∈ ti and $m'+1 \le m$ and $m', m \in$ N) := (($_{m-1}$ & $_{ti}$ E) ∨ $_{ti'}$ ($_{m+1}$ \$ $_{ti}$ E))(t), where |ti'| = 0.

10. CE(t) = (+ $_{ti1}$ (+ $_{ti2}$ E))(t) := (E(t_{11}) AND … AND E(t_{1m}), where $t_{11} \le ... \le t_{1m}$ and $t_{1m} = t_1$ and t_{11}, t_1 ∈ $ti2_1$ and $1 \le m$ and $m \in$ N) AND … AND (E(t_{n1}) AND … AND E($t_{nm'}$), where $t_{n1} \le ... \le t_{nm'}$ and $t_{nm'} = t_n$ and t_{n1}, t_n ∈ $ti2_n$ and $1 \le m'$ and $m' \in$ N), where $t_1 \le .. \le t_n$ and $t_n = t$ and t_1, t ∈ ti1 and $1 \le n$ and $n \in$ N and (|$ti2_1$|=…=|$ti2_n$|=|ti2|) = (E(t_{11}) AND…AND E(t_{1m}) AND … AND E(t_{1n}) … AND E($t_{nm'}$), where $t_{11} \le ... \le t_{1m} \le ... \le t_{1n} \le ... \le t_{nm'}$ and $t_{1m} = t_1$ and $t_{nm'} = t_n = t$ and (t_{11}, t_{1m} ∈ $ti2_1$) and (t_1, t ∈ ti1) and $1 \le m, m', n$ and $m, m', n \in$ N) = E(t_1) AND … AND E(t_k), where $t_1 \le ... \le t_k$ and $t_k = t$ and t_1, t ∈ ti = ti1+ti2 and $1 \le k$ and $k \in$ N: = (+ $_{ti}$ E)(t) where ti = ti1+ti2.

Decomposition Algorithm

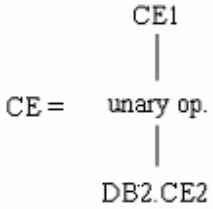
The following studies justify the respective steps of step c of the decomposition algorithm.

Proofs:

Suppose that the original event expression, CE, involves the databases DB1 and DB2, and that we are producing the sub-event for DB1, CE_{DB1}. Note that in the following we use a tree to visualize parts of the composite event CE. In the tree representation of e.g. CE = (CE_1 OP_2 CE_2) OP_1 CE_3, where OP1 is the root of the CE tree. For simplicity of the figures we usually represent the part of the tree including the root that is of no interest as a composite sub-event. For the previous example the part '(CE_1 OP_2 CE_2) OP_1' could be represented as CE_x connected via a line with CE_3. Also, if a part of CE has a subscript, we mean the respective event part containing only simple events of the database denoted by

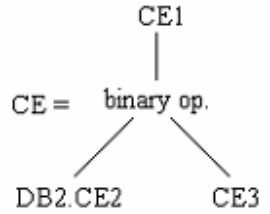
the subscript. For example $CE1_{DB1}$ is the CE1 part of CE from which we have eliminated the simple events that do not occur in DB1. Finally, by DBx.CEy we mean that the CEy part of CE contains only simple events of DBx.

1. Suppose that:

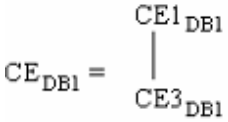


The part DB2.CE2 denotes a composite (or simple) event that consists only of simple events of DB2. It is obvious that all the information in CE concerning DB1 is captured by the part CE1; thus we can eliminate the part 'unary op.(DB2.CE2)' without loss of information in the structure of the DB1 sub-event. Thus CE_{DB1} = $CE1_{DB1}$, where $CE1_{DB1}$ is the part of CE1 that captures only the information for DB1.

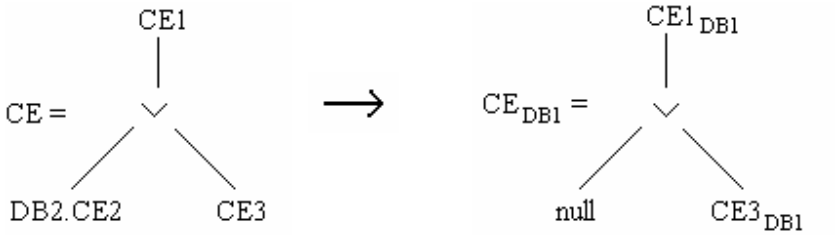
2. Suppose that:



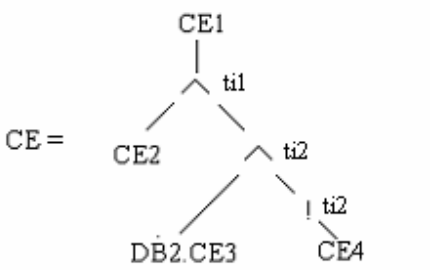
where 'binary op.' is either a $\wedge$ (conjunction) or a « (loose sequence) operator and CE3 is not a composite event of the ! (negation) operator. Again DB2.CE2 is an event that consists of DB2 simple events, only. It is obvious that the information about DB1 events is captured by the CE1 and CE3 parts of CE. Thus we can eliminate DB2.CE2 without loss of information. Also, the binary operator does not provide us any help in gathering information about DB1: If the 'binary op' is either a $\wedge$ or a «, we know that we want $CE3_{DB1}$ to occur definitely. Moreover, for an instance of $CE3_{DB1}$ the 'binary op' does not help us to decide if we can keep this instance as part of a valid CE instance, without using information about the DB2.CE2 instances. Thus, it is not useful in the sub-event of DB1. Thus we can eliminate the $\wedge$ and « in cases as the ones described. The DB1 sub-event is formed as:



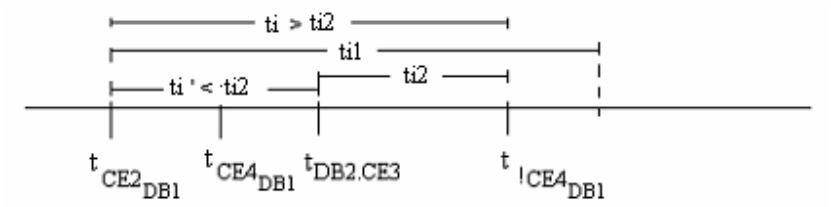
However, the case of a $\vee$ (disjunction) operator is different.
In contrast with the $\wedge$ and the «, this operator can produce valid composite events even if one of its two operands is not instantiated. For this reason CE_{DB1} should be valid either if $CE3_{DB1}$ is valid or not. Thus, we keep the $\vee$ in CE_{DB1}:



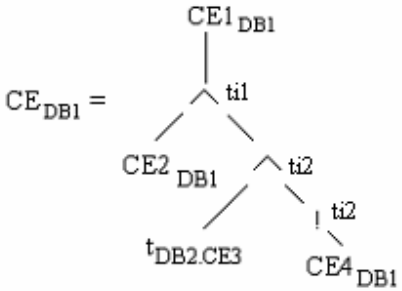
The $\vee$ in the above figure is evaluated always as true. The sub-event is valid (i.e., there is an instance) iff $CE1_{DB1}$ is valid. Thus, we avoid the loss of an instance of $CE1_{DB1}$ in case that $CE3_{DB1}$ is not valid. However, if the latter is also valid, the instance of it will be also captured as information of the DB1 part in CE.
Lets consider now the case of a binary operator one operand of which is a composite event of the ! operator. For example:



Suppose that, in the above figure, CE2 (thus $CE2_{DB1}$) occurs before the right operand of the $\wedge$ operator having the ti1 time interval and that DB2.CE3 occurs before !CE4 (thus before $!CE4_{DB1}$):



(Note that in the above figure the events $CE4_{DB1}$ and $!CE4_{DB1}$ occur exclusively). As we observe from this figure, the time distance between $CE2_{DB1}$ and $!CE4_{DB1}$ is longer than the time interval of the 'not' operator, st3. If we eliminate the intermediate $\wedge$, the starting point of the time interval of the event $!CE4_{DB1}$ will be bound to the occurrence time of $CE2_{DB1}$. If this time interval remains big as ti2, then we won't be able to capture the event $!CE4_{DB1}$ of the above figure. Someone could argue that this problem can be solved by changing the time interval of ! to ti1. Indeed this could be a solution, but then another problem could occur: If an instance of $CE4_{DB1}$ occurred in the time interval $[t_{CE2DB1}, t_{DB2.CE3}]$, then the instance of $!CE4_{DB1}$ would not be detected. Thus, we definitely need the occurrence time of DB2.CE as a reference point in order to preserve the equivalence between the original event expression and this sub-expression. The sub-event for DB1 is:



The reader might think that preserving $t_{DB2.CE3}$ is not correct because DB2.CE3 is an event from another database and the sub-event of DB1 is not supposed to accumulate information about other databases. However, in the 'close world' of DB1, the *time event* $t_{DB2.CE3}$ is an absolute time event. (Note that this holds only for the time occurrence of DB2.CE3 and not for the actual instance of this event).

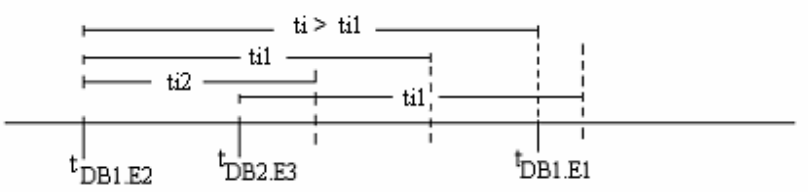
3. Consider the following example:

$$CE(t) = (DB1.E1 \wedge_{ti1} (DB1.E2 \wedge_{ti2} DB2.E3))(t)$$

In this case, as we have seen, the sub-event for DB1 is:

$$CE_{DB1}(t) = (DB1.E1 \wedge_{ti1} (DB1.E2 \wedge_{ti2} DB2.E3))(t) = (DB1.E1 \wedge_{ti} DB1.E2)(t)$$

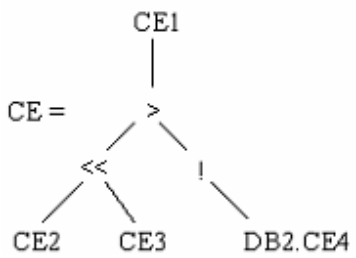
If ti remains equal to ti1, every instance of CE_{DB1} should satisfy the expression: $|t_{DB1.E2} - t_{DB1.E1}| < ti1$. In this case, we may experience loss of information as far as it concerns DB1, because of too tight time constraints. Observing the original event expression, CE, we can see that it can be satisfied by instances in which an instance of DB1.E2 occurs some time before an instance of DB2.E3, and DB1.E1 occurs some time after DB2.E3, so that the time distance between DB1.E2 and DB1.E1 is bigger than ti1. However, this distance between DB2.E3 and DB1.E1 is smaller than ti2:



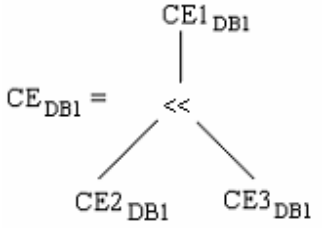
Thus, ti has to be bigger than ti1, and more specifically, it should be ti = ti1+ti2, the sum of the initial time interval of the 'outer' $\wedge$ and the time interval of the 'inner', eliminated $\wedge$. If more operators than one were eliminated in the above example, ti should be even bigger, equal to the sum of all the eliminated operators.
We don't have to do the described changes in the time interval of a « operator of which only the right operand has an eliminated 'underlying' operator. This is because, the « operator indicates precisely that the right operand should occur after the left one. Thus, an operator elimination in the right operand, does not change the total ti of the initial «.

4. Here we examine the special case of the binary operator > (strict sequence).

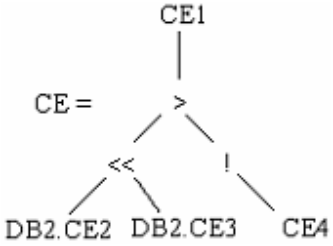
4a. Suppose:



In this case all the information about the DB1 event instances is accumulated by the parts CE1, CE2 and CE3, whereas DB2.CE4 gathers information only for DB2. Thus, we can eliminate the ! part of the > operator. After this elimination, it is obvious that the 'strict sequence' does not serve its role anymore, which is to detect two consecutive events instances without the intermediate occurrence of a third event. We can only capture information about consecutive occurrences of CE2 and CE3. Thus, the « part of the initial > is enough, and that is what we keep in the sub-event of DB1:



4b. Suppose:



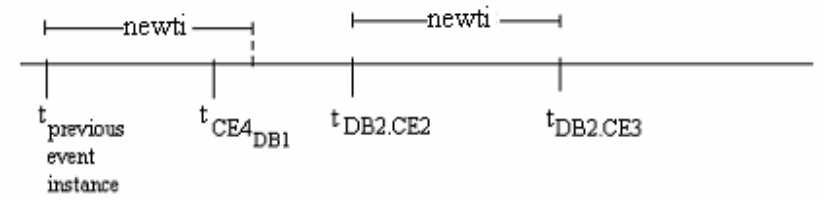
In this case all the information about the DB1 event instances is accumulated by the parts CE1 and CE4, whereas DB2.CE2 and DB2.CE3 gather information only for DB2. Thus we can eliminate the « part of the >. After this elimination, it is obvious that the > does not serve its role anymore. We can now detect only if CE4

does not occur in a time interval ti, which is the time interval of the initial >. However, this is not enough, because, if ti > $|t_{DB2.CE2} - t_{DB2.CE3}|$ some instances of !CE4 will not be captured: Imagine the following situation:

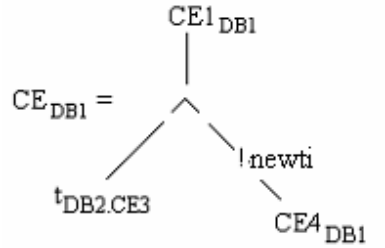
The event $CE4_{DB1}$ occurs after DB2.CE3. Thus, even though an instance of $!CE4_{DB1}$ (and possibly of !CE4) could be detected in an interval of size $|t_{DB2.CE2} - t_{DB2.CE3}|$, (for example, it would be detected in the exact interval $[t_{DB2.CE3}, t_{DB2.CE2}]$), if we keep ti as the time interval of $!CE4_{DB1}$ in the sub-event of DB1, such an instance may not be detected (it also depends on the operator in the previous level in the tree of the sub-event and how that would be instantiated). Thus the new ! operator should have an time interval of size:

$$newti = |t_{DB2.CE2} - t_{DB2.CE3}|$$

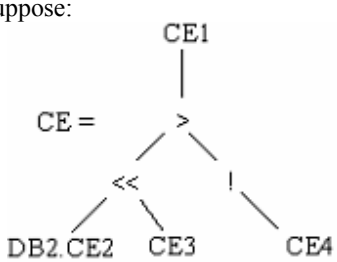
Still, this is not enough. If the ! is the root of the tree of the sub-event expression CE_{DB1}, we can assume that the starting point of newti is bound to $t_{DB2.CE3}$. Nevertheless, if ! is an operand of a binary operator in the sub-event of DB1, then the starting point of newti will be bound to the occurrence time of the other operand of this binary operator. In a situation like this:



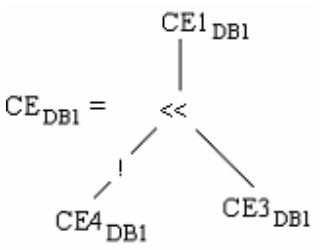
there is going to be no instance of $!CE4_{DB1}$, even though there should. In order to solve this problem we can put the ! in the sub-event of DB1 in a $\wedge$ operator that has as first operand the time event $t_{DB2.CE3}$. So the sub-event for DB1 in this case is:



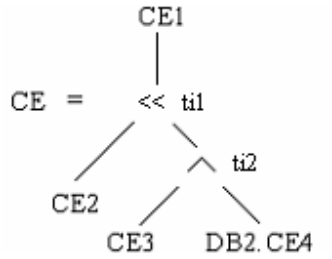
4c. The last case for the > operator is when only one operand of the « together with the ! part have information about DB1 events. For example, suppose:



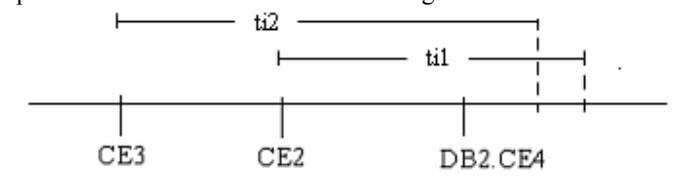
In this case, the part DB2.CE2 offers no information about any DB1 events, and, thus, it should be eliminated. In the sub-event of DB1, we can only accumulate information about the occurrence of $CE3_{DB1}$ and $!CE4_{DB1}$ and their relevant position in time: if $!CE4_{DB1}$ occurs before $CE3_{DB1}$ in a time interval ti, where ti is the time interval of the initial > operator. Thus, we substitute the latter with its left operand, the « and the left operand of this with $!CE4_{DB1}$:



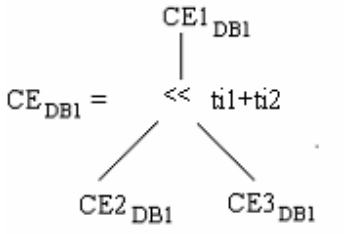
5. Suppose:



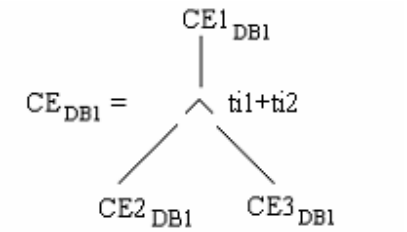
Also suppose that the sequence of event instances is as following:



According to this sequence an instance of CE occurs, because the composite event (CE3 $\wedge_{ti2}$ DB2.CE4) occurs after CE3.
If we keep the « operator in the sub-event of DB1:



The evaluation procedure of CE_{DB1} would falsely expect $CE3_{DB1}$ to occur always after $CE2_{DB1}$. Thus, we should change the « operator in order to make CE_{DB1} less restrictive in terms of the sequence of occurrences of its operands. However, we still want both of them to occur. Thus, the solution is to change it into a 'conjunction'. The sub-event CE_{DB1} is:



6. See section 4.4.