MANAGING EVENT PROCESSING NETWORKS

Louis Perrochon
Stephane Kasriel
David C. Luckham

Technical Report No.: CSL-TR-99-788

October 1999

This Project is in part supported by Air Force grant F30602-96-2-0191 and Navy grant N00014-93-1-1335.

Managing Event Processing Networks

Louis Perrochon, Stephane Kasriel, David C. Luckham

Technical Report No.: CSL-TR-99-788

October 1999

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
William Gates Computer Science Building, 4A-408
Stanford, CA 94305-9040
<pubs@shasta.stanford.edu>

Abstract

This technical report presents Complex Event Processing. CEP is a fundamental new technology that will enable the next generation of middleware based distributed applications. CEP gains information on distributed systems and uses this knowledge for monitoring, failure analysis or prediction of activities.

A very promising route in CEP research is that of Event Processing Networks, which is one of the main areas of research of the Program and Analysyis Group at Stanford University. Event Processing Networks are one way of describing and building CEP, by successively filtering meaningful information and aggregating the corresponding events into higher levels of abstraction.

This reports describes in detail the foundations and aims of Complex Event Processing. Then we will introduce the concept of Event Processing Networks and describe their use in the context of Complex Event Processing. Finally, we will describe the architecture of the CEP system.

Key Words & Phrases: Complex event processing, distributed applications, network and systems management

I. Complex Event Processing

This chapter introduces the concept of Complex Event Processing, the problem that it addresses and how it aims at solving those problems. The chapter also gives an overview of how Event Processing Networks contribute to Complex Event Processing. Finally the chapter briefly describe the Stanford University CEP technology and illustrates how to use it to solve a CEP problem.

A. Introduction

Complex Event Processing (CEP) gains knowledge of a complex system in real-time based on *events* that denote the system's activities. A system can be anything from a single semiconductor fabrication line to the interconnected check-out registers of a nation-wide retailer.

Such systems may be probed to *produce* events as the system operates.

Events are then *processed* in a multitude of ways: Unwanted events are filtered out, patterns of logically corresponding events are aggregated into one new complex event, repetitive events are counted and aggregated into a new single event with a count of how often the original event occurred, etc. More generally, events are said to be *mined* (hence the name of Event Mining or EM) because the user wants to discover new interesting patterns occurring in the system.

This process of producing fewer "better" events out of many "lesser" events can be iterated. This idea is the basis for Event Processing Networks, where agents mine incoming events and pass the resulting events to other agents down the Event Processing Network. The *presentation* of the mined events to the user is virtually unlimited.

CEP is particularly well suited for event based systems, but is applicable to other systems as well, e.g. updates in a database can be interpreted as events.

The following list gives a few application examples:

- *Security:* High target sites are attacked hundreds of times per day. Most attacks are unsuccessful, but a few attackers create havoc, from denial-of-service attacks to taking control of systems. CEP can detect attacks in real-time, reduce the number of false alarms and prioritize attack messages. Drill-down gives quick access to the original data to increase the effectiveness of countermeasures [1]. Section 3.1. gives an example of this application.

- *Message Brokering:* The CEP feed-back capabilities make it a powerful, content/context based message broker. Users of publish-subscribe middleware can subscribe to messages based on their context and content. A trader could e.g. subscribe to IBM quotes only when at least 10,000 shares were traded at that price and the last four quotes were all up-ticks.

- *Business applications:* EM based real-time decision support systems constantly gather information throughout the enterprise and immediately respond to changes in information. These systems are business event driven, where a business event represents any significant change in business data or conditions.

- *Enterprise network and systems management:* Pattern of events that may lead to a failure (e.g. an important disk filling up) or that could signal break-in attempts (i.e. connect requests to multiple targets from a single source over a short time) are detected as they occur. EM provides immediate notification of such conditions to the Subsystems of large, mission critical networks. Automatic prioritizing of alerts and quick root cause analysis leads to reduced response time, higher up time and allows network Subsystems to quickly respond to critical situations.

- *Manufacturing:* A fabrication line is a complex federated system from multiple vendors connected by communication middleware and running around the clock. Small irregularities in fablines can be very expensive. EM allows close monitoring and immediate propagation of all relevant data to the management.

Current event processing systems are rather simple. They support action-reaction rules based on single events, with boolean conditions over the parameters of events. Our Complex Event Processing (CEP) is based on the following innovative capabilities:

- *Additional semantics:* CEP supports not only *time* but also *causality*: one event *causes* another. Causality is used to represent dependent vs. independent execution and provides additional information that can be used in processing events. It also greatly reduces the search space of pattern matching since related events are grouped together, even if they occur widely separated in time or in the log file.

- *Powerful mathematical foundation:* Events are stored as complex objects together with their relationships as partial orders. In today's networked real-time environments not all events are time-ordered in respect to each other. In existing event-processing systems, the original partial order of events is implicitly reduced while tracing, information is lost, and non-determinism is introduced [2]. CEP maintains the partial order for both time and causality.

- *Expressive event query language:* A formal language [3] to specify event patterns is a crucial part of CEP. Working with a high level query language simplifies development and configuration and allows for query optimization within CEP. The CEP event pattern language allows the user to describe patterns of events. An event pattern matcher searches for all occurrences of a pattern of events in a partially ordered set. A typical example would search for all A events that cause both a B and a C event, with B and C either ordered or independent of each other. This pattern could be specified as:

  ```
  A->(B~C)
  ```

  As a comparison, in OQL, clumsily enhanced with a * operator denoting one or several repetitions of the path expression, this query would look like:

  ```
  select tuple(e.ID, f.ID, g.ID)
  from event e, e.(successor)* f, e.(successor)* g
  where e.type='A' and f.type='B' and g.type='C' and
    (NOT f.(successor)*=g) OR (NOT f.(successor)*=g) OR f=g)
  ```

  Using this pattern language, we can describe an agent with the following rule:

  ```
  A(?id)->(B~C) => generate C(?id)
  ```

Whenever the above pattern is matched among observed events, a new event C is generated. The rule also specifies that the parameter (or attribute) of the newly generated C has the same value as the parameter of the A event that triggered the rule.

- *Concept hierarchies:* Building hierarchies of more and more abstract events allows for high-level situation classification, description and reaction driven from high-level business strategies (similar to *views* in databases). Concept hierarchies are useful for understanding a system as they visualize levels of abstractions often used by humans when thinking about complex systems. Concept hierarchies are built using the event pattern language. The different levels of the hierarchy are fully interconnected, allowing for drill down diagnostic analysis. The examples give examples of concept hierarchies.

  Hierarchical organization of event pattern processing is a critical component in recognition and classification of complex activity. It ensures that users receive information at the appropriate semantic level. Low level events streams may be our main source of data, (e.g., raw network intrusion detector output). A technique is needed to correlate subsets of these events that signify possible parts of attacks. Hierarchical organization of CEP lets us configure CEP to detect patterns of low level events and generate high level events that correlate and summarize the data in the lower level pattern instances. Higher level events can be fed to even higher level event processing agents, transitively, until a complex activity is humanly recognizable. An abstraction hierarchy gives us two very important things:

  - it defines a set of concepts in terms of which views of the target system can be constructed,

  - it structures the concepts into a hierarchy of levels, and defines mathematical relationships between the concepts in different levels.

  We can use aggregator agents containing event pattern mapping rules to specify relationships between the concepts at different levels in the definition of an abstraction hierarchy. This gives us an EPN which is an executable form of hierarchy definition. This EPN will generate a hierarchy of events corresponding to the hierarchy of concepts. We can then add additional EPAs that construct views consisting of abstract events corresponding to concept levels in a specified hierarchy. When sets of events denoting instances of concepts at one level occur, they trigger aggregators in an EPN corresponding to the hierarchy mappings which then generate abstract events denoting instances of higher level concepts.

  Concept hierarchies also work the other way round. The user can specify actions on a high-level of abstractions and event processing agents translate these instructions into corresponding lower level events.

  Abstraction Hierarchies are useful for understanding a system as they visualize levels of abstractions often used by humans when thinking about complex systems. Hierarchies are implemented by sequence of aggregation steps (called *maps*) that create more and more abstract events, all deduced from a basic source of events. The same mechanism can be used to provide different views of the same system to different users. The relation of events is maintained between different views.

- *Support for systems architecture:* Complex middleware-based systems consist of components and communication pathways between these components. These components and pathways may change while the systems is executing (e.g. in mobile computing). Often the context of events is relevant: The reason why a message got lost may be that a required pathway was no longer existent when the message was sent.

- *Dynamic causal models* are one form of event processing. Events in event streams are automatically processed to include genetic information indicating which other events had to happen in order for them to happen. The genetic data in an event allows the events in its causal ancestry to be immediately uncovered by other EPAs, making their operation more efficient and accurate. Causal information must be recognized across multiple event streams. Extracting and developing accurate causal models for C2 environments is another research issue.

- *Flexibility:* CEP queries are flexible and configurable at runtime so the user can rapidly adjust them if needed. This includes starting a new query against an ongoing event stream, that either considers only new events, only old events, or both.

- *Efficiency:* Many applications produce relevant events at rates of millions/minute. CEP supports high event rates through query optimization, efficient pattern matching algorithms, as well as distribution of event processing.

CEP supporting these two features is part of Stanford University's **RAPIDE** project. We developed an extensive set of tools that supports logging, mining, storing, and viewing of events in real-time. **RAPIDE** events are related by time and cause. Each relations builds a partial order on all the events. A formal pattern language [3] supports the construction of *filters* and *maps*, constructs that aggregate simple events to complex events on a higher level of abstraction [4]. The same process can be used to query complex events, thus building a more and more abstract view of the system.

B. Event Processing Networks

In order to accurately reflect the complex nature of real computing environments, effective event processing must be component based. A variety of different components need to be able to process a broad spectrum of events from different technologies.

We have built *Event Processing Networks* (EPNs) consisting of any number of *Event Processing Agents* (EPAs), namely event *producers*, event *processors,* and event *consumers*. Figure 1 shows an overview over the three categories, with thin arrows indicating the (logical) flow of events from producers through processors to consumers.

The EPN is supported by an event service. The event service provides repositories for events and event schemas, templates for processors, and a consumer registry. Event Schemas define the format of events forwarded by the event service. The event repository stores events. Processor templates are predefined templates that can be activated to filter, aggregate, or otherwise process events. The consumer registry keeps track of which consumer is interested in which events.

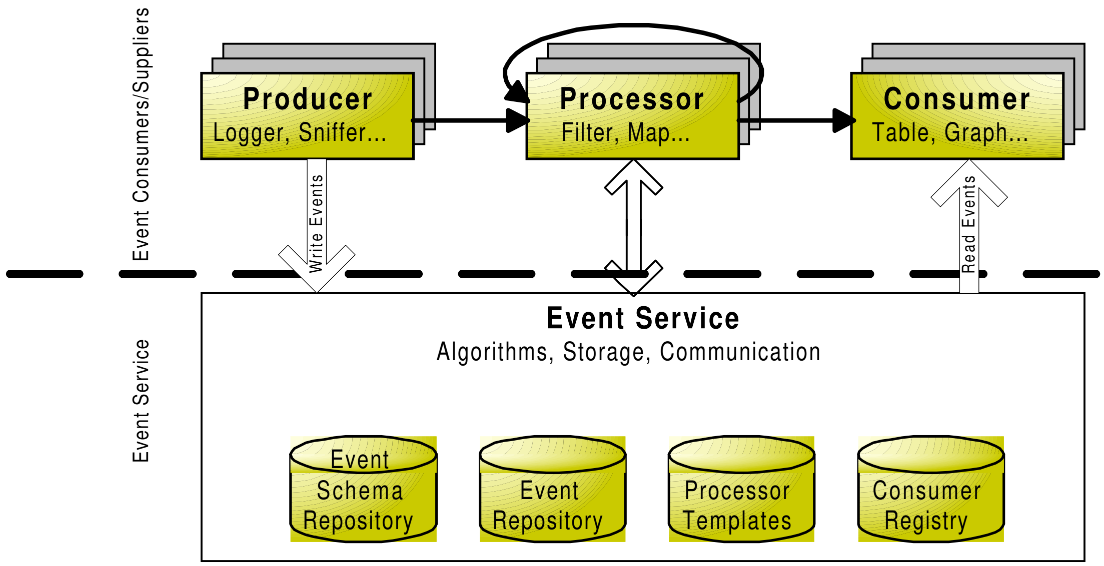


Figure 1 : Event Mining

Event sources in our applications are typically middleware *sniffers*. The system middleware can be pure TCP/IP, an event communication service based on a proprietary protocol like TIBCO Inc.'s TIB or Vitria, Inc.'s Communicator, or a military standard like the MIL STD 1553. We also automatically instrument the source code of system written in Java to intercept events within the Java engine [5].

Typical examples for event processors are *filters* and *maps*. Filters pass on only a subset of their input, maps aggregate multiple events in the input to output events, thus generating events on a higher level of abstraction. Any third party event processor can be inserted into an EPN allowing for the integration with other approaches.

Typical event viewers are a graphical viewer for partially ordered sets of events (POV), a tabular viewer of event frequency or a simple gauge metering the latest value of an important parameter.

Data needs to be stored persistently because agents may want to access past events, even long after they have happened. Also, because the number of objects currently under consideration may easily exceed the size of the available main memory, thus EM requires some way of storing objects temporarily to disk. **RAPIDE** EM includes a shared data store that keeps track of all the objects. New objects are written into the data store from where agents and viewers read them. A communication service notifies other EPAs when new objects are added.

On a conceptual level, splitting up the event processing load into a sequence of aggregation steps creates a *concept stack,* the concepts being more and more abstracted activities, all deduced from one basic source of events. Concept stacks are useful for understanding a system as they visualize levels of abstractions often used by humans when thinking about complex systems. Also, reusing partial results of one agent for multiple other agents reduces the amount of overall computation that needs to be done.

Events flow through the EPN in real-time and are displayed in viewers as soon as they are created, limited only by the speed of the underlying infrastructure. Processed events are displayed in viewers shortly after the underlying events have been created by the event source. EPNs are *dynamic* in that EPAs can be added, reconfigured and removed at runtime. Newly added agents can ignore all previous events and just start with the current event at the time they are added, or they can try to catch up all events from the beginning. As EPNs are distributed, EPAs can reside on machines distributed across a network.

C. Examples

1. Monitoring a Semiconductor Fabrication Line

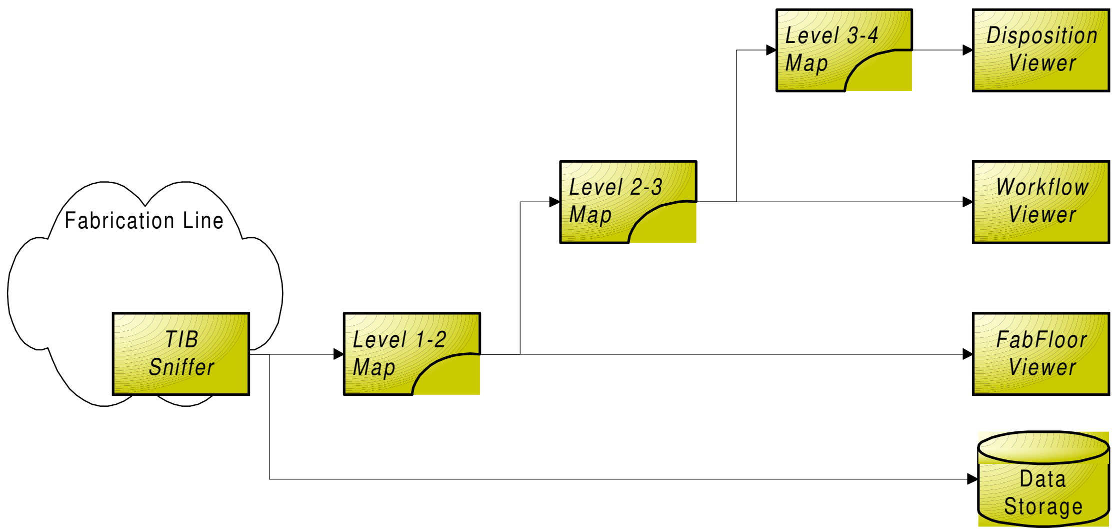


Figure 2 : The fabline EPN

A typical chip fabrication line may have some 600 components ranging from manufacturing machines to in-house developed applications. In our example, these components are interconnected using TIBCO Inc.'s Rendezvous. A detailed version of this example can be found in [4].

The EPN in our example consists of a TIB sniffer, three event processors, and three viewers (as shown in Figure 2. The RapNet reference manual gives a complete description of the graphical representation of Event Processing Networks).

The sniffer records the raw traffic on the TIB and produces events on a first level - the *Middleware Communication* level. In our example, only these events are stored persistently to disk.

A first map ("Level 1-2 Map") reads these events and creates new events on a higher level of abstraction - the *Point-to-Point Communication* level. Point-to-point events have inherently more meaning, since they represent flow from a sender to a receiver. The first map has to aggregate sets of broadcast messages that are all causally dependent and build a valid point-to-point message into one new event. The start and end point of this communication are deduced from the contents and causal relation of the level 1 events. This higher level is then read by a viewer that displays graphical animations of point-to-point messages within the fabline system ("Fabfloor Viewer").

A second map ("Level 2-3 Map") aggregates point-to-point events to *Workflow Steps*, events denoting the movement and processing of lots. These events are also displayed as animation ("Workflow Viewer").

Analogously, we build a fourth layer, *Disposition*, which describes the positions of the lots on the fab floor.

These four layers build our concept stack.

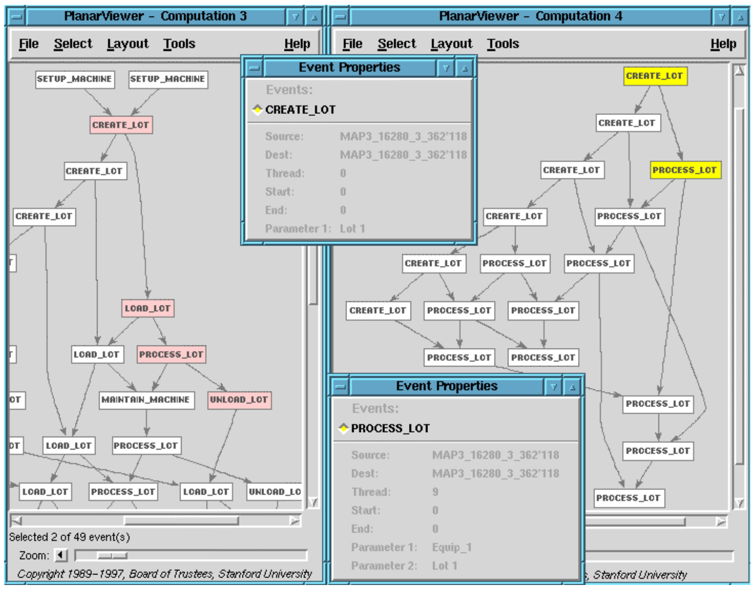


Figure 3:Monitoring a fabline

Figure 3 shows the result of the Level 3-4 Map. The nodes in the graphs represent events, the edges represent the causal relation. The selection in the left window shows 4 events on level 3 that are mapped to the 2 selected events on level 4 in the right window. The two small windows give in-depth information on single events .

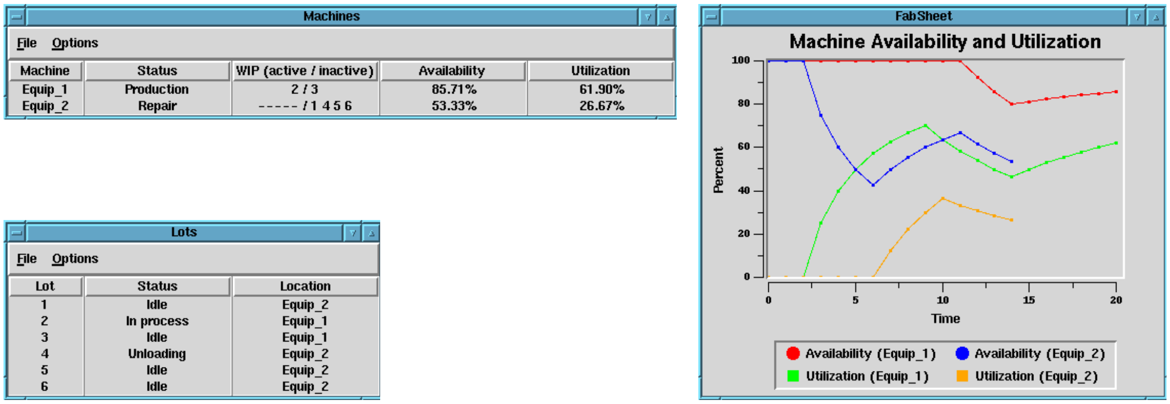


Figure 4 : Monitoring a fabline

This EPN can now be used in different scenarios: In a *decision support scenario*, the site Subsystem has constant on-line information on the production status, how many lots are in process, when they are expected to be done, when scheduled maintenance will slow down the fabline, etc. (Figure 4).

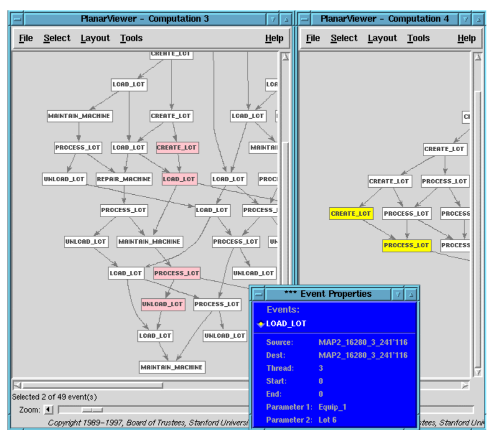


Figure 5 : Tracking the fate of lot 6

In a *monitoring scenario*, something might go wrong with lot 6, maybe it got stuck in equipment 2. An engineer now tracks the fate of lot 6 at level 3. He selects events leading to a problem at level 4 and looks at the corresponding events at level 3 (Figure 5). We describe this process down through all the levels to the root cause in [4]. An operator failed to reply to a message involving preventive maintenance on equipment 2. This scenario shows how our tools can be used to efficiently drill-down through huge amounts of data to resolve problems.

2. Cyber Warfare

Our second application example brings us to the world of cyber warfare. As organizations build ever-larger computer networks, security has become an increasingly important issue. The growth of the Internet has resulted in an increase in the size of individual networks as well as an increase in the volume of traffic flowing through the network. As system log files have grown to contain gigabits of data, it has become almost impossible to manually trace through log files to detect attempts at intrusion activity or other security violations. This has given rise to a need for software tools that automate network monitoring for security intrusions [6-12]. Some of the results that can be achieved by using context-based event correlation in cyber warfare are:

- *High-level view:* CEP provides high-level situation classification, event description, and deployment of new defensive actions driven from high level strategies. This is achieved through definition of concept hierarchies.

- *Flexibility:* CEP provides rapid reaction and interactive on-the-fly reconfiguration of strategies. This is made possible by communicating structures of event processing agents, called Event Processing Networks.

- *Analysis:* CEP provides drill down diagnostic analysis.

- *Consolidation:* CEP can correlate of a diversity of independent inputs from network-level intrusion detectors to application-level sensors.

- *False Alarm Reduction:* Alarms that are not subsequently confirmed by another alarm may be false or irrelevant. As an example, denial of service attacks may be used to create a diversion by flooding administrators with low priority alarms to mask penetration attacks. Compared to the ongoing penetration, the denial of service may be irrelevant.

- *Increased Detection Rate:* CEP detects coordinated but separate attacks no matter how widely separated in time by capturing of causal relations between events. Unobserved intrusions may be detected by deduction from observed attacks. On-line event correlation of early stage probing alerts may detect ongoing attack patterns in early stages.

CEP provides on on-line overview of the state of the cyber battlefield. Patterns of events that may lead to a failure (e.g. a DNS server having slower and slower response time) are detected as they occur. CEP provides immediate notification of such conditions. Because the context of events is maintained, user driven drill-down from a notification message back to the root cause is possible. Automatic prioritizing of alerts and quick root cause analysis leads to reduced response time, higher up time and allows system managers to quickly respond to critical situations. CEP also allows automatic response based on pre-defined rules.

Figure 6 shows how CEP is integrated into the emerging Common Intrusion Detection Framework (CIDF)[13]. The framework consists of middleware that provides secure transport of General Intrusion Detection Objects (GIDOs). These GIDOs are formulated in the Common Intrusion Specification Language (CISL). The middleware and the language provide a mechanism for components to communicate (shown as dashed arrows). The main components in the framework are detectors and correlators. Detectors are a part of the information infrastructure, such as a (sub)network, a host or the middleware. Correlators investigate the output for two major purposes: Reduction of false alarm and detection of large-scale attacks that involve simultaneous attacks to multiple parts.

CEP/CIDF integration starts with a sniffer agent that listens to CIDF traffic and intercepts relevant GIDOs. Each GIDOs is then translated into an event in the CEP internal format (level 0 events). As shown in Figure 6, these level 0 events are stored to disk for later analysis. They are also displayed in the CIDF Traffic Viewer. The level 0 events are also the basis for a first map. Because CEP has been optimized for strongly structured events, and CIDF events have a very flexible structure, we extract relevant information from the level 0 events and create another layer of events with a rigid structure (level 1). These events are then used as input for several other maps that may be built on top of each other as shown in Figure 6.

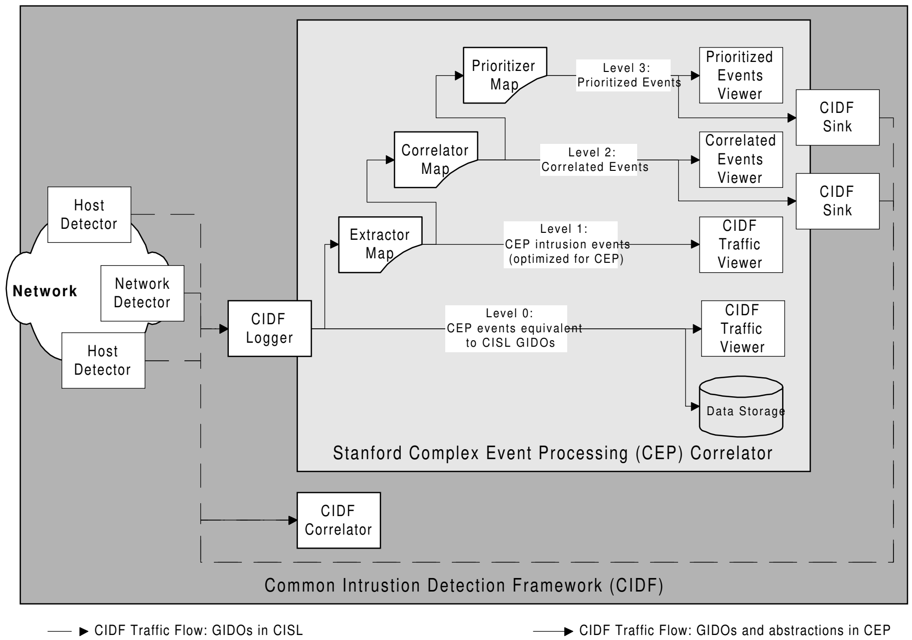


Figure 6 : The CIDF Event Processing Network

D. Related Work

1. General Event Services

The Open Group published a specification for Event Management Systems (XEMS) [14]. XEMS provides a set of APIs for event producers and consumers to connect to the The Open Group published a specification for Event Management Systems (XEMS) [14]. XEMS provides a set of APIs for event producers and consumers to connect to the service. Additionally, there are API calls for configuration and management of the service itself. XEMS supports a mechanism to define types of events. Event filters in XEMS are part of the event service and not external components. They are also rather simple, consisting of logical expressions on the attributes in an event (in disjunctive normal form) only. Simple and internal filters can be implemented more efficiently. In our technology, filters are outside the event service and can be arbitrarily complex. The service supports efficient filtering (and aggregation) through the API.

We have our own event processing infrastructure and event format. We supply translators that perform the translation from and to other event formats. We expect the cost of the translation to be negligible compared to the cost of the event processing. Our format is optimized for efficient processing of causal information. Our EPN management tool is optimized to control our event processing agents. However, by using proxies, third party agents can be controlled as well.

2. Event Correlation

Many groups are focusing on event correlation for network management. Event correlation circuits as described in [15] are limited to simple filtering and aggregation. SMART correlation books is restricted to simple event correlation [16]. The correlation is not done by an EPN, but rather by single process based on a pre-compiled table of correlations. Both [15] and [16] are focused on network management events, while CEP is applicable to any domain. Commercial products available include Computer Associates International Inc.'s Unicenter TNG, Tivoli Systems Inc.'s TME 10, and Hewlett Packard Co.'s OpenView. These commercial products are rather limited in two ways: First, their respective event processing capabilities are limited to simple filters and aggregations, similar to the Unix commands **`grep`**, **`sort`** and **`wc`**. Second, they are based on relational database technology, treating all events as tuples in a table. While this approach is sufficient for many applications, there are cases where more powerful aggregation is necessary.

Systems formally defined using communicating finite-state machines (CFSMs) introduce non-determinism (e. g. SDL [17], Estelle [18],). The events denoting state changes of the CFSM build a partial order with relations between events of the same machine or the same communication activity. However, in the CFSM this partial order is lost.

The same holds for most approaches to detect software failures (e. g. [19]). The partial order is implicitly reduced to a total order. This introduces additional complexity as multiple reductions are possible, depending on the delay in the communication pathways. CEP avoids this complication by analyzing partially ordered sets.

CEP based on-line validation can be viewed as a variation of the observer-worker approach described in [20] or the supervisor approach as described e. g. in [21]. An observer/supervisor can be defined using the CEP event pattern language. Using the on-line validation capabilities of our infrastructure the observer/supervisor constantly checks the specification against the running system.

3. Query Engine

The work we did in the event pattern matcher is related to work done so far in the database field. Optimizing our pattern matching is related to the work Qian and Wiederhold did for relational algebra [22, 23]. However, we hope to do similar optimization work on the level of our event pattern language, without having to go down to relational algebra.

Work done by Wolfson et al. [24] focused on incremental evaluation of rules in a deductive database where rules are added as the interpreter evaluates rules. They operarate on a more general level where rules can change after initial insertion with consequences for later rules. In our pattern language, once a pattern is found, it cannot be undone. Regarding real-time constraints, [24] just expects "that incremental rule processing … will realize real-time expert systems with large database".

[25] presents a logic oriented point of view of the problem. It involves backtracking and describes an algorithm to obtain an approximation of the ideal effective point in a constraint based incremental system.

It applies to another case of dynamism, where the query itself evolves dynamically, which can happen with the pattern language used to trigger events in the CEP model.

Our problem is a subset of those studied in [26], where our FOIES is almost insertion-only with (currently) non recursive auxiliary mappings. Unfortunately, the arity of our queries is potentially high, which makes an efficient general algorithm unlikely.

Actually, the problem we are faced with most closely resembles materialized view maintenance, for which many algorithms (see for instance [27, 28]) have been developed for relational systems. A large part of these studies focus on how to propagate changes over several materialized views, a problem that does not apply to EM. Only a few published papers address view materialization issues in OODBMS and some of the proposals have not been implemented. Croque [29] extends [22] to monoids and is much more general (and probably less efficient) than what we need. MultiView [30] is more concerned about schema changes. Using this approach, we would have to define a virtual class for each query that is derived from existing classes by a recursive path query. However, MultiView does not support recursive queries.

II. Event Processing Networks

In the following sections, we introduce the concept of event processing agents and of event processing networks, we proceed to a relational representation of them, which we use to optimize such networks. We will follow a bottom up approach: we begin by defining a relational representation of the RAPIDE pattern language, then use that representation to define typical Event Processing Agents, and explain how to combine those into full Event Processing Networks. This leads to a discussion of possible optimizations, efficient incremental algorithms to compute the outputs of an event processing network and opportunities to distribute and parallelize the network on a pool of machines.

A. Event Processing Agents

To monitor a distributed system and analyze its behavior, we have designed a framework that enables Complex Event Processing in a modular and distributed fashion. Each element of this modular framework acts as an agent, hence the name of Event Processing Agent, or EPA.

Event Processing Agents can be split in three categories:

- Suppliers: these agents either have no input (in which case we refer to them as generators), or have an input that is external to our system: they may listen to the messages being exchanged by the nodes of the monitored system and convert them into events for on line analysis (in which case we call them sniffers) or they can read some log file produced by the monitored system, usually for post-mortem analysis (in which case we call them loggers). A typical supplier will work under the assumption of orderly observation (no earlier event arrives after a later event) and will try to add meaningful causal information to the generated events.

- Processors: these agents do the bulk of the work. Filters filter out events that do not match a given pattern. Mappers aggregate multiple events into one or more new events. Constraint checkers ensure that a condition is always or never met.

- Consumers/Responders: these agents have no output within our system. They are usually used to notify the user of significant activities in the network. This is done either through a graphical or textual representation (viewers) or through taking specific actions (actors), such as sending email or paging the person in charge of the system.

Once we have those EPAs, we can compose them into an Event Processing Network, which can be thought of as a directed acyclic graph, where the vertices refer to event channels and the nodes are EPAs.

From these types of agents, we realize that a lot of commonality is shared. In general, we can represent an EPA as a black box, with a given (possibly variable) number of inputs and outputs, each with a given type (Rapide is a typed language).

This allows us to design our system to be modular and extensible: by defining an interface which all EPAs must implement, we can fully exploit the black box paradigm yet let the system grow by adding new EPAs as needed. This is further described in chapter III, which specifies the architecture of the RAPIDE CEP system, but it is important to note it here, because this is the key to Event Processing

Networks: because nodes can be represented as black boxes, agents can interact within an Event Processing Network and pass events to other agents in order to build the concept stack which was described in section I.C.

B. Relational Representation of the Pattern Language

The purpose of this section is not to describe the Rapide pattern language, which has been thoroughly defined in [3], but to define a representation that is useful in representing and optimizing Event Processing Networks.

1. Basics

We will be working in an algebra on the set ***C*** of all the event containers in a given universe (an event container is either a computation or a subcomputation. Although there is a difference between the two notions in the CEP context, the following theory is independent of that distinction and therefore applies to event containers in general).

We are either working in terms of binary relations over $\boldsymbol{C}^p x\ \boldsymbol{C}^q$ or in terms of predicates. The two notions are linked: For a relation definition, the following statement holds: $x\ R\ y\ =\ P_R(x,y)$, where P_R is a predicate that defines R. In other terms, $R=\{(x,y)|P_R(x,y)\}$. Considering this way of viewing a relation (i.e. by its graph), one can apply the usual set operations to relations.

Composition is defined as: $xR_1;R_2\,y \Leftrightarrow \exists z.xR_1 z \wedge zR_2\,y$. This leads to the notion of inverse of a relation R, written R^{-1}, where $R;R^{-1} = Id(\boldsymbol{C}^p)$ and $R^{-1};R = Id(\boldsymbol{C}^q)$

We will also need to use the concept of power set: $x\Lambda(R)\vec{y} \Leftrightarrow \vec{y} = \{\,u\,|\,xRu\,\}$, (where we write y as a vector as a reminder that it is actually a subset of $\boldsymbol{C}^q$).

Finally, because Event Processing Networks typically have fan-out portions, we also define the concept of pairs or tuples of relations:

$$(x_1, x_2)(R_1, R_2)(y_1, y_2) \Leftrightarrow x_1 R_1 y_1 \wedge x_2 R_2 y_2$$

A useful operator which is directly related to the concept of pairs is the *dup* operator, which transforms x into (x,x).

Finally, we can define conjugation by graphs: the graph of the conjugate of a relation R, written $\overline{R}$, is the complement of the graph of R.

One can prove that ***C***, with conjugation and composition has an algebraic structure.

2. Derived Definitions

From the basic definitions, we can define the usual set operators:

Intersection: $R_1 \cap R_2 = dup;(R_1, R_2); dup^{-1}$

Union: $R_1 \cup R_2 = \overline{\overline{R_1} \cap \overline{R_2}}$

Difference: $R - S = S^{-1}; R; S$

3. Patterns

Given this framework, we can define the Rapide patterns. A pattern p is a binary relation $p : \boldsymbol{C} x \boldsymbol{C}$ such that: $c\, p\, m$ if $m \subseteq c$ matches p.

A pattern macro in RAPIDE is a parametrizable pattern. RAPIDE has six predefined pattern macros[3], which take patterns as parameters and are the glue that makes up the pattern language. These predefined pattern macros express causal relationships; for instance, $a \rightarrow b$ means "b causally follows a", $a \| b$ means "b and a are causally independent", and $a \sim b$ means "b causally follows a or b and a are causally independent".

Because of their key importance in the language, and because we are following a bottom-up approach, it makes sense to start by expressing those predefined pattern macros, in terms of their predicates. This is summarized in the following table:

$P_{\rightarrow}(C_1, C_2)$	iff	$\forall e_1 \in C_1 . \forall e_2 \in C_2 . e_1 \prec e_2$
$P_{\sim}(C_1, C_2)$	iff	$C_1 \cap C_2 = 0$
$P_{\|}(C_1, C_2)$	iff	$\forall e_1 \in C_1 . \forall e_2 \in C_2 . \neg e_1 \prec e_2 \wedge \neg e_2 \prec e_1 \wedge \neg e_2 \cong e_1$
$P_{OR}(C_1, C_2)$	iff	$C_1 = 0 \vee C_2 = 0$
$P_{AND}(C_1, C_2)$	iff	$C_1 = C_2$
$P_{\cup}(C_1, C_2)$	iff	$true$

Here, we assume, according to the Rapide models, that a causal relation amongst events is known, namely $\prec$. This is a partial order, with the notion of equivalent events, noted $\cong$.

From the pattern macros, we can work towards more complex patterns, using the following relation: Given two patterns p_1 and p_2, and a pattern macro +, which is one of $\rightarrow$, ~, $\|$, OR, AND or U, and the corresponding predicates, we have:

$$\boxed{p_1 + p_2 = dup;(p_1, p_2); Id(C, P_+); \cup}$$

This is very powerful, as it lets us now write every non iterative Rapide pattern (parameters and guarded patterns can be described as well).

C. Relational Representation of Event Processing Agents

Most of the agents that are used in Event Processing Networks can be defined in terms of the algebra presented in the previous section, possibly using the pattern language. As the number of such Event Processing Agents is quite large (and growing), we will only present here some typical and interesting examples, namely simple filters and maps.

Although it is probably not possible to express all possible EPAs in relational form (maybe not even every possible map), this representation is still useful to optimize an EPN when possible.

1. Filter

The task of a filter is, given a pattern *p*, to only let pass through those events that are part of a match for *p*. Filters can be very complex, given the power of the RAPIDE filter language. They can maintain state between matches. Such state makes our representation much harder to write (although it does not make it impossible), and we therefore present the results for stateless filters.

Given this intuitive definition, two definitions are possible for filters:

- The first one produces a set of computations, and can be written as: $\boxed{filter_p = \Lambda(p)}$. Here a given event can participate in several matches.
- In the second definition, only the earliest maximal match is produced. Therefore a filter only produces one computation.

The second definition is the one that is currently implemented, primarily for efficiency reasons, and because, in most cases, only the earliest maximal match matters. However the first one is much easier to represent, because our framework does not include the notion of time, i.e. of which events came first, and which match is earliest.

To make up for the inaccuracy between theory and implementation and to make sure that following EPAs in the EPN get a computation rather than a set of computations as an input, we also define a special kind of EPA, called *choose*, which selects a computation from a set of computations.

2. Map

The purpose of this section is not to fully describe maps. A complete reference can be found in [31]. The main purpose of this section is to show how to specify simple maps in our relational scheme.

Maps are potentially very complex EPAs. The basic idea behind map is that, given a domain and a range, rules will be triggered when events of the domain match a given pattern, and those rules may produce events in the range of the map. A map is usually a list of rules. A rule is usually made of a trigger (a pattern which describes events which will start the map) and a statement (a pattern which defines which event(s) are produced when the trigger matches).

Because maps can be arbitrarily complex (the map language is a large superset of the pattern language), not much can be done to optimize or even predict map behavior in general.

Therefore we will only concentrate on simple maps, which are basically maps that do not maintain any state. These maps can be represented as a list of triggers, which are Rapide patterns, and a list of bodies, which are also Rapide patterns. Whenever a set of events matches a pattern in the list of triggers, then corresponding body is executed, leading to another set of events. In the end, maps operate on *C*x*C*.

Before introducing the representation of maps, we define the matching of a rule $p_l \Rightarrow p_r$ as $R_{match}(p_l, p_r) = \{(c_l, c_r) | c_l \in R(p_l) \wedge c_r \in R(p_r)\}$: this simply says that the graph of a matcher is such that for every match of the rule by events of the input computation, a set of events matching the statement of the rule is generated.

A single ruled map is a little more involved than that, because of the notion of consumption of events: a rule will only trigger on the first earliest set of events (in the causal sense). To represent that in our relational representation, we need to introduce the concept of prefixes. This notion is commonly defined in partial orders and therefore we will only say that a prefix is a subset of a computation which respects the causal ordering of events, i.e. if a prefix contains an event then it contains all the events which causally precede it.

We now have the tools to define a single-ruled map:

$$\boxed{P_{map\; p_l \Rightarrow p_b} = M(p_1, p_2) = \left\{(C_1, C_2) \in CxC \,\middle|\, C_1 \supseteq \breve{\bigcup} c_{l_i} \wedge C_2 = \bigcup c_{r_j} \wedge Disj(c_{l_i})\right\}}$$

Here, the c_l's and c_r's are taken from the corresponding matches, defined in R_{natch}, and $\breve{\bigcup}$ is the causal preserving disjoint union operator, i.e. it enforces elements to be disjoint while building them monotonically (following the causal order), which can be expressed using the notion of consistent cuts:

$\forall(c', c'').c' \prec c'' \wedge c' \in \breve{\bigcup} \Rightarrow c'' \notin \breve{\bigcup}$, where $\prec$ stands for "is a consistent cut of". This notion of monotonic union can also be expressed as:

$$\forall(c_1', c_2').c_1' \prec c_1 \wedge (c_1', c_2') \in M(p_1, p_2) \wedge (c_1, c_2) \in M(p_1, p_2) \Rightarrow c_2' \prec c_2$$

Multiple ruled maps can be defined in terms of single rule maps, depending on the semantics of triggering: when multiple rules can trigger, only one of them should trigger. The decision of which rule will actually trigger in such cases is not made as of now because it is unclear whether random, round-robin or user specified priorities would be most useful.

D. Sub-EPN

The concept of sub-EPNs is useful in several contexts:

- Sub-EPNs are the equivalent of a basic block in the process of EPN optimization.
- Sub-EPNs are used extensively in tools like RapNet, where they are the principal means of scalable/hierarchical design of EPNs and where they mimic the typical top-down design approach: an Architect designs building blocks in the form of sub-EPNs, which are then added to a library of Agents, which is used by engineers to build actual EPNs. This is further discussed in section II.H.2.a).

- Synchronous Groups (which we describe below) are a special case of sub-EPN.

Just like an EPN is a directed acyclic graph of agents, a sub-EPN is a subset of an EPN, which respects the following condition: one should be able to replace that subset in the EPN by a single node without creating a cycle in the resulting graph. The reason for that restriction is that you want to be able to treat a sub-EPN just like a regular EPA, because: 1/ it is then transparent to the user of RapNet whether a library element is an EPA or a sub-EPN, and 2/ optimizations can then be done in terms of sub-EPNs, with no side-effects outside of the sub-EPN (this is explained in section II.F).

Examples and counter-examples of sub-EPNs are shown on Figure 7, where the dotted blue area is the boundary separating the elements of the sub-graph and the elements outside the sub-graph.

The first example in the figure shows a typical sub-EPN, which contains a filter and two maps. The resulting sub-EPN can be represented as a black box itself, with one input (the input of the filter) and four outputs (the outputs of the maps).

The second example in the figure shows that if the sub-graph is not connected, then it may not be a sub-EPN: ($f_1; agent; m_1$) shows that ($f_1 \circ m_1$) is not a sub-EPN because the induced graph has a cycle.

The figure also proves that being connected is not a sufficient condition: in the third example, the selected sub-graph is connected, but it still is not a sub-EPN, because the EPN shown induces a cycle when the sub-graph is replaced by a single node.

To give a more formal definition of sub-EPNs, we need to introduce the notion of dominance. This notion is taken from graph theory and compiler optimization. In a graph, a node *a* dominates a node *b* if all paths starting from any entry node that lead to *b* go through *a*. In our case, an entry node is a node with no predecessor. Post dominance is dominance in the reverse graph (i.e. when reversing all the edges of the DAG).

In fact, it is possible to prove that the intuitive definition of sub-EPNs given above is equivalent to the following: a sub-EPN is a sub-graph of an EPN under which dominance and post-dominance are closed; i.e. if a node is both dominated by some node of the sub-graph and is post-dominated by some other node in the sub-graph, then it must be in that sub-graph in order for the sub-graph to be a sub-EPN.

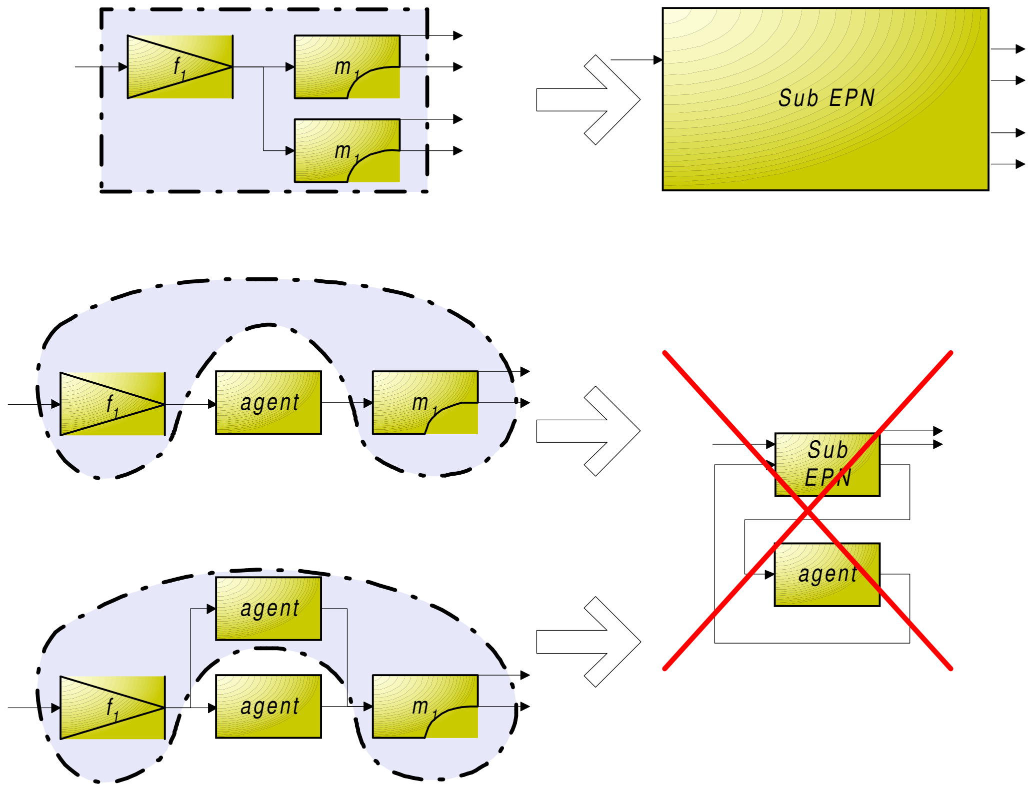


Figure 7 : Sub Event Processing Networks (Sub-EPNs)

E. Representation of Event Processing Networks

1. Definition

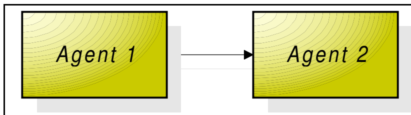
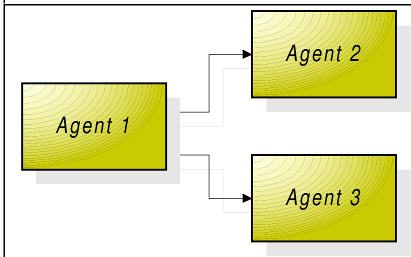
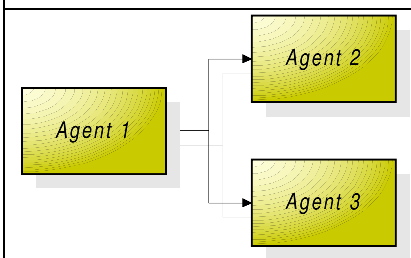
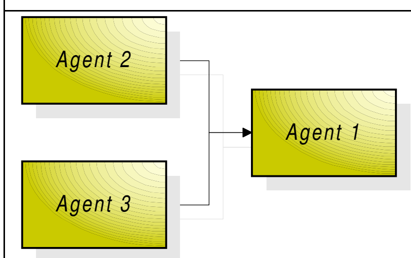
We can represent an Event Processing Network using our relational representation of EPAs, combined with operators to describe fan-in and fan-out connections:

- in our model, an EPN is a directed acyclic graph of agents. The acyclicity of the graph is not a strong requirement and may be relaxed in the future. We have already thought of examples where such a cycle may add expressiveness to the system, provided that the cycle does not recurse indefinitely.

- Each node is an agent, which can be represented using the previous relational expressions.

- Each vertex corresponds to an event container, which is the mean for carrying events between two agents.

There are many possible situations: the representation depends on the multiplicity of the connections on each input / output of agents, and on fan-in/fan-out positions in the network. The possible situations are summarized in Figure 8. This figure introduces a graphical representation of Event Processing Networks. Composing the elementary configurations presented in Figure 8 allows one to build all possible EPNs.

<table>
<tr><td>
</td><td>Error! Objects cannot be created from editing field codes.</td></tr>
<tr><td>
</td><td>$Agent_1;(Agent_2, Agent_3)$</td></tr>
<tr><td>
</td><td>$Agent_1;dup;(Agent_2, Agent_3)$</td></tr>
<tr><td>
</td><td>$(Agent_2, Agent_3);\cup;Agent_1$

This case is further discussed below.</td></tr>
</table>

Figure 8 : Connections in relational expressions

The semantics for a fan-out situation are rather clear, and we only define one type (the second case in Figure 8). The third case in Figure 8 is a special case of the previous one, where the fan-out is represented as a *dup* operator, i.e. everything happens as if the events where copied on each outgoing computation (in reality, for efficiency reasons, our implementation shares the events amongst the different observers which is fine as long as EPAs are not allowed to modify their inputs). This is exactly as if there was a *dup* EPA between *Agent 1* and *Agent 2, Agent 3*.

For fan-in nodes, as represented by the last case of Figure 8, many semantics can be defined. The most common ones are:

- Barrier semantics: an event can go through only if all the sources have produced an event. The name comes from the Barrier construct, which is typical in message passing systems.

- Pass-through semantics: anything that comes from either source will propagate through the node: this is what the union does, shown here.

We then have much more involved semantics, which use the concept of activity-equivalent event: each event has its own, unique event identifier (EID). Each event is *identical* only to itself, however, events denoting the same activity are said to be *activity equivalent* or in short, equivalent. Equivalence is context dependent, i.e. it depends on the view of the world and the implementation of the loggers. In certain cases, it might even never be resolved whether two events are equivalent. But in cases where the activities can be associated with identifiers (AIDs) and the events carry the AID of the corresponding activity, resolving the equivalence is possible. Two events are equivalent if they belong to the same activity class. We can then define a *merge* operator, which acts as a union on the quotient of the identity equivalence class – and is therefore yet another powerful way of implementing a fan-in node.

Building representations from Figure 8 defines an injective morphism between EPNs in the intuitive sense, as represented by the graphical representation and in the relational sense: given a graphical representation, it is possible to define a unique relational expression; the opposite is not true in general, simply because there are relational expressions for which there is no graphical representation.

Moreover, this defines an algorithm to go from the graphical representation (which is introduced in the next section and which is fully described in the RapNet reference section) to the relational representation – which is the basis for a tool like RapNet: starting from the graphical representation, it can convert into the relational sense, and optimize based on that representation.

2. Example

We can code a network composed of a map m_1 whose outputs is connected in parallel to two maps, m_2 and m_3, each of which is connected to a filter (respectively f_2 and f_3), which then are merged back. The representation in our relational language for this EPN is: $m_1; dup; (m_2, m_3); (f_2, f_3); \bigcup$

We also use another representation for Event Processing Networks, which is purely graphical. Below is an example of it, showing the same network in graphical form. In this form, an EPN appears as a Directed Acyclic Graph (DAG), where the nodes are EPAs and the vertices are computations.

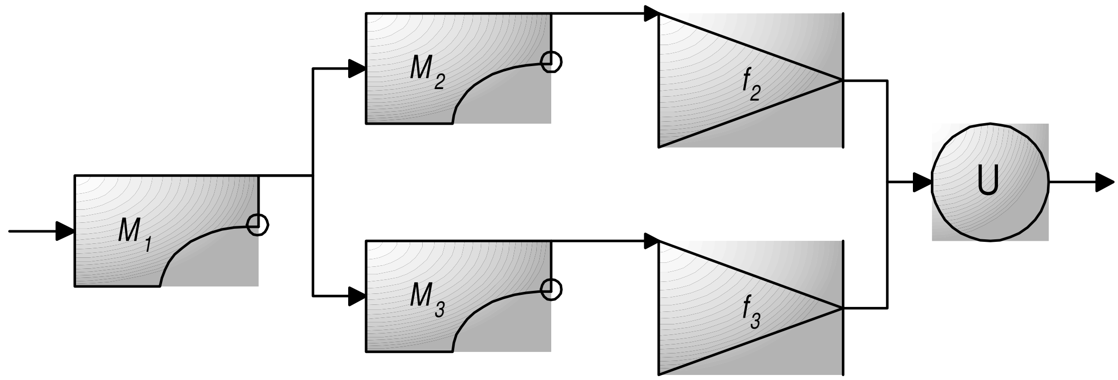


Figure 9:Connections in relational expressions

A complete reference of the graphical representation can be found in the RapNet reference section, as this chapter aims at using the relational form to optimize a network – which is something that cannot be done easily with our graphical representation.

F. Optimizations

The relational representation of Event Processing Networks gives an opportunity to optimize the run-time processing cost. Processing cost includes CPU, I/O and network bandwidth, part of which is implementation dependent. We are giving here a list of possible optimizations that the run-time engine can do and their respective tradeoffs.

All of these optimizations operate on sub-EPNs, i.e. a sub-graph of the EPN which can itself be represented as a node without creating any cycle in the resulting graph. Note that this kind of algorithm is inherently recursive: it starts with the smallest possible sub-EPNs, usually a couple of EPAs and progresses toward larger and larger sub-EPNs. Obviously, the number of combinations is exponential in the size of the complete EPNs, and we therefore use heuristics to try to avoid testing for non-optimal cases.

All of these optimizations respect the structure of sub-EPNs: the result of optimizing a sub-EPN leads to a sub-EPN (sub-EPNs are formally defined in section II.D). This is very important for Synchronous Groups, which are a special case of sub-EPNs: because the user of a tool like RapNet are able to specify where they want to split into Synchronous Groups, the optimizations should not change the corresponding structure internally, i.e. optimizations should be closed under Synchronous Group partition.

On an implementation standpoint, these optimizations can occur within RapNet, when the user hits the Go button.

1. Compression

It is possible to take advantage of certain conditions to test conditions earlier than they would normally be, hence reducing the number of events that flow through the EPN. For instance, a sequence of filters

$f_1;f_2$ can be replaced by a single filter f, where f is defined by: $f = \mathbf{U}\Lambda(p)$, where $p = p_1 \wedge p_2$, if p_1 is the pattern filtered by f_1 and p_2 the pattern filtered by f_2.

We call this optimization a compression, in the sense that it compresses the original (extended) graphical representation by removing as many filters as possible.

Why would anyone ever write such an EPN? There are mainly two reasons:

- For logical reasons: even though the two filters can be combined in one, it may aid the user conceptually to separate them. For instance, a system that needs to filter out chips that have passed test #1 and then, out of these, those that have passed test #2 would most logically be drawn as two filters in sequence.

- One (or both) of the filters may be hidden from the user. This is because the tool we use to build EPNs (RapNet) allows the user to use sub-EPNs, which behave like any regular EPA but may actually contain many EPAs, and may in particular contain a filter. Then, if the user builds an EPN using two sub-EPNs e_1 and e_2, i.e. $e_1;e_2$ and e_1 is $m_1;f_1$ and e_2 is $f_2;m_2$ then the user – without knowing it – actually creates the EPN: $m_1;f_1;f_2;m_2$, where there is an opportunity for compression.

In Figure 10, we show this EPN: the top part shows what the user builds and sees, i.e. two sub-EPNs connected to each other. Below is the internal representation, showing the expanded sub-EPNs. On that level, it is easy to see that the two filters can be compressed into a single one. This is what is done in the bottom part, which shows the resulting, optimized EPN.

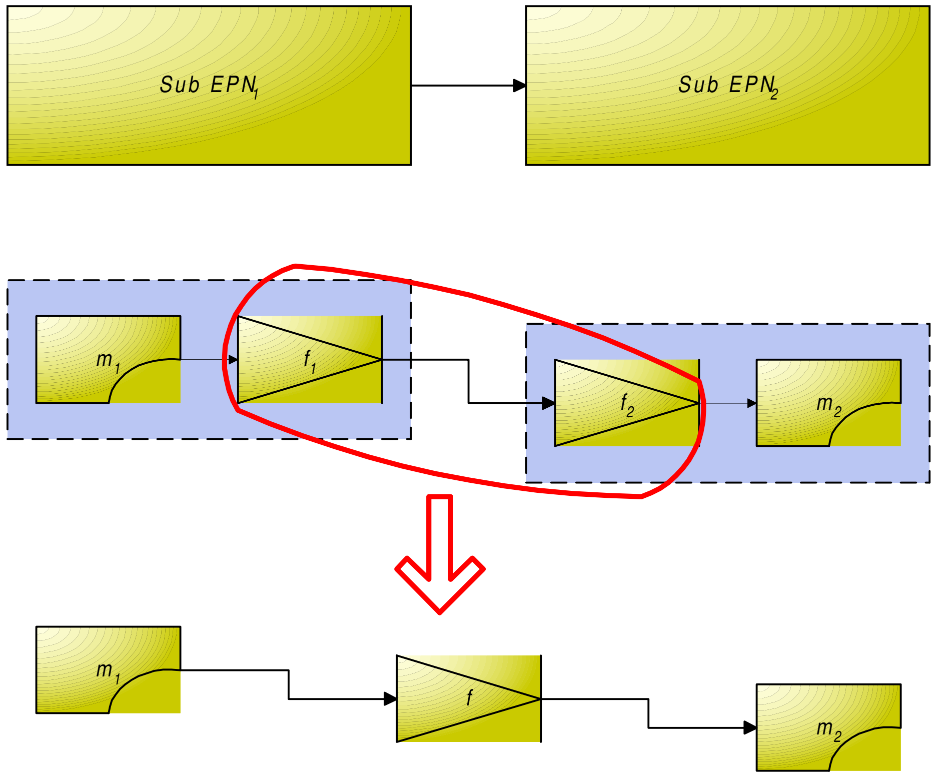


Figure 10: Optimization: Compression

Of course, further improvements can be made if the conditions in f1 are not independent from those in f2: these are fairly classical logic optimizations, which have been thoroughly described in both logic and database research.

This is a tremendous improvement in performance in our system, because of the cost of generating and distributing useless events (this costs CPU power, as well as network bandwidth – in the case of lightweight computations – or disk bandwidth – in the case of non-transient computations).

There is a disadvantage of processing to this optimization like all those presented here: it assumes that no other EPA x will ever be connected to the output of the first filter. If that is not the case, then we are replacing $f_1;dup;(x,f_2)$ by $f_1;f_2;x$, which is incorrect. This is avoided by having tools like RapNet lock EPNs in a way which does not allow adding incorrect EPAs once the EPN is optimized: basically, the user can turn optimization on or off. He will only turn it on when he knows that he will not need to change the EPN in a way that would be prohibited by RapNet.

2. Common Sub Expression Elimination

It is frequent that part of the computation needed to process an EPA can be reused somewhere in the network. This can be usefully put to use by changing the topology of the executed network to avoid redundant work.

For instance, if we have an EPN $f_1;m_1$, where the pattern in f_1 is the pattern in the left side of the map rule, then the filter is useless, and can be safely deleted.

Another case of common sub expressions occurs often when using sub-EPNs, and can be represented as $agent_1;dup;(agent_2,agent_2);(agent_3,agent_4)$. Provided that agents are not allowed to change their inputs (which is currently a requirement, due to the implementation), this sub-EPN can be replaced by $agent_1;agent_2;dup;(agent_3,agent_4)$. This is an obvious win: the new sub-EPN is less computationally intensive, it produces fewer intermediary events and therefore less disk/network contention.

3. Reordering

In general it makes sense to try to execute filters as early as possible in the EPN, as they tend to reduce the number of events that are further processed in the system.

This optimization assumes that the execution of the filter is relatively fast, i.e. that it makes up for the time that would be spent carrying more events through the EPN. This is not always the case, however: two commons counter examples are:

- Expensive predicates: if the pattern in the filter is complex (with respect to those in the other EPAs), then the cost of executing the filter where it would normally not be executed may kill the benefit of moving the filter up in the EPN.

- Low selectivity: if most of the events go through the filter, then the cost of executing the filter will be larger than the little time saved by not generating the corresponding events.

In general, it makes sense to try to put the filters with the highest selectivity and the lowest processing cost up front.

This optimization may involve rewriting the filter pattern. In particular, when inverting a map and a filter, the filter has to operate on a completely different set of events.

Note that this optimization may not always be possible or expressible:

- It may be incorrect to exchange a filter and a constraint.

- Because maps can aggregate events in very powerful ways, it may not be possible to express the filter in terms of vertically related events (with respect to the map), i.e. events of the domain and events of the range of the map, which correspond to each other by the triggering of one of the map rules.

4. Map Rule Elimination

If a map contains a rule that cannot be triggered, that rule is useless and can be eliminated. If a map does not contain any rule – due to elimination – it can itself be removed from the EPN (the mapped events of the map then become the empty computation of the corresponding type, and the unmapped computation is just the input computation of the map).

It is sometimes possible for our system to realize statically that a rule cannot trigger, and therefore to remove it before activating the EPN. This is done by looking at the type of the input computation and at the events that can possibly go through. A typical example of this is: $f(p_1);m(p_2 \Rightarrow q)$ where $p_1 = a$ and $p_2 = b$. Then only events with action declaration *a* go through the filter, but the map will only trigger on events with action declaration *b*.

A more complex example is the same EPN, but with: $p_1 = a(i)\ where\ i < 0$ and $p_2 = a(j)\ where\ j > 10$

5. Filtered Computations

We also define a special kind of computations, called filtered computations. A filtered computation is a pair <pattern, computation> such that all the events in the computation match the pattern. A filtered computation is generated from a normal computation by filtering out unmatched events.

Typically the pattern of a filtered computation has to be very simple (else one uses a real filter). Filtered computations were initially introduced as an implementation optimization: by filtering out useless events early, we avoid having them go through all the layers of the system, most notably through the persistent storage (hence reducing I/O bandwidth) and through the notification subsystem, hence reducing the network bandwith.

Even though filtered computations were introduced as an implementation optimization, they actually have a theoretical advantage as well, because they allow further optimizations. For instance, if the triggers of a map are simple patterns, the normal computation which is an input to the map can be replaced by a filtered computation of the disjunction of the patterns.

G. Synchronous Groups

1. Introduction

We have two paradigms of inter agent communications: in the first one, agents use a local/synchronous notification mechanism (basically, method calls); in the second one they use a remote/asynchronous notification mechanism (a message passing system).

Following certain constraints that we will explain below, these two communication mechanisms can be mixed. In this section we will explain what a Synchronous Group is, what they can be used for, and how to optimize an Event Processing Network by selecting the right Synchronous Groups and optimally allocating resources.

Within a synchronous group, all processing happens in a depth first, synchronous way: As soon as one EPA in the local SG updates its output computation(s), all local subscribers to that computation get called back. Therefore, even if EPAs run in multiple threads, the flow of control within one thread follows a depth first traversal of the DAG.

Splitting an EPN in several Synchronous Groups is an opportunity for distributing the EPN on several machines: within a Synchronous Group, all the EPAs are executed in the same process, and therefore on the same machine, but one can choose to run a Synchronous Group on one machine and another one somewhere else.

Seen this way, a Synchronous Group corresponds quite well to the COM concept of Apartment [32], which is neither a process nor a thread, but represents the ownership by a thread of some object, and therefore dictates whether objects can communicate with each other using regular calls or whether they should communicate indirectly using marshalling proxies.

Choosing the SGs for a given EPN can greatly improve its performance. We have some rules of thumbs to decide how to partition our EPNs when we design them. We are working on developing a more formal theory of optimal partitions.

2. Definition

A Synchronous Group is a sub-EPN which obeys to the following constraints:

- Head: to define Synchronous Groups, one can use the concept of *heads* of sub-EPNs: a node (i.e. an EPA) is the head of a sub-EPN iff all of its inputs are outside of the sub-EPN and if it blocks when started. With this terminology, a Synchronous Group has exactly one head: it obviously must have at least one (except if one considers a complete EPN to be a Synchronous Group); and if it has more than one, then only one of the heads can start.

- Fan-in Nodes: fan-in situations can cause starvation. Therefore, we enforce that either both or none of the connections in a fan-in slot are in the Synchronous Group of the EPA.

In the following paragraphs, we are explaining why we placed such restrictions in the definition of Synchronous Groups. We are illustrating with graphical representations of candidate Synchronous Groups. In those representations, gradient-filled boxes with thick discontinuous borders represent the partition of the EPN into Synchronous Groups.

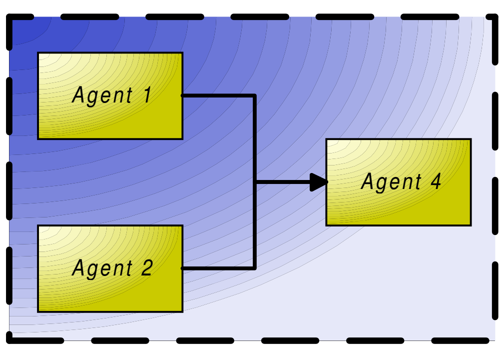


Figure 11 : SGs are restricted to one head

In Figure 11, we show why we restrict SGs to one head: if both *Agent 1*and *Agent 2* are in the same SG as *Agent 4*, then the SG has two heads (*Agent 1* and *Agent 2*). The flow of control will start from either *Agent 1* or *Agent 2*, then go to *Agent 4* (because of the depth-first propagation within an SG), then back to the caller, and so on, and therefore the other head will starve.

This explains why we only allow one head per SG. In reality, we can relax this constraint slightly: if a head agent can be implemented to work asynchronously and cooperatively then we can build a pseudo-head, which precedes all the heads in the SG, and which calls each of the heads in a round-robin fashion: in Figure 12, the pseudo head calls the first asynchronous head. If that agent has data available, it creates the corresponding events, which in turn calls *Agent 4*. Then, the control returns to the head. Because *Head 1* is instructed by the pseudo-head to cooperate, it returns the control to the pseudo-head, which gives a chance to the other head to process its events. This mechanism may seem simplistic, but it actually works very well, because many heads are sniffers, which typically use select() calls to retrieve their information and are therefore inherently asynchronous.

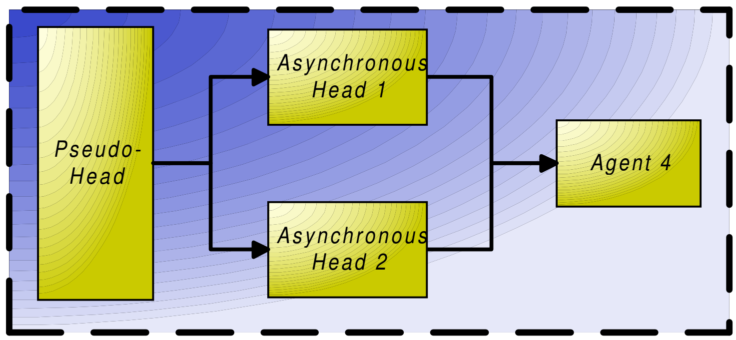


Figure 12: Pseudo-Head in SGs

More generally, fan-in nodes are an issue. On Figure 13, we show an incorrect SG partition. The problem is that *Agent 4* needs to be receiving asynchronous notifications from *Agent 2* (i.e. the corresponding computation has to be either lightweight or heavyweight global, see III.C for details), and therefore everything is as if there was a second head in the SG, called *Listener*, which handles all the asynchronous notifications (see Figure 14).

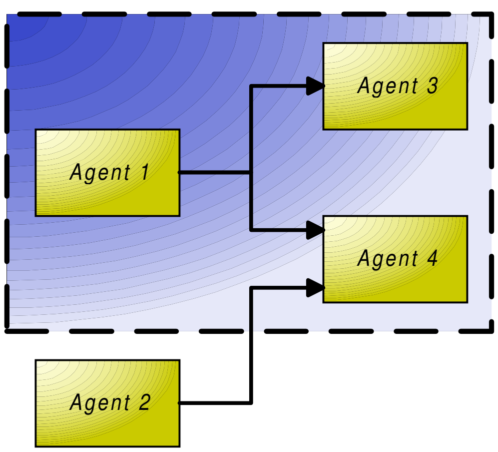


Figure 13: Incorrect SG Partition at Fan-In Node

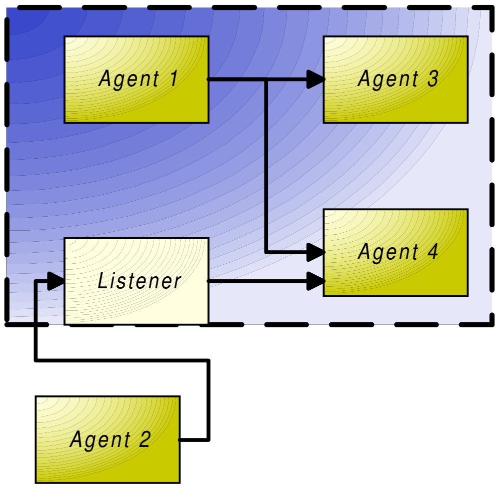


Figure 14: The asynchronous notification can be seen as an extra head in the SG

This restricts the possible SG partitions considerably, although the relaxation that we described above definitely applies to the *Listener*: it is easy to implement the notification subsystem in a way such that *Listener* is both asynchronous and cooperative. Thus, if *Agent 1* is also both asynchronous and cooperative, we can add a pseudo-head, which calls successively *Agent 1* and *Listener*, making the SG in Figure 13 a legal one.

If we cannot use the relaxation mechanism, then a possible SG partition for this EPN is shown on Figure 15: because *Agent 4* is now in its own SG, all the requirements are met to make this a legal partition. Indeed, *Agent 4* receives notification asynchronously from both *Agent 1* and *Agent 2*, which avoids starvation of those agents.

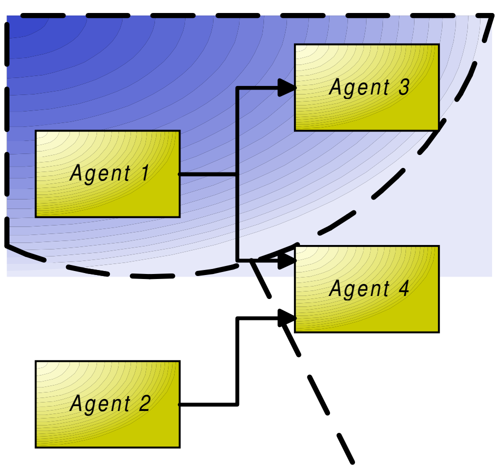


Figure 15: A legal SG partition

3. Example

Let us look at a simple example (based on our Fabline Demo, first introduced in the first chapter of this document, and Figure 2).

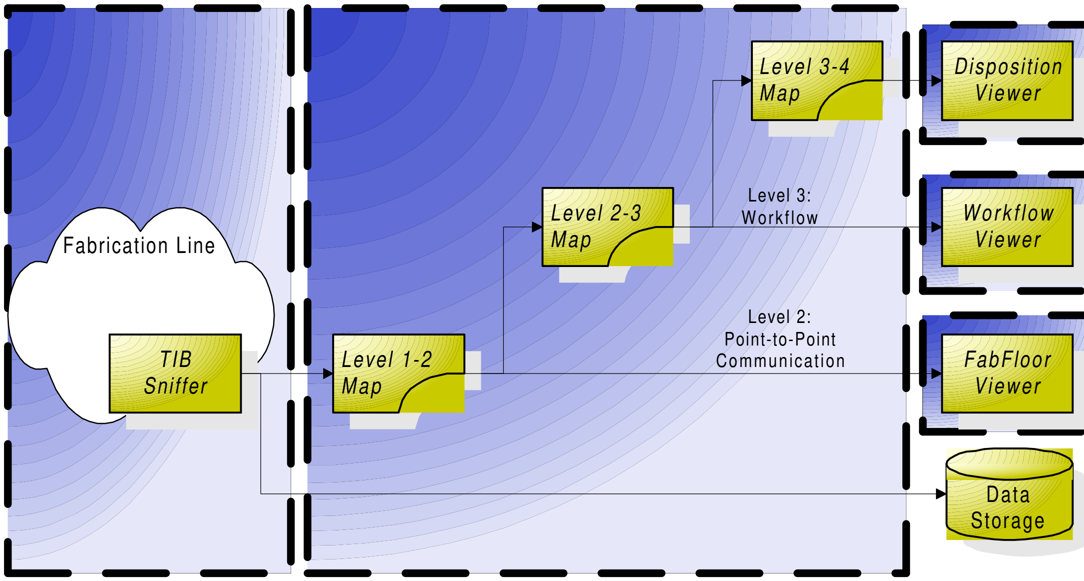


Figure 16 : FabLine EPN split into 5 SGs

In that EPN, the logger creates a first computation reflecting the raw (broadcast) traffic on the middleware. A first map transforms this first computation to a higher level of abstraction, point-to-point communication. This second computation is then read by a viewer that displays the number of point-to-point messages originating at the controller. Likewise, the Level 2-3 and Level 3-4 maps are creating higher levels in our conceptual stack.

We split the fabline EPN into five SGs (see Figure 16):

1. The first contains just the sniffer.
2. A second SG contains the three maps.
3. The remaining SGs consist of one viewer each.

Combining the sniffer and the maps into a single SG could work in this case too. Alternatively, provided we have global heavyweight computations (described in section III.C), we can also separate each map in its own Synchronous Group.

Whenever the sniffer registers an event on the middleware, it adds it to the first computation. This means that it is added to the persistent store and the first map is notified that a new event is available.

Immediately after that, the logger starts to look out for the next event. Independently of the sniffer, the first map is notified asynchronously of the new event and may generate another event in the second computation.

In that case, the second map is called for processing. Because it is in the same SG the first map will not process any more events for now: this is the depth first execution pattern of synchronous groups.

If the second map generates a new event, the third map is called in the same way. After all the maps are done, the first map starts looking at the next event. Independently each viewer is notified asynchronously of incoming events on their level.

4. Design Issues for SG partition

High speed loggers can be slowed down considerably by synchronous processing of events. The time spent calling back following EPAs might be longer than the time to the next event to log, eventually leading to missed events or to a growing queue of waiting events.

On the other hand, if the raw output of the logger is not relevant, calling a simple filter for each event immediately might remove the need to store that raw output altogether and lead to higher overall throughput.

As one can see there are trade-offs when partitioning an EPN into multiple SGs. There are some rules to efficiently decide on the partition:

- SGs follow the same rules as Sub EPNs. They partition the EPN in a way that is consistent with the vertexes of the EPN. This puts a topological constraints on the possible SGs and therefore narrows considerably the search space in an algorithm which tries to find the optimal SG partition.

- Any EPA that runs in a separate process and/or on a separate machine is by definition in a separate SG. This includes for instance all loggers which instrument some legacy code (e.g. eJava [5]).

- Viewers: If a viewer lives in its own process, then the preceding rule applies. Else, since the user cannot view thousands of events per second, viewers will typically receive events at a low rate and do a lot of processing on them (e.g. POV) or at a high rate but then only show aggregate information on them (e.g. count of events with a given signature). In the first case, if the processing time at the previous node of the EPN is not much greater than that required at the viewer, it may be wise to put the viewer in its own SG.

- Currently, distribution of EPNs occurs on an inter-SG level: within a single SG, all agents run in the same thread, on the same machine. Therefore, to take advantage of a cluster of machines, one tries to split the EPN in as many Synchronous Groups as there are machines, provided this does not overload the available network bandwidth of the cluster.

H. RapNet

1. A Tool for Managing Event Processing Networks

This section introduces RapNet. A user manual can be found on-line or as an appendix to this document. The architecture of the tool is described in section III.F.

Once we started designing a few hard-coded Event Processing Networks, we realized that it would be very valuable to have a tool that would allow one to create, instantiate and modify on the fly a network while the system is running. This allows to design networks incrementally, which fits well in the process: it allows the engineer to modify the network every time he discovers new information.

The tool we have developed to accomplish this is called RapNet. RapNet offers a GUI to instantiate and communicate with EPAs. The GUI lets one draw an EPN in a form that is based on the graphical representation we introduced above; given that EPN, RapNet is responsible for its configuration, its instantiation (i.e. creating the actual agents in the EPN and hooking them to each other using computations) and its optimized distribution on a pool of machines (using the Synchronous Group mechanism described in the previous section).

a) Requirements

- RapNet should be easy to use: most of our customers don't want to learn the CEP pattern language or *a fortiori* the map, filter or constraint languages. They are completely happy with having an architect write the EPAs for them, and just want to be able to assemble them as needed. Such EPAs will usually have a few free parameters which the user can configure.

- RapNet should be able to load a library of EPAs: the architect makes a given set of EPAs available to the engineer for building EPNs. Whether these EPAs are simple EPAs or complex EPAs should be indifferent to the engineer. Complex EPAs include Sub-EPNs, which are described in section II.D.

- It should be flexible: if an object is able to use the Computation Subsystem (a part of the RAPIDE CEP implementation, described in III.C) , then RapNet should be able to use it.

- It should be portable: if an EPA can work both under NT and Solaris, then RapNet should be able to start it on either of those platform, and should also be able to run there.

To meet these requirements, and although the complete infrastructure is written in C++, we decided to implement RapNet in Java. We therefore designed a Java interface to the Computation Library and to the EPA interface. The current implementation of this interface makes an extensive use of the Java Native Interface (JNI), after having tried other communication models.

The GUI of RapNet was strongly inspired by other products such as Visio[1]. However, we consider the rest of the architecture of RapNet to be original enough to be worth presenting in a separate section of this document (III.F).

[1] Visio is a line of tools made by Visio Corp. More information can be found at www.visio.com

b) Support for sub-EPNs.

RapNet supports a hierarchy concept, where EPNs can be constructed out of other EPNs. Such a sub-EPN can be viewed as a complex EPA itself. This means that a sub-EPN can be part of the library of available EPAs. The user does not need to know that the EPA is actually a sub-EPN.

In RapNet, the user can choose to either view a sub-EPN as an EPA, i.e. as a single node in the network, or to expand it to show its internal structure. This is possible, because by definition of sub-EPNs, doing so will not create a cycle in the original graph, i.e. expansion transforms an EPN into another EPN.

A sub-EPN may be password protected, in which case a password is necessary to expand it: this way, an architect can sell a library of EPAs without providing their source code.

c) Support for Synchronous Groups

Sub-EPNs are architecturally defined to be EPAs, enabling hierarchical design by making a sub-EPN just a special case of EPA.

Likewise, a synchronous group is by design a special case of sub-EPN. In a synchronous group, all the elements cooperate using a local notification mechanism. This has several consequences:

- The cost of execution is lower than using a global notification scheme, because notification goes through regular procedure calls instead of our external publish/subscribe mechanism. This gain is of several orders of magnitudes, but the overall improvement in not always that significant, because notification is often a small portion of the actual computing cost.

- The semantics within a synchronous group are slightly different from the external notification scheme: procedure calls are synchronous and we do them in a depth first order, whereas the asynchronous notification scheme tends to do the calls in a breadth first search fashion. In general, this does not change the output.

- The trade-offs involved in choosing synchronous groups within an EPN have been studied in the section covering synchronous groups (section II.G).

RapNet can allow synchronous groups by selecting a sub-EPN and deciding to make it a synchronous group. Some additional rules apply to synchronous groups, because it is not possible to interleave groups.

It is not possible to have Synchronous Groups as part of a library of available EPAs, though, and trying to do so upcasts the SG back to a sub-EPN. This is because adding connections to a Synchronous Group, as seen as a black box may result in an incorrect SG: in Figure 17 we show a simple and perfectly legal Synchronous Group, consisting of just a head and another agent. However, if that SG could be added to the library of EPAs as an SG rather than as a sub-EPN, and one connected another head, *Head 2* to the agents (shown on Figure 18), then an incorrect SG partition would result (as explained in section II.G.2).

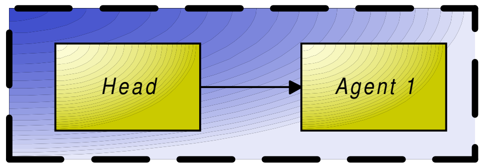


Figure 17: If one can add an SG into a library of EPAs...

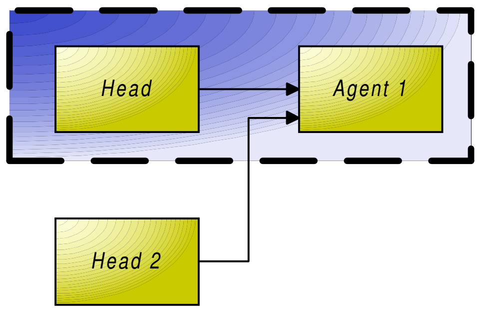


Figure 18:... then incorrect SG partitions may result

We don't expect many users to want to control the partition of their EPNs into multiple SGs. Therefore RapNet can be told to attempt to partition an EPN optimally, following the rules we mentioned in section II.G.4.

d) Support for strong type checking

The underlying language of Event Processing Networks, called RAPIDE, is strongly typed: computations have a type, which describes the kinds of events they can carry. Moreover, the type system in RAPIDE has a notion of inheritance, i.e. some types are subtypes of others, and an EPA can accept any subtype of its expected input types.

Because of that, EPAs themselves have a type, which is the combination of the types of their input computations and of the types of their output computations, each with a given cardinality. In our relational representation, EPAs are binary relations from the cross product of the input computations of the respective input types to the cross product of the output computations of the respective output types.

RapNet does both type checkings:

- EPA type: some EPAs can take multiple computations on a given slot, others cannot. This is specified by the EPA to RapNet, and RapNet only lets the user connect as many computations as are available for that EPA.

- RAPIDE type: RapNet ensures that the RAPIDE types are respected, and only lets the user connect EPAs in a way that respects these types: for example, the type of an output of an EPA must be a subtype of the input types of the EPAs to which this output is connected.

e) Support for persistence

So far, we only discussed "computations". In the real world, the single concept of computation is relaxed for efficiency reasons. While a first time user may build an EPN using only "computations", advanced users have more control over different aspects of computations. For one thing, computations can be declared non-persistence, or transient. Transient computations are called *event channel.* Within an EPN, they provide the same functionality that computations do, except that the data associated with them is not stored persistently. Using Event Channels instead of Computations improves throughput for an EPN.

Both, computations and event channels can optionally be declared lightweight. Lightweight means that the computations are made available outside the local synchronous group in a reduced form without information. Some EPAs need not more than lightweight, and these EPAs run more efficient on lightweight computations. However, the process of publishing lightweight is additional effort for the publisher. Lightweight should only be turned on if needed.

2. Usage Scenario

a) Roles

We are assuming two different roles in Complex Event Processing:

- An architect: the architect is responsible for creating the agents which constitute basic building blocks to design EPNs. There are many tools available to the architect in order to build those agents and the architect uses them to fulfill the following responsabilities:

  - Maps, Filters and Constraints can be designed graphically, using a tool like RapNet, usually by filling placeholders in predefined EPAs. Alternatively, for more flexibility, they can be written in RAPIDE, and then registered as new agents in the library.

  - Custom agents can be written in any language. This includes, for instance dedicated producers. Usually, the existing viewers should be sufficient to view almost any data in any possible format; however, if the architect wants to design a viewer with the same functionality as our viewers (mainly the capacity to relate vertically-equivalent events and the ability for two viewers to communicate on different sets of events), he can use our dedicated framework to do so – including APIs in multiple languages.

  - RapNet lets the architect build sub-EPNs, which are themselves considered to be EPAs. Once a sub-EPN is added to the library of EPAs, nothing distinguishes it from other EPAs.

  - More generally, tools like RapNet use a registry which describe the agents which are available to build networks. This registry is setup by the architect, which may get the help of installation scripts: when the architect installs a new EPA, that EPA can self-register. The agent registry is described in section III.D.3 and a full reference is available in sections V.A.

- An engineer: the engineer uses the agents which are available in RapNet to design new networks. If the architect has given him the right set of agents, he should be able to solve any problem he wants and to refine queries as needed, even while the network is running.

Typically there will be one architect and several engineers. Each engineer has his own view of the available tools (shown in a tree view in RapNet), because all engineers solve different problems.

b) Example

For instance, in an intrusion detection system, the engineer sets up a network of nodes, each of which is aimed at either watching or stopping intrusion attempts. This involves a set of producers, which either read gateway log files or sniff directly from the network (e.g. using tcpdump). The producers are connected to filters, because attempts will be watched for some given machine and because some port numbers are known to be weak points. Filters are connected to maps: some are convenience maps, which transform the data in a way that is understood by viewers; others are more involved and aim at discovering unusual patterns of behavior.

At some point, the engineer may detect a new kind of intrusion and want to add a node to stop that kind of intrusion while the system is running. RapNet makes it easy for him to create a new EPA (or configure an existing one), and to connect it to existing EPAs.

RapNet also lets the engineer reconfigure existing EPAs: for instance, he may want to start watching another port as well. This can be done in limited ways by RapNet, because it causes a problem in the way our architecture works: because patterns can express things which happened in the past, changing the configuration of EPAs may create undesirable discontinuities of behavior: for instance, if the filter now starts to let port 80 go through, although it stopped it before, then a pattern asking for all the past instances of port 80 will miss all the ones which predate the current time. RapNet enforces correct behavior by prohibiting creating such discontinuities while the system is running: major changes require one to either stop using past events (such as is the case with lightweight computations) or to stop the EPN, and then restart it.

RapNet lets the engineer chain from previous results: once the engineer has detected an intrusion pattern from a given address, he may want to monitor all activity coming from that address. There are several ways of doing so. The one which we currently use is a multi-input map, where one of the input is the raw data and the other input is the output of the intrusion detection system (which can be seen by this map as a sub-EPN).

III. CEP: Architecture of the system

The Event Processing Network system is a complete tool set of integrated products. It can also be used as an extensible framework for third party developers.

The whole system was designed in a layered architecture, as can be seen from Figure 19 . This chapter describes the different components of the system, giving a rough outline of those that only exist as infrastructural services and do not contribute to the understanding of Event Processing Networks.

In terms of actual products, the Event Processing Network system acts as an infrastructure for our management software: RapNet. The main point of this chapter is then to describe the underlying architecture and services of the system, as seen by RapNet, as well as RapNet itself.

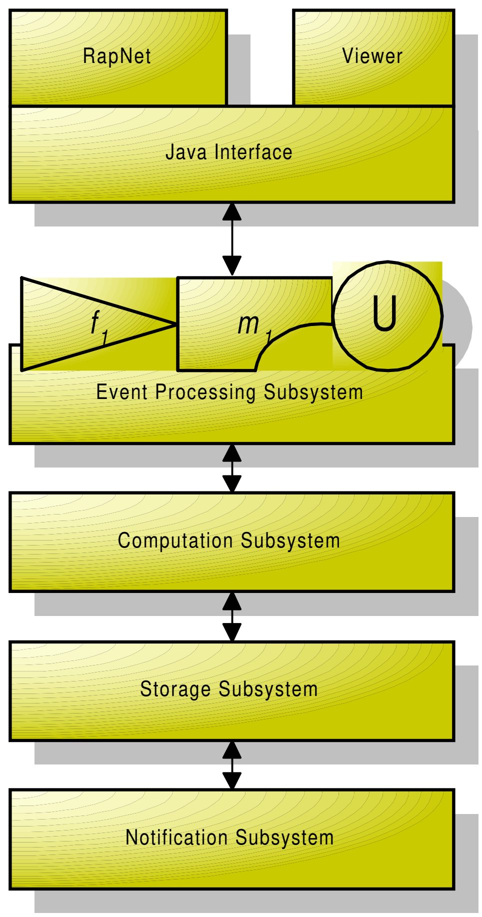


Figure 19 : Architecture of CEP

A. Notification Subsystem

The Notification Subsystem provides location transparent communication between Event Processing Agents. It provides a hierarchical publish/subscribe communication model: an agent subscribes to a set of subjects. When some other agent publishes under that subject or under a subject that inherits from it, the Notification Subsystem notifies the former agent by forwarding it the message.

The default notification paradigm is asynchronous, just like in the typical Information Bus architecture: a daemon (which can be distributed for scalability and/or replicated for availability) runs on the system. Subscribers subscribe to a given subject or set of subjects with the daemon. Producers send messages to the daemon, which then forward appropriately to the corresponding subscribers.

This is currently implemented with raw UDP sockets. However, the Notification Subsystem itself is layered in such a way that very little code has to be changed to adapt it to another communication medium or protocol. For instance, since the system typically sniffs messages from a piece of middleware, it may sometimes make sense to use that middleware for inter-agent communication as well.

However, for efficiency reasons, the Notification Subsystem supports another notification paradigm, which is synchronous: that mode allows Event Processing Networks to be subdivided into Synchronous Groups (SG). In such a group, all notifications are simple procedure calls, i.e. they are synchronous.

The NM hides this dual mechanism from the upper layers of the system.

B. Storage Subsystem

The role of the SM is to provide location transparent memory access to events and computations.

Currently there are two implementations of this layer. Both permanently store the objects in an object oriented database management system (OODBMS): one of them uses a commercial OODBMS from Objectivity, Inc., whereas the other one relies on a small custom database engine.

In this layer, events are stored on disk, and they are grouped in event containers (an event container corresponds to either a computation or a subcomputation). Other objects, such as time-, causal- and architecture stamps, as well as clocks and threads are stored on disk as well. This is not further discussed in this document, but reference to the class hierarchy in the Storage Subsystem and their correspondence to higher level objects can be found in [33, 34].

All objects are very small but point to many different objects, which influenced the design of that subsystem, and was one of the reasons to use an object oriented system (the other one being that the type of queries that are done by upper layers is highly path oriented, which is hard to express in SQL).

The database is optimized for the multiple reader - one writer scenario (MROW), where the readers always have access to the data, possibly in a stale version: most of the time only one logger will write in a given event container.

Data can be cached at the SM level, but this is completely hidden to the upper layer.

C. Computation Subsystem

The Computation Subsystem (also called Computation Library, or CL) layer provides many helpful algorithms related to causality and hides specifics of memory access and agent communication.

This layer drives the lowest two layers, and exposes an interface where communication amongst different agents is completely contained within the computation or event channel (for the rest of this section, we will not separately list event channels, what applies to computations also applies to event channels) through which they communicate: the way agents get notified of new events is that they register observers on the corresponding computation. The Computation Subsystem takes care of publishing on the notification Subsystem when an event is added to a computation. Observers register themselves to the notification daemon for the subject corresponding to the computation and therefore get called back when an event is added to a computation.

The two mechanisms of notification available at the Notification Subsystem level (namely synchronous vs. asynchronous) translate to a property of computations, which we call "lightweight". Computations can be one of three types with respect to the lightweight property:

- A local heavy weight computation uses the local (synchronous) notification mechanism. During the callbacks that implement notification, the events are read from the database. Nothing goes over the network. Due to current limitation in the implementation of the system (mainly, that it is not reentrant), only one heavyweight thread exists per database.

- A global heavy weight computation uses the remote (asynchronous) notification mechanism to pass object IDs over the wire. This way, an observer can retrieve information stored in the database, even though both the publisher and the subscriber may be running in completely different processes on different machines. This is currently not implemented as it requires major changes to the infrastructure, but it is one of the future directions of the RAPIDE CEP system.

- A light weight computation uses the remote (asynchronous) notification mechanism. Events in lightweight computations are sent (i.e. the events themselves, not their object IDs) to the Notification Daemon, i.e. transit over the wire and are forwarded to event observers. Only current events can be observed, i.e. clients are not notified of past events. Moreover, to save network bandwidth, not all of the information that we usually store in an event is generated, hence the name of "lightweight".

In general, lightweight computations are used for lightweight viewers, which are typically simple meters that do not use any causal information, but just plot some of the parameters of the incoming events.

Until the system (mainly the Storage Subsystem layer) is made re-entrant, the only way to distribute an Event Processing Network within one database is to use lightweight computations. This means that Synchronous Groups are currently the main way to distribute an Event Processing Network.

In addition to lightweight, the Computation Subsystem also manages another important property of computations: normally, computations are persistent, i.e. the Storage Subsystem is responsible for writing events to disk. However, some intermediary computations do not need to be persistent; for instance, mappers create temporary event containers to store incomplete matches. Therefore, for efficiency reasons, the Computation Subsystem is also able to create transient computations, which do not reside on disk and therefore decrease the disk throughput requirements.

D. Event Processing Subsystem

1. Event Processing Agent Interface

All EPAs implement the EPA interface. This enables a single tool – such as RapNet – to manage all possible EPAs, hence providing a general extensibility mechanism. The design of the interface is inspired by both Java's reflection mechanism and COM automation (which is described thoroughly in [32]).

We are presenting this interface in detail, as it is an important part of the whole architecture, most of all from RapNet's perspective.

```
class EPNEPA {
public:
  /** Virtual Destructor for protocol class. */
  virtual ~EPNEPA() {};
  /** Returns an argMap of EPN arguments.  The values of the EPNArgMaps are set to the
default or current values.  */
  virtual void getArgs(EPNArgMap & arguments) const epnthrow2 = 0;
  /** Sets the arguments in the EPNArgMap to the values provided with   the argMap.  Setting
arguments to the same values provided by getArgs should not change the state of the EPA.  */
  virtual void setArgs(const EPNArgMap & arguments) epnthrow2 = 0;
```

This is used extensively by RapNet and is the main way for RapNet to communicate with agents. Through this interface, EPAs can make some of their parameters available, and getArgs()/setArgs() allows a tool like RapNet to present those parameters in a property sheet and therefore to configure EPAs during the construction of the EPN, without having to rewrite or rebuild a single line of calls.

Different types of arguments may be set using these functions, corresponding to standard predefined types (arguments of boolean, integer, string or float types) or to more complex and RAPIDE specific arguments (such as computation names and types).

Maps and filters will often have a few free parameters which the architect expects the engineer to fill in: for example, in our intrusion detection agents, there is a way to filter out connections which do not go to a secure port; this translates to a filter which can accept a list of ports to scan for.

Other examples are viewers, which can typically be configured in many ways, and producers, which often need to be given the source of the data.

In RapNet, these properties can be edited by the engineer by filling in property sheets. To do this, the engineer double clicks on an agent, which pops up a property sheets containing the graphical representation of what RapNet got from the getArgs().

```
/** Returns an argMap of arguments that can be changed after start.  The values of the
EPNArgMaps are set to the default or current values.  */
  virtual void getRuntimeArgs(EPNArgMap & arguments) const epnthrow2 = 0;
```

```
/** Changes (after start) the arguments in the EPNArgMap to the values provided with the argMap. Use setArgs for changes before start. */
virtual void changeArgs(const EPNArgMap & arguments) epnthrow2 = 0;
```

Those methods are the equivalent of setArgs/getArgs in the case where the EPN is already running. Typically, you only want to be able to change a proper subset of the previous parameters, to avoid creating inconsistencies with past events. This has been described in section II.H.2.b).

```
/** Returns true if all required args have been set except EPNInterfaceArgs. Note that things can go wrong even if configured returns() true. */
virtual bool configured() const epnthrow2 = 0;
/** Start the EPA (if not started already). */
virtual void start( ) epnthrow2 = 0;
/** Terminate the EPA's execution. Complete's all output computations of the EPA. and initializes all state. Pausing and other ways of stopping an EPA are TBD. */
virtual void terminate( ) epnthrow2 = 0;
```

These methods are used by RapNet when the user hits the go button. After making RapNet level checks, the tools calls configured() on all the EPAs in the EPN, and if everything went fine, RapNet calls start on all of the EPAs, starting with the connections, in topological order – the order does matter, because filter and map outputs are subcomputations of their input.

```
/** Returns true if the EPA will not return from start until he's completely done. */
virtual bool blocking() const epnthrow2 = 0;
```

This is important for RapNet, because blocking EPAs correspond to heads of synchronous groups. For instance, at most one blocking EPA can exist per synchronous group, else the group will dead lock – which is enforced by RapNet. When RapNet discovers a blocking EPA, it starts it in its own thread. The rest of the SG joins that thread automatically, even though they are created originally in another thread.

```
// EPA status observers
struct EPAStatusMessage {}; // empty as of now...
typedef CMObserver<EPAStatusMessage> EPAStatusObserver;
/** Register an EPAStatusObserver */
virtual void epaObserverFromNow(EPAStatusObserver & obs) epnthrow2 = 0;
```

This is used by RapNet to register itself as an observer on the EPA. This is a means for the EPA to communicate information on its current status to RapNet. The basic functionality is used for the EPA to notify RapNet that it is completed, in which case RapNet can show this information visually. When all the EPAs are completed, RapNet considers the EPN to be completed, and starts using setArgs/getArgs again, instead of their run-time version. Other functionality can be added by deriving a new class from EPAStatusMessage and the corresponding observer in RapNet (This could potentially be added to the Agent Registry if needed). Such functionality could include more detailed information on the running EPA, such as performance statistics, progress report, etc.

```
typedef map<string, EPNEPACreator *, less<string> > EPARegistryMap;
```

This is the data structure which we use to register EPAs. RapNet uses the Agent Registry, through the Agent Library, to know which EPAs are available, by name. This is the structure which, given the name of an EPA, returns its factory class.

```
// The EPA registry
/** Register EPAs with a name. All EPAs should call this in a static block */
static void registerCreator(const string & name, const EPNEPACreator * creator) epnthrow2;
/** Lookup EPA's by name. */
static EPNEPA * lookup(const string & name) epnthrow(EPNUsageException);
/** Get List of all known EPAs */
static void EPAList(list<string> & list) epnthrow2;
```

In order for the EPA interface to create EPAs, given their name (the name is given to RapNet by the agent registry), we need two mechanisms:

- There needs to be a way to know which dynamically linked library (DLL under Windows, .so under Solaris/Linux) has to be loaded in order for the EPA to register itself and be found; this allows for late binding, hence avoiding to have to re-link the system every time a new EPA is to be added. This could have been added to the Agent Registry, but we felt that it was mainly an implementation detail which even the Architect should not have to know about, and therefore it is done as an EPN property, for instance through the .epnrc file.

- Once the dynamic library is loaded, all EPAs are supposed to contain static code to register their class factory in the registry. This is the role of registerCreator().

From RapNet's perspective, the most useful method here is lookup(): given the name of the EPA, it gets its class factory ("Creator") from the EPA registry, which creates a new instance of the EPA, which is returned back to RapNet, through the Java interface.

```
/** Look in the registered CM declarations for the type name. */
static CMAggregateType& findOrRegister( const string& typeName,
                                        const string& fileName = "") epnthrow2;
```

This is a convenience method, and is an alias to ExternalForm::findOrRegister(). This method is used by RapNet to get registered declarations for a given type, and is used amongst other things to get declarations of maps and filters to be passed to the corresponding EPAs (mappers and filter EPAs).

2. Agent Library

To create and manage EPAs, we need some way of knowing what agents are available, where they are physically located, how to activate them and communicate with them.

This is a rather classical problem. For instance, in Corba, it is addressed by the naming service and by locators. COM/DCOM has its own way of registering and discovering COM objects (mainly through the Win32 registry). Our case is simpler because we currently don't need to do run-time discovery: we can hard-code these locations in a simple registry. However, there is still the requirement that you do not want to rebuild the whole system, just because you want to make a new Agent available.

Some Event Processing Agents may be linked in with the Computation Library itself. Others may be running and can advertise themselves to interested parties, possibly through the CL. Yet others are available as executables but are not currently running and therefore have no way of notifying of their existence.

Therefore, we needed a library that would allow us to manage EPAs transparently, without having to know the details of their activation or communication protocols.

This was implemented through a small library, whose functionality is inspired by Corba activators and locators.

3. Agent Registry

To provide its functionality, the Agent Library reads information from a central registry. This registry is defined in XML, and the default XML file, along with its validating DTD are given in appendix. More details on how to specify available EPAs can be found in the RapNet reference manual. We believe that the protocol to update the registry will most likely be one of the following:

- One of the architects of the system updates the registry when he wants to make a new EPA available to an engineer.

- When new EPAs are available, the script which installs them also updates the registry. We expect to be able to make such EPAs available remotely through the web: a client tool such as RapNet can load a directory of available EPAs from any remote source, the architect can select useful EPAs on that list and download them automatically.

Part of the DTD is dedicated to RapNet, and is discussed in the RapNet reference.

From looking at the default XML file one can note that three categories are empty: map, filter and constraint. These correspond to mapper, filter and constraint agents respectively. Although those EPAs may have several different implementations that one might want to choose from – as they currently do – we feel that the clients of the Agent Library are more interested in the content than in the container, e.g. about the different types of available maps than in the available mappers.

The difference is that maps are static information stored in the storage Subsystem in the form of an AST, whereas the mappers are the actual EPAs that process them. Although we have different implementations of mappers, we feel that the maps themselves should be in the directory, not the mappers.

On the other hand, it does not make sense to enforce that the Computation Library, which maintains and stores the AST, updates the registry every time a new map, filter or constraint is created; hence the Agent Library considers the categories Map, Filter and Constraint to be special and queries the Computation Library to get the available operators.

The current implementation of the Agent Library uses Sun's XML parser, which allows to associate JavaBeans to XML elements, and hence facilitates the implementation of such semantics.

E. Proxies

In this section, we describe possible event processing agents that enable RapNet to talk with legacy agents: whereas RapNet can talk to linked-in EPAs using regular Java Native Interface (JNI) function calls, there needs to be some mechanism to talk to external EPAs and in particular, to agents that do not implement the EPA interface.

To do this, we define special kinds of Java EPA objects, which act as proxies to the actual EPA. Then the actual EPA may or may not need to be able to talk to its proxy, and in particular may only be able to execute a subset of the functions of a regular EPA implementation. Two proxies are currently useful:

- If the external EPA is a Java application/applet, then we use Remote Method Invocation (RMI): the EPA registers itself as an exported object, and the proxy binds to it, then can make all the regular function calls. On the EPA side, except for the code that does the binding, this is completely transparent and only involves implementing the EPN.EPA interface.

- If the external EPA is not written in Java (usually it will be C++ or Tcl/Tk), then we use a special form of proxy, which knows the configuration parameters that the program it proxies for accepts (through the XML registry described in section III.D.3) and lets the user of RapNet (the engineer) configure them. When the user clicks on GO, the proxy starts the actual EPA as an external process (possibly on a remote machine), and ceases to have control over it. This proxy is intended for legacy code, where the EPA does not have to support the EPA interface.

Note that external EPAs cannot be notified through local subscription in the Notification Subsystem, and therefore a lightweight computation is necessary (or a global heavyweight computation, once they are available). This means that a Notification Subsystem daemon needs to be run on some machine and both RapNet and the EPAs need to be aware of which machine/port the daemon operates from (this is done by configuring the .epnrc file accordingly). It also means that the computation to which the EPA is connected needs to be lightweight (in the case of RapNet, this can be set graphically by double clicking on the connection and selecting the lightweight checkbox).

If this is not done, then the external EPA has no way of getting any notification from its source, and therefore the events will be lost.

F. RapNet

This section describes the architecture and design decisions taken by the author in building RapNet. We believe that they are useful not only as documentation, but *per se* and therefore present the design of RapNet in its own section of this document.

The graphical interface presented by RapNet is strongly influenced by other existing products, such as Visio. The underlying design and architecture of RapNet is strongly inspired by the Design Patterns book[35], and the Concurrent Java book [36].

RapNet is a generic tool to design flow graphs. Any object which can be represented as a black box and connected to other objects can be driven by RapNet. In that sense, RapNet is a very powerful Visio-like (or BeanBox-like) tool. It adds other functionality, though and therefore it made sense to develop a new tool rather than to try to extend existing products (moreover Visio only runs on Windows platform, whereas most of RST's customers are expected to use Unix based systems).

To achieve this generality in RapNet, the design of the tool is separated in two separate packages:

- A main program, which is very generic and provides the environment in which agents are created, described in III.F.1.

- The framework to be implemented by actual agents, which is further decomposed into several components, described in III.F.2.

In the last section of this part, we describe the interactions between RapNet and the underlying CEP infrastructure.

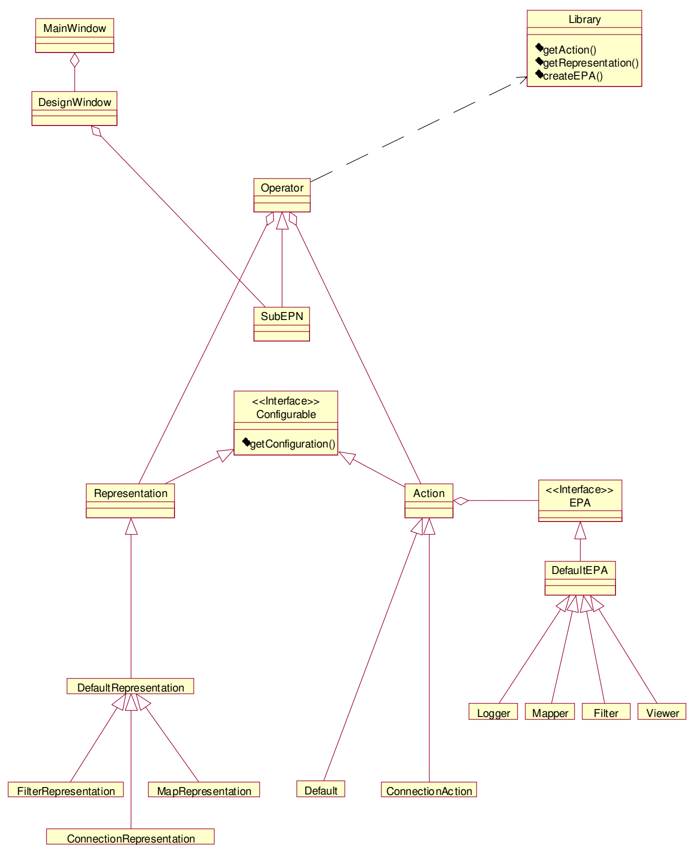


Figure 20: Architecture of RapNet (Class Diagram)

1. Main Environment

The MainWindow and DesignWindow classes represent the main classes of RapNet. MainWindow is a singleton class, which implements the frame that contains RapNet, as well as the menus, toolbars, and which creates both the DesignWindow and the Tools Window. Currently, DesignWindow is also a singleton because there has not been a need to work on several EPNs in the same instance of RapNet. This should be fairly easy to change, though, if the need were to arise.

An instance of DesignWindow contains a SubEPN, which represents the topmost sub-EPN, i.e. the EPN that the user is currently working on. Therefore, SubEPN can be considered has having two separate uses: it is either an actual sub-EPN or it is the EPN. This design is strongly inspired from the Composite Pattern in [36].

SubEPN both inherits from the class Operator and contains a list of instances of Operators. This design decision is what makes SubEPNs behave exactly like any operator, hence making them transparent to the user.

2. Agent Framework

Figure 20 shows a UML Class Diagram (a reference on UML diagrams can be found in [37]) of the agent framework within RapNet. To achieve generality in RapNet, the design separates agents (i.e. what the user creates graphically) into four parts:

- The Operator component is a thin shell which mainly contains pointers to the other components. When the user inserts an agent in the RapNet design canvas, RapNet actually goes and creates an Operator, which in turns instantiates the remaining components.

- A Representation component: this encodes the graphical aspect of the agent in the RapNet design canvas, as well as the interaction that the user can have with it (moving, resizing, etc.). Different agents have different graphical representations as well as different interactive features. For instance connections behave rather differently than regular agents; thanks to this design, though, this is completely transparent to RapNet. An extensive library of implementations of the representation exist, which makes it easy for the system Architect to specify unique shapes for unique components of the networks.

- An Action component: RapNet uses this component to talk to the EPA object. It is responsible for setting the arguments of the EPA according to the property sheet which the user can set by double clicking on the representation component. This component is also responsible for making persistent the state of the controlled EPA, by serializing it when the user wants to save EPNs.

- The EPA itself: this may be a Java class, a linked-in C++ class or even an external process – in which case RapNet can execute in on a remote machine using a special EPA, called a Distributor. The EPA is created by the Operator component, which uses the Agent Library to get the details of the construction, localization and activation mechanisms for that EPA. The EPA is controlled by the action component.

The actual implementation of this component model is done by deriving from the following classes:

- RapNet.Configurable is an interface which represents a tab in the property sheet (i.e. the dialog box which the user invokes by double clicking on the representation of an agent, and which lets him set the arguments of the EPA), and contains all editable properties of an object; it is also able to set the properties of that object, given the fields of the property sheet.

- RapNet.Operator is the way RapNet thinks of EPAs. An Operator is just a wrapper around an Action, a Representation and an EPA.

- RapNet.Representation represents the operator as a box on the screen, describes how many inputs/outputs the operator has, etc. Representation implements Configurable, hence allows the user to control GUI parameters of the operator.

- RapNet.Action, uses the CL to instantiate and manage the actual EPA. Since Action also implements Configurable, the implementor of an Action class must also provide a property sheet for its class.

The Action component is the same for all agents implementing the EPA interface. This interface enables (and forces) agents to implement methods that can give RapNet:

- The number, type, cardinality and name of inputs and outputs of the object.

- A way to ask the object whether it is configured properly.

- A way to start an object and to know whether it has completed.

- A way to change some of its arguments while it runs.

The default implementation of Action also provides a default property sheet, which is simply the list of all the editable properties of the agent. It is possible in the registry to specify the property sheet of the operator. This can be done to get a nicer GUI representation, or to provide several sets of property sheets, depending on the user's privileges or knowledge: the Architect may have her 'expert' property sheets, while the Engineer will have a more basic set of properties.

For EPAs that require more functionality, the Action component will depend directly on the agent being implemented, and can be given in the registry.

One typical such extension is the use of customized callbacks: as we described in the section describing the EPA interface, before starting an EPN, RapNet will register observers (callback objects) on all of its EPAs. The EPAs can then use this mechanism to transmit data to RapNet while they are running. This information can include the number of events processed so far, some load information or any kind of desired data. If the Action and/or Representation components are aware of this, they can react appropriately: for instance, the number of processed events can appear as a 'tool tip' when the user moves his mouse over the representation, or it can be drawn as a counter in the representation itself.

3. Interaction with other Components

RapNet only sees the upper layers of our infrastructure: at this level, both the SM and the NM are completely hidden, and even the CL is mostly hidden. The two components which RapNet uses the most are the EPA interface through which Action components start and interact with EPAs, and the Agent Library, which allows localization, creation and activation of EPAs, as well as giving a directory of available EPAs and how to represent them graphically.

a) Use of the Computation Subsystem by RapNet

In RapNet an EPN is represented visually as a DAG, where the boxes (nodes of the graph) represent EPAs and the connections (vertices of the graph) represent computations. Because of this, there is an interaction between RapNet and the computation Subsystem.

We are presenting in this section all the interactions between RapNet and the Computation Subsystem.

(1) Naming and Lightweigthness

When the user double clicks on a computation, he can specify the name of the computation. By default, heavy-weight computations are unnamed and lightweight computations are given a long, unique name automatically.

The user can also specify whether that computation should be heavy-weight or light-weight (description of these types and their trade-offs can be found in the section describing the computation Subsystem).

Inter-SG communication has to go through lightweight computations (in particular, the computation between an EPA and a lightweight viewer has to be lightweight: else, no message gets propagated to the viewer). This is enforced by RapNet: the frontier of a Synchronous Group is always made up of lightweight computations.

(2) Creation of a computation

When the user hits the GO button, RapNet calls a start() method on all the action components of connections. In general, this method creates a computation of the given type and with the given name. However, for filters and maps, this behaves differently: because the output of filters and mappers are subcomputations of their input, the action component will create them accordingly. This puts an additional constraint on RapNet: connections have to be started in a topological order, because one cannot create a sub-computation of a computation that has not yet been created. This is done in the SubEPN start() method, before any of the Action components is started.

b) Use of the EPA interface by RapNet

(1) Insertion

This happens when the user wants to add a new EPA to an EPN. The different steps involved are shown on the UML Interaction Diagram shown on Figure 22

When a new operator is inserted on the design window, RapNet first asks the Agent Library to create the corresponding EPA, then asks the library to create the appropriate Action component and links that action object to the EPA object. Then it creates the corresponding Representation component (once again, the Agent library knows how to create it). At that point, the representation component needs to know how many input and output slots there are, what their RAPIDE type should be, and where to draw them. To do that, it asks the EPA for its arguments, using EPA::getArgs(). This returns an ArgList which contains, among other things, arguments of type InterfaceArg, which describe the input and output slots of a representation, along with their cardinality (the number of connections that can start from or arrive to a given slot), and whether they are mandatory or optional (mandatory slots are represented as a red cross until they meet the cardinality requirements).

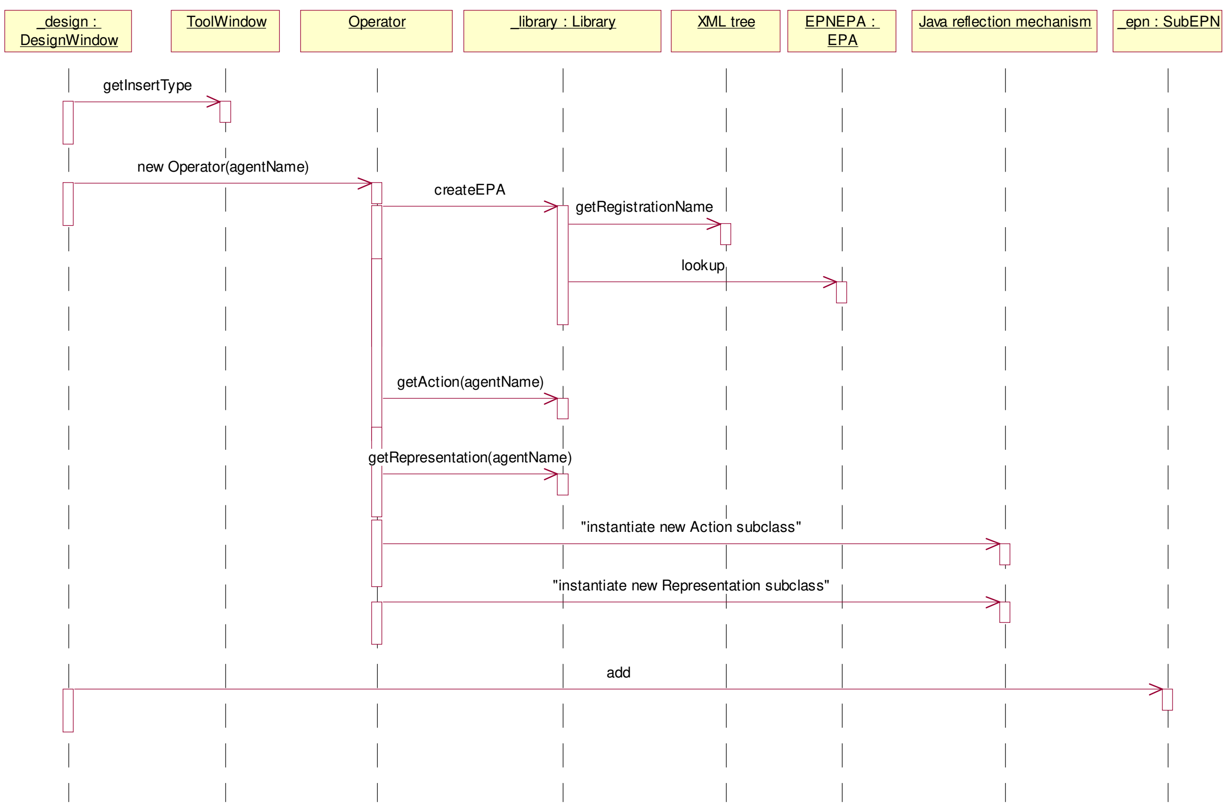


Figure 22: Creating a new EPA

(2) Property Sheet

When the user opens a property sheet on an operator, the action component will query the EPA for its arguments, using the EPA interface method getArgs(). The types of arguments corresponding to slots (of type InterfaceArg) are filtered out, and the others are added automatically to a form.

When the user opens a property sheet while the EPN is running, only the parameters which are returned by EPA.getRuntimeArgs() are shown – these are typically a proper subset of the arguments returned by getArgs(), because some parameters should not be changed while EPAs are running: heavyweight computations maintain all their past history and the pattern language has the potential to ask for past events. If one changes the rule of a filter while it is running – for instance – then the events which are already present in the computations may not be the ones which match the new filter, leading to incorrect results.

(3) Activation

When the user activates the EPN, RapNet first checks that all the EPAs are configured properly. This involves several phases:

- Connection checking: RapNet checks that all the connections are made, with the right cardinality.
- Type checking: Rapide is a type checked language. RapNet enforces type checking and only allows an EPA to be connected to the input of another EPA if it produces a subtype of the type expected by its connected EPA.
- RapNet calls EPA.isConfigured() on all the EPAs in the EPN.

Then RapNet calls start() on all the Connection action objects in the EPN, which create the appropriate computations. Finally, it calls start() on all the other action objects, which in turn call start() on their EPA. An exception to that rule, which is currently hard coded, is for the output of maps and filters, which are not computations but sub computations of their inputs.

c) Use of the Agent Library by RapNet

At startup, RapNet needs to know which operators are available. In addition, GUI features (such as the representation for an operator or the icon to represent it in the tree) may optionally be specified. This requires the functionality of the agent library.

The agent library manages a repository of available linked-in and external EPAs, along with default and specific GUI features for each of them.

The library is first used when RapNet starts up, to get the list of all the available EPAs, which are then displayed in the Tools Window.

Every time an EPA is inserted (i.e. the user selects an element from the Tools Window and places it in the design window), RapNet starts by querying the Library to create the new EPA. In the case of maps and filters, the behavior of the Library is slightly different, because they are not EPAs themselves, but just a parameter to an EPA (respectively the Mapper EPA and the Filter EPA). Therefore the library, which gets passed the element selected in the tree of the Tools Window creates a mapper/filter object and passes it the declaration which was selected.

G. Reference: Available EPAs

The CEP user manual [38] describes the following EPAs: Mappers, Filters, Loggers (sLogger, predLogger, netflow logger, tib sniffer), Viewers / Meters (nTable, nGraph, nTrail, RapView). These are the current agents which are available by default to engineers. Because these agents are fully described elsewhere, this section concentrates on agents which are directly related to RapNet.

1. Distributor

The distributor is a special kind of program which is used by RapNet to distribute agents on different machines. The current implementation is a simple script which takes an executable, its command line parameters and the name of the machine to run the process on, and which executes that process remotely.

This, admittedly, is a temporary hack, because agent distribution should ultimately be transparent to the user, unless he explicitly wants to take over the control of the Synchronous Groups.

2. Legacy Proxy

The Legacy Proxy EPA is used to run out of process agents. The proxy is designed in a way such that the agent does not need to implement the EPA interface; instead, it takes all the configurable arguments for a given agent from the Agent Registry. Because of that, the proxy only allows a subset of the capabilities of standard agents. For instance, because the spawned process does not implement any client-server mechanism, there is no way to change the arguments of a running EPA. Moreover, at least in the current implementation, there is no way for the proxy to know when the agent it started has completed and to notify RapNet accordingly.

The proxy implements the EPA interface, which means that it can be controlled by RapNet, and that, as far as the user can see, everything happens as if the agent was an actual EPA; i.e. the action of the proxy is transparent to the user.

Before the start() method is called on the proxy, the agent does not exit. The proxy knows which arguments the agent will understand, by reading that information from the Agent Registry. Therefore, a getArgs() returns the parameters which the proxy knows to be implemented by the agent – and which the proxy gets through the Agent Library which instantiated it. Likewise, a setArgs() only changes data within the proxy itself.

The interaction between the proxy and its agent only occur when the user hits the GO button in RapNet, i.e. when the start() method is called on the proxy. At that point, the proxy starts the agent by executing it in a new process, and passing all the arguments in the command line. More specifically, the arguments are passed as:

```
-D<argid>=<value>
```

This means that the argid, which is given in the Agent Registry, must correspond to the parameter that the agent expects on the command line.

Another consequence of this is that, with our current implementation, not all the types defined by the EPA layer can be passed this way. Currently, only strings, booleans, integers and reals can be passed through the proxy.

One special agent which may be started by the proxy is the Distributor, described above. A combination of the two pseudo-agents is used to distribute an EPN over several machines, respecting the boundaries of Synchronous Groups.

IV. Bibliography

1. Perrochon, L., E. Jang, and D.C. Luckham. *Enlisting Event Patterns for Cyber Battlefield Awareness*. in *DARPA Information Survivability Conference & Exposition (DISCEX'00)*. 25-27 January 2000. Hilton Head, South Carolina: IEEE Computer Society Press.
2. Pratt, V.R., *Modeling concurrency with partial orders.* Int. J. of Parallel Programming, 1986. **15**(1): p. 33-71.
3. RAPIDE, *Rapide 1.0 Pattern Language Reference Manual*. 1997, Stanford University: Stanford.
4. Luckham, D.C. and B. Frasca, *Complex Event Processing in Distributed Systems*. 1998, Stanford University: Stanford.
5. Santoro, A., *et al. eJava - Extending Java with Causality*. in *10th International Conference on Software Engineering and Knowledge Engineering (SEKE'98)*. Redwood City, CA, USA.
6. Network Associates Inc., *CyberCop Scanner*. 1999.
7. Internet Security Systems, *RealSecure*. 1999.
8. Security Dynamics Inc., *Kane Security Monitor*. 1999.
9. AXENT, *NetProwler*. 1999.
10. Net Flight Recorder Inc., *Net Flight Recorder*. 1999.
11. CISCO, *NetRanger*. 1999.
12. SRI, *Emerald*. 1999.
13. Kahn, C., *et al.*, *A Common Intrusion Detection Framework (CIDF)*. 1999, CIDF Working Group.
14. The Open Group, *Systems Management: Event Management Service*. 1997, The Open Group: Reading, Berkshire, UK.
15. Sheers, K.R., *HP OpenView Event Correlation Services.* Hewlett-Packard Journal, 1996(October).
16. Yemini, S., *et al.*, *High Speed & Robust Event Correlation.* 1996, System Management Arts (SMARTS): White Plains, NY 10601.
17. ITU, *[Z.100] Recommendation Z.100 (03/93) - CCITT specification and description language (SDL).* 1993, International Telecommunication Union: Geneva.
18. ISO/IEC, *Estelle: A formal description technique based on an extended state transition model. Amendment 1*. 1997, International Standards Organization: Geneva.
19. Savor, T. and R.E. Seviora, *Toward Automatic Detection of Software Failures.* IEEE Computer, 1998. **31**(8): p. 68-74.
20. Diaz, M., G. Juanole, and J.-P. Courtiat, *Observer - A Concept for Formal On-Line Validation of Distributed Systems.* IEEE Transactions on Software Engineering, 1994. **20**(12): p. 900-913.
21. Savor, T. and R.E. Seviora. *An Approach to Automatic Detection of Software Failures in Real-Time Systems*. in *IEEE Real-Time Technology & Applications Symposium (RTAS '97)*: IEEE Computer Society Press.
22. Qian, X. and G. Wiederhold, *Incremental Recomputation of Active Relational Expressions.* IEEE Trans. on Knowledge and Data Engineering, 1991. **3**: p. 337--341.
23. Griffin, T., L. Libkin, and H. Trickey, *An Improved Algorithm for the Incremental Recomputation of Active Relational Expressions.* IEEE Transactions on Knowledge and Data Engineering, 1997. **9**(3): p. 508-511.
24. Wolfson, O., *et al. Incremental Evaluation of Rules and its Relationship to Parallelism*. in *SIGMOD'91*

Conference on the Management of Data. Boulder, CO: ACM Press.

25. Lee, J.H.M. and H.F. Leung. *Incremental Querying in the Concurrent CLP language IFD-Constraint Pandora*. in *1996 ACM Symposium on Applied Computing*. February 1996. Philadelphia: ACM Press.

26. Dong, G. and J. Su. *Space-Bounded FOIES*. in *14th ACM Symposium on Principles of Database Systems (PODS'95)*. May 22-25, 1995. San Jose, CA: ACM PRess.

27. Zhuge, Y., *et al. View Maintenance in a Warehousing Environment*. in *Sigmod'95*. June 1995. San Jose, CA.

28. Labio, W.J. and H. Garcia-Molina, *Expiring Data from the Warehouse*. 1997, Stanford University: Stanford.

29. Gluche, D., *et al. Incremental Updates for Materialized OQL Views*. in *The 5th Int'l Conference on Deductive and Object-Oriented Databases (DOOD'97)*. December 1997. Montreux, Switzerland: Springer Verlag.

30. Kuno, H.A. and E.A. Rundensteiner, *The MultiView OODB View System: Design and Implementation.* Theory and Practice of Object Systems (TAPOS), 1997. **2**(3).

31. Mann, W., *The Language of Event Processing Objects*. 1999, Stanford University: Stanford.

32. Box, D., *Essential COM*. 1998: Addison, Wesley, Longman.

33. Perrochon, L., *The Storage Manager API*. 1997, Stanford University: Stanford.

34. Perrochon, L., *Storage Management in the Computation Library*. 1997, Stanford University: Stanford.

35. Gamma, E., *et al.*, *Design Patterns, Elements of Reusable Object-Oriented Software*. 1994: Addison, Wesley, Longman.

36. Lea, D., *Concurrent Programming in Java: Design Principles and Patterns*. 1998: Addison, Wesley, Longman.

37. Fowler, M. and K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*. 1998: Addison, Wesley, Longman.

38. Perrochon, L., *Complex Event Processing User Manual*. 1999, Stanford University: Stanford.

V.Appendix

In this appendix, we add the reference document for the Agent Library and Agent Registry described in section III.D.2.

A. The Agent Registry: Grammar

The agent registry is an XML file, and as such is validated by a grammar. The grammar is written in the file operators.dtd, which lives in the RapNet directory (epn/packages/viewers/rapnet). This file is presented here as a reference only. Its semantics are described in the RapNet reference manual.

```
<!-- DTD for the EPA library -->
<!-- Stephane Kasriel -->
<!-- Copyright 1997-1999 Stanford University, Board of Trustees-->

<!ENTITY % title 'TITLE,ICON?,URL?'>
<!ENTITY % arg.att '
  argid             CDATA #REQUIRED
  argdesc   CDATA          "[no description]"
  required  CDATA          "true"
  default           CDATA          ""
'>
<!ENTITY % interface.att '
  argid             CDATA #REQUIRED
  argdesc   CDATA          "[no description]"
  required  CDATA          "true"
  input             CDATA          #REQUIRED
  min               CDATA          "1"
  max               CDATA          "1"
  declaration       CDATA          #REQUIRED
  file              CDATA          #REQUIRED
'>

<!ELEMENT OPERATORS (%title;,RAPNET?,CATEGORY+)>
<!ELEMENT CATEGORY (%title;,RAPNET?,(CATEGORY|OPERATOR)*)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT ICON (#PCDATA)>
```

```
<!ELEMENT URL (#PCDATA)>
<!ELEMENT OPERATOR (%title;,RAPNET?)>

<!ELEMENT CLASS (#PCDATA)>
<!ELEMENT PARAM (#PCDATA)>

<!ELEMENT INTARG EMPTY>
<!ATTLIST INTARG %arg.att;>
<!ELEMENT BOOLARG EMPTY>
<!ATTLIST BOOLARG %arg.att;>
<!ELEMENT STRINGARG EMPTY>
<!ATTLIST STRINGARG %arg.att;>
<!ELEMENT INTERFACE EMPTY>
<!ATTLIST INTERFACE %interface.att;>
<!ELEMENT EPA ((CLASS,PARAM*, (INTARG|BOOLARG|STRINGARG|INTERFACE)*) |
PATH)>

<!-- The following is used by RapNet -->
<!ELEMENT ACTION (#PCDATA)>
<!ELEMENT ACTIONCONFIGURATION (#PCDATA)>
<!ELEMENT REPRESENTATION (#PCDATA)>
<!ELEMENT REPRESENTATIONCONFIGURATION (#PCDATA)>
<!ELEMENT PATH (#PCDATA)>

<!ELEMENT RAPNET (EPA?,ACTION?,REPRESENTATION?, ACTIONCONFIGURATION?,
REPRESENTATIONCONFIGURATION?)>
```