

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Relatório Técnico

RT-MAC-2001-01

**EVALUATING COMPOSITE EVENTS USING
SHARED TREES**

**Douglas Moreto
Markus Endler**

Janeiro de 2001

Evaluating Composite Events using Shared Trees

Douglas Moreto Markus Endler
Departamento de Ciência da Computação
IME-Universidade de São Paulo
Rua do Matão 1010, Cidade Universitária
05508-900 São Paulo, Brazil
E-mail: {moreto,endler}@ime.usp.br

Abstract

Distributed and concurrent systems tend to generate large numbers of events which require further processing, such as filtering and establishment of co-relations, in order to become useful for human or automated managers. This work focuses on the detection of composite events defined through event expressions involving primitive events and some operators: boolean conjunction and disjunction; sequence, repetition, and absolute, periodic and relative timer events.

A concrete result of this work was the design and implementation of a general-purpose Event Processing Service (EPS) which may be used by monitoring applications to be informed about the occurrence of primitive and composite events. Composite events are commonly represented as trees where the leaf nodes represent primitive event types and the intermediate nodes represent any of the supported event operators. The main contribution of this work is the description of a method to process composite events that share common sub-expressions with other composite events.

The Event Processing Service was implemented in Java within the framework of the Sampa Project.

1 Introduction

Collection and analysis of events are intrinsic tasks of monitoring computer systems, and may be used for a wide variety of purposes, such as management of resource usage, detection of program state transitions, capturing of user interactions, performance analysis, security, etc. Thus, event processing has been used in many areas such as active databases, computer networks, software development tools, distributed systems, and many others.

In networked and distributed systems, monitoring has always been a very important function for the management of resource utilization and sharing, as well as for the verification of proper functioning of a system [11]. But monitoring also plays an important role in testing, debugging and optimizing concurrent (i.e. parallel and distributed) programs, where a developer has to deal with many sorts of indeterminism. For most applications, monitoring also includes some form of event processing, such as filtering and detection of co-relations between event.

Composite events are essential for applications where it is impossible to detect a particular state of a system by looking only at isolated events. For example, in Local Area Network management, a problem with a physical link may be detected when the amount of incoming IP packets and the amount of outgoing IP packets at the attached network interfaces stops to grow (e.g. no IP/ICMP activity). Although the freezing of the number of packets processed at one interface is an isolated event that may or not be an important information, in this particular case it is the combination of several such events that is of relevance for link-layer network management.

Similarly, monitoring and correlating different types of events at different hosts, such as, for example, large numbers of open TCP connections to/from a given set of hosts, or unusual processes with root permission may also be important for detecting abnormal system or user behaviors.

Another class of applications that rely on the detection of composite events are process control systems, where the environment is continuously monitored through several sensors (e.g. for temperature, pressure, motion, etc.). In these systems primitive events are generated by those sensors whenever some of its input variable trespasses certain boundary values, and many control (re)actions depend on the detection of patterns of events that have some application-specific correlation, such as:

if (*temperature* < *X* or *pressure* > *Y*) and *tank_isfull* then *start_reaction*

A concrete result of this work was the design and implementation of a general-purpose Event Processing Service (EPS) which may be used by applications which require some form of monitoring. Through EPS these can be informed of the occurrence of a composite event, i.e. a set of related events occurring at components of a distributed system. Monitoring applications submit requests-for-notification (*RfNs*) to the EPS which contain an event expression and a condition on event attributes that must be satisfied for the application to be notified. The EPS also receives messages (event instances) from sensors¹ notifying the EPS of the occurrence of changes in either software or hardware components in the system.

Composite events are defined through an *event expression* based on primitive event types and combined by any of the following operators: boolean conjunction and disjunction; sequence and repetition, with arbitrary nesting degree. Moreover, it is possible to specify also periodic and absolute timer events and to relate the former to other primitive or composite events. Composite events are commonly represented as trees where the leaf nodes represent primitive event types and the intermediate nodes represent any of the supported event operators.

The main contribution of this work is the development of a method to process composite events that share common sub-expressions with other composite events. Unlike other event processing systems[1, 6] which do not support aggregated trees, this method allows for an optimized evaluation of event expressions since less leaf nodes need to be checked when a new event instance arrives from a sensor.

Due to the sharing of sub-trees derived from common sub-expressions within different *RfNs*, our method also takes into account that *RfNs* may specify different conditions on the attributes of event types appearing in the common sub-expressions. This makes the accounting of event usage for the different conditions more complicated, since for example, a primitive event instance that has not contributed to the satisfaction of

¹A sensor may be any system component capable of sending primitive event instances to the EPS.

one condition cannot be eliminated unless there is no other event condition to which it may contribute.

Our method consists of a four-phase tree-traversal algorithm, in which the first phase is done whenever a new event instance arrives at the EPS, the second and the third phases are done whenever an event condition of an *RfN* is satisfied, and the fourth phase can be done at any moment and aims at garbage-collecting the events that actually contributed to the satisfaction of condition. The algorithm used in our method has been formally described in an attribute grammar and has been implemented in the Event Processing Service. Despite being a general-purpose service, EPS is best suited for applications that require the detection of complex event patterns (eventually with similar structures), such as found in workflow control, network management or monitoring of automated processes (e.g. a chemical reaction, manufacturing, etc.).

This work was done in the framework of a larger project aiming at the development of a system for supporting the execution and management of fault-tolerant distributed applications and services. This project is called Sampa (*System for Availability Management of Process-based Applications*)[2], and also comprises other component services, such as monitoring, checkpointing, group membership and configuration management, which are used to monitor, control and dynamically configure systems in response to the detection of failures. In this project, event monitoring is a basic service which is to be used for the purpose of availability management. EPS is also being used as a basic component for exception handling in an Object-Oriented Software Architecture for Dependable Systems[3], within another project.

The remainder of the paper is organized as follows: In the next section we compare our approach to related work. In section 3 we present the underlying system model and in section 4 we introduce the Event Processing Service, its main components and their functionality. Then, in section 5 we describe the types of events supported, their attributes and the syntax of event expressions and conditions. Section 6 then explains each of the four processing phases in detail, based on an example of a simple event tree, and in section 7 we discuss some time-related features of the service. In section 8 we compare our algorithm with other ones and in 9 we show some factors that have influence on EPS's performance. Finally, in section 10 we show results of performance tests and discuss both achieved results and future research directions.

2 Related Work

Most of the works dealing with composite events are related to the area of Active Database and are based on centralized monitors. Only few works, such as GEM and Eve, aim at distributed systems.

While most of the works use trees as the basic data structure for implementing such event monitors, other computational models are used as well, such as Petri Nets or Finite Automata.

Samos [10] is a project that uses a colored Petri Net Model to specify the behavior in Active Databases. This model is very suitable for modeling and describing complex and concurrent behavior, but its implementation was shown to be complex and comparatively inefficient [4].

Ode [8] is a project in Object-oriented Databases that was developed at AT&T Bell Labs. Its main characteristic is the use of triggers to automatically launch actions depending on the state of the database. One of the results of the project is COMPOSE, a system for specification and detection of composite events

which is based on finite automata. Its main weakness is the impossibility to detect concurrent events and to re-order events according to their creation times.

The following projects use trees to represent composite events: Eve [1] is an event detector for workflow control; Snoop [7] is a monitor for databases and GEM [6] is a generic event monitor for distributed systems. The main reason for adopting trees is that they are simple, well suited to represent composition, and have been thoroughly studied in the past, leading to a large collection of efficient algorithms for traversal and manipulation.

In the Eve and GEM monitors, each request-for-notification (*RfN*, cf. section 4) for a composite event causes the creation of an entire new event tree, even when this tree contains identical subtrees of other trees already created by the monitor. Consequently, the monitor may process a same subtree several times, depending on the number of *RfNs* for similar composite events.

For example, in a local area network it may happen that several programs, such as a performance or security management tool, a distributed operating system or a middleware service may be interested in notifications about very similar composite events concerning, for example, TCP connections (or IP traffic) to or from a specific host.

Snoop is a monitor which is also concerned with the problem of repetitive processing of shared subtrees of composite events, and also uses aggregate trees to implement the composite event detector. Unlike our approach, however, Snoop defines a predefined set of different modes (called *contexts*) for processing composite events. The set consists of the following four processing modes, which will be explained considering a generic composite event type $E_C = (E_1, E_2)$ with constituent event types E_1 and E_2 .

Recent only the most recent instances of the events are considered during the processing, i.e. when several instances of a given event type (e.g. E_1) have been generated, then all but the most recent instance are ignored;

Chronological the instances are processed according to the order in which they have been generated.

Thus, an instance of a composite event type E_C is always built from the pair of the oldest instances of the constituent event types E_1 and E_2 . After the composite instance is processed, its constituent instances are not used again, and the next oldest pair of instances of E_1 and E_2 is considered, and so on;

Continuous in this mode, the instance of the constituent event type (e.g. E_1) that triggered the detection of the composite event E_C is combined with all the instances of E_2 . All the other instances of E_2 used in the composition are removed and will not be used in future instances of E_C . The instance of E_1 that triggered the detection of E_C may or not be used in future instances of E_C , which depends on the composition operator. For example, for operator conjunction it is re-used while for the operator sequence it is removed;

Cummulative when a composite event is detected in this mode, the instance of E_C contains all the instances of both E_1 and E_2 that have been generated so far, and the instance sets for this event types are reset.

3 The System Model

This work assumes a distributed application as being composed of processes executing on different hosts interconnected through a reliable communication network.

Some of the processes (the *sensors*) are specialized to probe run-time status information about other processes or system resources. We also assume the existence of one (or several) monitoring components (the *monitors*), whose task is to do the application-specific analysis and presentation of the information provided by the sensors. This information is transferred from sensors to monitors via *event messages* (of certain types) that contain a time-stamp.

In this model, we further assume that clocks are approximately synchronized (within a certain precision). Due to this assumption, and because sensors do not necessarily interact with each other, we chose to use time-stamps with real-time values instead of vector time-stamps to determine the precedence of event messages at arrival. Thus, in EPS time-stamp are supposed to be the value of the host's local clock at the moment the message is sent by the sensor.

4 The Event Processing Service

The Event Processing Service (EPS) is a general-purpose service that supports the detection of primitive and composite events for any application which requests this service. An application process makes a request by sending a message called *Request for Notification (RfN)* to the EPS. The *RfN* is composed of the following parts:

Event expression It specifies the structure of the composite event in terms of component primitive event types and five possible event operators, which are described in section 5. The set of all defined primitive event types is a parameter of the EPS, and each event type has associated a set of specific attributes.

Condition expression It determines constraints on attributes of event types which have to be satisfied for a notification to be sent to the requesting application process.

Callback port This is a reference to an entry point (e.g. a port or remote method) at the requesting application process, which is accessed/invoked by EPS to send a notification of occurrence of an event of interest.

Notification This is a character string that the requesting application process will receive from EPS as part of the event notification. It will typically describe the application specific meaning of the corresponding event occurrence.

Each Request for Notification (*RfN*) gets a unique *request-Id*, which may further be used by the application process to access it, as for example, to cancel a the particular request.

The monitoring of the system is actually performed by application specific *sensors* (i.e. software processes), which report the occurrence of any significant change in either software or hardware component,

by sending event messages to EPS. Each such message contains a single primitive event instance with some attribute values.

4.1 Architecture

Currently, the EPS is implemented as a centralized server program (shown in figure 1) composed of the following four functional units:

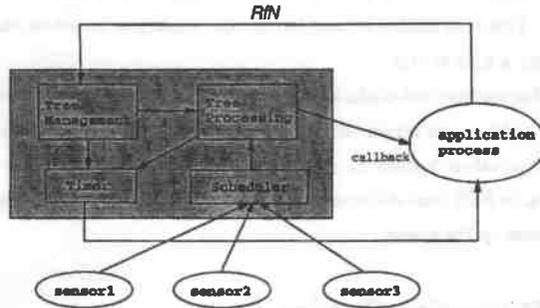


Figure 1: Structure of the EPS

Tree Management This unit handles the acceptance of new *RfNs*, adds the corresponding event nodes in the event tree and sends requests for timer events to the **Timer** unit.

Scheduler The scheduler receives event messages (with primitive event instances) from the sensors, implements a detection window to eliminate outdated events (see section 7.1) and schedules the delivery of event instances to the **Tree Processing** unit. In order to deliver the event instances in the correct order, the scheduler uses the *scheduler time* (cf. section 7) to wait for “late” instances.

Tree Processing This is the core unit within EPS. It performs all the event processing and detection, as well as garbage collection on one or more trees representing the event expressions specified in the *RfN*. Upon every delivery of a new primitive event instance by the **Scheduler**, this unit eventually generates new event instances at intermediate nodes of the tree to represent the occurrence of composite events. The algorithm for tree traversal used in this unit is described in detail in section 6.

Timer This unit essentially implements an alarm clock, which receives requests to generate either absolute or periodic timer events. These requests arrive either from the **Tree Management** or the **Tree Processing** units, where the first ones are for absolute and periodic timer events and the second ones are for absolute timer events.

5 Event Types and Attributes

Events are instances of either *primitive* or *composite event type*. Primitive event types have attributes that hold data values reflecting some change of state or behavior in either a software or hardware component of the distributed system. Composite event types describe complex events, such as the conjunction, disjunction, sequence, etc. of simpler primitive or composite event types. A composite event is defined by an event expression contained in a *RfN*.

Common to all event types are the attributes *origin* and *time-stamp*. While for primitive events the value of a time-stamp is determined by the generating sensor, for composite events the time-stamp is inferred from the time-stamps of its component events.

In order to combine primitive events, EPS has five operators: operator “&” (boolean and); operator “||” (boolean or) operator “;” (sequence of events); operator *TimesEq* and operator *TimesDf*. The last two operators represent the repetitive occurrence of a specific event. While operator *TimesEq* consider the repetition of an event whose instances have the same value for the specific attribute, operator *TimesDf* does the opposite, consider only the instances with different values for the specific attribute.

In addition to these operators, it is possible to group an event expression by using parenthesis. For example, the following event expression

$$TimesEq(n, E_1, attr_1) || (E_2; E_1)$$

describes the composite event of n occurrences of a event E_1 (all with the same value of attribute $attr_1$) or the composite event “ E_2 followed by E_1 ”. Such an event expression could be part of a *RfN*.

Although the ability to express the absence of events certainly adds expressive power to an event description language, we chose not to support negative event expressions in EPS, since this would require the implementation of complex evaluation methods based on interval logic [9].

Moreover, one can get around this problem by using special timer-equipped sensors (or interposing them between the normal sensors and EPS) that periodically check if some state change or event has occurred within some period of time, and if this is not the case, deliver an event to EPS of the type “*not event E within last T time units*”. Thus, we believe that for most applications EPS’s genuine lack of negative event expressions will not be a major problem.

Besides the composite operators, EPS is able to deal with timer events. The three kind of timer events which EPS can handle are: **Absolute** (specifying a unique moment in time); **Periodic** (specifying a repetitive event) and **Relative**. This last event type defines the generation of a timer event some time after the occurrence of an arbitrary composite or primitive event. For example, *at(E, 0/0/2 0:01:05)* specifies that exactly two days, one minute and five seconds after the occurrence of an event of type E a timer event will be generated.

6 Event Processing

In this section we give an overview of the event processing method used in EPS.

Like some other works, we chose to represent composite events as a tree, where the leaf nodes represent primitive event types (denoted by E_i) and the intermediate nodes represent one of the possible event operators shown in section 5. In addition, there is a third kind of node called an *application node*, which represents an *RfN* and may be connected to a leaf or an intermediate node. Such event trees are built according to the event expressions of the *RfNs* received by EPS from the application processes. Unlike most other work in this area, where each application request causes the creation of a separate tree representing the corresponding event expression, our approach is to aggregate the trees whenever an event expression from an *RfN* shares sub-expressions with another *RfN*, eventually from another application. The main advantage of this tree aggregation is that it facilitates the processing of events contained in several expressions. Only Snoop [7] provides similar functionality. However, unlike Snoop, which adopts four modes to control the event processing, EPS uses a generic mode where all possible combinations of events are considered. Details about Snoop's four modes and the advantage of using a generic mode in EPS are discussed in sections 2 and 10.

In what follows we will make a distinction between *nodes* of the event tree, e.g. describing primitive event types and operators, such as operator "&", operator "||", etc. and *event instances* (or simply, *instances*), which are the pieces of monitored data obtained through event messages from the sensors. Whenever an instance arrives at the EPS, it is associated with the corresponding leaf node and its occurrence is also propagated up the tree. This upward propagation is done by eventually creating new instances at the parent nodes of the leaf node, at the parent nodes of these, etc. Thus, while processing incoming event instances, several trees of instances are created according to the operators represented by each of the intermediate nodes of the event tree.

Figure 2 depicts a simple aggregated event tree built from two *RfNs* sent by applications AP_2 and AP_1 . In this example the event expression from AP_1 is $(E_1 \&_1 E_2) \&_2 E_3$ and from AP_2 is $E_1 \&_1 E_2$. Moreover, the Figure shows the instance trees corresponding to the arrival of four primitive event instances 0, 1, 2 and 3. In this and the following figures we will represent the event trees by circles (the nodes) and thick lines, and the instance trees by small rectangles (the instances) and thin lines. In the rectangles representing the instances, the upper and lower parts are used to represent the attributes *states* and *users* of each event instance, which will be explained in the following.

Since we are using aggregate trees, each node can have one or more parent nodes. Therefore, for the event processing we need to decide if an instance has or not been already used in a composite event including a specific parent node. This decision is supported by the following two attributes associated with every instance e at a tree node E .

states_e is a set which contains one flag for each parent node of E . Each flag is either *marked* or *not marked*, which indicates whether this instance has been used in an application notification involving the corresponding parent node, or not;

users_e is a set of identifiers that stores all the applicationIDs which were notified due to the occurrence of a composite event instance of which instance e is a component.

The event processing in EPS consists of four phases, each of which is responsible for the following

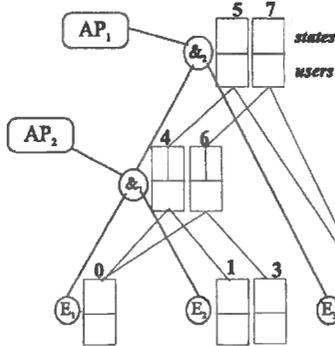


Figure 2: First Phase of the event processing

actions: the first phase creates intermediate event instances; the second and third phases eventually notify the application and mark all instances which participated in the notification; and the fourth phase does garbage collection, by removing parts of (unusable) instance trees.

6.1 First Phase

The first phase builds the instance trees. It is an incremental, bottom-up process, which happens every time an instance is delivered to a leaf node by the scheduler. The arrival of such instance causes the creation of additional instances at the corresponding parent node. The number of new instances that are created at such a node depends on the particular operator represented by the node and on the set of already existing instances at its child nodes. For example, at a node representing an operator “&” the instance set is the Cartesian product of the instance sets of the corresponding child nodes. At an operator “||” node the instance set is the union of the instance sets at the child nodes. This is because any of these instances may be used as an instance of the parent node (representing operator “||”).

The event operator “;” is a particular case of the operator “&”, with the additional constraint that an instance from the right branch (right child node) must be combined only with instances of the left branch which have a lower timestamp attribute. For example, if we assume that the left child of a node (representing operator “;”) has two instances, e_{l1} and e_{l2} with time-stamps $e_{l1}.t = 10$ and $e_{l2}.t = 12$ respectively, and a new instance, e_r with $e_r.t = 11$, arrives at the node’s right child, then because of the precedence constraint only one new composite instance, $e_{l1}; e_r$ is created at this composite node (representing operator “;”).

At nodes representing either operator `TimesEq` or operator `TimesDf` the set of instances is the set of all possible choices of N elements out of the set L of instances at the (single) child node. Thus, in this case we build an instance set with cardinality:

$$\binom{\text{card}(L)}{N}$$

The rules above are used for the creation of new intermediate instances at all levels of the event tree

except for the bottom level, which is composed of leaf nodes.

Notice that by our method we build composite instances by forming all possible combinations of the child instances. This form of composite event derivation (*All Combinations*) has some advantages compared to the more restrictive derivations defined by Snoop's[7] modes *recent*, *chronological*, *continuous* and *cumulative*, as discussed in Section 2.

Figure 2 depicts an event tree and the instance tree after processing the First Phase for the instances 0, 1, 2 and 3.

6.2 Second Phase

The second phase starts when a new set S of composite instances arrives at an application node representing an *RfN*. For each composite instance in S , the event condition of *RfN* is tested. If any composite instance satisfies the condition, then the corresponding instance tree is traversed down, and all the instances which collaborated to build the composite instance are marked. Otherwise, no action is taken.

For each new set S of composite instances that arrives at an application node, only one instance in S can trigger a notification to the application. This is because all composite instances in S have at least one primitive instance in common, but the application should be notified only once of the occurrence of this "common" primitive event. Accordingly, all other instances of set S (and its component instances) should be marked so that they are not used again, i.e. they are not able to cause new notifications. This marking is performed during the third phase (section 6.3).

Figure 3 shows how the instance tree looks like after the second phase, assuming that instance 7 satisfied the event condition of an *RfN*, causing a notification to application AP_1 . The black dots in this figure represent the marks at the event instances, as explained above.

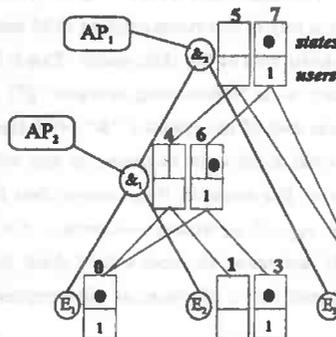


Figure 3: Second Phase of the event processing

6.3 Third Phase

In the second phase we could mark the instances which constituted the final composite event. However, many other instances use the same set or subset of the primitive instances of the final composite event. As mentioned in the section 6.2, all of these instances are not supposed to generate new instances and therefore should be marked too.

In order to mark all these instances, the third phase traverses the event tree up marking all instances that contain at least one primitive instance that is part of the composite event in S which caused the notification to the application (i.e. instance 7 in figure 3).

To ensure that no wrong instances will be marked, the third phase checks only the instances associated with the nodes of the event tree corresponding to the specific event expression which caused the notification and which initiated the second phase.

Figure 4 shows the instance tree at the end of the third phase.

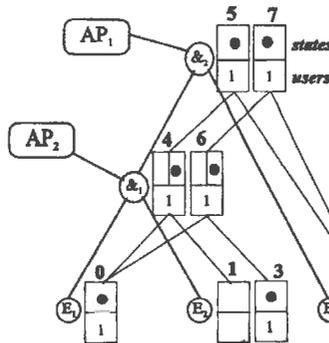


Figure 4: Third Phase of the event processing

6.4 Fourth Phase

The main goal of the fourth phase is to avoid an explosion of the number of event instances. After a notification, the instance tree may have many marked instances, and these instances cannot be used any more. The fourth phase traverses the event tree down, starting at the application nodes that have had a notification, and removing all the instances which (i) have a mark in *all* elements of states and (ii) do not have any more a parent instance.

Applying this phase we guarantee that an instance is only kept in the tree if it can be part of a composite component for some future notification.

Figure 5 shows the final instance tree after the whole event processing (phases one through four) caused by the occurrence of the primitive instances 0, 1, 2 and 3.

Although by this event processing method the first phase tends to generate large numbers of intermediate event instances, the marking and removal of instances performed in the phases two through four leaves

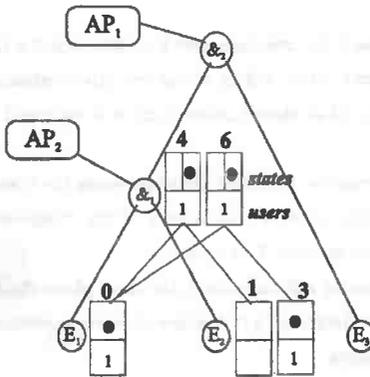


Figure 5: Fourth Phase (garbage collection)

only the minimum number of instances eligible for contributing to future notifications.

This is very important for ensuring a good processing performance. Since EPS uses the generic processing mode (c.f. section 6.1), the more instances are kept at a given level of the tree, the more composite instances at the upper levels will be generated from the levels below, yielding a direct impact on the monitor's performance.

7 Timing issues

In this section we discuss several time-related issues and explain how they are handled by the EPS.

Some well-known problems when monitoring distributed systems are the impossibility of achieving exact synchronization among the clocks of the hosts in a network and the unpredictable delay of message transmission. These problems have been taken into account in the design of EPS, in particular, in the way it orders incoming event instances and handles communication delays.

7.1 Handling of delays

Concerning the first problem, in EPS we assume that machine clocks are

approximately synchronized (with error rate $\leq \epsilon$) through the use of some time-synchronization protocol [5]. Based on this assumption it makes sense to compare the timestamp of events (received from sensors) in order to determine an approximate relative precedence.

In order to deal with the problem of message transmission delay in the EPS, we use the following two parameters:

detection window : is a window size Σ (i.e. a period of time in *ms*) used to determine acceptance of event instances for processing by EPS. An instance e can be accepted if its timestamp attribute $e.t$ is greater or equal to $CT - \Sigma$, where CT is the current time at the arrival of the event instance.

An appropriate choice for a value of Σ mainly depends on the kind of application which is using the EPS. A large Σ is recommended for applications with relatively long periods of stable states, i.e., where event instances are not generated frequently. Examples for such applications are monitors for chemical reactions. A small Σ is recommended for applications with frequent state changes, as for example, a monitor aiming to detect dependencies among concurrent computation tasks.

scheduler time : is a delay τ (in *ms*). This time is used to wait for event instances which have eventually arrived at the EPS with some delay. This is particularly important in a distributed environment, where the events instances sent by the sensors may arrive in a different order than their generation times.

7.2 Event Aging

Another problem related to the monitoring of events is that some instances of primitive and composite events will never be marked by any application, mainly because their attributes do not satisfy the corresponding event conditions in the *RfN*.

In order to get rid of those events, every event processing system needs to implement a garbage collection based on event aging. In the EPS, such garbage collection (based on event age) is done by traversing the tree top-down, scanning for and removing all instances e which have their timestamp attribute $e.t$ satisfying $e.t \leq CT - (1/n) * \Sigma$, where Σ is the detection window and n is the frequency in which the garbage collection is executed. During this traversal, whenever a composite instance is to be removed due to its age, we know also that all of its child instances can be removed as well. This is due to the fact that the time-stamp of a composite instance is always the maximum of the time-stamps of its child instances.

The frequency n of the garbage collection execution has significant influence on the performance of EPS. If the frequency is high, the number of event instances in the tree is lower, but each garbage collection execution will remove few instances. If the frequency is low, the instance tree will have more instances for a larger period of time, but each garbage collection execution will remove many instances. Thus, a reasonable periodicity for age-based garbage collection must be found.

8 Comparison with Related Algorithms

In this section we compare the algorithm for processing event trees used in EPS with those used in Snoop [7] and GEM [6] monitors. Unfortunately, no performance data of these other monitors have been published or made available, making it impossible to compare execution times for similar composite event expressions. Table 1 summarizes the differences in the algorithms used in the monitors.

GEM

Both GEM and EPS use a single algorithm for processing the trees representing composite events. The major differences among the algorithms are summarized in table 1.

Monitor	Trees	Instance Ordering	Tree Processing	Modes
EPS	shared	before tree processing	four Phases	generic
Snoop	shared	no ordering	one phase	four modes
GEM	simple	during tree processing	two phases	generic

Table 1: Comparison between the algorithms

The main difference between GEM and EPS is that the former does not consider the sharing of subtrees. This may cause the construction of several copies of a tree representing the same composite event requested by different monitoring application. Because of its simpler structuring of composite events (i.e. without sharing), GEM has also a simpler tree processing algorithm which consists of two phases: in the ascending phase the composite instances are created, and in the descending phase used instances are removed. Essentially, these two phases correspond to EPS's first and fourth phases.

The main disadvantage of GEM's lack of subtree sharing is that the following two actions have to be carried out whenever a new primitive event instance is delivered to the monitor: first, all trees with the corresponding event type have to be identified and then the processing associated with that instance has to be carried out in all the corresponding trees.

In order to reduce processing at the different trees, GEM employs *filters* that specify which instances are allowed to ascend the tree. This filter is a logical condition on the instance's attributes and may be associated with any node in the trees. By using these filters it is possible to reduce significantly the repetitive processing of instances in different trees.

Yet another important difference is related to the ordering of instances which arrive at the monitor. Unlike in EPS, where the ordering is performed before starting the first phase, in GEM the ordering is done during tree processing. This means that when an old instance arrives at GEM, it is put at the correct position with respect to the other instances at the same node.

Sentinel

As mentioned in section 2 Snoop defines four modes for processing composite events. However, sometimes there exist composite events of interest to some applications which cannot be expressed by any of these modes, as will be explained below.

Unlike Snoop, in EPS we chose to process events in a complete way, i.e. always building composite instances from all possible combinations of the instances at the child nodes. This generic processing mode is required when the instances are to be combined by considering only their *logical dependence* as expressed by the operators, rather than their temporal attributes.

This form of generic processing is required by some applications where not only the most recent events are relevant for detecting and managing abnormal situations. Examples of applications are network security management or dataflow monitoring in a distributed system.

For example, consider the case in which the goal is to detect cycles in message transmissions among

a set of processes, for example to avoid indefinite forwarding. In particular, let us consider the specific problem for three processes, P_1, P_2 and P_3 . Here we may have two types of events, *send type* and *receive type*, denoted by $[st, procID, msgID]$ and $[rt, procID, msgID]$, where *procID* and *msgID* are the attributes that indicate, respectively, the process at which the event occurred and a unique message identifier.

In this case, a cycle can be detected if a process receives a message that it has previously sent. This is essentially a sequence of the primitive event *send* followed by the primitive event *receive*, with matching attributes. Translating this into an event detection condition, it means that we are interested in composite events of the form $[st, X, Y]; [rt, Z, W]$, where $X = Z$ and $Y = W$ hold.

Considering the following order of instance arrivals (where j in e^j denotes the moment in which instance e has arrived):

$$[st, 1, 2]^1, [st, 2, 1]^2, [rt, 3, 1]^3, [rt, 2, 2]^4, [st, 3, 1]^5, [rt, 2, 1]^6,$$

it can be seen that the condition above can only be detected by considering all combinations of events.

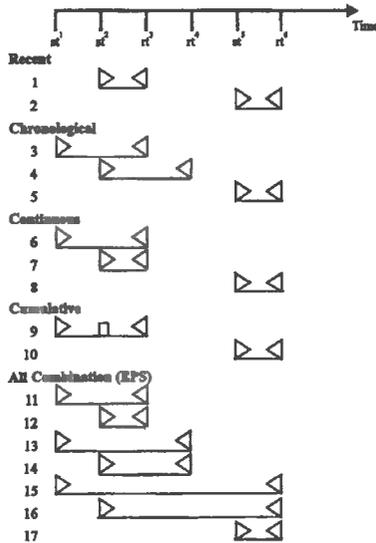


Figure 6: Derivation of composite instances in Snoop and EPS

Figure 6 shows which composite events would be generated in each of Snoop's modes (*recent*, *chronological*, *continuous* and *cumulative*) and EPS's generic mode (*all combination*).

After processing the six primitive event instances above, in the generic mode the set of derived composite instances would be $(st1,rt3)$; $(st2,rt3)$; $(st1,rt4)$; $(st2,rt4)$; $(st1,rt6)$, $(st2,rt6)$; $(st5,rt6)$, (lines 11 through 17 in Figure 6).

Now let us consider Snoop's modes, where the upward propagation of events is determined by the instances that start (\triangleright) and finish (\triangleleft) the detection of a composite event (the box symbol just represents the membership of a primitive instance in a composite instance). In *recent* mode the set of composite instance

is (st2,rt3); (st5,rt6). In *chronological* mode the set is (st1,rt3); (st2,rt4); (st5,rt6). In the *continuous* mode the set is (st1,rt3);(st2,rt3);(st5,rt6). Finally, the *cumulative* mode the set is (st2,st2,rt3);(st5,rt6).

The union of the sets generated by each of Snoop's mode would have four elements plus the additional three-instance element (st1,st2,rt3). Thus, the three other possible composite instances detected by EPS (lines 13, 15, and 16) would never be generated by Snoop, implying that such communication cycle would never be detected by Snoop.

As a consequence of adopting four specific modes, Snoop just goes up in the tree. During the upward propagation of instances in the tree, the Snoop monitor removes instances from the lower levels according to the processing mode in use. This means that Snoop combines the EPS's First and the Fourth phases in only one, while the Second and the Third phases are not necessary because a copy of each instance of the lower level is passed to the upper level. However, this approach is not suitable for generating all combinations (generic mode), where it is necessary to keep track of all instances that have arrived at the monitor. Without this, some combinations could be lost.

9 Performance of the EPS

In this section we present the factors which have the largest influence on the performance of the event processing algorithm presented in section 6. Unfortunately, a mathematically rigorous analysis of the order of complexity is very difficult in this case, because of the large number of factors and the difficulty of quantifying them. Based on the experience gained by testing EPS for various forms of shared event trees and different numbers of instances and arrival rates, we identified the main factors that determine the algorithm's performance. Below is a list in decreasing order of influence:

Types of Operators: Among the possible types of operators, some contribute with more event instances in the tree than others. For example, the operator "||" does not cause the creation of many instances, because it does not employ the Cartesian product of the instances at the corresponding child nodes. However, if the majority of nodes in the tree is of type operator "&", operator ";" or repetition (operator TimesDf and operator TimesEq), the number of instances will grow exponentially with the height of the tree.

Frequency of new primitive event instances: the more new event instances arrive at the EPS during a given period of time increases, the lower will be the overall performance of the algorithm. This is due to the fact that more instances remain in the tree and may be used when applying the Cartesian product for deriving higher-level composite instances.

Size of the detection window: it determines the period of time during which an instance is used to generate new intermediate instances in the tree. Thus, if the detection window is large, more instances remain in the tree and hence cause the creation of even more instances.

Distribution of instances in the tree: this is also a factor that has influence on the total number of instances in the tree. If the distribution of instances at the leaf nodes is unbalanced, most intermediate

instances will be concentrated on certain parts of the tree, and thus will contribute less to the generation of more intermediate instances. Thus, an uniform distribution of instance arrivals for all event types is worse for the overall performance than a non-uniform distribution.

Structure of the tree: the tree structure also has a twofold effect on the generation of intermediate instances, i.e. on the total number of instances in a tree. For example, comparing a binary balanced tree and a backbone-like tree (a linear tree where each node has a single additional node) with a same distribution of instances at the leaf nodes, the former will have less intermediate nodes than the latter, and hence will require less processing according to our algorithm.

Condition expression: the event condition associated with a RfN determines the number of instances that can be eliminated from the tree once the corresponding application node has been reached. For example, the more restrictive the condition is, the smaller is the chance of being able to satisfy the condition with a set of composite instances. Hence, less instances are consumed or cancelled and remain in the tree.

Most of the factors above depend on the nature of the application using the EPS, such as, common structures and conditions associated with composite events of interest, primitive instance arrival rates for the different types of events, etc. However, fine tuning of parameters such as the size of the detection window, or the periodicity of the execution of the Fourth Phase helps to improve the services performance. For example, if outdated instances are removed more frequently less instances are kept in the tree, which in turn reduces the amount of composite instances that are generated and processed.

10 Conclusion

Detection of composite events has been an active area of research in the database community, but more recently it is also being applied to other domains such as distributed computing, security and surveillance systems or analysis of financial markets.

Our interest in this subject arose from the area of distributed system monitoring and control, where we are developing an infra-structure for availability management of distributed programs. In particular, we required a service capable of detecting composite events, expressed in terms of primitive events detected by *sensors*, and describing complex relations of events concerning availability problems of software and hardware components. This lead to the design of a generic event detection service that can be used by any application program, e.g. a monitoring program, and which handles delays in the delivery of event messages from sensors caused by network latency.

The most significant contribution of this work is the use of shared trees to represent event expressions with common sub-expressions. The major advantage of this optimization is that it reduces significantly the cost of scanning the leaf nodes and evaluating sub-events that are common to many event trees. With our technique, the arrival of an instance of an event type appearing in a shared tree only once triggers the detection procedure for composite events.

As part of the design process, we have formally described this event processing method using a customized attribute grammar. In this description we specified each node of the event tree as an object and the interactions related to each of processing phase as invocations of the corresponding methods at adjacent tree nodes. Based on this formal description the actual programming of this event processing was carried out without any major problems.

The event processing service described in this paper was fully implemented in Java with the use of the RMI facility as the basic means for communication among the EPS clients, i.e. application processes and the sensors². We chose the Java language because it is object-oriented, portable across many environments, and because RMI provides a simple method invocation interface for remote applications. Inheritance was very helpful for implementing the classes for the tree nodes (representing the primitive event types and operators) and for structuring the EPS code in general. We used threads to implement concurrency within the EPS, and this was also very simple with Java. Since many threads manipulate the event and instance trees, we were forced to synchronize the access to them using Java's synchronization facility.

We tested the performance of the EPS for composite event expressions corresponding to extreme tree configurations. Figure 7 shows the total processing time for a binary balanced tree versus a backbone-like tree for different numbers of stored instances, i.e. instances that had previously arrived at the EPS. We measured the processing time from the moment of arrival of the last instance necessary for a notification until the moment of the actual notification to the application process. This instance was always corresponding to a deeply nested event type in the event expression. Unlike our initial belief the processing time with the backbone-like tree lead to a linear increase of time while the balanced tree shows an exponential behavior. This occurred because although we kept the total number of instances the same, their distribution was different in each tree. And this fact apparently had a bigger influence on the total processing time than the shape of the tree.

We also compared the total processing time for a tree shared by two applications with the total processing time for two copies of the same tree, and for different distributions of previous instances in the trees. Again, here we measured the time between the arrival of the last instance and the moment of the second notification to the application program. Figure 8 shows that the speedup gained when sharing trees is less for a binary balanced tree (tests 1 through 4) than for a backbone-like tree (test 5).

In all tests, the processing of events using shared trees was faster than using simple trees, as expected. But this difference corresponds only to the time saved in the tree processing. In addition, it is important to consider also the time spent to find the trees which have to be processed. Then, the total speedup gained is more than the one showed in the Figure 8. Because related work does not present any data of the time spent to find the trees, this information was not included in the graphics.

Our performance tests, and in particular Figure 7, also showed that EPS is able to process approximately up to 1000-2000 event instances per minute, and therefore is suitable for systems with a compatible instance creation rate (by all sensors).

Although the performance of the EPS was very good, its implementation could be optimized and the results could be even better, meaning a larger number of instances processed by minute.

²The EPS is public domain software and may be downloaded from <http://www.ims.usp.br/~moreto/eps/eps.tar>

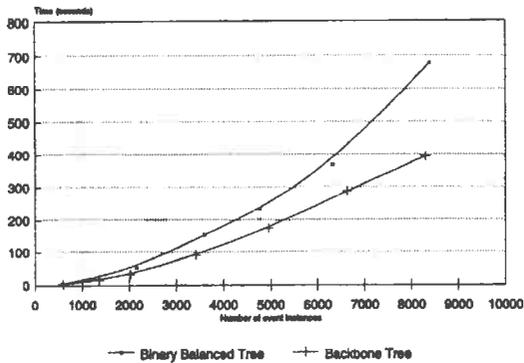


Figure 7: Processing time of balanced binary tree versus Backbone-like tree

10.1 Future Work

We are aware of the major limitations of our service, namely its centralized implementation and its assumptions about approximately synchronized machine clocks, and we plan to address these issues in future work.

A centralized implementation of any basic service embodies the problem of contention at concurrent access. In order to reduce the contention, the best solution is to have several servers, each being in charge of monitoring events of a certain class. Although this reduces the contention at each server, it makes the detection of composite events much more complicated. Distributed event detection is an interesting and challenging problem which we plan to investigate in future.

One direction of future work is also to implement a class which acts as an interface between the EPS and the application-specific sensors. In addition to facilitating the programming of these sensors this class would also implement a clock synchronization protocol among the communicating machines, making the EPS portable and less dependent on the existence of network clock synchronization services.

Another line of future work is to investigate methods for identifying event sub-expressions within a *RfN* that are already present in a previous event tree and implement simple transformations on event expressions in order to take advantage of the structure of an already existing event tree.

References

- [1] D. Tombros & K. R. Dittrich A. Geppert, M. Kradolfe. EvE, An Event Engine for Distributed Workflow Enactment. Technical report, Department of Computer Science, University of Zurich, May 1996.

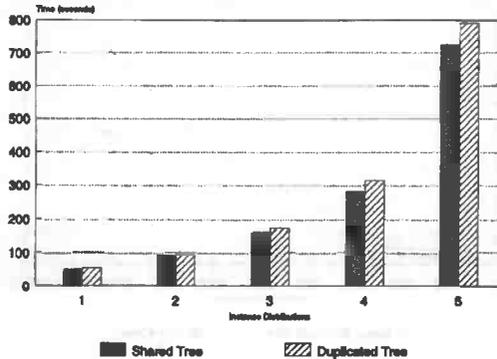


Figure 8: Comparison of shared tree versus duplicated tree

- [2] M. Endler and D.'Souza. Supporting Distributed Application Management in SAMPA. In *Proc. 3rd International Conference on Configurable Distributed Systems, Annapolis, USA*, pages 177–184. IEEE Computer Society, May 1996.
- [3] A.F. Garcia, D.M. Beder, and C.M.F. Rubira. An exception handling mechanism for developing dependable object-oriented software based on meta-level approach. In *Proc. of ISSRE'99, Boca Raton, FL*. IEEE, November 1999.
- [4] W. Hseush and Kaiser. Modeling concurrency in parallel debugging. *ACM SIGPLAN Notices*, 25(3):11–20, March 1990.
- [5] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.
- [6] Masoud Mansouri-Samani. *Monitoring of Distributed Systems Distributed*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, December 1995.
- [7] S. Chakravarthy & D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. Technical Report UF-CIS-TR-93-007, University of Florida, July 1993.
- [8] H. V. Jagadish & O. Shmueli N. H. Gehani. COMPOSE A System for Composite Event Specification and Detection. Technical report, AT&T Bell Laboratories, December 1992.
- [9] Y.S. Ramakrishna, P.M. Melliar-Smith, L.E. Moser, L.K. Dillon, and G. Kutty. Interval Logics and Their Decision Procedures, Part I: An Interval Logic. *Theoretical Computer Science*, 166(1 and 2):1–47, October 1996.
- [10] A. Geppert & K. R. Dittrich S. Gatzju. The SAMOS Active DBMS Prototype. Technical Report 94.16, Institut fr Informatik, University Zurich, 1994.

- [11] Jeffrey J.P. Tsai and Steve J.H. Yang, editors. *Monitoring and Debugging Distributed Real-Time Systems*. IEEE Computer Society Press, 1995.

RELATÓRIOS TÉCNICOS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO Instituto de Matemática e Estatística da USP

A listagem contendo os relatórios técnicos anteriores a 1997 poderá ser consultada ou solicitada à Secretaria do Departamento, pessoalmente, por carta ou e-mail (mac@ime.usp.br).

Flávio Soares Corrêa da Silva e Daniela Vasconcelos Carbogim
A TWO-SORTED INTERPRETATION FOR ANNOTATED LOGIC
RT-MAC-9801, fevereiro de 1998, 17 pp.

Flávio Soares Corrêa da Silva, Wamberto Weber Vasconcelos, Jaume Agustí, David Robertson e Ana Cristina V. de Melo.
WHY ONTOLOGIES ARE NOT ENOUGH FOR KNOWLEDGE SHARING
RT-MAC-9802, outubro de 1998, 15 pp.

J. C. de Pina e J. Soares
ON THE INTEGER CONE OF THE BASES OF A MATROID
RT-MAC-9803, novembro de 1998, 16 pp.

K. Okuda and S. W. Song
REVISITING HAMILTONIAN DECOMPOSITION OF THE HYPERCUBE
RT-MAC-9804, dezembro 1998, 17pp.

Markus Endler
AGENTES MÓVEIS: UM TUTORIAL
RT-MAC-9805, dezembro 1998, 19pp.

Carlos Alberto de Bragança Pereira, Fabio Nakano e Julio Michael Stern
A DYNAMIC SOFTWARE CERTIFICATION AN VERIFICATION PROCEDURE
RT-MAC-9901, março 1999, 21pp.

Carlos E. Ferreira e Dilma M. Silva
BCC DA USP: UM NOVO CURSO PARA OS DESAFIOS DO NOVO MILÊNIO
RT-MAC-9902, abril 1999, 12pp.

Ronaldo Fumio Hashimoto and Junior Barrera

A SIMPLE ALGORITHM FOR DECOMPOSING CONVEX STRUCTURING ELEMENTS

RT-MAC-9903, abril 1999, 24 pp.

Jorge Euler, Maria do Carmo Noronha e Dilma Menezes da Silva

ESTUDO DE CASO: DESEMPENHO DEFICIENTE DO SISTEMA OPERACIONAL LINUX PARA CARGA MISTA DE APLICAÇÕES.

RT-MAC-9904, maio 1999, 27 pp.

Carlos Humes Junior e Paulo José da Silva e Silva

AN INEXACT CLASSICAL PROXIMAL POINT ALGORITHM VIEWED AS DESCENT METHOD IN THE OPTIMIZATION CASE

RT-MAC-9905, maio 1999, pp.

Carlos Humes Junior and Paulo José da Silva e Silva

STRICT CONVEX REGULARIZATIONS, PROXIMAL POINTS AND AUGMENTED LAGRANGIANS

RT-MAC-9906, maio 1999, 21 pp.

Ronaldo Fumio Hashimoto, Junior Barrera, Carlos Eduardo Ferreira

A COMBINATORIAL OPTIMIZATION TECHNIQUE FOR THE SEQUENTIAL DECOMPOSITION OF EROSIONS AND DILATIONS

RT-MAC-9907, maio 1999, 30 pp.

Carlos Humes Junior and Marcelo Queiroz

ON THE PROJECTED PAIRWISE MULTICOMMODITY FLOW POLYHEDRON

RT-MAC-9908, maio 1999, 18 pp.

Carlos Humes Junior and Marcelo Queiroz

TWO HEURISTICS FOR THE CONTINUOUS CAPACITY AND FLOW ASSIGNMENT GLOBAL OPTIMIZATION

RT-MAC-9909, maio 1999, 32 pp.

Carlos Humes Junior and Paulo José da Silva e Silva

AN INEXACT CLASSICAL PROXIMAL POINT ALGORITHM VIEWED AS A DESCENT METHOD IN THE OPTIMIZATION CASE

RT-MAC-9910, julho 1999, 13 pp.

Markus Endler and Dilma M. Silva and Kunio Okuda

A RELIABLE CONNECTIONLESS PROTOCOL FOR MOBILE CLIENTS

RT-MAC-9911, setembro 1999, 17 pp.

David Robertson, Fávio S. Corrêa da Silva, Jaume Agustí and Wamberto W. Vasconcelos
A LIGHTWEIGHT CAPABILITY COMMUNICATION MECHANISM
RT-MAC-9912, novembro 1999, 14 pp.

Flávio S. Corrêa da Silva, Jaume Agustí, Roberto Cássio de Araújo and Ana Cristina V. de Melo
KNOWLEDGE SHARING BETWEEN A PROBABILISTIC LOGIC AND BAYESIAN BELIEF NETWORKS
RT-MAC-9913, novembro 1999, 13 pp.

Ronaldo F. Hashimoto, Junior Barrera and Edward R. Dougherty
FINDING SOLUTIONS FOR THE DILATION FACTORIZATION EQUATION
RT-MAC-9914, novembro 1999, 20 pp.

Marcelo Finger and Wamberto Vasconcelos
SHARING RESOURCE-SENSITIVE KNOWLEDGE USING COMBINATOR LOGICS
RT-MAC-2000-01, março 2000, 13pp.

Marcos Alves e Markus Endler
PARTICIONAMENTO TRANSPARENTE DE AMBIENTES VIRTUAIS DISTRIBUÍDOS
RT-MAC-2000-02, abril 2000, 21pp.

Paulo Silva, Marcelo Queiroz and Carlos Humes Junior
A NOTE ON "STABILITY OF CLEARING OPEN LOOP POLICIES IN MANUFACTURING SYSTEMS"
RT-MAC-2000-03, abril 2000, 12 pp.

Carlos Alberto de Bragança Pereira and Julio Michael Stern
FULL BAYESIAN SIGNIFICANCE TEST: THE BEHRENS-FISHER AND COEFFICIENTS OF VARIATION PROBLEMS
RT-MAC-2000-04, agosto 2000, 20 pp.

Telba Zalkind Irony, Marcelo Lauretto, Carlos Alberto de Bragança Pereira and Julio Michael Stern
A WEIBULL WEAROUT TEST: FULL BAYESIAN APPROACH
RT-MAC-2000-05, agosto 2000, 18 pp.

Carlos Alberto de Bragança Pereira and Julio Michael Stern
INTRINSIC REGULARIZATION IN MODEL SELECTION USING THE FULL BAYESIAN SIGNIFICANCE TEST
RT-MAC-2000-06, outubro 2000, 18 pp.

Douglas Moreto and Markus Endler
EVALUATING COMPOSITE EVENTS USING SHARED TREES
RT-MAC-2001-01, janeiro 2001, 26 pp.