Semantic Mapping of Events

Fabio Casati
Dipartimento di Elttronica Informazione
Politecnico di Milano
Milano, Italy

Weimin Du and Ming-Chien Shan
Hewlett-Packard Laboratories
Palo Alto, CA 94304, USA
{du, shan}@hpl.hp.com

Abstract

This paper addresses the problem of efficient management of events, in particular in those environments where events carry information useful to multiple applications, possibly operating in different domains and at different levels of abstraction. We investigate the problems and opportunities offered by such environments, and define a framework that enables a *semantic mapping* of events, i.e., enables the processing and successive refinement of events at different levels of abstraction, so that they can be understood and efficiently consumed by business applications.

We identify the requirements of an event mapping system and present a specification language, integrating high-level Petri nets and database query languages, which provides the required expressive power to specify complex event processing functions and includes a set of constructs that support the design process and allows efficient implementations.

Keywords: Interoperability issues and information modeling, event processing, semantic mapping, business events, Petri nets.

1 Introduction

In a number of environments and application domains there is the need of handling a huge number of events. Such environments typically include management and control platforms, such as telecommunication network management or automated driving systems, where both the software and the hardware components (called *event sources* or *event producers*) generate events in order to enable the ordinary and exceptional control of the system. A peculiarity of such systems, which complicates their management, is that events often arrive in storm, as a result of some failure or exceptional condition. For instance, a broken wire in a network may cause a storm of thousands of events notifying failure of every node on the wire, messages for traffic re-routing, notifications of undelivered packets, and so on. When the problem is fixed, a restoration storm of similar size occurs. Event management in such environments has been traditionally performed "by hand": human operators, like doctors, look at the events (symptoms) and try to extract useful information (such as the problem that caused the symptoms). While this approach can be acceptable for simple or small systems, it does not scale to complex systems, where the number of events is unmanageable without automated support.

Such events may be of interest to many applications (called *event sinks* or *event consumers*), possibly in different domains. For instance, a storm of events notifying node failures carries information which can be used by a diagnostic tool, by an application that monitors network performance, or by a customer care business process that checks whether service level agreements with the customers can be satisfied or not. Thus, multiple applications, possibly operating at different *levels of abstraction*, may be interested in the same events.

In this paper we present a service that enables *semantic mapping of events*, i.e., the processing of events notified by one or more event sources, in order to generate semantically meaningful events at different levels of abstraction for the different applications interested in the information that can be extracted from the notified events. The use of an event mapping facility raises applications from the task of performing event processing by themselves. A sophisticated event processing capability is in fact missing in most applications, and in particular in high-level, business applications, since these have not been designed for efficient event handling. For instance, a customer care workflow process would have to extract the knowledge relating the impact of the detected node failures on the customers by using the constructs and mechanisms offered by the Workflow Management System (WFMS) itself. This means that event processing can not be efficiently modeled or implemented. Furthermore, the use of specialized event processing components enables both the *factorization* of event processing, since different applications may share some event processing requirements, and its *distribution*, since event processing components can be structured in arbitrary architectures according to the physical locations of event producers and consumers and to the characteristics of the required mappings.

In this paper we first analyze the requirements of an event mapping system and present them by means of a simple example taken from the telecommunications domain, which is one of the environments with stronger need for efficient event processing. We then show how flexible event mapping can be performed by a set of distributed, independent event processing components, each capable of performing complex processing functions involving event correlation and access to external databases.

The complexity of the mapping needed by some applications requires a rich model for the specification of event processing. Traditional event processing models, such as those offered by active databases or event correlators, are not suited for this purpose, either because they lack the adequate expressive power or do not allow event processing based on the state of external databases. In order to overcome these limitations, we propose a formalism, integrating high-level Petri nets and database query languages, which provides the required expressive power to specify complex event processing functions and includes a set of constructs that support the design process and allows efficient implementations.

The paper is structured into two main sections: Section 2 details the requirements for event mapping, starting from the telecommunications network management case study, and shows the inadequateness of current approaches with respect to these requirements, while Section 3 introduces the formalism we propose for the specification of event mapping and discusses its main features. Section 4 concludes the paper with some remarks and details our future agenda.

2 Requirements for Semantic Mapping of Events

This section details the requirements for an event mapping service. We first introduce an example, taken from the domain of telecommunication management network (TMN). This simple example, serves the purpose of showing the requirements for developing an event mapping service[1]. We then analyze the requirements and discuss related work in this area, showing why we believe that current approaches to event processing do not meet these requirements.

2.1 The Network Management Case Study

In the case study we assume that a network is composed of interconnected nodes; nodes can be recursively grouped into subnetworks, thereby originating a tree of subnetworks, whose root is the entire network. Analogously, physical components in the network (e.g., routers, wires, adapters, etc.) are also organized into tree structures, so that a component may be made of multiple sub-components. We assume that the network structure and topology are stored in a database.

Many events can be generated by the network components; typical notifications include loss or restoration of network signals, loss of packets, protocol errors, or equipment malfunctions. We will refer to these events as *raw events*. Raw events are detected and notified by network

[1] The case study is based on the TMN international standards *SNMP/CMIP* [21] and *NMF - Business Agreements* [18]

monitoring components, and may carry several parameters denoting for instance the affected component or the timestamp of the event occurrence.

Such events may be of interest to many applications. In our example we consider three of them.

- A **Network diagnostic application** that needs to detect failures in the network, such as equipment malfunctions or broken wires. We refer to the events of interest to the network maintenance application as *network physical events*, since this application wants to know what physically happened in the network, i.e., what is the problem that caused the event notifications.
- A **Network performance monitor** that keeps track of the performance and quality of service provided by the network. We refer to the events of interest to the network performance monitor as *network impact events*, since the application is interested in understanding the impact on the network performance caused by network failures
- A **Customer care** application (e.g., a workflow process) that is interested in determining the impact of performance degradation on each customer, for instance to open trouble reports or offer discounts to damaged customers. We refer to the event of interest to this business process as *service events*, since the application is interested in understanding how the service (for specific customers) is affected.

Our aim is to develop a mechanism enabling the mapping of raw events into network physical, network impact, and service events, in order to provide each application with events at the appropriate level of abstraction. In the following we characterize the four categories of events and show what kind of processing is needed in order to generate high-level events starting from raw events.

- **Raw events** are those raised by the network equipment or by low-level monitoring applications. Examples of raw events and parameters are *LossOfSignal(NodeId)*, *AdapterError(ComponentId)*, *ComponentMalfunction(ComponentId)*, *MessageDelivered(Message)* or *CommunicationsProtocolError (PairOfNodes)*.
- **Network physical events** are the result of a processing over raw events that aims at determining the causes that actually generated an event storm. Examples of network physical events are *netFailure(networkId)* or *ComponentMalfunction(ComponentId)*. Event *netFailure(N)* is raised as node failures are notified from every node in a network N within a specified time interval, and if the same event can not be raised for an ancestor of N. An analogous process is followed for determining the occurrence of the *ComponentMalfunction* event. The detection of these events involves correlating raw events and/or previously detected network physical events. In this example the

correlation is particularly complex since the number of events to be correlated is not fixed but depends on the considered subnetwork and on its topology (we thus refer to it as *dynamic correlation*). Note that knowledge of the network structure, stored in a database, is also required. The detection of network physical events also requires *filtering* capabilities: for instance, a diagnostic tool might only be interested in notifications coming from a given subnetwork, so that events involving other subnetworks are disregarded, or it might want to discard false alarm, so that when a node failure event is immediately followed by a node restoration event, it is not processed further.

- **Network impact events** describe the impact of a failure on the network performance. Examples of such events are *PerformanceLevelChange(networkId, oldLevel, newLevel)* or *BandwidthChange(PairOfNode, oldBw, newBw)*. The semantics of these events can be defined in terms of network physical events, rather than referring directly to raw events. The performance level is changed on the basis of some functions over the message delivery times with respect to the expected ones, of the frequency of failure of critical components, and so on. The event *BandwidthChange* is raised whenever the bandwidth between two nodes changes. Note that this could be deduced by failures in nodes or system components, but could also be caused by new node connections or more powerful equipment. These latter changes can be described by database modifications.

- **Service events** define how network impact events affect the Service Level Agreements (SLAs) and the customers. Examples are *create(TroubleReport)* or *SLAviolation(SLA, Trouble)*. A trouble ticket, for instance, is created whenever a problem that affects a user of the network is detected. This requires knowledge of which user is affected by which failures, and again this can be deduced by analyzing network impact events and the database describing the service level agreements stipulated with the customers.

2.2 Requirements Analysis

The case study shows that mapping events to higher levels of abstraction can be obtained by a sequence of *event mappers*, organized as shown in Figure 1: mapping from *raw* to *service* events can be achieved in multiple steps, where intermediate results represent events at the appropriate semantic level for an event consuming application. Each event mapper receives events from other (lower level) mappers or from generic event sources, processes the events on the basis of the event parameters, the state of the application's database, and the occurred event history, and feeds the output events to event consuming applications or to other (higher level) mappers.

Mappers at different levels typically have different focuses: mappers closer to event sources have a greater event load and their processing mainly consists in filtering and correlating events. On the other hand higher lever mappers must usually handle a reduced number

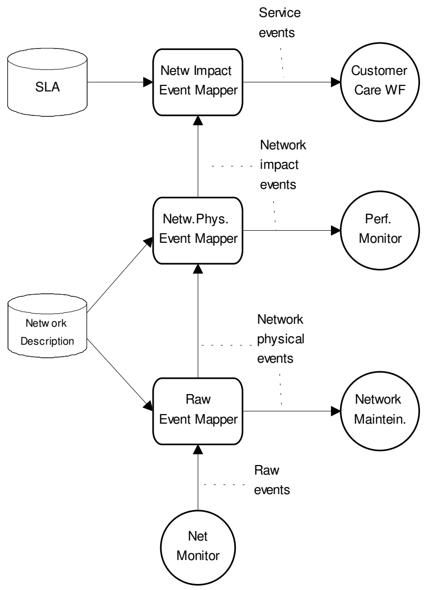


Figure 1: An architecture for event mapping in the telecommunication environment

of events and their processing involves interaction with the applications' data bases. In the following section we define the requirements for a *generic* mapper, suitable for both low-level and high-level event mapping, so that the mapping of events from the producers to the consumers can be achieved by a set of distributed, independent mappers, that can be combined in a flexible way, according to the characteristics of the environment and of the applications (e.g. distribution of producers, consumers, and data required in order to perform the mapping), as shown in Figure 2.

1. **advanced filtering/correlation capabilities**: the event processing service should be capable of performing complex filtering and correlating functions. The language for specifying filtering and correlation functions should offer "traditional" correlation constructs such as conjunction, disjunction, sequence, and repetition, and should enable correlation based on event attributes. Furthermore, it should allow the specification of:

   (a) *domain-specific correlation*, i.e., correlation based on domain specific knowledge. For instance, the network topology is typically needed in order to determine the impact of node failures on network performance, and knowledge of service level agreements is required in order to determine when these cannot be met.

   (b) *dynamic correlation*, i.e., a correlation in which the number and type of events involved can only be determined at run-time (as it is, for instance, in the

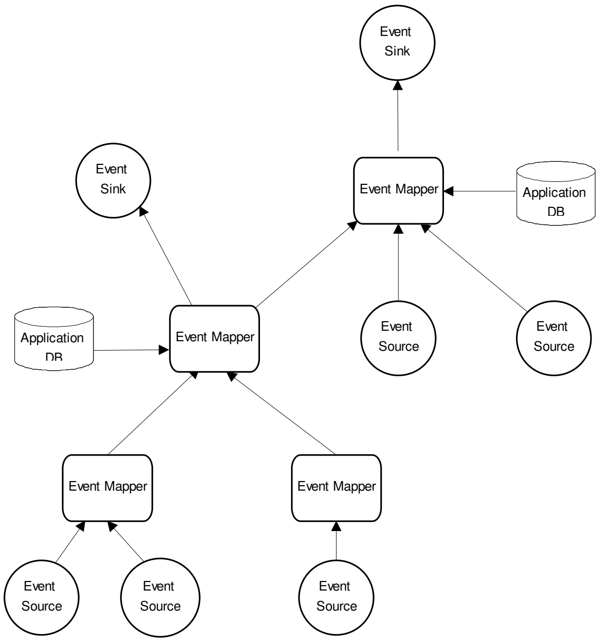


Figure 2: An architecture for semantic mapping of events

correlation that determines the occurrence of the *netFailure* event).

(c) *raise unless* behavior: the formalism must allow the definition of events to be raised unless a summarizing event can be raised. For instance, we might want to notify that a component is malfunctioning unless we detect that the containing component is malfunctioning, in which case only the latter event is raised. Note that the raise unless behavior also implies a sophisticated event consumption mechanism.

2. **detection of data and temporal events**: meaningful events include modifications of external application's database (e.g., changes in the service level agreements) as well as temporal events, raised as a specified temporal instant is reached. It should be possible to capture these events and define reactions to these events.

3. **transparent event base management**: in order to perform event correlation, occurred events must be stored in a suitable event base, and retrieved when needed for correlation. The mapping system should take care of managing the event base according to the defined correlation functions, without requiring explicit maintenance by the user.

4. **performance**: some applications may have stringent performance requirements. This typically happens when dealing with events at the lower levels of abstraction, where event storms may be composed of thousands of events, all generated within a very narrow time window (usually a few seconds).

In the remainder of this section we describe current approaches to event processing and show why they do not meet these requirements.

2.3 Current Approaches to Event Processing

Two significant research areas in which event management has been studied are *active databases* and network *event correlation engines*. In active databases (see [22] for a review) event processing is specified by means of Event-Condition-Action (ECA) rules: as the specified event occurs, the condition (usually a query over the DB) is tested, and if it is satisfied the action part is executed. Active database languages are well-suited for defining data events and their processing based on the database state. Some of these (e.g., Ode [12, 13], HiPAC [6], Snoop [5], Samos [11], and Chimera [17]) allow to correlate events by providing an *event algebra* that enables the composition of elementary events in order to define *composite events*. Typical composition operators include conjunction, disjunction, sequence, and repetition.

Drawbacks of the Active Database Approach: The constructs and functionality provided by active databases are well suited for processing business, high-level events typical of database applications, but do not meet basic requirements for event mapping, such as the capability of performing *dynamic correlation* and the *raise-unless* behavior. In fact, event composition operators do not allow the definition of correlations whose exact form can only be determined dynamically, having arbitrary dependencies on factors such as event history, current database state, or timing constraints. With active database languages, dynamic correlation can only be "simulated" by the designer, who has to define dummy rules, events, and data structures in order to specify the required semantics. Analogously, no constructs are provided for defining the *raise-unless* behavior, which must also be simulated with a considerable modeling effort, required in order to specify when a detected composite event should or should not actually be raised. These drawbacks are also due to the limited flexibility which is offered by the event consumption mechanism. Finally, active databases do not provide the required performance, particularly for processing storms of events in real time. Similar drawbacks can be found in other rule-based event processing systems such as the WIDE exception handler [2] and the YEAST general purpose event processing system [16].

On the opposite side, network event correlators (such as ECS developed by HP [8, 20] or InCharge by Smarts [23]) have been designed in order to efficiently process low-level events. The formalisms for the definition of event processing reflect this need, by requiring detailed and low-level specifications, resulting in products capable of performing very fast event correlation. For instance, the ECS model provides the user with a set of *event processing nodes*, each performing a specific processing function. The nodes can be combined in a graph, and events flow through the graph, being transformed at every node. Special nodes are provided in order to store events to be used for future event correlations. Insertion and deletion of events in these nodes must be explicitly specified.

Drawbacks of the Event Correlator Approach: The main drawbacks of network event correlators are the lack of integration with database systems and the complexity of event processing. These systems do not allow the specification of domain based correlations, i.e., of event processing based on the content of a database, but only consider event attributes and event history; this is mainly due to the fact that they have been designed to achieve filtering/correlation over event storms. The complexity of event specification is instead due to the need for very efficient processing, which causes the specification language to be closer to an assembly language rather than to a high level language, although graphical interfaces are provided to the designer. Furthermore, the event base needs to be explicitly maintained by specifying when events can be discarded and are not needed for future correlations.

3 Event Processing Nets

This section introduces a formalism for specifying event mapping, called *Event Processing Net (EPN)*, which aims at satisfying the requirements and at overcoming the limitations of current specification formalisms. We first describe EPN and show its functionality by means of a few examples, then we introduce design support constructs, and finally we discuss the main features of the model, relating them to the current approaches. A complete description of the formalism can be found in [3].

3.1 The EPN Model

The EPN model is based on a modification of *Coloured Petri Nets (CPN)* [14], and reuses concepts defined in [10], where Petri nets are used in order to detect the occurrence of composite events. EPN combines the integration with database systems offered by active databases languages with the powerful correlation mechanisms offered by graph-based formalisms typical of network events processors. The basic idea is to represent events occurrences with typed tokens, and to specify the event processing by means of a high level Petri Net, where the event correlation is defined by the net structure and the interaction with the database is captured by *transition guards*, corresponding to queries over the data and event base state that (1) determine if the transition can indeed be activated, (2) define the events (tokens) that must be removed from the input places, and (3) allow the extraction of data from the DB to be used for enriching the semantics of the output token.

Every event type biunivocally corresponds to a place in the net, and the occurrence of an event of a given type corresponds to the insertion of a token into the associated place. Event types are distinguished by their names, and may have several atomic attributes. Using the CPN terminology, the event type determines the *colour set* of the corresponding place, while the attribute values of the occurred event determine the colour of the token. Event types, along with their attributes, are defined in a declaration section associated to each net. A distinguished attribute implicitly defined for each token and assigned by the system is the *timestamp* of the event detection.

We assume that events are notified to the mapper by external applications, and we do not discuss the issue of their detection here, which depends on the context in which the mapper is embedded. We further assume that events are delivered in the required format and to the appropriate mapper by means of suitable *event dispatchers* (such as, for instance, Ambrosia [1]). The only exception is represented by *temporal events*, whose occurrence is detected by the mapper itself: tokens in the corresponding place are inserted by the system as a specific or periodic temporal instant is reached, according to the event definition provided in the declaration section (e.g., at Christmas 1998 - specified as *"1998-12-25 0:00:00"* - or "every 3 days" - specified as *"every 3 0:00:00"*).

Formally, an EPN is a tuple $EPN = (\Sigma, P, T, A, C, G, E)$ where:

Σ is a finite set of event types.

$P = P_I \cup P_O \cup P_M$ is a finite set of places; the set of places is divided into three subsets: P_I represents *input places* of the net, and tokens in these places are inserted as a result of events notifications or temporal event occurrences. P_O denotes the set of *output places*, from which tokens are immediately removed, meaning that they are dispatched to other mappers or to event consumers. P_M denotes *mapping places*, and are introduced in order to ease the specification of event mapping (otherwise, a one-step mapping would require very complex transition guards).

$T = T_I \cup T_O \cup T_M$ is a finite set of transitions. T_I represents *input transitions*, which fire on event notification or detection of event occurrences. They have no input place and one output place, belonging to the P_I set. T_O denotes the set of *output transitions*: they have exactly one input place, belonging to the P_O set, and no output place. They remove the tokens from the output place, thereby forwarding the corresponding event to the dispatcher. T_M denotes the set of *mapping transitions*: they may have one or more input places and zero or more output places, and are those that actually perform the mapping. If a mapping transition has no output places, its firing causes tokens to be removed from the input places but no token to be generated.

$A = IA \cup OA$ is a finite set of arcs; IA is the set (p, t) of arcs in input to transitions, where $p \in P$ and $t \in T_M \cup T_O$. OA denotes instead the set (t, p) of arcs in output from transitions, where $p \in P$ and $t \in T_I \cup T_M$.

$C : P \leftrightarrow \Sigma$ is a *colour function*, associating an event type (colour set) to each place.

G is a (biunivocal) *transition guard function*, associating a query to every mapping transition T_M.

E is an *arc expression* function. Input arcs (IA) are associated to couples $\langle V, t\rangle$, where V is an object variable that will define tokens to be removed from the input place and t is time interval defining how long tokens must stay in the place before they become *active*, i.e., they can be considered by the transition (default is that tokens can be immediately considered). Arcs in output to transitions (OA) are instead associated to a list of variables $(V_1, V_2, ..., V_n)$ defining the colour of the generated output tokens.

The token game has rules analogous to high-level Petri nets: a transition is enabled as each input place contains at least one active token and as the transition guard evaluates to true (i.e., the result of the associated query is non-empty). Token colours may be inspected by the transition guard, in order to determine if the transition is enabled, and the colours of the input token concur in determining the colour of the output tokens. Transitions guards in EPN are expressed in the declarative object-oriented database language *Chimera*. The language is not described in the following, and self-explanatory examples will be used. The interested reader is referred to [4] for a detailed description of the language. The Chimera query may access both the state of an external database and the colour of the tokens in the input places. In order to provide an uniform way to access data, each place is associated to an object class and tokens in the input places are referred by the query language as objects of the corresponding class. Thus, transition guards access the database state and the event attributes with the same formalism. For instance, if a transition has `netFailure` as input place, the query may refer to tokens of the place as objects of a `netFailure` class, implicitly defined.

Arc expressions define event consumption and generation, i.e., define tokens that must be removed from the input places and inserted in the output ones. Each input arc is associated with an object variable, ranging over the objects (tokens) in the input place. Conditions over these variables may be imposed within the transition query, that will possibly restrict their range. Tokens which remain bound to the variable after the condition evaluation will be removed from the input place. Output arcs are instead associated to a sequence of variables $(V_1, V_2, ..., V_n)$, whose value is again defined by the evaluation of the guard. The semantics is that a token is inserted for each different element of the cartesian product of the bindings determined for variables $V_1 \times V_2 \times ... \times V_n$. The values of the variables for each binding are assigned to the attributes of the generated token, with a positional notation. The use of these constructs is exemplified in the next subsection.

3.2 Examples of EPN Specifications

A simple transition guard implementing a filter is shown Figure 3(a). Tokens in the input places denote notifications of network failures. The purpose of the mapping is to filter failures of low-priority networks, focusing on high priority ones. The transition guard of Figure 3(a) determines the relevant network object in the topology database (`F.netId=N.netId`), and checks if it is a high priority network (`N.priority="high"`). The query thus restricts variable

F to range over tokens denoting notifications of failed high priority networks, which are then removed from the input place. For every token removed from the input place, a token whose value will be determined by variable `F.netId` is inserted in the output one. In the Figure, the `netFailure` place is an input place, meaning that tokens are inserted due to event notifications from external applications, while `criticalFailure` is an output place, meaning that tokens are dispatched to other mappers or to event sinks.

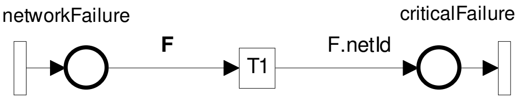
Figure 3(b) shows a more complex EPN, defining a part of the raw to network physical event mapping. The only events in input to the mapper are `netDown` and `netOK`. Transition T1 filters erroneous or temporary network failures, i.e., those followed within 5 seconds by a network restoration notification (note the time interval specification included in the arc expression, meaning that tokens becomes active for that transition after 5 seconds). This is achieved by removing tokens from the `netDown` place if a token with the same `netId` attribute value is inserted in `netOK` (the guard expression of T1 performs the matching). Notice that T1 has no output place, meaning that no token is generated due to the consumption of tokens in input places. Transition T2 removes tokens that remained in `netDown` for more than 5 seconds, so that they cannot be any more accessed by transition T1. Thus, actual net failures (`actualNetFailure`) are determined. Transition T3 provides a summarized view of net failures: if all subnetworks N of a parent network P notify a failure within a 10 second window, then transition T3 causes the notification of failure of P rather than notifying the many subnetworks failures (a token notifying the failure of P is generated, while tokens relating failures of subnetworks N are removed). Finally, transition T4 produces the output events defining which are the relevant component that possibly caused failures (`affectingComponent`) and the failed networks (`summarizedFailure`). The EPN specification therefore allows mapping network failure notifications, by filtering false alarms and providing a summarized, high level view of failed networks and of the relevant component.

3.3 EPN Design Support Constructs

Analogously to hierarchical CPNs [14], the EPN model offer modularization features that allow constructing a complex net by composing simpler nets. A *hierarchical EPN* specification is composed of a several diagrams, organized in a tree structure. Transitions in higher level diagrams can be expanded in lower level diagrams which precisely define the activity performed by the transition. Thus, the top-level diagram gives an overall view of the event processing specification, while the details are hidden in low-level diagrams. Diagrams can also be organized into a library and reused for different EPN specifications.

Another feature that simplifies the EPN design and allows efficient implementations involves the identification of widely used event processing *patterns*. In particular, we recognize the need of frequently occurring filtering structures, such as the one of Figure 3(a), or *unless* structures, forwarding an event A unless an event B is raised within a defined time window.

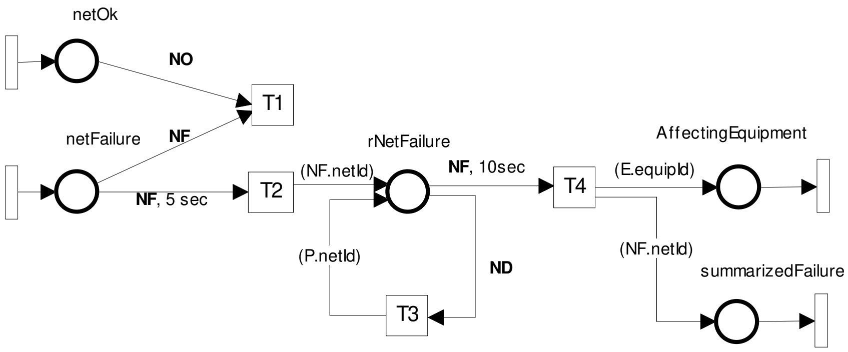
Declarations
Events:
network Failures
attributes netId: integer
end;
criticalFailures
attributes netId: integer
end;



T1: network(N), F.netId=N.netId,
N.priority="high"

(a)

Declarations
Events:
netOk
attributes netId: integer
end;
netFailure
attributes netId: integer
end;
r*NetFailure*
attributes netId: integer
end;
summarizedFailures
attributes netId: integer
end;
affectingEquipment
attributes eqiupId:integer
end;



Transition Guards:
T1: NF.netId=NO.netId,
NF.timestamp<NO.timestamp;

T2 : -

T3: network(N), network(P), N.parent=P, N.netId=NF.netId,
card(N)=card(A where network(A), A.parent=P);

T4: Equipment(E), NF in E.affectedNetworks;

(b)

Figure 3: Examples of EPNs. (a) A simple filtering net, and (b) a more complex EPN describing the mapping from *raw* to *network physical* events

The EPN model offers a number of special-purpose, predefined transitions that allow a simple specification of the most common mapping functions. For instance, the distinguished *unless* transition is a transition having two input places A and B and one output place C: when a token is inserted in place A, it is forwarded to place C *unless* a token is inserted in place B in the next x seconds. Predefined transitions considerably reduce the modeling effort and represent a basic mechanism for achieving the required performance, by allowing ad-hoc implementations. This is particularly important in low-level mappers, where event processing usually involves the use of a small set of processing patterns, and where the need for fast event processing is stronger. A similar approach is in fact followed by network event correlators, that offer a number of predefined and ad-hoc implemented event processing nodes that must be composed to specify the overall event processing.

3.4 Discussion of EPN Features

In the following we summarize the main strengths of the model. EPN overcomes the limitations of current approaches to event processing, and satisfies the requirements stated in section 2.2: in fact, the integration of the net structure with a query language provides the required expressive power for defining complex correlation functions, including dynamic and domain-based correlations. Event processing can be based on event attributes, occurred event history, and state of an external database, and knowledge can be extracted from the database in order to enrich the semantics and informative content of the mapped events. Furthermore, temporal constraints can be easily intregrated into an EPN specification, as well as definition of reactions to temporal events.

EPN also allows a flexible event base management, implicitly specified in the definition of the net: events that must be kept in the event base are specified by tokens in the net; when a token is removed from a place, the corresponding event can be discarded from the event base, since it cannot be used any more for future event correlations. No explicit insertion or deletion from the event base must be specified, but still different consumption policies can be defined for each event. The *raise unless* behavior is guaranteed by the net structure, where only tokens in the output places are in fact forwarded to the event dispatcher. In the example of Figure 3(b) the raise unless behavior appears in two cases of different nature: transitions T1 and T2 cause net failures to be propagated *unless* a notification of a restoration occurs shortly after. Transition T3 causes a net failure event to be raised *unless* a failure of its containing network can be determined.

This EPN specification would be practically unmanageable with active database languages, since it would require the creation of dummy classes and the design of a complex, interacting set of rules whose cooperation must be carefully designed, in particular in order to simulate the dynamic correlations and raise-unless behaviors required by this mapping. Analogously, these semantics (and also that of Figure 3(a)) can not in general be specified by means of network event correlation engines, due to their inability of performing domain-based correlations.

EPN also enables the specification of flexible and complex mapping functions with a reduced design effort: in fact, the graph-based nature of the formalism helps in structuring event processing specification (mapping performed only by guards becomes extremely complex and practically unmanageable); furthermore, the design is supported by modularization constructs and by predefined, special purpose transitions that implement frequently occurring event mapping patterns, besides allowing the realization of efficient implementations.

4 Concluding Remarks

In this paper we have introduced the notion of semantic mapping of events, in order to bridge the semantic gap between events at different levels of abstraction, and we have shown that this corresponds to a need of many applications aiming at extracting information from the same set of events. Appropriate events suited for the (different) levels of abstraction of the applications can be generated by a set of independent, general purpose event processing components, which we have called event mappers. Event mappers enable the processing of events and feed mapped events to event consuming applications or to other event mappers. A wide range of applications can benefit by such components, since they do not need to perform complex event processing that can in turn be delegated to specialized components. Based on a case study relating the telecommunication network environment, we have analyzed the requirements that must be satisfied by the event mappers. In particular, we have identified that mappers need to process events according to event parameters, previously occurred events, and possibly according to the state of an external database. We found that the formalism for specifying event processing must allow the definition of complex correlation functions, allowing in particular the specification of dynamic and domain-based correlations and of the raise unless behavior. Also, we showed that a flexible mechanism for defining event consumption and event base maintenance is needed in order to a define a variety of mapping functions which are needed in practice. Finally, we have introduced a formalism, called EPN, which integrates high level Petri nets and database query languages in order to enable a flexible and powerful event processing specification, which meets the specified requirements and provides a set of constructs for supporting the design process, such as net hierarchies and predefined transitions.

Our future agenda includes the development of static and dynamic EPN analysis criteria that helps the event processing designer in understanding the properties of the mapping net, and we plan to translate other event processing specification formalisms into EPN, in order to enable the analysis and implementation of different formalisms within the same framework.

5 References

[1] Ambrosia. Ambrosia Event Management System, Version 1.1: Concepts and User Guide. *Open Horizon Inc*, 1996.

[2] F. Casati, S. Ceri, B. Pernici, G. Pozzi. Specification of the Rule Language and Active Engine of Foro V.1, *WIDE Technical Report 3008-6*, 1997

[3] F. Casati, W. Du, M.C. Shan. *Semantic Mapping of Events*, *Hewlett Packard Labs., HPL-97-69*, Palo Alto, CA, 1997

[4] S. Ceri, R. Manthey. Consolidated Specifications of Chimera, *Technical Report IDEA DE.2P.006.01*, 1993

[5] S. Chakravarthy, V. Krishnaprasad, E. Anwar, S-K. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection, *Procs. of the 20th Int'l Conf. on Very Large Databases*, Santiago, Chile, Sept. 1994.

[6] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. Carey, M. Livny, and R. Jauhari. The HiPAC Project: Combining Active Databases and Timing Constraints, *SIGMOD Record* 17(1): 51-70 (1988)

[7] O. Diaz, N.Paton, P. Gray. Rule Management in Object Oriented Databases: A Uniform Approach. *Procs. of the Int'l. Conf. On Very Large Data Bases (VLDB)*, Barcelona, Spain, 1991

[8] HP. HP OpenView Event Correlation Services: Technical Evaluation Guide. *Hewlett-Packard Document*, 1996

[9] S. Gatziu, K. Dittrich. Events in an Active Object-Oriented Database System. *Procs. of the Workshop on Rules in Database Systems*, Edinburgh, 1993

[10] S. Gatziu, K. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. *Proc. of the Int'l Workshop on Research Issue in Data Engineering*, Houston, Texas, 1994

[11] S. Gatziu, H Fritschi, A. Vaduva. SAMOS an active Object-Oriented Database System: Manual. *University of Zurich, Internal Report 96.02*, 1996

[12] N. Gehani, H. Jagadish, O. Shmueli. Event Specification in an Active Object-Oriented Database. *Procs. of the Int'l Conf. On Management of Data (SIGMOD)*, 1992

[13] N. Gehani, H. Jagadish, O. Shmueli. Composite Event Specification in an Active Databases: Model and Implementation. *Proc. of the Int'l Conf. On Very Large Data Bases (VLDB)*, Vancouver, Canada, 1992

[14] K. Jensen, Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1992

[15] G. Kappel, S. Rausch-Schott, W. Retschitzegger, S. Vieweg. TriGS - Making a Passive Object-Oriented Database System Active, *Journal of Object-Oriented Programming (JOOP)*, 1994

[16] B. Krishnamurthy, D. Rosenblum. Yeast: A general Purpose Event-Action System *IEEE Transactions on Software Engineering* Oct. 1995

[17] R. Meo, G. Psaila, S. Ceri. Composite Events in Chimera. *Procs. of the Int'l Conf. On Extending Database Technology (EDBT)*, Avignon, France, 1996

[18] The Network Management Foundation. Service Provider to Customer Performance Reporting Business Agreement, NMF document 503, Issue 1.0, March 1997.

[19] N. Paton, O. Diaz, M. L. Barja. Combining active rules and metaclasses for enhanced extensibility in object-oriented systems. *Data and Knowledge Engineering 10*, North-Holland, 1993

[20] K. Sheers. HP OpenView Event Correlation Service, *Hewlett-Packard Journal*, 1996

[21] W. Stallings. SNMP, SNMPv2, and CMIP: The practical Guide to Network-Management Standards. Addison Wesley, 1993

[22] J. Widom, S. Ceri. Active Database Systems, Morgan-Kaufmann, 1996

[23] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, D. Ohsie. High Speed and Robust Event Correlation, *http://www.smarts.com/products.html*