# Processing Flows of Information: From Data Stream to Complex Event Processing

GIANPAOLO CUGOLA and ALESSANDRO MARGARA
Dip. di Elettronica e Informazione
Politecnico di Milano, Italy

A large number of distributed applications requires continuous and timely processing of information as it flows from the peripheral to the center of the system. Examples are intrusion detection systems, which analyze network traffic in real-time to identify possible attacks; environmental monitoring applications, which process raw data coming from sensor networks to identify critical situations; or applications performing on-line analysis of stock prices to identify trends and forecast future values.

Traditional DBMSs, with the need of storing and indexing data before processing it, have an hard time in fulfilling the requirement of timeliness coming from such domains. Accordingly, during the last decade different research communities, often working in isolation with each other, developed a number of tools, which we collectively call *Information Flow Processing (IFP) Systems*, to support these scenarios. They differ in their system architecture, data model, rule model, and rule language. In this paper we survey these systems to help researchers, often coming from different backgrounds, in understanding how the different approaches may complement each other.

In particular, we propose a general, unifying model to capture the different aspects of an IFP system and use it to provide a complete and precise classification of the different systems and mechanisms proposed so far.

Categories and Subject Descriptors: H.4 [**Information Systems Applications**]: Miscellaneous; I.5 [**Pattern Recognition**]: Miscellaneous; H.2.4 [**Database Management**]: Systems—*Query Processing*; A.1 [**General**]: Introductory and Survey

General Terms: Design, Documentation

Additional Key Words and Phrases: Complex Event Processing, Publish-Subscribe, Stream Processing

## 1. INTRODUCTION

An increasing number of distributed applications requires processing of continuously flowing data from geographically distributed sources with unpredictable rate to obtain timely responses to complex queries. Examples of such applications come from the most disparate fields: from wireless sensor networks to financial tickers, from traffic management to click-stream inspection. In the following we collectively refer to these applications as building the *Information Flow Processing (IFP) domain*, while we call *Information Flow Processing (IFP) engine* a tool capable of timely processing large amount of information as it flows from the peripheral to the center of the system.

The concepts of "timeliness" and "flow processing" are crucial to justify the need of a new class of systems. Indeed, traditional DBMSs, (i) require data to be (persistently) stored and indexed before it could be processed, and (ii) they process data only when explicitly asked by the users, i.e., asynchronously with respect to its arrival. Both aspects contrast with the requirements of IFP applications. As an example, consider the need of detecting possible fire in a building by using

temperature and smoke sensors. On one hand, the fire alert has to be notified as soon as the relevant data becomes available. On the other hand, there is no need to store sensor readings if they are not relevant for fire detection, while those relevant can be discarded as soon as the fire is detected, since all the information they carried, like the area where the fire occurred, if important for the application should be already part of the fire alert.

These requirements led to the development of a number of systems specifically designed to process information as a flow (or a set of flows) according to a set of pre-deployed *processing rules*. Despite their common goal, these systems differ in a wide range of aspects, including architectures, data models, rule languages, and processing mechanisms. In part, this is also the fact that they were the result of the research efforts of different communities, each one bringing its own view of the problem and its background for the definition of a solution, not to mention its own vocabulary [Bass 2007]. After several years of research and development we can say that two models emerged and are today competing: the *data stream processing* model [Babcock et al. 2002] and the *complex event processing* model [Luckham 2001].

As suggested by its own name, the data stream processing model describes the IFP problem as that of processing *streams of data* coming from different sources to produce streams of new data as an output, and see this problem as an evolution of traditional data processing as supported by DBMSs. Accordingly, *Data Stream Management Systems (DSMSs)* were developed as special DBMSs. While traditional DBMSs are designed to work on persistent data, where updates are relatively infrequent, DSMSs are specialized in dealing with transient data that is continuously updated. Similarly, while DBMSs run queries just once to return a complete answer, DSMSs execute *standing queries*, which run continuously and provide updated answers as new data arrives. Despite these differences, DSMSs resemble DBMSs, especially in the way they process incoming data through a sequence of transformations based on common SQL operators like selections, aggregates, joins, and in general all the operators defined by the relational algebra.

On the other side, the complex event processing model sees flowing information items as *notifications* of *events* happened in the external world, which have to be filtered and combined to deduce what is happening in terms of *higher-level* events. Accordingly, the focus of this model is on detecting occurrences of particular *patterns* of (low-level) events, which represent the higher-level events whose occurrence has to be notified to the interested parties. While the contributions to this model come from different communities, including those of distributed information systems, business process automation, control systems, network monitoring, sensor networks, and middleware in general; we set the main root of this model in the publish-subscribe domain [Eugster et al. 2003]. Indeed, while traditional publish-subscribe systems consider each event separately from the others, filtering them (based on their topic or content) to decide if they are relevant for subscribers, *Complex Event Processing (CEP)* systems extend this functionality, by increasing the expressive power of the subscribing language to consider complex patterns of events that involve the occurrence of multiple, related events.

In this paper we claim that neither of the two models above may entirely satisfy

the needs of a full fledged IFP engine, which requires features coming from both worlds. Accordingly, we draw a general framework to compare the results coming from the different communities, and use it to provide an extensive review of the state of the art in the area, with the overall goal of reducing the effort to merge the results produced so far.

In particular, in Section 2 we describe the IFP domain in more details, provide an initial description of the different technologies that have been developed to support it, and explain the need of combining the best of different worlds to fully support IFP applications. In Section 3 we describe a framework to model and analyze the different aspects that are relevant for an IFP engine, from its functional architecture, to its data and processing models, to the language it provides to express how information has to be processed, to its run-time architecture. We use this framework in Section 4 to describe and compare the state of the art in the field, discussing the results of such classification in Section 5. Finally, in Section 6 we review related works, while in Section 7 we provide some conclusive remarks and a list of open issues.

## 2. BACKGROUND AND MOTIVATION

In this section we provide a precise description of the application domain that we call Information Flow Processing (and motivate why we introduce a new name); we describe the different technologies that have been proposed to support such a domain; and we conclude by motivating our work.

### 2.1 The IFP Domain

With the term *Information Flow Processing (IFP)* we refer to an application domain in which users need to collect information produced by multiple, distributed sources, to process it in a timely way, in order to extract new knowledge as soon as the relevant information is collected.

Examples of IFP applications come from the most disparate fields. In environmental monitoring, users needs to process data coming from sensors deployed on field to acquire information about the observed world, detect anomalies, or predict disasters as soon as possible [Broda et al. 2009; EventZero 2010]. Similarly, several financial applications require a constant analysis of stocks to identify trends [Demers et al. 2006], while fraud detection requires continuous streams of credit card transactions to be observed and inspected to prevent frauds [Schultz-Moeller et al. 2009]. To promptly detect and possibly anticipate attacks to a corporate network, intrusion detection systems needs to analyze network traffic in real-time, generating alerts when something unexpected happens [Debar and Wespi 2001]. RFID-based inventory management needs to perform a continuous analysis of RFID readings to track valid paths of shipments and to capture irregularities [Wang and Liu 2005], while manufacturing control systems often require anomalies to be detected and alerted by looking at the information that describe how the system is behaving [Lin and Zhou 1999; Park et al. 2002].

What all these examples have in common is the need of processing information as it flows from the peripheral to the center of the system without requiring, at least in principle, this information to be persistently stored. Once the flowing information has been processed, producing new information, the original one can be discarded,

while the new one leaves the system as its output[1].

The broad spectrum of applications domains that require "information flow processing" in the sense above, explains why several research communities focused their attention to the IFP domain, each bringing its own expertise and point of view, but also its own vocabulary. The result was a typical tower of Babel syndrome, with misunderstandings among researchers that negatively impacted the spirit of collaboration required to advance the state of the art [Etzion 2007]. This explains why in this paper we decided to adopt our own vocabulary, moving away from terms like "event", "data", "stream", or "cloud", to use more general (and unbiased) terms like "information" and "flow". Notice that in doing so we neither have the ambition to propose a new standard terminology nor we want to contribute "raising the tower". Our only desire is to help the reader looking at the field with her's mind clear from any bias toward the meaning of the various terms used.

Pushed by this goal, we model an IFP system as in Figure 1. At the center we place the *IFP engine*, a tool that operates according to a set of *processing rules*, which describe how incoming *flows of information* have to be processed to timely produce new flows of information as outputs. We call *information sources* the entities that create the flows of information entering the IFP engine. Examples of such information are notifications about events happening in the observed world or, more in general, any kind of data that reflects some knowledge generated by the source. *Information items* that are part of the same flow are neither necessarily ordered nor of the same kind. They are just pieces of information, with their semantics and relationships (including total or partial ordering, in some cases). The IFP engine takes such information items as input and process them, as soon as they are available, according to a set of processing rules. They specify how to filter, combine, and aggregate different flows of information, item by item, to generate new flows, which represent the output of the engine. We call *information sinks* the recipients of such output, while we call *rule managers* those entities that are in charge of adding new processing rules or removing them. In some situations the same entity may play different roles. For example, it is not uncommon for an information sink to submit the rules that produce the information it needs.

While the definition above is general enough to capture every IFP system we are interested in classifying, it encompasses some key points that characterize the domain and come directly from the requirements of the applications we mentioned above. First of all the need to perform real-time or quasi real-time processing of incoming information to produce new knowledge (i.e., outgoing information). Second, the need for an expressive language to describe how incoming information have to be processed, with the ability of specifying complex relationships among the information items that flow into the engine and are relevant to sinks. Third, the need for scalability to effectively cope with situations in which a very large number of geographically distributed information sources and sinks have to cooperate.

[1]Complex applications may also need to store data, e.g., for historical analysis, but in general this is not the goal of the IFP subsystem.

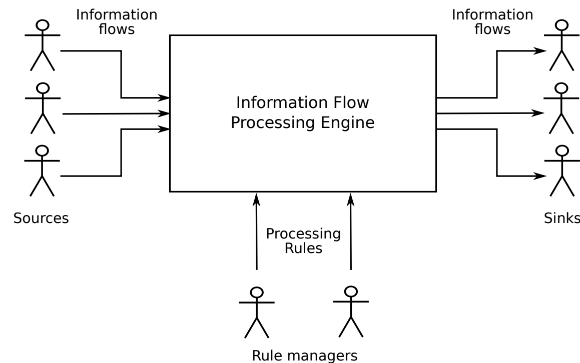



Fig. 1. The high-level view of an IFP system

### 2.2 IFP: One Name Different Technologies

As we mentioned, IFP has attracted the attention of researchers coming from different fields. The first contributions came from the database community in the form of *active database systems* [McCarthy and Dayal 1989], which were introduced to allow actions to automatically execute when given conditions arise. *Data Stream Management Systems (DSMSs)* [Babcock et al. 2002] pushed this idea further, to perform query processing in the presence of continuous data streams.

In the same years that saw the development of DSMSs, researchers with different backgrounds identified the need of developing systems capable of processing not generic data but event notifications, coming from different sources, to identify interesting situations [Luckham 2001]. These systems are usually known as *Complex Event Processing (CEP) Systems*.

2.2.1 *Active Database Systems.* Traditional DBMSs are completely passive, as they present data only when explicitly asked by users or applications: this form of interaction is usually called *Human-Active, Database-Passive (HADP)*. Using this interactional model it is not possible to ask the system to send notifications when predefined conditions are detected. *Active database systems* have been developed to overcome this limitation: they can be seen as an extension of classical DBMSs where the reactive behavior can be moved, totally or in part, from the application layer into the database.

There are several tools classified as active database systems, with different software architectures, functionalities, and oriented toward different application domains; still, it is possible to classify them on the basis of their *knowledge model* and *execution model* [Paton and Díaz 1999]. The former describes the kind of active rules that can be expressed, while the latter defines the system's runtime behavior.

The knowledge model usually considers active rules as composed by three parts:

*an event* defines which sources can be considered as event generators: some systems only consider internal operators, like a tuple insertion or update,

while others also allow external events, like those raised by clocks or external sensors;

*a condition* specifies when an event must be taken into account; for example we can be interested in some data only if it exceeds a predefined limit;

*an action* identifies the set of tasks that can be executed as a response to an event detection: some systems only allow the modification of the internal database, while others allow the application to be notified about the identified situation.

To make this structure explicit, active rules are usually called *Event-Condition-Action (ECA)* rules.

The execution model defines how rules are processed at runtime. Here, five phases have been identified:

*signaling* refers to the detection of an event;

*triggering* refers to the association of an event with the set of rules defined for it;

*evaluation* refers to the evaluation of the conditional part for each triggered rule;

*scheduling* refers to the definition of an execution order between selected rules;

*execution* refers to the execution of all the actions associated to selected rules.

Active database systems are used in three kinds of applications: as a database extension, in closed database applications, and in open database applications. As a database extension, active rules refer only to the internal state of the database, e.g., to implement an automatic reaction to constraint violations. In closed database applications active rules can support the semantics of the application level but external sources of events are not allowed. Finally, in open database applications events can come both from the inside of the database and from external sources, for example monitoring sensors. This is the domain that is closer to IFP.

2.2.2 *Data Stream Management Systems.* Even when taking into account external event sources, active database systems are built, like traditional DBMSs, around a persistent storage where all the relevant data is kept, and whose updates are relatively infrequent. This approach negatively impacts their performance when the number of rules expressed increases over a certain limit or when the rate of arrival of events (internal or external ones) is too high. For most of the applications that we classified as IFP such limitations are excessively penalizing.

Motivated by this limitation, the database community developed a new class of systems oriented toward processing large *streams of data* in a timely way: the *Data Stream Management Systems (DSMSs)*. Data streams differ from conventional databases in several ways:

—streams are usually unbounded;

—no assumption can be made on data arrival order;

—size and time constraints make it difficult to store and process data stream elements after their arrival; one-time processing is the typical mechanism used to deal with streams.

Users of a DSMS install *standing* (or *continuous*) *queries*, i.e., queries that are deployed once and continue to produce results until removed. Standing queries

can be executed periodically or continuously as new stream elements arrive; they introduce a new interactional model w.r.t. traditional DBMSs: users do not have to explicitly ask for updated information, rather the system actively notifies them according to installed queries. This form of interaction is usually called *Database-Active, Human-Passive (DAHP)* [Abadi et al. 2003].

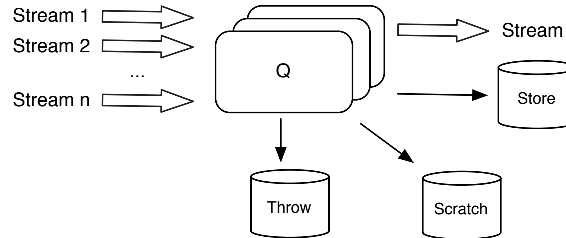



Fig. 2. The typical model of a DSMS

Several implementations were proposed for DSMSs. They differ for the semantics they associate to standing queries. In particular, an answer to a query can be seen as an append only output stream, or as an entry in a storage that is continuously modified as new elements flow inside the processing stream. Also, an answer can be either exact, if the system is supposed to have enough memory to store all needed elements of input streams' history, or approximate, if computed on a portion of the required history [Tatbul et al. 2003; Babcock et al. 2004].

In Figure 2 we report a general model for DSMSs, directly taken from [Babu and Widom 2001]. The purpose of this model is to make several architectural choices and their consequences explicit. A DSMS is modeled as a set of standing queries $Q$, one or more *input streams* and four possible outputs:

the *Stream* is formed by all the elements of the answer that are produced once and never changed;

the *Store* is filled with parts of the answer that may be changed or removed at a certain point in the future. The *Stream* and the *Store* together define the current answer to queries $Q$;

the *Scratch* represents the working memory of the system, i.e. a repository where it is possible to store data that is not part of the answer, but that may be useful to compute the answer;

the *Throw* is a sort of recycle bin, used to throw away unneeded tuples.

To the best of our knowledge, the model described above is the most complete to define the behavior of DSMSs. It explicitly shows how DSMSs alone cannot entirely cover the needs for IFP: being an extension of database systems, DSMSs focus on producing query answers, which are continuously updated to adapt to the constantly changing contents of their input data. Detection and notification of complex patterns of elements involving sequences and ordering relations are usually out of the scope of these systems.

2.2.3 *Complex Event Processing Systems.* The limitation above originates from the same nature of DSMSs, which are generic systems that leave to their clients the responsibility of associating a semantics to the data being processed. *Complex Event Processing (CEP) Systems* adopt the opposite approach. As shown in Figure 3, they associate a well precise semantics to the information items being processed: they are *notifications* of *events* happened in the external world and *observed* by sources. The CEP engine is responsible for filtering and combining such notifications to deduce what is happening in terms of *higher-level* events (sometime also called *composite events* or *situations*) to be notified to sinks, which act as *event consumers*.

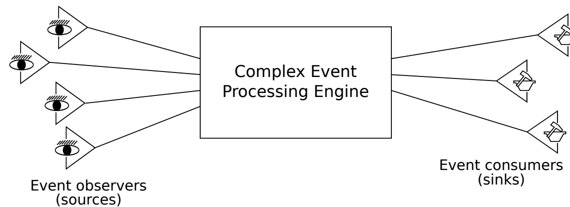



Fig. 3. The high-level view of a CEP system

Historically, the first event processing engines [Rosenblum and Wolf 1997] focused on filtering incoming notifications to extract only the relevant ones, thus supporting an interaction style known as *publish-subscribe*. This is a message oriented interaction paradigm based on an indirect addressing style. Users express their interest in receiving some information by *subscribing* to specific classes of events, while information sources *publish* events without directly addressing the receiving parties; they are dynamically chosen by the publish-subscribe engine based on the received subscriptions.

Conventional publish-subscribe comes in two flavors: topic and content-based [Eugster et al. 2003]. *Topic-based* systems allow users to subscribe only to predefined topics. Publishers choose the topic each event belongs to before publishing. *Content-based systems* allow subscribers to use complex *event filters* to specify the events they want to receive based on their content. Several languages have been used to represent event content and subscription filters: from simple attribute/value pairs [Carzaniga and Wolf 2003] to complex XML schema [Altinel and Franklin 2000; Ashayer et al. 2002].

Whatever language is used, subscriptions may refer to single events only [Aguilera et al. 1999] and cannot take into account the history of already received events or relationships between events. To this end, CEP systems can be seen as an extension to traditional publish-subscribe, which allow subscribers to express their interest in *composite events*. As their name suggests, these are events whose occurrence depends on the occurrence of other events, like the fact that a fire is detected when three different sensors, located in an area smaller than $100\,m^2$, report a temperature greater than $60^o$C, within $10\,sec.$ one from the other.

Being designed to detect the occurrence of composite events, CEP systems put great emphasis on the issue that represent the main limitation of most DSMSs: the ability of detecting complex patterns of incoming items, involving sequencing and

ordering relationships. Indeed, the ability of specifying composite events through *event patterns* that match incoming event notifications on the basis of their content and on some ordering relationships that join them, is at the basis of the CEP model.

Apart from this attentions toward event patterns, the focus on information representing occurrences of events had also an impact on the architecture of CEP engines. In fact, these tools often have to interact with a large number of widely distributed and heterogeneous information sources and sinks, which observe the external world and operate on it. This is the case for environmental monitoring, but also business process automation, control systems, and in general most of the scenarios for CEP. This suggested, at least in the most advanced proposals, the adoption of a distributed architecture for the CEP engine itself, organized (see Figure 4) as a set of *event brokers* (or *event processing agents* [Etzion and Niblett 2010]) connected in an *overlay network* (the *event processing network*), which implement specialized routing and forwarding functions, with scalability as their main concern. For the same reason, CEP systems research focused on optimizing parameters, like bandwidth utilization and end-to-end latency, which were usually ignored in DSMSs.

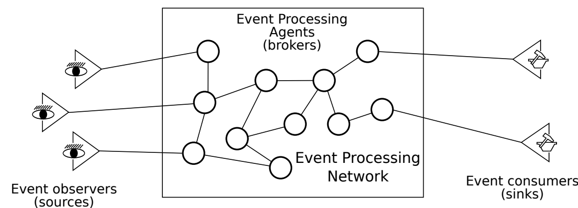



Fig. 4. CEP as a distributed service

This peculiarity allows classifying CEP systems on the basis of the architecture of the CEP engine (centralized, hierarchical, acyclic, peer-to-peer) [Carzaniga et al. 2001; Eugster et al. 2003], on the forwarding schemes adopted by brokers [Carzaniga and Wolf 2002], and on the way processing of event patterns is distributed among brokers [Amini et al. 2006; Pietzuch et al. 2006].

## 2.3 Our Motivation

At the beginning of this section we introduced IFP as a new application domain demanding timely processing of flows of information according to their content and to the relationships among them. IFP engines have to filter, combine, and aggregate a huge amount of information according to a given set of processing rules. Moreover, they usually need to interact with a great number of heterogeneous information sources and sinks, possibly dispersed over a wide geographical area. Users of such systems can be high-level applications, other processing engines, or end-users, possibly in mobile scenarios. For these reasons expressiveness, scalability, and flexibility are key requirements in the IFP domain.

We have seen how IFP has been addressed by different communities, which come out with two classes of systems: DSMSs and CEP engines. The former mainly focus on flowing data and data transformations. Only few works allow the easy capture of sequences of data involving complex ordering relationships, not to mention taking

into account the possibility to perform filtering, correlation, and aggregation of data directly in-network, as streams flow from sources to sinks. Finally, to the best of our knowledge, network dynamics and heterogeneity of processing devices have never been studied in depth.

On the other hand, CEP engines, either those developed as extensions of publish-subscribe middleware or those developed as totally new systems, define a quite different model; their focus is on processing event notifications with their ordering relationships to capture complex event patterns; and on the communication aspects involved in event processing, such that the ability to adapt to different network topologies, as well as to various processing devices, together with the ability of processing information directly in-network to distribute the load and reduce communication cost are their main concern.

As we stated in Section 1, it is our claim that IFP needs to consider all these aspects: effective data processing, including the ability to capture complex ordering relationships among data, as well as efficient event delivery, including the ability to process data in a strongly distributed fashion. To pursue this goal, researchers are required to keep the best coming from the different communities involved. They need to examine the solutions provided so far and to understand how the different approaches complement each other. As we will see in Section 4, some of the most advanced proposals in both areas are going in this direction. This work is a general step in the same direction, as it provides an extensive framework to model an IFP system and use it to classify and compare existing systems: from active databases, to DSMSs, to CEP engines.

## 3. A MODELLING FRAMEWORK FOR IFP SYSTEMS

In this section we define a framework to compare the different proposals in the area of IFP. It includes several models that focus on the different aspects relevant for an IFP system.

### 3.1 Functional Model

Starting from the high-level description of Figure 1, we now present an abstract architecture that describes the main functional components of an IFP engine. This architecture brings two contributions: (i) it allows a precise description of the functionalities offered by an IFP engine; (ii) it can be used to describe the differences among the existing IFP engines, providing a versatile tool for their comparison.

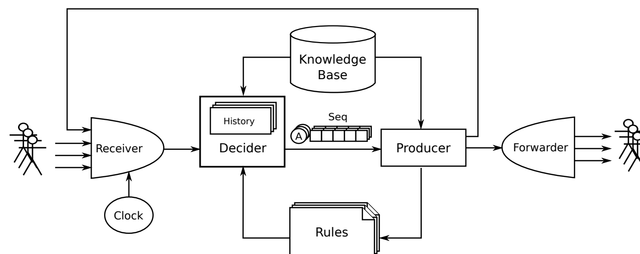



Fig. 5. The functional architecture of an IFP system

As we said before, an IFP engine takes flows of information coming from different sources as its input, processes them, and produces other flows of information directed toward a set of sinks. Processing rules describe how to filter, combine, and aggregate incoming information to produce outgoing information. This general behavior can be decomposed in a set of elementary actions performed by the different components shown in Figure 5.

Incoming information flows enter the *Receiver*, whose task is to manage the channels connecting the sources with the IFP engine. It implements the transport protocol adopted by the engine to move information around the network. It also acts as a demultiplexer, receiving incoming items from multiple sources and sending them, one by one, to the next component in the IFP architecture. As shown in figure, the *Receiver* is also connected to the *Clock*: the element of our architecture that is in charge of periodically creating special information items that hold the current time. Its role is to model those engines that allow periodic (as opposed to purely reactive) processing of their inputs, as such, not all engines currently available implement it.

After traversing the *Receiver*, the information items coming from the external sources or generated by the *Clock* enter the main processing pipe, where they are elaborated according to the processing rules currently stored into the *Rules* store.

From a logical point of view we find important to consider rules as composed by two parts: $C \rightarrow A$, where $C$ is the *condition part*, while $A$ is the *action part*. The condition part specifies the constraints that have to be satisfied by the information items entering the IFP engine to trigger the rule, while the action part specifies what to do when the rule is triggered.

This distinction allows us to split information processing into two phases: a *detection phase* and a *production phase*. The former is realized by the *Decider*, which gets incoming information from the *Receiver*, item by item, and looks at the condition part of rules to find those enabled. The action part of each triggered rule is then passed to the *Producer* for execution.

Notice that the *Decider* may need to accumulate information items into a local storage until the constraints of a rule are entirely satisfied. As an example, consider a rule stating that a fire alarm has to be generated (action part) when both smoke and high temperature are detected in the same area (condition part). When information about smoke reaches the *Decider*, the rule cannot fire, yet, but the *Decider* has to record this information to trigger the rule when high temperature is detected. We model the presence of such "memory" through the *History* component.

The detection phase ends by passing to the *Producer* an action $A$ and a sequence of information items *Seq* for each triggered rule, i.e., those items (accumulated in the *History* by the *Decider*) that actually triggered the rule. The action $A$ describes how the information items in *Seq* have to be manipulated to produce the expected results of the rule. The maximum allowed length of the sequence *Seq* is an important aspect to characterize the expressiveness of the IFP engine. There are engines where *Seq* is composed by a *single* item; in others the maximum length of *Seq* can be determined by looking at the set of currently deployed rules (we say that *Seq* is *bounded*); finally, *Seq* is *unbounded* when its maximum length depends on the information actually entering the engine.

The *Knowledge Base* represents, from the perspective of the IFP engine, a read-

only memory that contains information used during the detection and production phases. This component is not present in all IFP engines; usually, it is part of those systems developed by the database research community, which allow accessing persistent storage (typically in the form of database tables) during information processing[2].

Finally, the *Forwarder* is the component in charge of delivering the information items generated by processing rules to the expected sinks. Like the *Receiver*, it implements the protocol to transport information along the network up to the sinks.

In summary, an IFP engine operates as follows: each time a new item (including those periodically produced by the *Clock*) enters the engine through the *Receiver*, a *detection-production cycle* is performed. Such a cycle first (detection phase) evaluates all the rules currently present in the *Rules* store to find those whose condition part is true. Together with the newly arrived information, this first phase may also use the information present in the *Knowledge Base*. At the end of this phase we have a set of rules that have to be executed, each coupled with a sequence of information items: those that triggered the rule and that were accumulated by the *Decider* in the *History*. The *Producer* takes this information and executes each triggered rule (i.e., its action part). In executing rules, the *Producer* may combine the items that triggered the rule (as received by the *Decider*) together with the information present in the *Knowledge Base*, to produce new information items. Usually, these new items are sent to sinks (through the *Forwarder*), but in some engines they can also be sent internally, to be processed again (*recursive processing*). This allows information to be processed in steps, by creating new knowledge at each step, e.g., the composite event “fire alert”, generated when smoke and high temperature are detected, can be further processed to notify a “chemical alert” if the area of fire (included in the “fire alert” event) is close to a chemical deposit. Finally, some engines allow actions to change the rule set by adding new rules or removing them.

### 3.2 Processing Model

In our functional model the *Decider* and the *Producer* represent the core of the IFP engine. The *Decider* processes the incoming flow, accumulating relevant information items according to the set of deployed rules, and sending them to the *Producer* when the condition part of a rule is satisfied. The *Producer* uses the data received from the *Decider* to produce new information. The output produced in a detection-production cycle is fully determined by the last information item entering the *Decider*, the set of deployed rules, the information stored in the *History* and in the *Knowledge Base*, and the specific *selection*, *consumption*, and *load shedding* policies adopted by the system.

**Selection policy.** In presence of situations in which a single rule $R$ may fire more than once, picking different items from the *History*, the selection policy [Zimmer 1999] specifies if $R$ has to fire once or more times, and which items are actually

[2]Issues about initialization and management of the *Knowledge Base* are outside the scope of an IFP engine, which simply uses it as a pre-existing repository of stable information.

selected and sent to the *Producer*.

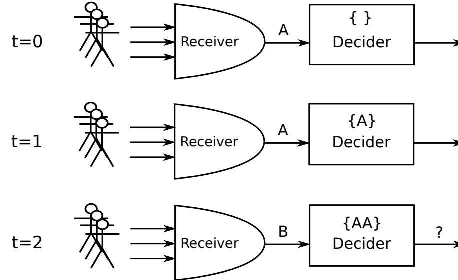



Fig. 6. Selection policy: an example

As an example, consider the situation in Figure 6, which shows the information items (modeled as capital letters) sent by the *Receiver* to the *Decider* at different times, together with the information stored by the *Decider* into the *History*. Suppose that a single rule is present in the system, whose condition part (the action part is not relevant here) is $A \wedge B$, meaning that something has to be done each time both $A$ and $B$ are detected, in any order. At $t = 0$ information $A$ exits the *Receiver* starting a detection-production cycle. At this time the pattern $A \wedge B$ is not detected, but the *Decider* has to remember that $A$ has been received since this can be relevant to recognize the pattern in the future. At $t = 1$ a new instance of information $A$ exits the *Receiver*. Again, the pattern cannot be detected, but the *Decider* stores the new $A$ into the *History*. Things change at $t = 2$, when information $B$ exits the *Receiver* triggering a new detection-production cycle. This time not only the condition part of the rule is satisfied, but it is satisfied by two possible sets of items: one includes item $B$ with the first $A$ received, the other includes the same $B$ with the second $A$ received.

Systems that adopt the *multiple* selection policy let rules fire more than once at each detection-production cycle. This means that in the situation above the *Decider* would send two sequences of items to the *Producer*, one including item $B$ followed by the $A$ received at $t = 0$, the other including item $B$ followed by the $A$ received at $t = 1$.

Conversely, systems that adopt a *single* selection policy allow rules to fire at most once at each detection-production cycle. In the same situation above the *Decider* of such systems would choose one among the two $A$s received, sending a single sequence to the *Producer*: the sequence composed of item $B$ followed by the chosen $A$. It is worth noting that the single selection policy actually represents a whole family of policies, depending on the items actually chosen among all possible ones. In our example, the *Decider* could select the first or the second $A$.

Finally, some systems offer a *programmable* policy, by including special language constructs that enable users to decide, often rule by rule, if they have to fire once or more than once at each detection-production cycle, and in the former case, which elements have to be actually selected among the different possible combinations.

**Consumption policy.** Related with the selection policy is the consumption policy [Zimmer 1999], which specifies whether or not an information item selected in a given detection-production cycle can be considered again in future processing cycles.

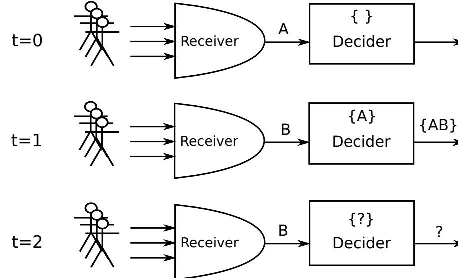



Fig. 7. Consumption policy: an example

As an example, consider the situation in Figure 7, where we assume again that the only rule deployed into the system has a condition part $A \wedge B$. At time $t = 0$, an instance of information $A$ enters the *Decider*; at time $t = 1$, the arrival of $B$ satisfies the condition part of the rule, so item $A$ and item $B$ are sent to the *Producer*. At time $t = 2$, a new instance of $B$ enters the system. In such a situation, the consumption policy determines if pre-existing items $A$ and $B$ have still to be taken into consideration to decide if rule $A \wedge B$ is satisfied. The systems that adopt the *zero* consumption policy do not invalidate used information items, which can trigger the same rule more than once. Conversely, the systems that adopt the *selected* consumption policy "consume" all the items once they have been selected by the *Decider*. This means that an information item can be used at most once for each rule.

The zero consumption policy is the most widely adopted in DSMSs. In fact, DSMSs' usually introduce special language constructs (windows) to limit the portion of an input stream from which elements can be selected (i.e. the valid portion of *History*); however, if an item remains in a valid window for different detection-production cycles, it can trigger the same rule more than once. Conversely, CEP systems may adopt either the zero or the selected policy depending from the target application domain and from the processing algorithm used.

As it happens for the selection policy, some systems offer a *programmable* selection policy, by allowing users to explicitly state, rule by rule, which selected items should be consumed after selection and which have to remain valid for future use.

**Load shedding.** Load shedding is a technique adopted by some IFP systems to deal with bursty inputs. It can be described as an automatic drop of information items when the input rate becomes too high for the processing capabilities of the engine [Tatbul et al. 2003]. In our functional model we let the *Decider* be responsible for load shedding, since some systems allow users to customize its behavior (i.e., deciding when, how, and what to shed) directly within the rules. Clearly,

those systems that adopt a fixed and pre-determined load shedding policy could, in principle, implement it into the *Receiver*.

### 3.3 Deployment Model

Several IFP applications include a large number of sources and sinks, dispersed over a possibly wide geographical area, producing and consuming a large amount of information that the IFP engine has to process in a timely manner. Hence, an important aspect to consider is the *deployment architecture* of the engine, i.e., how the components that actually implement the functional architecture presented in Figure 5 can be distributed over multiple nodes to increase the system's scalability.

In particular, we distinguish between *centralized* vs. *distributed* IFP engines, further differentiating the latter into *clustered* vs. *networked*.

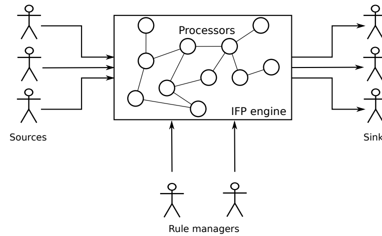



Fig. 8. The deployment architecture of an IFP engine

With reference to Figure 8, we define as *centralized* an IFP engine in which the actual processing of information flows coming from sources is realized by a single node in the network, in a pure client-server architecture. The IFP engine acts as the server, while sources, sinks, and rule managers acts as clients.

To provide better scalability, a *distributed* IFP engine processes information flows through a set of *processors*, each running on a different node of a computer network, which collaborate to perform the actual processing of information. The nature of this network of processors allows to further classify distributed IFP engines. In a *clustered* engine scalability is pursued by sharing the effort of processing incoming flows of information among a cluster of well connected machines, usually part of the same local area network. As a result, we may define a clustered IFP engine as one in which the links connecting processors among themselves perform much better than the links connecting sources and sinks with the cluster itself, while the processors are under the same administrative domain. Several advanced DSMSs are clustered.

Conversely, a *networked* IFP engine focuses on minimizing network usage by dispersing processors over a wide area network, with the goal of processing information as close as possible to the sources. As a result, in a networked IFP engine the links among processors are similar to the links connecting sources and sinks to the engine itself (usually, the closer processor in the network is chosen to act as an entry point in the IFP engine), while processors are widely distributed and usually run under different administrative domains. Some CEP systems adopt this architecture.

In summary, in their seek for better scalability, clustered and networked engines focus on different aspects: the former on increasing the available processing power by sharing the workload among a set of well connected machines, the latter on minimizing bandwidth usage by processing information as close as possible to the sources.

### 3.4 Interaction Model

With the term *interaction model* we refer to the characteristics of the interaction among the main components that builds an IFP application. In particular, with reference to Figure 1, we distinguish three different sub-models: the *observation model* refers to the interaction between information sources and the IFP engine; the *notification model* refers to the interaction between the IFP engine and the sinks; finally, for those systems that provide a distributed implementation of the IFP engine, the *forwarding model* defines the characteristics of the interaction among processors.

In all cases, we distinguish between a *push* and a *pull* style of interaction [Rosenblum and Wolf 1997; Cugola et al. 2001]. In particular, we have a pull observation model if the IFP engine is the initiator of the interaction to bring information from sources to the engine itself, otherwise we have a push model. Similarly, a pull notification model leave to the sinks the responsibility of pulling from the engine those informaition relevant for them, otherwise we have a push model. Finally, the forwarding model is pull if each processor is reponsible for pulling information from the processor upstream in the chain from sources to sinks, we have a push model otherwise. While the push style is the most common for either the observation, notification, and particularly forwarding models; a few systems exist, which prefer a pull model or supports both. We survey them in Section 4.

### 3.5 Data Model

The various IFP systems available today differ in how they represent single information items flowing from sources to sinks, and in how they organize them in flows. We collectively refer to both issues with the term *Data model*.

As we mentioned in Section 2, one of the aspects that distinguishes DSMSs from CEP systems is that the former manipulate generic data items, while the latter manipulates event notifications. We refer to this aspect as the *nature of items*. It represent a key issue for an IFP system as it impacts several other aspects, like the rule language. We come back to this issue later.

A further distinction among IFP systems, orthogonal with respect to the nature of the information items they process, is the way such information items are actually represented, i.e., their *format*. The main formats adopted by currently available IFP systems are *tuples*, typed or untyped ones; *records*, organized as sets of key-value pairs; *objects*, as in object-oriented languages or databases; or *XML* documents.

A final aspect regarding information items is the ability of an IFP system to deal with *uncertainty*, modeling and representing it explicitly when required [Wasserkrug et al. 2008; Liu and Jacobsen 2004]. In many IFP scenarios, in fact, information received from sources has an associated degree of uncertainty. As an example, consider sources that provide only rounded data [Wasserkrug et al. 2008].

Besides single information items, we classify IFP systems based on the nature of

information flows, distinguishing between those systems whose engine process *homogeneous* information flows and those that may also manage *heterogeneous* flows.

In the first case, all the information items in the same flow have the same format, e.g., if the engine organizes information items as tuples, all the items belonging to the same flow must have the same number of fields. Most DSMSs belong to this class. They view information flows as typed data streams, which they manage as transient, unbounded database tables, to be filtered and transformed while they get filled by data flowing into the system.

The opposite case is that of engines that allow different types of information items in the same flow, e.g., one record with four fields, followed by one with seven, and so on. Most CEP engines belong to this class. Information flows are viewed as heterogeneous channels connecting sources with the CEP engine. Each channel may transport items (i.e., events) of different types as soon as they come from the same source.

### 3.6 Time Model

With the term *time model* we refer to the relationship between the information items flowing into the IFP engine and the passing of time. More precisely, we refer to the ability of the IFP system of associating some kind of *happened-before* relationship [Lamport 1978] to information items.

As we mentioned in Section 2, this issue is very relevant as a whole class of IFP systems, namely CEP engines, are characterized by the fact of processing not generic data but event notifications, a special kind of information strongly related with time. Moreover, the availability of a native concept of ordering among items has often an impact on the operators offered by the rule language. As an example, without a total ordering among items it is not possible to define sequences and all the operators meant to operate on sequences.

Focusing on this aspect we distinguish four cases. First of all there are systems that do not consider this issue as prominent. Several DSMSs, in fact, do not associate any special meaning to time. Data flows into the engine within streams but timestamps (when present) are used mainly to order items at the frontier of the engine (i.e., within the *receiver*), and they are lost during processing. The only language construct based on time, when present, is the windowing operator (see Section 3.8), which allows to select the set of items received in a given time-span. After this step the ordering is lost and timestamps are not available for further processing. In particular, the ordering (and timestamps) of the output stream are conceptually separate from the ordering (and timestamps) of the input streams. In this case we say that the time model is *stream-only*.

At the opposite of the spectrum are those systems, like most CEP engines, which associate each item with a timestamp that represent an absolute time, usually interpreted as the time of occurrence of the related event. This way, timestamps define a total ordering among items. In such systems timestamps are fully exposed to the rule language, which usually provide advanced constructs based on them, like sequences (see Section 3.8). Notice how such kind of timestamps can be given by sources when the event is observed, or by the IFP engine as it receives each item. In the former case, a buffering mechanism is required to cope with out-of-order arrivals [Babcock et al. 2002].

Another case is that of systems that associate each item with some form of label, which, while not representing an absolute instant in time, allows to define a partial ordering among items, usually reflecting some kind of causal relationship, i.e., the fact that the occurrence of an event was caused by the occurrence of another event. Again, what is important for our model is the fact that such labels and the ordering they impose are fully available to the rule language. In this case we say that the time model is *causal*.

Finally, there is the case of those systems that associate items with an interval, i.e., two timestamps taken from a global time, usually representing: the first one the time when the related event started, the second one the time when it ended. In this case, depending on the semantics associated with intervals, a total or a partial ordering among items can be defined [Galton and Augusto 2002; Adaikkalavan and Chakravarthy 2006; White et al. 2007].

### 3.7 Rule Model

Rules are much more complex entities than data. Looking at existing systems we can find many different approaches to represent rules, which depend on the rule language adopted. However, we can roughly classify them in two macro classes: *transforming rules* and *detecting rules*.

Transforming rules define an *execution plan* composed of *primitive operators* connected to each other in a graph. Each operator takes several flows of information items as its inputs and produces new items, which can be forwarded to other operators or directly sent out to sinks. The execution plan can be either user-defined or compiled. In the first case, rule managers are allowed to define the exact operators graph to be executed; in the second case, they write their rules in a high-level language, which the system compiles into an execution plan. The latter approach is adopted by a wide number of DSMSs, where rules are typically expressed using SQL-like statements.

Transforming rules are often used with homogeneous information flows, so that the definition of an execution plan can take advantage of the predefined structure of input and output flows.

Detecting rules are those that present a clear distinction between a condition and an action part. Usually, the former is represented by a logical predicate that captures *patterns* of interest in the sequence of information items, while the latter uses ad-hoc constructs to define how relevant information has to be processed and aggregated to produce new information. Examples of detecting rules can be found in many CEP systems, where they are adopted to specify how new events originate from the detection of others. Other examples can be found in active data-bases, which often use detecting rules to capture undesired sequences of operations within a transaction (the condition), in order to output a *roll-back* command (the action) as soon as the sequence is detected.

The final issue we address in our rule model is the ability to deal with *uncertainty* [Wasserkrug et al. 2008; Liu and Jacobsen 2004]. In particular, some systems allow to distinguish between *deterministic* and *probabilistic* rules. The former define deterministically their outputs from their inputs, while the latter allow to associate a degree of uncertainty (or a confidence) to the outputs. Notice that the issue is only partially related with the ability of managing uncertainty data (see Section 3.5.

Indeed, a rule could introduce a certain degree of uncertainty even in presence of definite and precise inputs. As an example, one could say that there is a good probability of fire if a temperature higher than $70^oC$ is detected. The input information is precise, but the rule introduce a certain degree of uncertainty (other situations could explain the raising of temperature).

### 3.8 Language Model

The rule model described in previous section provides a first characterization of the languages used to specify processing rules in currently available IFP systems. Here we give a more precise and detailed description of such languages: in particular, we first define the general classes into which all existing languages can be divided and then we provide an overview of all the operators we encountered during the analysis of existing systems. For each operator we specify if and how it can be defined inside the aforementioned classes.

3.8.1 *Language Type.* Following the classification introduced into the rule model, the languages used in existing IFP systems can be divided into the following two classes:

- *Transforming languages* define transforming rules, specifying one or more operations that process the input flows by filtering, joining and aggregating received information to produce one or more output flows. Transforming languages are the most commonly used in DSMSs. They can be further divided into two classes:
  - *Declarative languages* express processing rules in a declarative way, i.e. by specifying the expected results of the computation rather than the desired execution flow. These languages are usually derived from relational languages, in particular relational algebra and SQL [Eisenberg and Melton 1999], which they extend with additional ad-hoc operators to better support flowing information.
  - *Imperative languages* define rules in an imperative way, by letting the user specify a plan of primitive operators that the information flows have to follow. Each primitive operator defines a transformation over its input. Usually, systems that adopt imperative languages offer visual tools for rules' definition.
- *Detecting, or pattern-based languages* define detecting rules by separately specifying the firing conditions and the actions to be taken when such conditions hold. Conditions are usually defined as *patterns* that select matching portions of the input flows using logical operators, content and timing constraints; actions define how the selected items have to be combined to produce new information. This type of languages is common in CEP systems, which aim to detect relevant information items before starting their processing.

It is worth mentioning that existing systems sometimes allow users to express rules using more than one paradigm. For example, many commercial systems offer both a declarative language for rule creation, and a graphical tool for connecting defined rules in an imperative way. Moreover, some existing declarative languages embed simple operators for pattern detection, blurring the distinction among transforming and detecting languages.

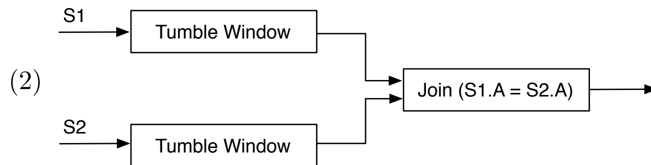
In the following, as a representative example for declarative languages we consider CQL [Arasu et al. 2006], created within the Stream project [Arasu et al. 2003] and

currently adopted by Oracle [Oracle 2009]. It defines three classes of operators: relation-to-relation operators are similar to the standard SQL operators and define queries over database tables; stream-to-relation operators allow the users to build database tables selecting portions of a given information flow, while relation-to-stream operators create flows starting from queries on fixed tables. Stream-to-stream operators are not explicitly defined, as they can be expressed using the existing ones. Here is an example of a CQL rule:

(1)
```
Select IStream(*)
From F1 [Rows 5], F2 [Rows 10]
Where F1.A = F2.A
```

This rule isolates the last 5 elements of flow F1 and the last 10 elements of flow F2 (using the stream-to-relation operator `[Rows n]`), then combines all elements having a common attribute A (using a relation-to-relation operator) and produces a new flow with the created items (using the relation-to-stream operator `IStream`).

Imperative languages are well represented by Aurora's Stream Query Algebra (SQuAl), which adopts a graphical representation called by the authors *boxes and arrows* [Abadi et al. 2003]. In Figure 2 we show how it is possible to express a rule similar to the one we introduced in Example 1 using Aurora's language. The tumble window operator selects portions of the input stream according to given constrains, while join merges elements having the same value for attribute A.

(2)



Detecting languages can be exemplified by the composite subscription language of Padres [Li and Jacobsen 2005]. Example 3 shows a rule expressed with such language.

(3) `A(X>0) & (B(Y=10);`$_{\texttt{[timespan:5]}}$ `C(Z<5))`$_{\texttt{[within:15]}}$

A, B, and C represent item types or topics, while X, Y, and Z are inner fields of items. After the topic of an information item has been determined, filters on its content are applied as in content-based publish-subscribe systems. The rule of Example 3 fires when an item of type A having an attribute X>0 enters the systems and also an item of type B with Y=10 is detected, followed (in a time interval of 5 to 15 sec.) by an item of type C with Z<5. Like other systems conceived as an evolution of traditional publish-subscribe middleware, Padres defines only the detection part of rules, the default and implicit action being that of forwarding a notification of the detected pattern to proper destinations. However, more expressive pattern languages exist, which allow complex transformations to be performed on selected data.

In the following, whenever possible, we use the three languages presented above to build the examples we need in our presentation.

3.8.2 *Available Operators.* We now provide a complete list of all the operators that we found during the analysis of existing IFP systems. Some operators are typical of only one of the classes defined above, while others cross the boundaries of classes. In the following we highlight, whenever possible, the relationship between each operator and the language types in which it may appear.

It is worth noting that there is not a direct mapping between available operators and the language's expressiveness: usually it is possible to express the same rule combining different sets of operators. Whenever possible we will show such equivalence, especially if it involves operators provided by different classes of languages.

**Single-Item Operators.** A first kind of operators that IFP systems provide are *single-item* operators, i.e. all those operators that process information items one by one. Two classes of single-item operators exist:

—*Selection operators* filter items according to their content, throwing away those elements that do not satisfy a given constraint. As an example, they can be used to keep only those information items that contain temperature readings whose value is greater than $20^oC$.

—*Elaboration operators* transform information items. As an example, they can be used to change a temperature reading, moving from Celsius to Fahrenheit. Among those elaboration operators, it is worth mentioning those coming from relational algebra, and in particular:
 —*Projection* extracts only some parts of the information contained in the considered item. As an example, it is used to process items containing data about a person in order to extract only her name.
 —*Renaming* changes the name of a field in languages based on records.

Single-item operators are present in every class of language. Declarative languages usually inherits selection, projection, and renaming from relational algebra, while imperative languages offer primitive operators both for selection and for some kind of elaboration. In pattern-based languages selection operators are used to filter those items that should be part of a complex pattern. Notably, they are the only operators available in publish-subscribe systems to select items to be forwarded to sinks. When allowed, elaborations can be expressed inside the action part of rules, where it is possible to define how selected items have to be processed.

**Logic Operators.** Logic operators are used to define rules that combine the detection of several information items. They differ from sequences (see below) in that they are *order independent*, i.e., they define patterns that rely only on the detection (or non detection) of single information items and not on some specific ordering relation defined for them.

—A *conjunction* of items $I_1$, $I_2$, .. $I_n$ is satisfied when all the items $I_1$, $I_2$, .. $I_n$ have been detected.

—A *disjunction* of items $I_1$, $I_2$, .. $I_n$ is satisfied when at least one of the information items $I_1$, $I_2$, .. $I_n$ has been detected.

—A *repetition* of an information item $I$ of degree $< m, n >$ is satisfied when $I$ is detected at least $m$ times and not more than $n$ times (it is a special form of conjunction).

—A *negation* of an information item $I$ is satisfied when $I$ is not detected.

Usually it is possible to combine such operators with each others or with other operators to form more complex patterns. For example, it is possible to specify disjunctions of conjunctions of items, or to combine logic operators and single-items operators to dictate constraints both on the content of each element and on the relationships among them.

It is worth noting how the use of logic operators allows the definition of expressions whose value cannot be verified in a finite amount of time, unless explicit bounds are defined for them. This is the case of repetition and negation, which require elements to remain undetected in order to be satisfied. For this reason existing systems combine these operators with other linguistic constructs known as *windows* (see below).

Logic operators are always present in pattern-based languages, where they represent the typical way to combine information items. Example 4 shows how a disjunction of two conjunctions can be expressed using the language of Padres (A, B, C, and D are simple selections).

(4)
```
(A & B) || (C & D)
```

On the contrary, declarative and imperative languages do not provide logic operators explicitly; however they usually allow conjunctions, disjunctions, and negations to be expressed using rules that transform input flows. Example 5 shows how a conjunction can be expressed in CQL: the `From` clause specifies that we are interested in considering (portions of) both flows F1 and F2, while the `Where` clause defines which constraints items should satisfy to be combined.

(5)
```
Select IStream(F1.A, F2.B)
From F1 [Rows 50], F2 [Rows 50]
Where F1.A = F2.A
```

**Sequences.** Similarly to logic operators, *sequences* are used to capture the arrival of a set of information items, but they take into consideration the order of arrival. More specifically, a sequence defines an ordered set of information items $I_1$, $I_2$, .. $I_n$, which is satisfied when all the elements $I_1$, $I_2$, .. $I_n$ have been detected in the specified order (see Section 3.6).

The sequence operator is present in many pattern-based languages, while transforming languages usually do not provide it explicitly. Still, in such languages it is sometimes possible (albeit less natural) to mimic sequences, for example when the ordering relation is based on a timestamp field explicitly added to information items, as in the following example: [3]

(6)
```
Select IStream(F1.A, F2.B)
From F1 [Rows 50], F2 [Rows 50]
Where F1.timestamp < F2.timestamp
```

[3] CQL adopts a stream-based time model. It associates implicit timestamps to information items and use them in time-based windows, but they cannot be explicitly addressed within the language; consequently they are not suitable to define sequences.

**Iterations.** *Iterations* express possibly unbounded sequences of information items satisfying a given *iterating condition*. Like sequences, iterations rely on the ordering of items; however, they do not define the set of items to be captured explicitly; on the contrary, they define it implicitly using the iterating condition.

(7)
```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
a.type = 'contaminated' and
b[1].from = a.site and
b[i].from = b[i-1].to }
WITHIN 3 hours
```

Example 7 shows an iteration written in the Sase+ language [Gyllstrom et al. 2008]. The rule detects contamination in a food supply chain: it captures an alert for a contaminated site (item $a$) and reports all possible series of infected shipments (items $b[i]$). Iteration is expressed using the + operator (usually called *Kleene plus* [Agrawal et al. 2008]), defining sequences of one or more *Shipment* information items. The iterating condition $b[i].from = b[i-1].to$ specifies the collocation condition between each shipment and its preceding one. *Shipment* information items need not to be contiguous within the input flow; intermediate items are simply discarded (*skip_till_any_match*). The length of captured sequences is not known a priori, but depends on the actual number of shipments from site to site. To ensure the termination of pattern detection, a time bound is expressed, using the *WITHIN* operator (see *Windows* below).

While most pattern-based languages include a sequence operator, it is less common to find iterations. In addition, like sequences, iterations are generally not provided in transforming languages.

It is worth mentioning, however, that some recent works have investigated the possibility to include sequences and iterations in declarative languages (e.g., [Sadri et al. 2004]). These efforts usually result in embedding pattern detection into traditional declarative languages.

As a final remark, we notice that iterations are strictly connected with the possibility for an IFP system to read its own output and to use it for recursive processing. In fact, if a system provides both a sequence operator and recursive processing, it can mimic iterations through recursive rules.

**Windows.** As mentioned before, it is sometimes necessary to define which portions of the input flows have to be considered during the execution of operators. For this reason almost all the languages used in existing systems define *windows*. Windows cannot be properly considered operators; rather, they are language constructs that can be applied to operators to limit the scope of their action.

To be more precise, we can observe that operators can be divided into two classes: *blocking operators* need to read the whole input flows before producing any result, while *non-blocking operators* can successfully return their results as items enter the system [Babcock et al. 2002]. An example of blocking operator is negation, which has to process the entire flow before deciding that the searched item is not present. The same happens to repetitions when an upper bound on the number of items to consider is provided. On the contrary, conjunctions and disjunctions are

non-blocking operators, as they can stop parsing their inputs as soon as they find the searched items. In IFP systems information flows are, by nature, unbounded; consequently, it is not possible to evaluate blocking operators as they are. In this context, windows become the key construct to enable blocking operators by limiting their scope to (finite) portions of the input flows. On the other hand, windows are also extensively used with non-blocking operators, as a powerful tool to impose a constraint over the set of items that they have to consider.

Existing systems define several types of windows. First of all they can be classified into *logical* (or *time-based*) and *physical* (or *count-based*) [Golab and Özsu 2003]. In the former case, bounds are defined as a function of time: for example to force an operation to be computed only on the elements that arrived during the last five minutes. In the latter case, bounds depend on the number of items included into the window: for example to limit the scope of an operator to the last ten elements arrived.

An orthogonal way to classify windows considers the way their bounds move, resulting in the following classes [Carney et al. 2002; Golab and Özsu 2003]:

—*Fixed windows* do not move. As an example, they could be used to process the items received between 1/1/2010 and 31/1/2010.

—*Landmark windows* have a fixed lower bound, while the upper bound advances every time a new information item enters the system. As an example, they could be used to process the items received since 1/1/2010.

—*Sliding windows* are the most common type of windows. They have a fixed size, i.e., both lower and upper bounds advance when new items enter the system. As an example, we could use a sliding window to process the last ten elements received.

—*Pane and tumble windows* are variants of sliding windows in which both lower and upper bounds move by $k$ elements, as $k$ elements enter the system. The difference between pane and tumble windows is that the former have a size greater than $k$, while the latter have a size smaller than (or equal to) $k$. In practice, a tumble window assures that every time the window is moved all contained elements change; so each element of the input flow is processed at most once. The same is not true for pane windows. As an example, consider the problem of calculating the average temperature in the last week. If we want such a measure every day at noon we have to use a pane window, if we want it every Sunday at noon we have to use a tumble window.

Example 8 shows a CQL rule that uses a count-based, sliding window over the flow F1 to count how many among the last 50 items received has $A > 0$; results are streamed using the IStream operator. Example 9 does the same but considering the items received in the last minute.

(8)
```
Select IStream(Count(*))
From F1 [Rows 50]
Where F1.A > 0
```

(9)
```
Select IStream(Count(*))
From F1 [Range 1 Minute]
Where F1.A > 0
```

Interestingly there exist languages that allow users to define and use their own windows. The most notable case is ESL [Bai et al. 2006], which provides *user defined aggregates* to allow users to freely process input flows. In doing so, users are allowed to explicitly manage the part of the flow that they want to consider, i.e., the window. As an example, consider the ESL rule in Example 10: it calculates the smallest positive value received and delivers it as its output every time a new element arrives. The window is accessible to the user as a relational table, called `inwindow`, whose inner format can be specified during the aggregate definition; it is automatically filled by the system when new elements arrive. The user may specify actions to be taken at different times using three clauses: the `INITIATE` clause defines special actions to be executed only when the first information item is received; the `ITERATE` clause is executed every time a new element enters the system, while the `EXPIRE` clause is executed every time an information item is removed from the window. In our example the `INITIATE` and `EXPIRE` clauses are empty, while the `ITERATE` clause removes every incoming element having a negative value, and immediately returns the smallest value (using the `INSERT INTO RETURN` statement). It is worth noting that the aggregate, as defined in Example 10, can be applied to virtually all kinds of windows, which are then modified during execution in order to contain only positive values.

(10)
```
WINDOW AGGREGATE positive_min(Next Real): Real {
   TABLE inwindow(wnext real);
   INITIALIZE : { }
   ITERATE : {
      DELETE FROM inwindow
      WHERE wnext < 0
      INSERT INTO RETURN
      SELECT min(wnext)
      FROM inwindow
   }
   EXPIRE : { }
}
```

Generally, windows are available in declarative and imperative languages. Conversely, only a few pattern-based languages provide windowing constructs; some of them, in fact, simply do not include blocking operators, while others include explicit bounds as part of blocking operators to make them unblocking (we can say that such operators "embed a window"). For example, Padres does not provide the negation and the iteration operators and does not allow repetitions to include an upper bound. Similarly, CEDR [Barga et al. 2007] can express negations through the UNLESS operator, shown in Example 11. The pattern is satisfied if A is not followed by B within 12 hours. Notice how the operator itself requires explicit timing constraints to become unblocking.

```
     EVENT Test-Rule
(11) WHEN UNLESS(A, B, 12 hours)
     WHERE A.a < B.b
```

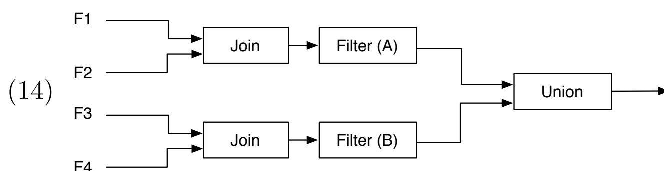
**Flow Management Operators.** Declarative and imperative languages require ad-hoc operators to merge, split, organize, and process flows of information. They include:

—The *Join* operator, used to merge two flows of information exactly as in traditional DBMS. Being a blocking operator the join is usually applied to portions of the input flows, which are processed as standard tables. As an example, the CQL Rule 12 uses the join operator to combine the last 1000 items of flows $F1$ and $F2$ by merging those items that have the same value in field $A$.

```
     Select IStream(F1.A, F2.B)
(12) From F1 [Rows 1000], F2 [Rows 1000]
     Where F1.A = F2.A
```

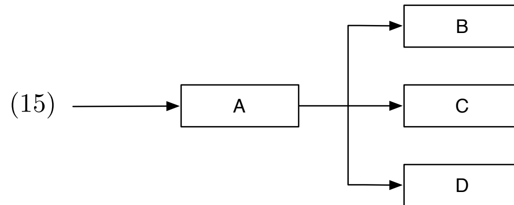
—*Bag operators*, which combine different flows of information considering them as bags of items. In particular, we have the following bag operators:
  —The *union* merges two or more input flows of the same type creating a new flow that includes all the items coming from them.
  —The *except* takes two input flows of the same type and outputs all those items that belong to the first one but not to the second one. It is a blocking operator.
  —The *intersect* takes two or more input flows and outputs only those items that are contained in all of them. It is a blocking operator.
  —The *remove-duplicate* removes all duplicates from an input bag.
  Rule 13 provides an example of using the union operator in CQL to merge together two flows of information. Similarly, in Example 14 we show the union operator as provided by Aurora.

```
     Select IStream(*)
     From F1 [Rows 1000], F2 [Rows 1000]
     Where F1.A = F2.A
(13) Union
     Select IStream(*)
     From F3 [Rows 1000], F4 [Rows 1000]
     Where F3.B = F4.B
```

(14) 


—The *duplicate* operator allows a single flow to be duplicated in order to use it as an input for different processing chains. Example 15 shows an Aurora rule that

takes a single flow of items, processes it through the operator `A`, then duplicates the result to have it processed by three operators `B`, `C`, and `D`, in parallel.

(15) 


—The *group-by* operator is used to split an information flow into partitions in order to apply the same operator (usually an aggregate) to the different partitions. Example 16 uses the group-by operator to split last 1000 rows of flow $F1$ based on the value of field $B$, to count (i.e., the aggregate) how may items exist for each value of $B$.

(16)
```
Select IStream(Count(*))
From F1 [Rows 1000]
Group By F1.B
```

—The *order-by* operator is used to impose an ordering to the items of an input flow. It is a blocking operator so it is usually applied to well defined portions of the input flows.

**Parameterization.** In many IFP applications it is necessary to filter some flows of information based on information that are part of other flows. As an example, the fire protection system of a building could be interested in being notified when the temperature of a room exceeds $40^oC$ but only if some smoke has been detected *in the same room*. Declarative and imperative languages address this kind of situations by first joining the two flows of information, i.e., that about room temperature and that about smoke detection, then applying a filter to the resulting flow to keep only those items related to the same room. On the other hand, pattern-based languages do not provide the join operator. They have to capture the same situation using a rule that combines, through the conjunction operator, detection of high temperature and detection of smoke in the same room. This latter condition can be expressed only if the language allows filtering to be *parametric*.

Given the importance of this characteristic we introduce it here even if it cannot be properly considered an operator by itself. More specifically, we say that a pattern-based language provides parameterization if it allows the parameters of an operator (usually a selection, but sometimes other operators, as well) to be constrained using values taken from other operators in the same rule.

(17)
```
(Smoke(Room=$X)) & (Temp(Value>40 AND Room=$X))
```

Example 17 shows how the rule described above can be expressed using the language of Padres; the operator $ allows parameters definition inside the filters that select single information items.

**Flow creation.** Some languages define explicit operators to create new information flows from a set of items. In particular, flow creation operators are used in declarative languages to address two issues:

—some of them have been designed to deal indifferently with information flows and with standard tables a-la DBMS. In this case, flow creation operators can be used to create new flows from existing tables, e.g., when new elements are added to the table.

—other languages use windows as a way to transform (part of) an input flow into a table, which can be further processed by using the classical relational operators. Such languages use flow creation operators to transform tables back into flows. As an example, CQL provides three operators called *relation-to-stream* that allow the creation of a new flow from a relational table $T$: at each evaluation cycle, the first one (IStream) streams all new elements added to $T$; the second one (DStream) streams all the elements removed from $T$; while the last one (Rstream) streams all the elements of $T$ at once. Consider Example 18: at each processing cycle the rule populates a relational table ($T$) with the last ten items of flow F1; then, it streams all elements that are added to $T$ at the current processing cycle (but were not part of $T$ in the previous cycle). In Example 19, instead, the system streams all elements that were in $T$, but have been removed during the current porcessing cycle. Finally, in Example 20, all items in $T$ are put in the output stream, independently from previous processing cycles.

(18)
```
Select IStream(*)
From F1 [Rows 10]
```

(19)
```
Select DStream(*)
From F1 [Rows 10]
```

(20)
```
Select RStream(*)
From F1 [Rows 10]
```

**Aggregates.** Many IFP applications need to aggregate the content of multiple, incoming information items to produce new information, as an example by calculating the average of some value or its maximum. We can distinguish two kinds of aggregates:

—*Detection aggregates* are those used during the evaluation of the condition part of a rule. In our functional model, they are computed and used by the *Decider*. As an example, they can be used in detecting all items whose value exceeds the average one (i.e., the aggregate), computed over the last 10 received items.

—*Production aggregates* are those used to compute the values of information items in the output flow. In our functional model, they are computed by the *Producer*. As an example, they can be used to output the average value among those part of the input flow.

Almost all existing languages have predefined aggregates, which include minimum, maximum, and average. Some pattern-based languages offer only production aggregates, while others include also detection aggregates to capture patterns that involve computations over the values stored in the *History*. In declarative and imperative languages aggregates are usually combined with the use of windows to limit their scope (indeed, aggregates are usually blocking operators and windows allow to process them on-line). Finally, some languages also offer facilities to create *user-defined aggregates (UDAs)*. As an example of the latter, we have already shown the ESL rule 10, which defines an UDA to compute the smallest positive value among the received It has been proved that adding complete support to UDAs makes a language Turing-complete [Law et al. 2004].

## 4. IFP SYSTEMS: A CLASSIFICATION

In the following we use the concepts introduced in Section 3 to present and classify existing IFP systems. We provide a brief description of each system and summarize its characteristics by compiling four tables.

Table I focuses on the functional and processing models of each system: it shows if a system includes the *Clock* (i.e., it supports periodic processing) and the *Knowledge Base*, and if the maximum size of the sequences (*Seq*) that the *Decider* sends to the *Producer* is bounded or unbounded, given the set of deployed rules. Then, Table I analyzes if information items produced by fired rules may re-enter the system (*recursion*), and if the rule set can be changed by fired actions at run-time. Finally, it presents the processing model of each system, by showing its selection, consumption, and load shedding policies.

Table II focuses on the deployment and interaction models, by showing the type of deployment supported by each system, and the interaction styles allowed in the observation, notification, and forwarding models.

Table III focuses on the data, time, and rule models, presenting the nature of the items processed by each systems (generic data or event notifications), the format of data, the support for data uncertainty, and the nature of flows (homogeneous or heterogeneous). It also introduces how the notion of time is captured and represented by each system, the type of rules adopted, and the possibility to define probabilistic rules.

Finally, Table IV focuses on the language adopted by each system, by listing its type and the set of operators available.

Given the large number of systems reviewed, we organize them in four groups: active databases, DSMSs, CEP systems, and commercial systems (while systems belonging to previous groups are research prototypes). The distinction is sometimes blurred but it helps us better organize our presentation.

The presentation of each class does not follow a strict chronological order: systems that share similar characteristics are listed one after the other, to better emphasize common aspects.

### 4.1 Active databases

**HiPac.** Hipac [Dayal et al. 1988; McCarthy and Dayal 1989] has been the first project proposing Event Condition Action (ECA) rules as a general formalism for

Table I. Functional and Processing Models

| | Functional Model | | | | | Processing Model | | |
|---|---|---|---|---|---|---|---|---|
| **Name** | **Clock** | **K. Base** | **Seq** | **Recursion** | **Dynamic Rule Change** | **Select. Policy** | **Consum. Policy** | **Load Shedding** |
| **HiPac** | Present | Present | Bounded | Yes | Yes | Multiple | Zero | No |
| **Ode** | Absent | Present | Unbounded | Yes | Yes | Multiple | Zero | No |
| **Samos** | Present | Present | Unbounded | Yes | Yes | ? | ? | No |
| **Snoop** | Present | Present | Unbounded | Yes | Yes | Program. | Program. | No |
| **TelegraphCQ** | Present | Present | Unbounded | No | No | Multiple | Zero | Yes |
| **NiagaraCQ** | Present | Present | Unbounded | No | No | Multiple | Selected | Yes |
| **OpenCQ** | Present | Present | Unbounded | No | No | Multiple | Selected | No |
| **Tribeca** | Absent | Absent | Unbounded | No | Yes | Multiple | Zero | No |
| **CQL / Stream** | Absent | Present | Unbounded | No | No | Multiple | Zero | Yes |
| **Aurora / Borealis** | Absent | Present | Unbounded | No | No | Program. | Zero | Yes |
| **Gigascope** | Absent | Absent | Unbounded | No | No | Multiple | Zero | No |
| **Stream Mill** | Absent | Present | Unbounded | No | No | Program. | Program. | Yes |
| **Traditional Pub-Sub** | Absent | Absent | Single | No | No | Single | Single | No |
| **Rapide** | Absent | Present | Unbounded | Yes | No | Multiple | Zero | No |
| **GEM** | Present | Absent | Bounded | Yes | Yes | Multiple | Zero | No |
| **Padres** | Absent | Absent | Unbounded | No | No | Multiple | Zero | No |
| **DistCED** | Absent | Absent | Unbounded | No | No | Multiple | Zero | No |
| **CEDR** | Absent | Absent | Unbounded | Yes | No | Program. | Program. | No |
| **Cayuga** | Absent | Absent | Unbounded | Yes | No | Multiple | Zero | No |
| **NextCEP** | Absent | Absent | Unbounded | Yes | No | Multiple | Zero | No |
| **PB-CED** | Absent | Absent | Unbounded | No | No | Multiple | Zero | No |
| **Raced** | Present | Absent | Unbounded | Yes | No | Multiple | Zero | No |
| **Amit** | Present | Absent | Unbounded | Yes | No | Program. | Program. | No |
| **Sase** | Absent | Absent | Bounded | No | No | Multiple | Zero | No |
| **Sase+** | Absent | Absent | Unbounded | No | No | Program. | Zero | No |
| **Peex** | Absent | Absent | Unbounded | Yes | No | Multiple | Zero | No |
| **Aleri SP** | Present | Present | Unbounded | No | No | Multiple | Zero | ? |
| **Coral8 CEP** | Present | Present | Unbounded | No | No | Program. | Program. | Yes |
| **StreamBase** | Present | Present | Unbounded | No | No | Multiple | Zero | ? |
| **Oracle CEP** | Absent | Present | Unbounded | No | No | Program. | Program. | ? |
| **Esper** | Present | Present | Unbounded | No | No | Program. | Program. | No |
| **Tibco BE** | Absent | Absent | Unbounded | Yes | No | Multiple | Zero | ? |
| **IBM System S** | Present | Present | Unbounded | Yes | No | Program. | Program. | Yes |

Table II. Deployment and Interaction Models

| | Deployment Model | Interaction Model | | |
|---|---|---|---|---|
| **Name** | **Deployment Type** | **Observation** | **Notification** | **Forwarding** |
| **HiPac** | Centralized | Push | Push | n.a. |
| **Ode** | Centralized | Push | Push | n.a. |
| **Samos** | Centralized | Push | Push | n.a. |
| **Snoop** | Centralized | Push | Push | n.a. |
| **TelegraphCQ** | Clustered | Push | Push | Push |
| **NiagaraCQ** | Centralized | Push/Pull | Push/Pull | n.a. |
| **OpenCQ** | Centralized | Push/Pull | Push/Pull | n.a. |
| **Tribeca** | Centralized | Push | Push | n.a. |
| **CQL / Stream** | Centralized | Push | Push | n.a. |
| **Aurora / Borealis** | Clustered | Push | Push | Push |
| **Gigascope** | Centralized | Push | Push | n.a. |
| **Stream Mill** | Centralized | Push | Push | n.a. |
| **Traditional Pub-Sub** | System dependent | Push | Push | System dependent |
| **Rapide** | Centralized | Push | Push | n.a. |
| **GEM** | Networked | Push | Push | Push |
| **Padres** | Networked | Push | Push | Push |
| **DistCED** | Networked | Push | Push | Push |
| **CEDR** | Centralized | Push | Push | n.a. |
| **Cayuga** | Centralized | Push | Push | n.a. |
| **NextCEP** | Clustered | Push | Push | Push |
| **PB-CED** | Centralized | Push/Pull | Push | n.a. |
| **Raced** | Networked | Push | Push | Push/Pull |
| **Amit** | Centralized | Push | Push | n.a. |
| **Sase** | Centralized | Push | Push | n.a. |
| **Sase+** | Centralized | Push | Push | n.a. |
| **Peex** | Centralized | Push | Push | n.a. |
| **Aleri SP** | Clustered | Push | Push/Pull | Push |
| **Coral8 CEP** | Clustered | Push | Push/Pull | Push |
| **StreamBase** | Clustered | Push | Push/Pull | Push |
| **Oracle CEP** | Clustered | Push | Push/Pull | Push |
| **Esper** | Clustered | Push | Push/Pull | Push |
| **Tibco BE** | Networked | Push | Push | Push |
| **IBM System S** | Clustered | Push | Push | Push |

Table III. Data, Time, and Rule Models

| | Data Model | | | | Time Model | Rule Model | |
|---|---|---|---|---|---|---|---|
| **Name** | **Nature of Items** | **Format** | **Support for Uncert.** | **Nature of Flows** | **Time** | **Rule Type** | **Probab. Rules** |
| **HiPac** | Events | Database and External Events | No | Heterogeneous | Absolute | Detecting | No |
| **Ode** | Events | Method Invocations | No | Heterogeneous | Absolute | Detecting | No |
| **Samos** | Events | Methon Invocation and External Events | No | Heterogeneous | Absolute | Detecting | No |
| **Snoop** | Events | Database and External Events | No | Heterogeneous | Absolute | Detecting | No |
| **TelegraphCQ** | Data | Tuples | No | Homogeneous | Absolute | Transforming | No |
| **NiagaraCQ** | Data | XML | No | Homogeneous | Stream-only | Transforming | No |
| **OpenCQ** | Data | Tuples | No | Heterogeneous | Stream-only | Transforming | No |
| **Tribeca** | Data | Tuples | No | Homogeneous | Stream-only | Transforming | No |
| **CQL / Stream** | Data | Tuples | No | Homogeneous | Stream-only | Transforming | No |
| **Aurora / Borealis** | Data | Tuples | No | Homogeneous | Stream-only | Transforming | No |
| **Gigascope** | Data | Tuples | No | Homogeneous | Causal | Transforming | No |
| **Stream Mill** | Data | Tuples | No | Homogeneous | Stream-only | Transforming | No |
| **Traditional Pub-Sub** | Events | System dependent | System dependent | Heterogeneous | System dependent | Detecting | System dependent |
| **Rapide** | Events | Records | No | Heterogeneous | Causal | Detecting | No |
| **GEM** | Events | Records | No | Heterogeneous | Interval | Detecting | No |
| **Padres** | Events | Records | No | Heterogeneous | Absolute | Detecting | No |
| **DistCED** | Events | Records | No | Heterogeneous | Interval | Detecting | No |
| **CEDR** | Events | Tuples | No | Homogeneous | Interval | Detecting | No |
| **Cayuga** | Events | Tuples | No | Homogeneous | Interval | Detecting | No |
| **NextCEP** | Events | Tuples | No | Homogeneous | Interval | Detecting | No |
| **PB-CED** | Events | Tuples | No | Heterogeneous | Interval | Detecting | No |
| **Raced** | Events | Records | No | Heterogeneous | Absolute | Detecting | No |
| **Amit** | Events | Records | No | Heterogeneous | Absolute | Detecting | No |
| **Sase** | Events | Records | No | Heterogeneous | Absolute | Detecting | No |
| **Sase+** | Events | Records | No | Heterogeneous | Absolute | Detecting | No |
| **Peex** | Events | Records | Yes | Heterogeneous | Absolute | Detecting | Yes |
| **Aleri SP** | Data | Tuples | No | Homogeneous | Stream-only | Transforming | No |
| **Coral8 CEP** | Data | Tuples | No | Homogeneous | Stream-only | Transforming | No |
| **StreamBase** | Data | Tuples | No | Homogeneous | Stream-only | Transforming | No |
| **Oracle CEP** | Data | Tuples | No | Homogeneous | Stream-only | Transforming | No |
| **Esper** | Data | Tuples | No | Homogeneous | Stream-only | Transforming | No |
| **Tibco BE** | Events | Records | No | Heterogeneous | Interval | Detecting | No |
| **IBM System S** | Data | Various | No | Homogeneous | Stream-only | Transforming | No |

active databases. In particular, the authors introduce a model for rules in which three kinds of primitive events are taken into account: database operations, temporal events, and external notifications. Primitive events can be combined using disjunction and sequence operators to specify composite events. Since these operators force users to explicitly write all the events that have to be captured, we can say that, given a set of rule, the size of the sequence of events that can be selected (i.e. the size of the *Seq* component) is bounded.

The condition part of each rule is seen as a collection of queries that may refer to database state, as well as to data embedded in the considered event. We represent the usage of the database state during the condition evaluation through the presence of the *knowledge base* component. Actions may perform operations on the database as well as execute additional commands: it is also possible to specify activation or deactivation of rules within the action part of a rule, thus enabling dynamic modification of the rule base. Since the event composition language of HiPac includes a sequence operator, the system can reason about time: in particular it uses an absolute time, with a total order between events.

A key contribution of the HiPac project is the definition of *coupling modes*. The

Table IV. Language Model

| | | Single-item | | | Logic | | | | | | Windows | | | | | | Flow Management | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Type | Selection | Projection | Renaming | Conjunction | Disjunction | Repetition | Negation | Sequence | Iteration | Fixed | Landmark | Sliding | Pane | Tumble | User Defined | Join | Union | Except | Intersect | Remove Dup | Duplicate | Group By | Order By | Parameterization | Flow Creation | Detection Aggr | Production Aggr |
| **HiPac** | Det | X | | | | X | | | X | | | | | | | | | | | | | | | | X | | | X |
| **Ode** | Det | X | | | X | X | X | X | X | X | | | | | | | | | | | | | | | X | | | X |
| **Samos** | Det | X | | | X | X | X | X | X | X | X | | | | | | | | | | | | | | X | | | X |
| **Snoop** | Det | X | | | X | X | X | | X | X | | | | | | | | | | | | | | | X | | | X |
| **TelegrahCQ** | Decl | X | X | X | | | | | | | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | | X | X |
| **NiagaraCQ** | Decl | X | X | X | | | | | | | | | | | | | X | | | | | | X | X | X | | X | X |
| **OpenCQ** | Decl | X | X | X | X | X | | X | | | | | | | | | X | X | X | X | X | X | X | X | X | | X | X |
| **Tribeca** | Imp | X | X | | | | | | | | | | X | | X | | | X | | | | X | X | | X | | X | X |
| **CQL/Stream** | Decl | X | X | X | | | | | | | | | X | X | X | | X | X | X | X | X | X | X | X | X | X | X | X |
| **Aurora/Borealis** | Imp | X | X | | | | | | | | | | X | | X | | X | X | | | | X | X | | X | | X | X |
| **Gigascope** | Decl | X | | | | | | | | | | | | | | | X | X | | | | | X | | X | | X | X |
| **Stream Mill** | Decl | X | X | X | | | | | | | X | X | X | X | X | X | X | X | X | X | | | X | X | X | X | X | X |
| **Traditional Pub-Sub** | Det | X | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Rapide** | Det | X | | | X | X | X | X | X | X | X | | X | | | | X | X | | | | | | | X | | X | |
| **GEM** | Det | X | | | X | X | | X | X | | X | X | X | | | | | | | | | | | | X | | | |
| **Padres** | Det | X | | | X | X | | | X | X | | | | | | | | | | | | | | | X | | | |
| **DistCED** | Det | X | | | X | X | | X | X | X | | | X | | | | | | | | | | | | X | | | |
| **CEDR** | Det | X | X | X | X | X | | X | X | | | | | | | | X | X | X | X | | | X | | X | X | | X |
| **Cayuga** | Det | X | X | X | X | X | | | X | X | | | | | | | X | | | | | | | | X | | X | X |
| **NextCEP** | Det | X | X | X | X | X | | | X | X | | | X | | | | | | | | | | | | | | X | X |
| **PB-CED** | Det | X | | | X | X | | X | X | | | | | | | | | | | | | | | | X | | | |
| **Raced** | Det | X | | | X | X | X | X | | | X | X | X | | | | | | | | | | | | X | | | |
| **Amit** | Det | X | | | X | X | X | X | X | X | | | X | | | | | | | | | | | | X | | X | X |
| **Sase** | Det | X | | | X | X | | X | X | | | | X | | | | | | | | | | | | X | | | |
| **Sase+** | Det | X | | | X | X | | X | X | X | | | X | | | | | | | | | | | | X | | X | X |
| **Peex** | Det | X | | | X | X | | X | X | | | | | | | | | | | | | | | | X | | | |
| **Aleri SP** | Imp+Det | X | X | X | X | X | | X | X | | | | X | X | X | | X | X | | | | X | X | X | X | X | X | X |
| **Coral8 CEP** | Decl+Imp+Det | X | X | X | X | X | | X | X | | X | X | X | X | X | X | X | X | X | X | | X | X | X | X | X | X | X |
| **StreamBase** | Decl+Det+Imp | X | X | X | X | X | | X | X | | X | | X | X | X | X | X | X | X | X | | | X | X | X | X | X | X |
| **Oracle CEP** | Decl+Det+Imp | X | X | X | X | | X | | X | X | | | X | | | | X | X | X | X | X | X | X | X | X | X | X | X |
| **Esper** | Decl+Det | X | X | X | X | X | X | X | X | X | | | X | X | X | X | X | X | | | | | X | X | X | X | X | X |
| **Tibco BE** | Det | X | | | X | X | X | X | X | X | X | | X | | | | X | X | | | | | | | X | | X | |
| **IBM System S** | Decl+Imp | X | X | X | | | | | | | X | X | X | X | X | X | X | X | | | | | X | X | X | X | X | X |

coupling mode between an event and a condition specifies when the condition is evaluated w.r.t. the transaction in which the triggering event is signalled. The coupling mode between a condition and an action specifies when the action is executed w.r.t. the transaction in which the condition is evaluated. HiPac introduces three types of coupling modes: *immediate*, *deferred* (i.e., at the end of the transaction) or *separate* (i.e., in a different transaction). Since HiPac captures all the set of events that match the pattern in the event part of a rule, we can say that it uses a fixed multiple selection policy, associated with a zero consumption one.

**Ode.** Ode [Lieuwen et al. 1996; Gehani and Jagadish 1991] is a general-purpose object-oriented active database system. The database is defined, queried, and manipulated using the O++ language, which is an extension of the C++ programming language adding support for persistent objects.

Reactive rules are used to implement *constraints* and *triggers*. Constraints provide a flexible mechanism to adapt the database system to different, application-specific, consistency models: they limit access to data by blocking forbidden operations. Conversely, triggers are associated to allowed operations to specify a set of actions to be automatically executed when specific methods are called on database objects.

Constraints are associated with a class definition. All objects of a class must satisfy the constraints associated with the class. Constraints consist of a condition and an optional handler; the handler is a method performing operations needed to bring the database back to a consistent state. In the case it is not possible to repair the violation of a constraint (undefined handler) Ode will abort the transaction; according to the type of constraint (hard or soft) a constraint violation may be repaired immediately after the violation is reported (*immediate* semantics) or just before the commit of the transaction (*deferred* semantics). Consistently with the object oriented nature of the system, constraints conditions are checked at the boundaries of public methods calls, which represent the only way to interact with the database.

Like integrity constraints, triggers monitor the database for some conditions; the only difference is that these conditions do not represent consistency violations. A trigger, like a constraint, is specified in the class definition and it consists of two parts: a condition and an action. Triggers apply only to the specific objects with respect to which they were activated. Triggers are parameterized, and can be activated multiple times with different parameter values. When the condition of an active trigger becomes true, the corresponding action is executed. Activation and deactivation of rules can be dynamically executed inside the action part of other rules. Unlike a constraint handler, which is executed as part of the transaction violating the constraint, a trigger action is executed as a separate transaction (*separate* semantics).

Both constraints and triggers can be classified as detecting rules expressed using a pattern-based language, where the condition part of the rule includes constraints on inner database values, while the action part is written in C++.

Ode does not offer a complete support for periodic evaluation of rules but it is possible to specify actions to be executed when an active trigger is not fired within a given amount of time.

To map the behavior of Ode to our functional and data model, we can say that the information items that the system processes are the method calls to database objects and evaluation is performed by taking into account the database state (i.e., the *Knowledge Base* in our model).

So far we only mentioned processing involving single information items: support for multiple items processing has been introduced in Ode [Gehani et al. 1992] to allow users to capture complex interactions with the database involving multiple methods call. In particular, it is possible to specify rules that make use of logic operators to combine single calls; sequences are allowed as the centralized nature of the system makes it possible to define a complete ordering relation between calls (absolute time). The goal of the Ode designers was to provide a language that is as expressive as regular expressions, the only addition being the support for parameterization. Negation, for example, although present, expresses only negative constraints on the following information items to be detected. This way all supported operators become non-blocking. The presence of an iteration operator makes it possible to detect sequences of events whose length is unknown a priori (i.e. *Seq* is unbounded). Thanks to their similarities with regular expressions, rules are translated into finite state automata to be efficiently triggered. In [Gehani et al. 1992], the authors also address the issue of defining precise processing policies; in particular, they propose a detection algorithm in which all valid set of events are captures (multiple selection policy), while the possibility to consume events is not mentioned.

**Samos.** Samos [Gatziu and Dittrich 1993; Gatziu et al. 1992] is another example of a general-purpose object-oriented active database system. Like Ode, it uses a detecting language derived from regular expression (including iterations); rules can be dynamically activated, deactivated, created and called during processing.

The most significant difference between Samos and Ode is the possibility offered by Samos to consider *external events* as part of the information flows managed by rules. External events include all those events not directly involving the database structure or content. Among external events, the most important are temporal events, which allow Samos programmers to use time inside rules. As a consequence, time assumes a central role in rule definition: first, it can be used to define periodic evaluation of rules; second, each rule comes with a validity interval, which can be either explicitly defined using the `IN` operator, or implicitly imposed by the system. The `IN` operator defines time-based windows whose bounds can be either fixed time points or relative ones, depending on previous detections. To do so, it relies on an absolute time. The `IN` operator is also useful as not all the operators defined in Samos (like for example the negation) are non-blocking; by providing an explicit window construct through the `IN` operator, Samos offers more flexibility than Ode does.

Great attention has been paid to the specification of precise semantics of rule processing; in particular it is possible to specify when a condition has to be evaluated: immediately, as an event occurs, at the end of the active transaction, or in a separate transaction. Like in HiPac, similar choices are also associated to the execution of actions. Evaluation of rules is performed using Petri Nets.

**Snoop.** Snoop [Chakravarthy and Mishra 1994] is an event specification language

for active databases developed as a part of the Sentinel project [Chakravarthy et al. 1994]. Unlike Ode and Samos, it is designed to be independent from the database model, so it can be used, for example, on relational databases as well as on object-oriented ones.

The set of operators offered by Snoop is very similar to the ones we described for Ode and Samos, the only remarkable difference being the absence of a negation operator. Like Samos, Snoop considers both internal and external events as input information items. Unlike Samos it does not offer explicit windows constructs: instead, it embeds windows features directly inside the operators that require them. For example iterations can only be defined within specific bounds, represented as points in time or information items.

Snoop enables parameter definition as well as dynamic rule activation. The detection algorithm is based on the construction and visiting of a tree.

The most interesting feature of Snoop is the possibility to specify a *context* for each rule. Contexts are used to control the semantics of processing. More specifically, four types of contexts are allowed: recent, chronicle, continuous, and cumulative. In the presence of multiple sets of information items firing the same rule, contexts specify exactly which sets will be considered and consumed by the system (i.e., they specify the selection and consumption policies). For example, the "recent" context only selects the most recent information items, consuming all of them; while the cumulative context selects all the relevant information items, resulting in a multiple policy.

Snoop, like HiPac, Ode, Samos, and like many other proposed languages for active DBMSs (e.g., [Buchmann et al. 1995; Engstrm et al. 1997; Bertino et al. 1998] considers events as points in an time. Recently the authors of Snoop have investigated the possibility to extend the semantics of the language to capture events that have a duration. This led to the definition of a new language, called [Adaikkalavan and Chakravarthy 2006], which offers all the operators and language construct of Snoop, including contexts, but uses interval based timestamps for events.

### 4.2 Data Stream Management Systems

To better organize the presentation, we decided to start our description with all those systems, namely TelegraphCQ, NiagaraCQ, and OpenCQ, that belong to the class of DSMSs, but present a strong emphasis on periodic, as opposed to purely reactive, data gathering and execution; usually, these systems are designed for application domains in which the timeliness requirement is less critical, like for example Internet updates monitoring. On the contrary, all remaining systems are heavily tailored for the processing of high rate streams.

**TelegraphCQ.** TelegraphCQ [Chandrasekaran et al. 2003] is a general-purpose continuous queries system based on a declarative, SQL-based language called StreaQuel.

StreaQuel derives all relational operators from SQL, including aggregates. To deal with unbounded flows, StreaQuel introduces the `WindowIs` operator, which enables the definition of various types of windows. Multiple `WindowIs`, one for each input flow, can be expressed inside a `for` loop, which is added at the end of each rule and defines a time variable indicating when the rule has to be processed. This mechanism is based on the assumption of an absolute time model. By adopting

an explicit time variable, TelegraphCQ enables users to define their own policy for moving windows. As a consequence, the number of items selected at each processing cycle cannot be bounded a priori, since it is not possible to determine how many elements the time window contains.

Since the *WindowIs* operator can be used to capture arbitrary portions of time, TelegraphCQ naturally supports the analysis of historical data. In this case, disk becomes the bottleneck resource. To keep processing of disk data up with processing of live data, a shedding technique called OSCAR (Overload-sensitive Stream Capture and Archive Reduction) is proposed. OSCAR organizes data on disk into multiple resolutions of reduced summaries (such as samples of different sizes). Depending on how fast the live data is arriving, the system picks the right resolution level to use in query processing [Chandrasekaran and Franklin 2004].

TelegraphCQ has been implemented in C++ as an extension of PostgreSQL. Rules are compiled into a query plan, which is then extended using adaptive modules [Avnur and Hellerstein 2000; Raman et al. 1999; Shah et al. 2003], which dynamically decide how to route data to operators and how to order commutative operators. Such modules are designed also to distribute the TelegraphCQ engine over multiple machines, as explained in [Shah et al. 2004a]. The approach is that of clustered distribution, as described in Section 3: operators are placed and, if needed, replicated in order to increase performance, by only taking into account processing constraints and not transmission costs, which are assumed negligible.

**NiagaraCQ.** NiagaraCQ [Chen et al. 2000] is an IFP system for Internet databases. The goal of the system is to provide a high-level abstraction to retrieve information, in the form of XML data sets, from a frequently changing environment like Internet sites. To do so, NiagaraCQ specifies transforming rules using a declarative, SQL-like language called XML-QL [Deutsch et al. 1999]. Each rule has an associated time interval that defines its period of validity. Rules can be either *timer-based* or *change-based*; the former are evaluated periodically during their validity interval, while processing of the latter is driven by notifications of changes received from information sources. In both cases, the evaluation of a rule does not directly produce an output stream, but updates a table with the new results; if a rule is evaluated twice on the same data, it does not produce further updates. We capture this behavior in our processing model, by saying that NiagaraCQ presents a selected consumption policy (information items are used only once). Users can either ask for results on-demand, or can be actively notified (e.g., with an e-mail) when new information is available (i.e., the notification model allows both a push and a pull interaction). For each rule, it is possible to specify a set of actions to be performed just after the evaluation. Since information items processed by NiagaraCQ comes directly from Internet databases, they do not have an explicit timestamp associated.

NiagaraCQ can be seen as a sort of active database system where timer-based rules behave like traditional queries, the only exception being that they are re-evaluated periodically, while change-based rules implement the reactive behavior. NiagaraCQ offers a unified format for both types of rules.

The main difference between NiagaraCQ and a traditional active database is the distribution of information sources over a possibly wide geographical area. It is worth noting that only information sources are actually distributed, while Nia-

garaCQ engine is designed to run in a totally centralized way. To increase scalability NiagaraCQ uses an efficient caching algorithm, which reduces access time to distributed resources and an *incremental group optimization*, which splits operators into groups; members of the same group share the same query plan, thus increasing performance.

**OpenCQ.** Like NiagaraCQ, OpenCQ [Liu et al. 1999] is an IFP system developed to deal with Internet-scale update monitoring. OpenCQ rules are divided into three parts: a SQL *query* that defines operations on data, a *trigger* that specifies when the rule has to be evaluated, and a *stop* condition that defines the period of validity for the rule. OpenCQ presents the same processing and interaction models as NiagaraCQ.

OpenCQ has been implemented on top of the DIOM framework [Liu and Pu 1997], which uses a client-server communication paradigm to collect rules and information and to distribute results; processing of rules is therefore completely centralized. Wrappers are used to perform input transformations, thus allowing heterogeneous sources to be taken into account for information retrieval. Depending on the type of the source, information may be pushed into the system, or it may require the wrapper to periodically ask sources for updates.

**Tribeca.** Tribeca [Sullivan and Heybey 1998] has been developed with a very specific domain in mind: that of network traffic monitoring. It defines transforming rules taking a single information flow as input and producing one or more output flows for the upper level applications. Rules are expressed using an imperative language, which defines the sequence of operators an information flow has to pass through.

Tribeca rules are expressed using three operators: selection (called qualification), projection, and aggregate. Multiplexing and demultiplexing operators are also available, which allow programmers to split and merge flows. Tribeca supports both count based and time based windows, which can be sliding or tumble. They can be exploited to better define portions of flows to be considered when performing aggregates. Since items do not have an explicit timestamp, the processing is performed in arrival order. The presence of a timing window makes it impossible to know the number of information items captured at each detection-production cycle (i.e., the *Seq* is unbounded). Moreover, a rule may be satisfied by more than one set of elements (multiple selection policy), while an element may participate in more than one processing cycle, as it is never explicitly consumed (zero consumption policy).

Each rule is translated into a direct acyclic graph to be executed; during translation the plan can be optimized in order to improve performance.

**CQL/Stream.** CQL [Arasu et al. 2006] is an expressive SQL based declarative language that creates transforming rules with a unified syntax for processing both information flows and stored relations.

The goals of CQL are those of providing a clear semantics for rules while defining a simple language to express them. To do so, CQL specifies three kinds of operators: relation-to-relation, stream-to-relation, and relation-to-stream. Relation-to-relation operators directly derive from SQL: they are the core of the CQL language,

which actually defines processing and transformation. The main advantage of this approach is that a large part of the rule definition is realized by using the standard notation of a widely used language. In order to add support for flow processing, CQL introduces the notion of windows, and in particular sliding, pane, and tumble windows, which are intended as a way to store a portion of each input flow inside relational tables to perform processing; for this reason CQL denotes windows as stream-to-relation operators. Windows can be based both on time and on the number of contained elements. The last kind of operators that CQL provides is that of relation-to-stream operators, which define how processed tuples can become part of a new information flow. Three relation-to-stream operators exist: `IStream` put a tuple into the output flow when it is added to the table, `DStream` considers only removed tuples, while `RStream` considers every tuple contained in the table. CQL is based on a time model that explicitly associates a timestamp to input items; however, timestamps cannot be addressed from the language. Moreover, since output is based on the sequence of updates performed on a relational table, its order is separate from input order.

Items selection is performed using traditional SQL operators and never consumed (until they remain in the evaluation window), so we can say that CQL uses a multiple selection policy, associated with a zero consumption.

CQL has been implemented as part of the Stream project [Arasu et al. 2003]. Stream computes a *query plan* starting from CQL rules; then, it defines a schedule for operators taking into account predefined performance policies.

The Stream project has proposed two major shedding techniques to deal with resource overload problem on data streams: the first addresses the problem of limited computational resources by applying load shedding on a collection of sliding window aggregation queries [Babcock et al. 2004]; the second addresses the problem of limited memory, by discarding operator state for a collection of windowed joins [Srivastava and Widom 2004].

**Aurora/Borealis.** Aurora [Abadi et al. 2003] is a general-purpose DSMS; it defines transforming rules created with an imperative language called SQuAl. SQuAl defines rules in a graphical way, by adopting the *boxes and arrows* paradigm, which makes connections between different operators explicit.

SQuAl defines two types of operators: *windowed* operators apply a single input (user-defined) function to a window and then advance the window to include new elements before repeating the processing cycle; *single-tuple* operators, instead, operate on a single information item at a time; they include single-item operators like selection and flow operators like join, union, and group by. This allows information items to be used more than once (multiple selection policy), while they are never consumed.

SQuAl allows users to define plans having multiple input and multiple output flows and, interestingly, allows users to associate a QoS specification to each output. As output flows may be directly connected to higher-level applications, this makes it possible to customize system behavior according to application requirements. QoS constraints are used by Aurora to automatically define shedding policies: for example some application domain may need to reduce answer precision in order to obtain faster response times. Input and output flows do not have an associated

timestamp, but are processed in their arrival order.

An interesting feature of Aurora is the possibility to include intermediate storage points inside the rule plan: such points can be used to keep historical information and to recover after operators failure.

Processing is performed by a scheduler, which takes an optimized version of the user defined plan and chooses how to allocate computational resources to different operators according to their load and to the specified QoS constraints.

The project has been extended to investigate distributed processing both inside a single administrative domain and over multiple domains [Cherniack et al. 2003]. In both cases the goal is that of efficiently distributing load between available resources; in these implementations, called Aurora* and Medusa, communication between processors took place using an overlay network, with dynamic bindings between operators and flows. The two projects were recently merged and all their feature have been included into the Borealis stream processor [Abadi et al. 2005].

**Gigascope.** Gigascope [Cranor et al. 2002; Cranor et al. 2003] is a DSMS specifically designed for network applications, including traffic analysis, intrusion detection, performance monitoring, etc. The main concern of Gigascope is to provide high performance for the specific application field it has been designed for.

Gigascope defines a declarative, SQL-like language, called GSQL, which includes only filters, joins, group by, and aggregates. Interestingly, it uses processing techniques that are very different from those of other data stream systems. In fact, to deal with the blocking nature of some of its operators, it does not introduce the concept of windows. Instead, it assumes that each information item (tuple) of a flow contains at least an *ordered attribute* i.e. an attribute that monotonically increases or decreases as items are produced by the source of the flow, for example a timestamp defined w.r.t. an absolute time but also a sequence number assigned at source. Users can specify which attributes are ordered, as part of the data definition, and this information is used during processing. For example, the join operator, by definition, must have a constraint on an ordered attribute for each involved stream.

These mechanisms make the semantics of processing easier to understand, and more similar to that of traditional SQL queries. However, they can be applied only on a limited set of application domains, in which strong assumptions on the nature of data and on arrival order can be done.

Gigascope translates GSQL rules into basic operators, and composes them into a plan for processing. It also uses optimization techniques to re-arrange the plan according to the nature and the cost of each operator.

**Stream Mill.** Stream Mill [Bai et al. 2006] is a general-purpose DSMS based on ESL, a SQL-like language with ad-hoc constructs designed to easily support flows of information. ESL allows its users to define highly expressive transforming rules by mixing the declarative syntax of SQL with the possibility to create custom aggregates in an imperative way directly from within the language. To do so, the language offers the possibility to create and manage custom tables, which can be used as variables of a programming language during information flow processing. Processing itself is modeled as a sort of loop on information items, which users control by associating behaviors to the beginning of processing, to internal iterations,

and to the end of processing. Behaviors are expressed using the SQL language.

This approach is somehow similar to the one we described for CQL: flows are temporarily seen as relational tables and queried with traditional SQL rules. ESL extends this idea by allowing users to express exactly how tables have to be managed during processing. Accordingly, the selection and consumption policies can be programmed rule by rule. Since processing is performed on relational tables, the ordering of produced items may be separate from the input order.

Besides this, Stream Mill keeps the traditional architecture of a database system; it compiles rules into a query plan whose operators are then dynamically scheduled. The implementation of Stream Mill as described in [Bai et al. 2006] shows a centralized processing, where the engine exchanges data with applications using a client-server communication paradigm.

### 4.3 Complex Event Processing Systems

**Traditional content-based publish-subscribe systems.** As stated in Section 2 traditional content-based publish-subscribe middleware [Eugster et al. 2003; Mühl et al. 2006] represent one of the historical basis for complex event processing systems. In the publish-subscribe model, information items flow into the system as messages coming from multiple *publishers* while simple detecting rules define interests of *subscribers*. The detection part of each rule can only take into account single information items and select them using constraints on their content; the action part simply perform information delivery. These kinds of rules are usually called *subscriptions*. The exact syntax used to represent messages and subscriptions varies from system to system, but the expressive power of the system remains similar, hence we group and describe them together.

Many publish-subscribe systems [Carzaniga et al. 2000; Strom et al. 1998; Chand and Felber 2004; Pietzuch and Bacon 2002; Fiege et al. 2002; Balter 2004; Cugola et al. 2001; Pallickara and Fox 2003] have been designed to work in large scale scenarios. To maximize message throughput and to reduce as much as possible the cost of transmission, such systems adopt a distributed core, in which a set of brokers, connected in a dispatching network, cooperate to realize efficient routing of messages from publishers to subscribers.

**Rapide.** Rapide [Luckham 1996; Luckham and Vera 1995] is considered one of the first steps toward the definition of a complex event processing system. It consists of a set of languages and a simulator that allows users to define and execute models of system architectures. Rapide is the first system that enables users to capture the timing and causal relationships between events: in fact, the execution of a simulation produces a *causal history*, where relationships between events are made explicit.

Rapide models an architecture using a set of components, and the communication between components using events. It embeds a complex event detection system, which is used both to describe how the detection of a certain pattern of events by a component brings to the generation of others, and to specify properties of interest for the overall architecture. The pattern language of Rapide includes most of the operators presented in Section 3: in particular it defines conjunctions, disjunctions, negations, sequences, and iterations, with timing constraints. Notice that

Rapide does not assume the existence of an absolute time (however, event can be timestamped with the values of one or more clocks, if available): as a consequence sequences only take into account a causal order between events. During pattern evaluation, Rapide allows rules to access the state of components; we capture this interaction with the presence of the *Knowledge Base*, in our functional model.

The processing model of Rapide captures all possible set of events matching a given pattern, which means that it applies a multiple selection and a zero consumption policies. Notice that complex events can be re-used in the definition of other events.

Since Rapide also uses its pattern language to define general properties of the system, it embeds operators that are not found in other proposals, like logical implications and equivalences between events. To capture these relations, Rapide explicitly stores the history of all events that led to its occurrence. This is made easy by the fact that simulations are executed in a centralized environment.

**GEM.** GEM [Mansouri-Samani and Sloman 1993; 1997] is a generalized monitoring language for distributed systems. It has been designed for distributed deployment on monitors running side-by-side to event generators. Each monitor is configured with a script containing detecting rules that detect patterns of events occurring in the monitored environment and perform actions when a detection occurs. Among the possible actions that a monitor can execute, the most significant is notification delivery: using notifications each monitor becomes itself the generator of an information flow, thus enabling detection of events that involve multiple monitors in a distributed fashion. It is worth noting, however, that distribution is not completely automatic, but needs the static configuration of each involved monitor. To make distributed detection possible, the language assumes the existence of a synchronized global clock. An *event-specific delaying technique* is used to deal with communication delays; this technique allows detection to deal with out-of-order arrival of information items.

GEM supports filtering of single information items, parameterization, and composition using conjunction, disjunction, negation (between two events), and sequence. Iterations are not supported. The time model adopted by GEM considers events as having a duration, and allows users to explicitly refer to the starting and ending time attributes of each event inside the rules. This allows users to define fixed and landmark windows for pattern evaluation. Moreover, an implicit sliding window is defined for each rule having blocking operators; the size of the window is not programmable by the users but depends from system constraints and load (i.e., main memory and input rates of events). Since the definition of a composite event has to explicitly mention all the components that need to be selected, the maximum number of events captured at each processing cycle is determined by deployed rules (i.e., *Seq* is bounded.

Detection is performed using a tree based structure; the action part of a rule includes, beside notification delivery, also the possibility to execute external routines and to dynamically activate or deactivate rules. The GEM monitor has been implemented in C++ using the REGIS/DARWIN [Mansouri-Samani and Sloman 1996; Magee et al. 1994] environment that provides its communication and configuration platform.

**Padres.** Padres [Li and Jacobsen 2005] is an expressive content-based publish-subscribe middleware providing a form of complex event processing. As in traditional publish-subscribe it offers primitives to publish messages and to subscribe to specified information items, using detecting rules expressed in a pattern-based language. Unlike traditional publish-subscribe, expressed rules can involve more than one information item, allowing logic operators, sequences, and iterations. Padres uses a time model with an absolute time, which can be addressed in rules to write complex timing constrains. This way it provides fixed, landmark, and sliding windows. All possible set of events satisfying a rule are captured; so we can say that the system adopts a multiple selection and zero consumption policies. The presence of an operator for iterations together with a timing window, makes it impossible to determine the maximum number of items selected by each rule a priori (*Seq* is unbounded).

Processing of rules, including those involving multiple information items, is performed in a fully distributed way, exploiting a hierarchical overlay network. To do so, the authors propose an algorithm to decompose rules in order to find a convenient placement for each operator composing them; its goal is that of partially evaluating rules as soon as possible during propagation of information elements from sources to recipients. Assuming that applying an operator never increases the amount of information to be transmitted, this approach reduces network usage.

Information processing performed inside single nodes is realized using Rete trees [Forgy 1982]. After detection, information items are simply delivered to requiring destinations: only juxtaposition of information inside a single message is allowed; more complex actions, like aggregates, cannot be specified.

**DistCED.** DistCED [Pietzuch et al. 2003] is another expressive content-based publish-subscribe system; it is designed as an extension of a traditional publish-subscribe middleware and consequently it can be implemented (at least in principle) on top of different existing systems.

DistCED rules are defined using a detecting language. Like in Padres it is possible to combine single information items using logic operators, sequences, and iterations. DistCED also allows users to specify the time of validity for detection: this is done using a construct that builds up a time-based sliding window. DistCED uses a time model that takes into account the duration of events and uses two different operators to express sequences of events that are allowed to overlap (*weak sequence*), and sequences of events with no overlapping (*strong sequence*). The authors consider duration of events as a simple way to model the timing uncertainty due to transmission; however no details are provided about this.

Processing is performed in a distributed way: rules are compiled into finite state automata and deployed as detectors; each detector is considered as a mobile agent, that can also be split into its components to be better distributed. When a new rule is added, the system checks whether it can be interpreted by exploiting already defined detectors; if not, it creates new automata and pushes them inside the dispatching network. To better adapt dispatching behavior to application requirements, DistCED provides different distribution policies: for example, it can maximize automata reuse or minimize the overall network usage.

**CEDR.** In [Barga et al. 2007] the authors present the foundations of CEDR, de-

fined as a general purpose event streaming system. Different contributions are introduced. First of all, the authors discuss a new temporal model for information flows, based on three different timings, which specify system time, validity time, and occurrence/modification time. This approach enables the formal specifications of various consistency models among which applications can choose. Consistency models deal with errors in flows, such as latency, or out-of-order delivery. In CEDR flows are considered as notification of state updates; accordingly the processing engine keeps an history of received information and sends a notification when the results of a rule changes, thus updating the information provided to connected clients. From this point of view, CEDR strongly resembles DSMSs for Internet update monitoring, like NiagaraCQ and OpenCQ; however, we decided to put CEDR in the class of event processing systems since it strongly emphasizes detection of event occurrences, using a powerful language based on patters, including temporal operators, like sequences.

The presence of a precise temporal model enables authors to formally specify the semantics of each operator of the language. It also greatly influences the design of the language, which presents some interesting and singular aspects: windows are not explicitly provided, while validity constraints are embedded inside most of the operators provided by the system (not only the blocking ones). In the CEDR language, instances of events play a central role: rules are not only specified in term of an event expression, which defines how individual events have to be filtered, combined and transformed; they also embed explicit instance selection and consumption policies (through a special `cancel-when` operator), as well as instance tranformations, which enable users to specify: (i) which specific information items have to be considered during the creation of composite events, (ii) which ones have to be consumed, and (iii) how existing instances have to be transformed after an event detection.

Notice that the presence of a time window embedded into operators, together with the ability to use a multiple selection policy, makes the number of items sent by the *Decider* to the *Producer* unbounded a priori. Consider for example a simple conjunction of items $(A \wedge B)$ in a time window $W$ with a multiple selection policy: it is not possible to know a priori how many couples of items $A$ and $B$ will be detected in $W$.

**Cayuga.** Cayuga [Brenna et al. 2007] is a general purpose event monitoring system. It is based on a language called CEL (Cayuga Event Language). The structure of the language strongly resembles that of traditional declarative languages for database: it is structured in a *SELECT* clause that filters input stream, a *FROM* clause that specifies a *streaming expression*, and a *PUBLISH* clause that produces the output. It includes typical SQL operators and constructs, like selection, projection, renaming, union, and aggregates. Despite its structure, we can classify CEL as a detection language: in fact, the streaming expression contained in the *FROM* clause enables users to specify detection patterns, including sequences (using the *NEXT* binary operator) as well as iterations (using the *FOLD* operator). Notice that CEL does not introduce any windowing operator. Complex rules can be defined by combining (nesting) simpler ones. Interestingly, all events are considered as having a duration, and much attention is paid in giving a precise semantics for

operator composability, so that all defined expressions are left-associated, or can be broken up into a set of left-associated ones. To do so, the semantics of all operators is formally defined using a query algebra [Demers et al. 2006]; the authors also show how rules can be translated into non deterministic automata for event evaluation. Different instances of automata work in parallel for the same rule, detecting all possible set of events satisfying the contraints in the rule. This implicitly defines a multiple selection policy; Cayuga uses a zero consumption policy, as events can be used many times for the detection of different complex event occurrences. Since Cayuga is explicitly designed to work on large scale scenarios, the authors put much effort in defining efficient data structures: in particular, they exploit custom heap management, indexing of operator predicates and reuse of shared automata instances. Since detecting automata of different rules are strictly connected with each other, Cayuga does not allow distributed processing.

**NextCEP.** NextCEP [Schultz-Moeller et al. 2009] is a distributed complex event processing system. Similarly to Cayuga, it uses a language that includes traditional SQL operators, like filtering and renaming, together with pattern detection operators, including sequences and iterations. Detection is performed by translating rules into non deterministic automata, that strongly resemble those defined in Cayuga in structure and semantics. In NextCEP, however, detection can be performed in a distributed way, by a set of strictly connected node (*clustered* environment). The main focus of the NextCEP project is on rule optimization: in particular, the authors provide a cost model for operators, that defines the output rate of each operator according to the rate of its input data. NextCEP exploits this model for *query rewriting*, a process that changes the order in which operators are evaluated without changing the results of rules. The objective of query rewriting is that of obtaining the best possible evaluation plan, i.e. the one that minimizes the usage of CPU resources and the processing delay.

**PB-CED.** In [Akdere et al. 2008], the authors present a system for complex event detection using data received from distributed sources. They call this approach Plan-Based Complex Event Detection (PB-CED). The emphasis of the work is on defining an efficient plan for the evaluation of rules, and on limiting as much as possible transmissions of useless data from sources.

PB-CED uses a simple detecting language, including conjunctions, disjunctions, negations, and sequences, but not iterations, or reuse of complex events in pattern. The language does not offer explicit windowing constraints, but embeds time limits inside operators, like. PB-CED offers a timing model that takes into account the duration of events.

PB-CED compiles rules into non deterministic automata for detection and combines different automata to form a complex detection plan. Although PB-CED uses a centralized detection, in which the plan is deployed on a single node, simple architectural components are deployed near sources; these components just store information received from sources (without processing them) and are directly connected with the detecting node. Components near sources may operate both in push and in pull-based mode. The authors introduce a cost model for operators and use it to dynamically generate the plan with minimum cost. Since each step in the plan involves acquisition and processing of a subset of the events, the plan is

created with the basic goal of postponing the monitoring of high frequency events to later steps in the plan. As such, processing the higher frequency events conditional upon the occurrence of lower frequency ones eliminates the need to communicate the former (requested in pull-based mode) in many cases. Interestingly, the plan optimization procedure can take into account different parameters, besides the cost function; for example it can consider local storage available at sources, or the degree of timeliness required by the application scenario.

**Raced.** Raced [Cugola and Margara 2009] is a distributed complex event processing middleware. It offers a very simple detecting language which strongly resembles the one described in Padres, including conjunctions, disjunctions, negations, parameters, and sliding windows. It does not offer any processing capability to compute aggregates. Like Padres, Raced is designed for large scale scenarios, and supports distributed detection.

Interestingly, Raced uses a hybrid push/pull approach for the communication between the different detecting nodes. More in particular, nodes are organized in a hierarchical (tree) topology; complex subscriptions are delivered from the root to the leaves, and progressively decomposed according to the information advertised at each node. Each node combines information items coming from its children and delivers composed information to its parent. Each node continuously monitor the rate of information items produced at each child; rare information are then asked in a push way (i.e. they are received immediately, as they are available), while other information is asked only when needed. Using this approach Raced gets the benefits of distributed processing (load is split, and information is filtered near the sources), while limiting the transmission of unneeded information from node to node.

Raced relies on a simple timing model in which all elements receive a timestamp that represent the absolute time of occurrence. This way complex events can be used to form other (more complex) ones. Its processing model captures all possible occurrences of events, using a multiple selection and zero consumption policies.

**Amit.** Amit is an application development and runtime control tool intended to enable fast and reliable implementation of reactive and proactive applications [Adi and Etzion 2004]. Amit embeds a component, called *situation manager*, specifically designed to process notifications received from different sources in order to detect patterns of interests, called *situations*, and to forward them to demanding *subscribers*. The situation manager is based on a strongly expressive and flexible detecting language, which includes conjunctions, negations, parameters, sequences, and repetitions (in the form of *counting* operators). Timing operators are introduced as well, enabling periodic evaluation of rules.

Amit introduces the concept of *lifespan* as a valid time window for the detection of a situation. A lifespan is defined as an interval bounded by two events called *initiator* and *terminator*. Amit's language enables user-defined policies for lifespans management: these policies specify whether multiple lifespans may be open concurrently (in presence of different valid initiators) and which lifespans are closed by a terminator (e.g., all open ones, only the most recently opened, etc.).

At the same time Amit introduces programmable event selection policies by applying a quantifier to each operator. Quantifiers specify which events an operator should refer to (e.g., the first, the last, all valid ones). Similarly, Amit allows pro-

grammable consumption policies by associating a consumption condition to each operator.

It is worth mentioning that detected situations can be used as part of a rule, as event notifications. This enables the definition of *nested situations*. Being focused on the detection of patterns, and not on complex processing of information, Amit does not include aggregates.

Amit has been implemented in Java and is being used as the core technology behind the E-business Management Service of IBM Global Services [IBM 2010]. It uses a centralized detection strategy: all events are stored at a single node, partitioned according to the lifespans they may be valid for. When a terminator is received, Amit evaluates whether all the conditions for the detection of a situation have been met and, if needed, notifies subscribers.

**Sase.** Sase [Wu et al. 2006] is a monitoring system designed to perform complex queries over real-time flows of RFID readings.

Sase defines detecting rules language based on patterns; each rule is composed of three parts: *event*, *where* and *within*. The event clause specifies which information items have to be detected and which are the relations between them; relations are expressed using logic operators and sequences. The where clause defines constraints on the inner structure of information items included into the event clause: referring to the list of operators of Section 3, the where clause is used to define selections for single information items. Finally, the within clause expresses the time of validity for the rule; this way it is possible to define time-based, sliding windows. The language adopted by Sase allows only detection of given patterns of information items; it does not include any notion of aggregation.

Sase compiles rules into a query plan having a fixed structure: it is composed of six blocks, which sequentially process incoming information elements realizing a sort of pipeline: the first two blocks detect information matching the logic pattern of the event clause by using finite state automata. Successive blocks check selections constraints, windows, negations, and build the desired output. Since all these operators explicitly specify the set of events to be selected, it is not possible to capture unbounded sequences of information items (*Seq* is bounded).

**Sase+.** Sase+ [Gyllstrom et al. 2008; Agrawal et al. 2008] is an expressive event processing language from the authors of Sase. Sase+ extends the expressiveness of Sase, by including iterations and aggregates as possible parts of detecting patterns.

Non deterministic automata are used for pattern detection, as well as for providing a precise semantics of the language. An interesting aspect of Sase+ is the possibility for users to customize selection policies using *strategies*. Selection strategies define which events are valid for an automaton transition: only the next one (if satisfying the rule's constraints), or the next satisfying one, or all satisfying ones that satisfy. Consumption of events, instead, is not taken into account.

An important contribution of [Agrawal et al. 2008] is the formal analysis of the expressive power of the language, and of the complexity of its detecting algorithm. On one side this analysis enables a direct comparison with other languages (e.g., traditional regular expressions, Cayuga language). On the other side, the analysis applies only to a limited set of pattern-based language and cannot yet capture the multitude of languages defined in the IFP domain.

**Peex.** Peex (Probabilistic Event Extractor) [Khoussainova et al. 2008] is a system designed for extracting complex events from RFID data. Peex presents a pattern-based rule language with four clauses: *FORALL*, which defines the set of readings addressed in the rule, *WHERE*, which specifies pattern constraints, *CREATE EVENT* and *SET*, which define the type of the event to be generated and set the content of its attributes. Patterns may include conjunctions, negations, and sequences, but not iterations.

The main contribution of Peex is its support for data uncertainty: in particular, it addresses data errors and ambiguity, which can be frequent in the specific application domain of RFID data. To do so, it changes the data model, by assigning a probability to all information items. More in details, when defining rules, system administrators can associate a confidence to the occurrence and attribute values of the defined composite events, as functions of composing ones. For example, they can state that, if three sequential readings are detected in a given time window, then an event "John enters room 50" occurs with a probability of 70%, while an event "John enters room 51" occurs with probability of 20% (probabilities need not sum to 100%). Peex enable users to re-use event definitions in the specification of others: to compute the probability of composite events, it fully takes into account the possible correlations between components. Another interesting aspect of Peex is the possibility to produce *partial events*, i.e. to produce an event even if some of its composing parts are missing. Obviously, the confidence on the occurrence of a partial event is lower than if all composing events were detected. Partial events are significant in the domain of RFID readings since sometimes source may fail in detecting or transmitting an event.

From an implementation point of view, Peex uses a relation DBMSs, where it stores all information received from sources and information about confidence. Rules are then translated into SQL queries and run periodically. For this reason, the detection time of events may be different from real occurrence time.

### 4.4 Commercial Systems

**Aleri Streaming Platform.** The Aleri Streaming Platform [Aleri 2009] is an IFP system that offers a simple imperative, graphical language to define processing rules, by combining a set of predefined operators. To increase system's expressiveness custom operators can be defined using a scripting language called Splash, which includes the capability of defining variables to store past information items, so that they can be referenced for further processing. Pattern detection operators are provided as well, including sequences. Notice, however, that pattern matching can appear in the middle of a complex computation, and that sequences may use different attributes for ordering, not only timestamps. As a consequence, the semantics of output ordering does not necessarily reflect timing relationships between input items.

The platform is designed to scale by exploiting multiple cores on a single machine or multiple machines in a clustered environment. However, no information is provided on the protocols used to distribute operators. Interestingly, the Aleri Streaming Platform is designed to easily work together with other business instruments: probably the most significant example is the Aleri Live OLAP system, which

extends traditional OLAP solutions [Chaudhuri and Dayal 1997] to provide near real time updates of information. Aleri also offers a development environment that simplifies the definition of rules and their debugging. Additionally, Aleri provides adapters to enable different data formats to be translated into flows of items compatible with the Aleri Streaming Platform, together with an API to write custom programs that may interact with the platform using either standing or one-time queries.

**Coral8 CEP Engine.** The Coral8 CEP Engine [Coral8 2009; 2010], despite its name, can be classified as a data stream system. In fact, it uses a processing paradigm in which flows of information are transformed through one or more processing steps, using a declarative, SQL-like language, called CCL (Continuous Computation Language). CCL includes all SQL statements; in addition it offers clauses for creating time-based or count-based windows, for reading and writing data in a defined window, and for delivering items as part of the output stream. CCL also provides simple constructs for pattern matching, including conjunctions, disjunctions, negations, and sequences; instead, it does not offer support for repetitions. Like in Aleri, the Coral8 engine does not rely upon the existence of an absolute time model: users may specify, stream by stream, the processing order (e.g., increasing timestamp value, if a timestamp is provided, or arrival order). The results of processing can be obtained in two ways: by subscribing to an output flow (push), or by reading the content of a *public* window (pull).

Together with its CEP Engine, Coral8 also offers a graphical environment for developing and deploying, called Coral8 Studio. This tool can be used to specify data sources and to graphically combine different processing rules, by explicitly drawing a plan in which the output of a component becomes the input for others. Using this tool, all CCL rules become the primitive building blocks for the definition of more complex rules, specified using a graphical, plan-based, language.

Like Aleri, the Coral8 CEP engine may execute in a centralized or clustered environment. The support for clustered deployment is used to increase the availability of the system, even in presence of failures. It is not clear, however, which policies are used to distribute processing on different machines. Load shedding is implemented by allowing administrator to specify a maximum allowed rate for each input stream.

During early 2009, Coral8 merged with Aleri. The company now plans a combined platform and tool set under the name of Aleri CEP.

**StreamBase.** StreamBase [Streambase 2009] is a software platform that includes a data stream processing system, a set of adapters to gather information from heterogeneous sources, and a developer tool based on Eclipse. It shares many similarities with the Coral8 CEP Engine: in particular, it uses a declarative, SQL-like language for rule specification, called StreamSQL [Streambase 2010]. Beside traditional SQL operators, StreamSQL offers customizable time-based and count-based windows. Plus, it includes a simple pattern based language that captures conjunctions, disjunctions, negations, and sequences of items.

Operators defined in StreamSQL can be combined using a graphical plan-based rule specification language, called EventFlow. User-defined functions, written in Java or C++, can be easily added as custom aggregates. Another interesting feature

is the possibility to explicitly instruct the system to permanently store a portion of processed data for historical analysis.

StreamBase supports both centralized and clustered deployments; networked deployments can be used as well, but also to provide high availability in case of failures. Users can specify the maximum load for each used server, but the documentation does not specify how the load is actually distributed to meet these constraints.

**Oracle CEP.** Oracle [Oracle 2009] launched its event-driven architecture suite in 2006 and added BEA's WebLogic Event Server to it in 2008, building what is now called "Oracle CEP", a system that provides real time information flow processing. Oracle CEP uses CQL as its rule definition language, but, similarly to Coral8 and StreamBase, it adds a set of relation-to-relation operators designed to provide pattern detection, including conjunctions, disjunctions, and sequences. An interesting aspect of this pattern language is the possibility for users to program the selection and consumption policies of rules.

Like in Coral8 and StreamBase, a visual, plan-based language is also available inside a development environment based on Eclipse. This tool enables users to connect simple rules into a complex execution plan. Oracle CEP is integrated with existing Oracle solutions, which includes technology for distributed processing in clustered environment, as well as tools for analysis of historical data.

**Esper.** Esper [Esper 2009] is considered the leading open-source CEP provider. Esper defines a rich declarative language for rule specification, called EPL (Event Processing Language). EPL includes all the operators of SQL, adding ad-hoc construct for windows definition and interaction, and for output generation. The Esper language and processing algorithm are integrated into the Java and .Net (NEsper) as libraries. Users can install new rules from their programs and then receive output data either in a push-based mode (using listeners) or in a pull-based one (using iterators).

EPL embeds two different ways to express patterns: the first one exploits so called EPL Patterns, that are defined as nested constraints including conjunctions, disjunctions, negations, sequences, and iterations. The second one uses flat regular expressions. The two syntax offer the same expressiveness. An interesting aspect of Esper pattern is the possibility to explicitly program event selection policies, exploiting the *every* and *every-distinct* modifiers.

Esper supports both centralized and clustered deployments; in fact, using the EsperHa (Esper High Availability) mechanisms it is also possible to take advantage of the processing power of different, well-connected, nodes, to increase the system's availability and to share the system's load according to customizable QoS policies.

**IBM WebSphere Business Events.** IBM acquired CEP pioneer system AptSoft during 2008 and renamed it WebSphere Business Events [IBM 2008]. Today, it is a system fully integrated inside the WebSphere platform, which can be deployed on clustered environment for faster processing. IBM WebSphere Business Events provides a graphical front-end, which helps users writing rules in a pattern-based language. Such a language allows detection of logical, causal, and temporal relationships between events, using an approach similar to the one described for Rapide and Tibco Business Events (see below).

**Tibco Business Events.** Tibco Business Events [Tibco 2009] is another widespread complex event processing system. It is mainly designed to support enterprise processes and to integrate existing Tibco products for business process management. To do so, Tibco Business Events exploits the pattern-based language of Rapide, which enables the specification of complex patterns to detect occurrences of events and the definition of actions to automatically react after detection. Interestingly, the architecture of Tibco Business Events is capable of decentralized processing, by defining a network of event processing agents: each agent is responsible for processing and filtering events coming from its own local scope. This allows, for example, correlating multiple RFID readings before their value is sent to other agents.

**IBM System S.** In May 2009, IBM announced a *Stream Computing Platform*, called System S [Amini et al. 2006; Wu et al. 2007; Jain et al. 2006]. The main idea of System S is that of providing a computing infrastructure for processing large volumes of possibly high rate data streams to extract new information. The processing is split into basic operators, called *Processing Elements* (PEs): PEs are connected to each other in a graph, thus forming complex computations. System S accept rules specified using a declarative, SQL-like, language called SPADE [Gedik et al. 2008], which embeds all traditional SQL operator for filtering, joining, and aggregating data. Rules written in SPADE are compiled into one or more PEs. However, differently from most of the presented systems, System S allows users to write their own PEs using a full featured programming language. This way users can write virtually every kind of function, explicitly deciding when to read an information from an input stream, what to store, how to combine information, and when to produce new information. This allows the definition of component performing pattern-detection, which is not natively supported by SPADE.

System S is designed to support large scale scenarios by deploying the PEs in a clustered environment, in which different machine cooperate to produce the desired results. Interestingly System S embeds a monitoring tool that uses past information about processing load to compute a better processing plan and to move operators from site to site to provide desired QoS.

**Other commercial systems.** Other widely adopted commercial systems exist, for which, unfortunately, documentation or evaluation copies are not available. We mention here some of them.

**IBM WebSphere Business Events.** IBM acquired CEP pioneer system AptSoft during 2008 and renamed it WebSphere Business Events [IBM 2008]. Today, it is a system fully integrated inside the WebSphere platform, which can be deployed on clustered environment for faster processing. IBM WebSphere Business Events provides a graphical front-end, which helps users writing rules in a pattern-based language. Such a language allows detection of logical, causal, and temporal relationships between events, using an approach similar to the one described for Rapide and Tibco Business Events.

**Event Zero.** A similar architecture, based on connected event processing agents, is used inside Event Zero [EventZero 2009], a suite of products for capturing and processing information items as they come. A key feature of Event Zero is its ability of supporting real-time presentations and analysis for its users.

**Progress Apama Event Processing Platform.** The Progress Apama Event Processing Platform [Progress-Apama 2009] has been recognized as a market leader for its solutions and for its strong market presence [Gualtieri and Rymer 2009]. It offers a development tool for rule definition, testing and deployment, and a high-performance engine for detection.

## 5. DISCUSSION

The first consideration that emerges from the analysis and classification of existing IFP systems done in the previous section is a distinction between those systems that mainly focus on data processing and those that focus on event detection[4].

This distinction is clearly captured in the data model (Table III), by the nature of items processed by the different systems. Moreover, by looking at the nature of flows, we observe that systems focusing on data processing usually manage homogeneous flows, while systems focusing on event detection allow different information items to be part of the same flow.

If we look at the rule and language models (Tables III and IV), we may also notice how the systems that focus on data processing usually adopt transforming rules, defined through declarative or imperative languages, including powerful windowing mechanisms together with a join operator to allow complex processing of incoming information items. Conversely, systems focusing on event processing usually adopt detecting rules, defined using pattern-based languages that provide logic, sequence, and iteration operators as a mean to capture the occurrence of relevant events.

Another issue that appear evident by looking at the data and rule models (Table III) is the fact that data uncertainty and probabilistic rules have been rarely explored in existing IFP systems. Since these aspects are critical in many IFP applications, i.e., when data coming from sources may be imprecise or even incorrect, we think that support for uncertainty deserves more investigation. It could increase the expressiveness and adaptability, and hence the diffusion, of IPF systems.

Coming back to the distinction between DSP and CEP, the time model (Table III) emphasizes the central role of time in event processing: in fact, all systems designed to detect composite events introduce an order among information items, which can be a partial order (*causal*), or a total order (using an *absolute* or an *interval* semantics). Conversely, stream processing systems often rely on a *stream-only* time model: timestamps, when present, are mainly used to partition input streams using windows; then, processing is performed using relational operators inside windows.

The importance of time in event detection becomes even more evident by looking at Table IV: almost all systems working with events include the sequence operator, and some of them also provide the iteration operator. These operators, instead, are not provided by stream processing systems, at least those coming from the research. As it emerges from Table IV, in fact, commercial systems adopt a peculiar approach. While they usually adopt the same stream processing paradigm of DSP research prototypes, they also embed pattern-based operators, including sequence and it-

[4]This distinction does not necessarily map to the grouping of systems we adopted in Section 4. There are active databases which focus on event detection, while several commercial systems, which classify themselves as CEP, actually focus more on data processing than on event notification.

eration. While at first this approach could appear the most effective to combine processing abstractions coming from the two world, a closer look to the languages proposed so far reveals many open issues. In particular, since all processing is performed on relational tables defined through windows, it is often unclear how the semantics of operators for pattern detection maps to the partitioning mechanisms introduced by windows. In general, it looks like a bunch of mechanisms were put together without putting too much effort at integrating them and at studying the best and minimal set of operators to offer both DSP and CEP processing support in a single language. In general, we can observe that this research area is new and neither the academy nor the various companies involved have found how to combine in a unifying proposal the two processing paradigms [Chakravarthy and Adaikkalavan 2008].

If we focus on the interaction style, we notice (Table II) that the push-based style is the most used both for observation and notification of items. Only a few research systems adopt pull-based approaches: some of them (NiagaraCQ and OpenCQ) motivate this choice through the specific application domain in which they work, i.e., monitoring updates of web pages, where the constraint on timeliness is less strict. Among the systems focusing on such domain, only PB-CED is able to dynamically decide the interaction style to adopt for data gathering according to the monitored load.

As for the notification model, different commercial systems use an hybrid push/pull interaction style: this is mainly due to the integration with tools for the analysis of historical data, which are usually accessed on-demand, using pull-based queries. Finally, we observe that almost all systems that allow distributed processing adopt a push-based forwarding approach for sending information from processor to processor. The only exception is represented by Raced, which switches between push and pull dynamically, to adapt the forwarding model to the actual load in order to minimize bandwidth usage and increase throughput.

Closely related with the interaction style is the deployment model. Looking at Table II we may notice a peculiarity that was anticipated in Section 2: the few systems that provide a networked implementation to perform filtering, correlation, and aggregation of data directly in network, focus on event detection. It is an open issue if this situation is an outcome of the history that brought to the development of IFP systems from different communities, having different priorities, or if it has to do with some technical aspect, like the difficulty of using a declarative, SQL-like language in a networked scenario, in which distributed processing and minimization of communication costs are the main concern. Clustered deployments, instead, have been investigated both in the DSP and CEP domains.

Along the same line, we may notice (Table I) how the systems that support a networked deployment do not provide a knowledge base and vice versa. In our functional model, the knowledge base is an element that has the potential to increase the expressiveness of the system but that could be hard to implement in a fully networked scenario. This may explain the absence of such a component in those systems that focus on efficient event detection in a strongly distributed and heterogeneous network.

Another aspect related with the functional model of an IFP system has to do

with the presence or absence of the *Clock* component. Table I shows how half of the systems include such a (logical) component and consequently allows rules to fire periodically, while the remaining systems provide a purely reactive behavior. The presence of this component seems to be independent from the class of the system.

We cannot say the same if we focus on load shedding. Indeed, Table I shows that all systems providing such a mechanism belong to the DSP world. Load shedding, in fact, is used to provide guaranteed performance to users, and this is strictly related with the issue of agreeing QoS levels between sources, sinks, and the IFP system. While all CEP systems look at maximize performance through efficient processing algorithms and distribution strategies, they never explicitly take into account the possibility of negotiating specific QoS levels with their clients. It is not clear to us whether the adoption of these two different approaches (i.e., negotiated QoS vs. fixed, usually best-effort, QoS) really depends from the intrinsically different nature of data and event processing, or if it simply depends from the different attitudes and backgrounds of the communities working on the two kinds of systems.

Another aspect differentiating CEP from DSP systems has to do with the possibility of performing recursive processing. As shown by Table I, indeed, such mechanism is often provided by systems focusing on events, while it is rarely offered by systems focusing on data transformation. This can be explained by observing how recursion is a fundamental mechanism to define and manage hierarchies of events, with simple events coming from sources used to define and detect first-level composite events, which in turn may become part of higher-level composite events.

Another aspect that impacts the expressiveness of an IFP system is its ability of adapting the processing model to better suit the need of its users. According to Table I this is a mechanism provided by a few systems, while we feel that it should be offered by all of them. Indeed, while the multiple selection policy (with zero consumption) is the most common, in some cases it is not the most natural. As an example, in the fire alarm scenario described in Section 3.1, a single selection policy would be better suited; in presence of a smoke event followed by three high temperature events a single fire alarm, not multiple, should be generated.

Similarly, in Section 3.1 we observed that the maximum length of the sequence *Seq* of information items that exits the *Decider* and enters the *Producer* has an impact on the expressiveness of the system. Here we notice how this characteristic allows to draw a sharp distinction between traditional publish-subscribe systems and all other systems, which allows multiple items to be detected and combined. On the other hand, it is not clear if there is a real difference in expressiveness between those rule languages that result in a "bound" length for sequence *Seq* and those that have it "unbound". As an example, a system that allows to combine (e.g., join) different flows of information in a time-based window requires a potentially infinite *Seq*. The same is true for systems that provide an unbounded iteration operator. Conversely, a system that would provide count-based windows forcing to use them each time the number of detectable items grows, would have a bounded *Seq*. While it seems that the first class of languages is more expressive than the second, how to formally define, measure, and compare the expressiveness of IFP rule languages is still an open issue, which requires further investigation.

As a final consideration, we may notice (Table I) how the ability to dynamically

change the set of active rules was available in most active databases developed years ago, while it disappeared in more recent systems (with the remarkable exception of GEM and Tribeca). This is a functionality that could be added to existing systems to increase their flexibility.

## 6. RELATED WORK

In this section we briefly discuss the results of on-going or past research to provide a more complete understanding of the IFP domain. In particular we cover four different aspects: (i) we present works that study general mechanisms for IFP; (ii) we review specific models used to describe various classes of systems, or to address single issues; (iii) we provide an overview of systems presenting similarities with IFP ones; (iv) we discuss existing attempts to create a standard for the IFP domain.

### 6.1 General Mechanisms for IFP

Many researchers focused on developing general mechanisms for IFP, by studying (i) processing strategies, (ii) operator placement and load balancing algorithms, (iii) communication protocols for distributed rule processing, (iv) techniques to provide adequate levels of QoS, and (v) mechanisms for high availability and fault tolerance.

Query optimization has been widely studied by the database community [Jarke and Koch 1984; Ioannidis 1996; Chaudhuri 1998]. Since in IFP systems different rules co-exist, and may be evaluated simultaneously when new input is available, it becomes important to look at the set of all deployed rules, to minimize the overall processing time and resource consumption. This issue is sometimes called the *multi-query optimization* problem [Sellis 1988]. Different proposals have emerged to address this problem: they include shared plans [Chen et al. 2000], indexing [Chandrasekaran and Franklin 2002; Madden et al. 2002; Brenna et al. 2007], and query rewriting [Schultz-Moeller et al. 2009] techniques. In the area of publish-subscribe middleware, the throughput of the system has always been one of the main concerns: for this reason different techniques have been proposed to increase message filtering performance [Aguilera et al. 1999; Carzaniga and Wolf 2003; Campailla et al. 2001].

When the processing is distributed among different nodes, beside the definition of an optimal execution plan, a new problem emerges: it is the *operator placement* problem, that aims at finding the optimal placement of entire rules or single operators (i.e., rule fragments) at the different processing nodes, to distribute load and to provide the best possible performance according to a system-defined or user-defined cost function. Some techniques have been proposed to address this issue [Balazinska et al. 2004; Ahmad and Çetintemel 2004; Li and Jacobsen 2005; Abadi et al. 2005; Kumar et al. 2005; Pietzuch et al. 2006; Zhou et al. 2006; Amini et al. 2006; Repantis et al. 2006; Wolf et al. 2008; Khandekar et al. 2009; Cugola and Margara 2009], each one adopting different assumptions on the underlying environment (e.g., cluster of well connected nodes or large scale scenarios with geographical distribution of processors) and system properties (e.g., if operators can be replicated or not). Most importantly, they consider different cost metrics; most of them consider (directly or indirectly) the load at different nodes, thus realizing load balancing, while others take into account network parameters, like latency and/or bandwidth. A classification of existing works can be found in [Lakshmanan et al. 2008].

In distributed scenarios it is of primary importance to define efficient communication strategies between the different actors (sources, processors, and sinks). While most existing IFP systems adopt a push-based style of interaction among components, some works have investigated different solutions. A comparison of push and pull approaches for Web-based, real-time event notifications is presented in [Bozdag et al. 2007]; some works focused on efficient data gathering using a pull-based approach [Roitman et al. 2008], or a hybrid interaction, where the data to received is partitioned into push parts and pull ones according to a given cost function [Bagchi et al. 2006; Akdere et al. 2008]. A hybrid push/pull approach has been proposed also for the interaction between different processing nodes [Cugola and Margara 2009].

Different contributions, primarily from people working on stream processing, focused on the problem of bursty arrival of data (which may cause processors' overload) [Tatbul and Zdonik 2006a], and on providing required levels of QoS. Many works propose load shedding techniques [Tatbul et al. 2003; Babcock et al. 2004; Srivastava and Widom 2004; Chandrasekaran and Franklin 2004; Tatbul and Zdonik 2006b; Chi et al. 2005] to deal with processors' overload. As already mentioned in Section 4, the Aurora/Borealis project allows users to express QoS constraints as part of a rule definition [Ahmad et al. 2005]: such constraints are used by the system to define the deployment of processing operators and to determine the best shedding policies.

Finally, a few works have focused on high-availability and fault tolerance for distributed stream processing. Some of them focused on fail-stop failures of processing nodes [Shah et al. 2004b; Hwang et al. 2005], defining different models and semantics for high-availability according to specific application needs (e.g., the fact that results have to be delivered to sinks at least once, at most once, or exactly once). They proposed algorithms based on replication. Interestingly, [Balazinska et al. 2008] also takes into account network failures and partitioning, proposing a solution in which, in case of a communication failure, a processor does not stop, but continues to produce results based on the only information it is still able to receive; such information will be updated as soon as the communication can be reestablished. This approach defines a trade-off between availability and consistency of data, which can be configured by sinks.

### 6.2 Other Models

As we have seen, IFP is a large domain, including many systems. Although the literature misses a unifying model, which is able to capture and classify all existing works, various contributions are worth mentioning. First, there are the community specific system models, which helped us understand the different visions of IFP and the specific needs of the various communities involved. Second, there are models that focus on single aspects of IFP, like timing models, event representation models, and language related models.

6.2.1 *System Models.* As described in Section 2, two main models for IFP have emerged, one coming from the database community and one coming from the event-processing community.

In the database community active database systems represent the first attempt

to define rules to react to information in real-time, as it becomes available. The research field of active database systems is now highly consolidated and several works exist offering exhaustive descriptions and classifications [McCarthy and Dayal 1989; Paton and Díaz 1999; Widom and Ceri 1996].

To overcome the limitations of active database systems in dealing with high rate flows of external events, the database community gave birth to the Data Stream Managements Systems. Even if research on DSMSs is relatively young, it appears to have reached a good degree of uniformity in its vocabulary and in its design choices. There exist a few works that describe and classify such systems, emphasizing common features and open issues [Babcock et al. 2002; Golab and Özsu 2003]. Probably the most complete model to describe DSMSs comes from the author of Stream [Babu and Widom 2001]; it greatly influenced the design of our own model for a generic IFP system. Stream is also important for its rule definition language, CQL [Arasu et al. 2006], which captures most of the common aspects of declarative (SQL-like) languages for DSMS clearly separating them through its stream-to-relation, relation-to-relation, and relation-to-stream operators. Not surprisingly many systems, even commercial ones [Oracle 2009; Streambase 2009] directly adopt CQL or a dialect for rule definition. However, no single standard has emerged so far for DSMSs languages and competing models have been proposed, either adopting different declarative approaches [Law et al. 2004] or using a graphical, imperative approach [Abadi et al. 2003; Aleri 2009].

Our model has also been developed by looking at currently available event-based systems, and particularly publish-subscribe middleware. An in depth description of such systems can be found in [Mühl et al. 2006; Eugster et al. 2003]. Recently, several research efforts have been put in place to cope with rules involving multiple events; some general model of CEP can be found in [Luckham 2001; Etzion and Niblett 2010].

6.2.2 *Models for Single Aspects of IFP.* A relevant part of our classification focuses on the data representations and rule definition languages adopted by IFP systems, and on the underlying time model assumed for processing. Some works have extensively studied these aspects, and deserve to be mentioned here. As an example, a few papers exist that provide an in depth analysis of specific aspects of languages, like blocking operators, windows, and aggregates [Arasu et al. 2002; Law et al. 2004; Golab and Özsu 2005; Jain et al. 2008]. Similarly, even if a precise definition of the selection and consumption policies that ultimately determine the exact semantics of rule definition languages is rarely provided in existing works, there are some papers that focus on this issue, both in the area of active databases [Zimmer 1999] and in the area of CEP systems [Konana et al. 2004; Adi and Etzion 2004].

In the areas of active databases and event processing systems, much has been said about event representation: in particular a long debate exists on timing models [Adaikkalavan and Chakravarthy 2006; Galton and Augusto 2002], distinguishing between a *detection*, and an *occurrence* semantics for events. The former associates events with a single timestamp, representing the time in which they were detected; it has been shown that this model provides a limited expressiveness in defining temporal constraints and sequence operators. On the other hand, the *occurrence semantics* associates a duration to events. In [White et al. 2007], the

authors present a list of desired properties for time operators (i.e., sequence and repetition); they also demonstrate that it is not possible to define a time model based on the occurrence semantics which is able to satisfy all those properties, unless time is represented using timestamps of unbounded size.

It has been demonstrated that, when the occurrence semantics is used, an infinite history of events should be used by the processing system to guarantee a set of desired properties for the sequences operators [White et al. 2007].

Finally, as we observed in Sections 4 and 5, DSMSs, and particularly commercial ones, have started embedding (usually simple) operators for capturing sequences into a traditional declarative language: this topic has been extensively studied in a proposal for adding sequences to standard SQL [Sadri et al. 2004].

All these works complement and complete our effort.

## 6.3 Related Systems

Besides the systems described in Section 4, other tools exist, which share many similarities with IFP ones.

6.3.1 *Runtime Verification Tools.* Runtime verification [Giannakopoulou and Havelund 2001; Havelund and Rosu 2002; Baresi et al. 2007; Barringer et al. 2004] defines techniques that allow checking whether the run of a system under scrutiny satisfies or violates some given rules, which usually model correctness properties [Leucker and Schallhart 2009].

While at first sight, runtime verification may resemble IFP, we decided not to include the currently available runtime verification tools into our list of IFP systems for two reasons: first, they usually do not provide general purpose operators for information analysis and transformation; on the contrary they are specifically designed only for checking whether a program execution trace satisfies or violates a given specification of the system; second, the specificity of their goal has often led to the design of ad-hoc processing algorithms, which cannot be easily generalized for other purposes.

In particular, as these tools are heavily inspired by model checking [Clarke and Schlingloff 2001], rules are usually expressed in some kind of linear temporal logic [Pnueli 1997]. These rules are translated into *monitors*, usually in the form of finite state or Büchi automata, which read input events coming from the observed system and continuously check for rules satisfaction or violation. According to the type of monitored system, events may regard state changes, communication or timing information. Guarded systems may be instrumented to provide such pieces of information, or they can be derived through external observation. It is worth noting that while the monitored systems are often distributed, the currently available runtime verification systems are centralized.

Focusing on the language, we may observe that temporal logics enable the creation of rules that resemble those that it is possible to define using pattern-based languages, as described in Section 3. In particular, they allow composition of information through conjunctions, disjunctions, negations, and sequences of events. Additionally many works are based on logics that also include timing constraints [Maler et al. 2006; Drusinsky 2000]; this introduces a flexible way to manage time, which can be compared to user defined windows adopted by some IFP systems. An-

other aspect explored in some works is the possibility to express parameterized rules through variables [Stolz 2007], which are usually associated with universal and existential quantifiers.

Finally, we observe a notable difference between IFP systems and runtime verification tools: while the former only deal with the history of past events to produce their output, the latter may express rules that require future information to be entirely evaluated. As the output of runtime verification systems is a boolean expression (indicating whether a certain property is satisfied or violated) different semantics have been proposed to include all the cases in which past information is not sufficient to evaluate the truth of a rule. Some works use a three values logic, where the output of the verification can be *true*, *false*, or *inconclusive* [Bauer et al. 2006b; 2009]. Other works consider a given property as satisfied until it has not been violated by occurred events [Giannakopoulou and Havelund 2001]. Another approach is that of adopting a four values logic, including *presumably true* and *presumably false* truth values [Bauer et al. 2007]. When a violation is detected some of the exisiting tools may also be programmed to execute a user defined procedure [Bauer et al. 2006a], for example to bring back the system in a consistent state.

6.3.2 *Runtime Monitoring Tools.* Similarly to IFP systems, runtime monitoring tools consider not only the satisfaction or violation of properties, as it happens in runtime verification, but also manipulation of data collected from the monitored system. Accordingly, some monitoring tools, even when developed with the specific domain of monitoring in mind, present design choices that can be easily adapted to the more general domain of IFP. For this reason, we included some of them in the list of IFP systems analyzed in Section 4. Others are more strongly bound to the domain they are studied for: we describe them here.

In particular, much effort has been recently put in runtime monitoring of service oriented architectures (SOAs) and, more in particular, of web services: the focus is on the analysis of the quality of service in compositions, to detect bottlenecks and to eliminate them whenever possible. Different rule languages have been proposed for these SOA-oriented monitoring systems; some of them are similar to the pattern-based languages described in Section 3 and express content and timing constraints on the events generated by services [Barbon et al. 2006a; 2006b]; others adopt imperative or declarative languages [Beeri et al. 2007] to express transforming rules. Events usually involve information about service invocations, including the content of exchanged messages. Moreover, runtime monitoring languages often provide constructs to define variables and to manipulate them, making it possible to store and aggregate information coming from multiple invocations.

From an implementation point of view, some systems integrate monitoring rules within the process definition language (e.g., BPEL) that describes the monitored process, thus enabling users to express conditions that must hold at the bounds of service invocations [Baresi and Guinea 2005b; 2005a; Baresi et al. 2008]. Other systems, instead, adopt an autonomous monitor.

6.3.3 *Scalable Distributed Information Management Systems.* Like many IFP systems, distributed information management systems [Van Renesse et al. 2003;

Yalagandula and Dahlin 2004] are designed to collect information coming from distributed sources to monitor the state and the state's changes in large scale scenarios. Such systems usually split the network into non overlapping zones, organized hierarchically. When information moves from layer to layer it is aggregated through user-defined functions, which progressively reduce the amount of data to forward. This way, such systems provide a detailed view of nearby information and a summary of global information. Getting the real data may sometimes require visiting the hierarchy in multiple steps.

If compared with the IFP systems presented in Section 4, distributed information management systems present some differences: first, they are not usually designed to meet the requirements of applications that make strong assumption on timeliness. This also reflects in the interaction style they offer, which is usually pull-based (or hybrid). Second, they are generally less flexible. In fact, they usually focus on data dissemination and not on data processing or pattern detection. Aggregation functions are only used as a summarizing mechanisms, and are not considered as generic processing functions [Van Renesse et al. 2003]. As a consequence they usually offer a very simple API to applications, which simply allows to install new aggregation functions, and to get or put information into the system.

### 6.4 IFP Standardization

Recently, much effort has been put in trying to define a common background for IFP systems. Proposals came mainly from the event processing community, where a large discussion on the topic is undergoing, starting from the book that introduced the term complex event processing [Luckham 2001] and continuing on web sites and blogs [Luckham 2009; Etzion 2009]. An *Event Processing Technical Society* [EPTS 2009] has been founded as well, to promote understanding and advancement in the field of event processing and to develop standards.

All these works and discussions put a great emphasis on possible applications and uses of complex event processing systems, as well as on the integration with existing enterprise solutions. For these reasons, they received a great attention from the industry that is rapidly adopting the term "complex event processing". On the other hand this work is still in its infancy and no real unifying model has been proposed so far to describe and classify complex event processing systems. Our work goes exactly in this direction.

## 7. CONCLUSIONS

The need of processing large flows of information in a timely manner is becoming more and more common in several domains, from environmental monitoring to finance. This need was answered by different communities, each bringing its own expertise and vocabulary and each working mostly in isolation in developing a number of systems we collectively called IFP systems. In this paper we surveyed the various IFP systems developed so far, which include active databases, DSMSs, CEP systems, plus several systems developed for specific domains.

Our analysis shows that the different systems tailor different domains: from data processing, to event detection; focusing on different aspects: from the expressiveness of the rule language, to performance in large scale scenarios; adopting different approaches and mechanisms.

From the discussion in Section 5 we identify several research challenges for research in the IFP domain. First of all, it would be nice to better analyze the differences in term of expressiveness among the different rule languages. In fact, while our analysis allows to draw a line among languages oriented toward data processing and languages oriented toward event detection, putting in evidence the set of operators offered by each language, it is still unclear how each operator contribute to the actual expressiveness of the language and which is the minimal set of operators to combine both full data processing and full event detection capabilities in a single proposal.

Related with the issue of expressiveness is the ability of supporting uncertainty in data and rule processing. As we have seen, a single system support it among those we surveyed, but we feel that this is something useful in several domains and we imagine that more systems should offer such kind of processing.

Along the same line, we noticed how the ability for a rule to programmatically manipulate the set of deployed rules, by adding or removing them, was common in active databases but is rarely offered by newer systems. Again, we think that this is something potentially useful in several cases, which should be offered to increase the expressiveness of the system.

A final issue has to do with the topology of the system and the interaction style. In our survey we observed that IFP systems either focus on throughput, looking at the performance of the engine with an efficient centralized or clustered implementation, or they focus on minimizing the communication costs, by performing filtering, correlation, and aggregation of data directly in network. Both aspects are relevant for a system that have to maximize its throughput in a widely distributed scenario and they are strongly related with the forwarding model adopted (either push or pull). We started looking at these issues in developing Raced [Cugola and Margara 2009] but more has to be done in the future to fully support very large scale scenarios like analysis of environmental data, processing of information coming from social networks, or monitoring of Web sources.

In general we notice that even if the IFP domain is mature enough, with a lot of systems developed by the academy and the industry, there is still space for new systems, possibly integrating in a smart way the different ideas already present in the various systems surveyed.

## Acknowledgments

This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom; and by the Italian Government under the projects FIRB INSYEME and PRIN D-ASAP.

## REFERENCES

Abadi, D., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Erwin, C., Galvez, E., Hatoun, M., Maskey, A., Rasin, A., Singer, A., Stonebraker, M., Tatbul, N., Xing, Y., Yan, R., and Zdonik, S. 2003. Aurora: a data stream management system. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 666–666.

Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and S., Z. 2003. Aurora: A new model and architecture for data stream management. *VLDB Journal 12,* 2.

Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. B. 2005. The design of the borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*. ACM, Asilomar, CA, USA.

Adaikkalavan, R. and Chakravarthy, S. 2006. Snoopib: interval-based event specification and detection for active databases. *Data Knowl. Eng. 59,* 1, 139–165.

Adi, A. and Etzion, O. 2004. Amit - the situation manager. *The VLDB Journal 13,* 2, 177–203.

Agrawal, J., Diao, Y., Gyllstrom, D., and Immerman, N. 2008. Efficient pattern matching over event streams. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 147–160.

Aguilera, M. K., Strom, R. E., Sturman, D. C., Astley, M., and Chandra, T. D. 1999. Matching events in a content-based subscription system. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*. ACM, New York, NY, USA, 53–61.

Ahmad, Y., Berg, B., Cetintemel, U., Humphrey, M., Hwang, J.-H., Jhingran, A., Maskey, A., Papaemmanouil, O., Rasin, A., Tatbul, N., Xing, W., Xing, Y., and Zdonik, S. 2005. Distributed operation in the borealis stream processing engine. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 882–884.

Ahmad, Y. and Çetintemel, U. 2004. Network-aware query processing for stream-based applications. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*. VLDB Endowment, 456–467.

Akdere, M., Çetintemel, U., and Tatbul, N. 2008. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow. 1,* 1, 66–77.

Aleri. 2009. http://www.aleri.com/. Visited Sep. 2009.

Altinel, M. and Franklin, M. J. 2000. Efficient filtering of xml documents for selective dissemination of information. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 53–64.

Amini, L., Andrade, H., Bhagwan, R., Eskesen, F., King, R., Selo, P., Park, Y., and Venkatramani, C. 2006. Spc: a distributed, scalable platform for data mining. In *DMSSP '06: Proceedings of the 4th international workshop on Data mining standards, services and platforms*. ACM, New York, NY, USA, 27–37.

Amini, L., Jain, N., Sehgal, A., Silber, J., and Verscheure, O. 2006. Adaptive control of extreme-scale stream processing systems. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society, Washington, DC, USA, 71.

Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., and Widom, J. 2003. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin 26*, 2003.

Arasu, A., Babu, S., and Widom, J. 2002. An abstract semantics and concrete language for continuous queries over streams and relations. Tech. Rep. 2002-57, Stanford InfoLab.

Arasu, A., Babu, S., and Widom, J. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal 15,* 2, 121–142.

Ashayer, G., Leung, H. K. Y., and Jacobsen, H.-A. 2002. Predicate matching and subscription matching in publish/subscribe systems. In *Proceedings of the Workshop on Distributed Event-based Systems, co-located with the 22nd International Conference on Distributed Computing Systems*. IEEE Computer Society Press, Vienna, Austria.

Avnur, R. and Hellerstein, J. M. 2000. Eddies: continuously adaptive query processing. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 261–272.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. 2002. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, New York, NY, USA, 1–16.

Babcock, B., Datar, M., and Motwani, R. 2004. Load shedding for aggregation queries over data streams. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 350.

Babu, S. and Widom, J. 2001. Continuous queries over data streams. *SIGMOD Rec. 30,* 3, 109–120.

Bagchi, A., Chaudhary, A., Goodrich, M., Li, C., and Shmueli-Scheuer, M. 2006. Achieving communication efficiency through push-pull partitioning of semantic spaces to disseminate dynamic information. *IEEE Transactions on Knowledge and Data Engineering 18,* 10, 1352–1367.

Bai, Y., Thakkar, H., Wang, H., Luo, C., and Zaniolo, C. 2006. A data stream language and system designed for power and extensibility. In *CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management*. ACM, New York, NY, USA, 337–346.

Balazinska, M., Balakrishnan, H., Madden, S. R., and Stonebraker, M. 2008. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst. 33,* 1, 1–44.

Balazinska, M., Balakrishnan, H., and Stonebraker, M. 2004. Contract-based load management in federated distributed systems. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 15–15.

Balter, R. 2004. JORAM: The open source enterprise service bus. Tech. rep., ScalAgent Distributed Technologies SA, Echirolles Cedex, France. Mar.

Barbon, F., Traverso, P., Pistore, M., and Trainotti, M. 2006a. Run-time monitoring of instances and classes of web service compositions. In *IEEE International Conference on Web Services (ICWS 2006)*. 63–71.

Barbon, F., Traverso, P., Pistore, M., and Trainotti, M. 2006b. Run-time monitoring of the execution of plans for web service composition. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006)*. 346–349.

Baresi, L., Bianculli, D., Ghezzi, C., Guinea, S., and Spoletini, P. 2007. Validation of web service compositions. *IET Software 1,* 6, 219–232.

Baresi, L. and Guinea, S. 2005a. Dynamo: Dynamic monitoring of ws-bpel processes. In *In Proceedings of the 3rd International Conference on Service-Oriented Computing (ICSOC 2005)*. 478–483.

Baresi, L. and Guinea, S. 2005b. Towards dynamic monitoring of ws-bpel processes. In *In Proceedings of the 3rd International Conference on Service-Oriented Computing (ICSOC 2005)*. 269–282.

Baresi, L., Guinea, S., Kazhamiakin, R., and Pistore, M. 2008. An integrated approach for the run-time monitoring of bpel orchestrations. In *ServiceWave '08: Proceedings of the 1st European Conference on Towards a Service-Based Internet*. Springer-Verlag, Berlin, Heidelberg, 1–12.

Barga, R. S., Goldstein, J., Ali, M. H., and Hong, M. 2007. Consistent streaming through time: A vision for event stream processing. In *In Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR 2007)*. 363–374.

Barringer, H., Goldberg, A., Havelund, K., and Sen, K. 2004. Rule-based runtime verification. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI 2004)*. 44–57.

Bass, T. 2007. Mythbusters: Event stream processing v. complex event processing. Keynote speech at the 1st Int. Conf. on Distributed Event-Based Systems (DEBS'07).

Bauer, A., Leucker, M., and Schallhart, C. 2006a. Model-based runtime analysis of distributed reactive systems. In *17th Australian Software Engineering Conference (ASWEC 2006)*. 243–252.

Bauer, A., Leucker, M., and Schallhart, C. 2006b. Monitoring of real-time properties. In *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, S. Arun-Kumar and N. Garg, Eds. Lecture Notes in Computer Science, vol. 4337. Springer-Verlag, Berlin, Heidelberg.

BAUER, A., LEUCKER, M., AND SCHALLHART, C. 2007. The good, the bad, and the ugly, but how ugly is ugly? In *Runtime Verification, 7th International Workshop (RV 2007)*. 126–138.

BAUER, A., LEUCKER, M., AND SCHALLHART, C. 2009. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*.

BEERI, C., EYAL, A., MILO, T., AND PILBERG, A. 2007. Monitoring business processes with queries. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 603–614.

BERTINO, E., FERRARI, E., AND GUERRINI, G. 1998. An approach to model and query event-based temporal data. *Temporal Representation and Reasoning, International Syposium on 0*, 122.

BOZDAG, E., MESBAH, A., AND VAN DEURSEN, A. 2007. A comparison of push and pull techniques for AJAX. *Report TUD-SERG-2007-016a*.

BRENNA, L., DEMERS, A., GEHRKE, J., HONG, M., OSSHER, J., PANDA, B., RIEDEWALD, M., THATTE, M., AND WHITE, W. 2007. Cayuga: a high-performance event processing engine. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 1100–1102.

BRODA, K., CLARK, K., 0002, R. M., AND RUSSO, A. 2009. Sage: A logical agent-based environment monitoring and control system. In *AmI*. 112–117.

BUCHMANN, A. P., DEUTSCH, A., ZIMMERMANN, J., AND HIGA, M. 1995. The reach active oodbms. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 476.

CAMPAILLA, A., CHAKI, S., CLARKE, E., JHA, S., AND VEITH, H. 2001. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 443–452.

CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. 2002. Monitoring streams: a new class of data management applications. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 215–226.

CARZANIGA, A., ROSENBLUM, D., AND WOLF, A. 2001. Design and evaluation of a wide-area event notification service. *ACM Trans. on Comp. Syst. 19,* 3, 332–383.

CARZANIGA, A., ROSENBLUM, D. S., AND WOLF, A. L. 2000. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. ACM, Portland, Oregon, 219–227.

CARZANIGA, A. AND WOLF, A. L. 2002. Content-based networking: A new communication infrastructure. In *IMWS '01: Revised Papers from the NSF Workshop on Developing an Infrastructure for Mobile and Wireless Systems*. Springer-Verlag, London, UK, 59–68.

CARZANIGA, A. AND WOLF, A. L. 2003. Forwarding in a content-based network. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, New York, NY, USA, 163–174.

CHAKRAVARTHY, S. AND ADAIKKALAVAN, R. 2008. Events and streams: harnessing and unleashing their synergy! In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*. ACM, New York, NY, USA, 1–12.

CHAKRAVARTHY, S., ANWAR, E., MAUGIS, L., AND MISHRA, D. 1994. Design of sentinel: an object-oriented dmbs with event-based rules. *Information and Software Technology 36,* 9, 555 – 568.

CHAKRAVARTHY, S. AND MISHRA, D. 1994. Snoop: An expressive event specification language for active databases. *Data Knowledge Engineering 14,* 1, 1–26.

CHAND, R. AND FELBER, P. 2004. Xnet: a reliable content-based publish/subscribe system. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. 264–273.

CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., REISS, F., AND SHAH, M. A. 2003. Telegraphcq: continuous dataflow processing. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 668–668.

Chandrasekaran, S. and Franklin, M. 2004. Remembrance of streams past: overload-sensitive management of archived streams. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*. VLDB Endowment, 348–359.

Chandrasekaran, S. and Franklin, M. J. 2002. Streaming queries over streaming data. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 203–214.

Chaudhuri, S. 1998. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, New York, NY, USA, 34–43.

Chaudhuri, S. and Dayal, U. 1997. An overview of data warehousing and olap technology. *SIGMOD Rec. 26,* 1, 65–74.

Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. 2000. Niagaracq: a scalable continuous query system for internet databases. *SIGMOD Rec. 29,* 2, 379–390.

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., and Zdonik, S. 2003. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*. ACM, Asilomar, CA.

Chi, Y., Wang, H., and Yu, P. S. 2005. Loadstar: load shedding in data stream mining. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 1302–1305.

Clarke, E. M. and Schlingloff, B.-H. 2001. Model checking. 1635–1790.

Coral8. 2009. http://www.coral8.com/. Visited Sep. 2009.

Coral8. 2010. http://www.aleri.com/WebHelp/coral8_documentation.htm. Visited Mar. 2010.

Cranor, C., Gao, Y., Johnson, T., Shkapenyuk, V., and Spatscheck, O. 2002. Gigascope: high performance network monitoring with an sql interface. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 623–623.

Cranor, C., Johnson, T., Spataschek, O., and Shkapenyuk, V. 2003. Gigascope: a stream database for network applications. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 647–651.

Cugola, G., Di Nitto, E., and Fuggetta, A. 2001. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. Softw. Eng. 27,* 9, 827–850.

Cugola, G. and Margara, A. 2009. Raced: an adaptive middleware for complex event detection. In *ARM '09: Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware*. ACM, New York, NY, USA, 1–6.

Dayal, U., Blaustein, B., Buchmann, A., Chakravarthy, U., Hsu, M., Ledin, R., McCarthy, D., Rosenthal, A., Sarin, S., Carey, M. J., Livny, M., and Jauhari, R. 1988. The hipac project: combining active databases and timing constraints. *SIGMOD Rec. 17,* 1, 51–70.

Debar, H. and Wespi, A. 2001. Aggregation and correlation of intrusion-detection alerts. In *Recent Advances in Intrusion Detection*. 85–103.

Demers, A., Gehrke, J., Hong, M., Riedewald, M., and White, W. 2006. Towards expressive publish/subscribe systems. In *In Proc. EDBT*. 627–644.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D. 1999. A query language for xml. *Comput. Netw. 31,* 11-16, 1155–1169.

Drusinsky, D. 2000. The temporal rover and the atg rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. Springer-Verlag, London, UK, 323–330.

Eisenberg, A. and Melton, J. 1999. Sql: 1999, formerly known as sql3. *SIGMOD Rec. 28,* 1, 131–138.

Engstrm, H., Engstrm, H., Berndtsson, M., Berndtsson, M., Lings, B., and Lings, B. 1997. ACOOD Essentials.

EPTS. 2009. http://www.ep-ts.com/. Visited Sep. 2009.

Esper. 2009. http://www.espertech.com/. Visited Sep. 2009.

Etzion, O. 2007. Event processing and the babylon tower. Event Processing Thinking blog: http://epthinking.blogspot.com/2007/09/event-processing-and-babylon-tower.html.

Etzion, O. 2009. Event processing thinking. http://epthinking.blogspot.com/. Visited Sep. 2009.

Etzion, O. and Niblett, P. 2010. *Event Processing in Action*. Manning Publications Co.

Eugster, P., Felber, P., Guerraoui, R., and Kermarrec, A.-M. 2003. The many faces of publish/subscribe. *ACM Comput. Surveys 2,* 35 (June).

EventZero. 2009. http://www.eventzero.com/. Visited Sep. 2009.

EventZero. 2010. http://www.eventzero.com/solutions/environment.aspx. Visited Mar. 2010.

Fiege, L., Mühl, G., and Gärtner, F. C. 2002. Modular event-based systems. *Knowl. Eng. Rev. 17,* 4, 359–388.

Forgy, C. L. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence 19,* 1, 17 – 37.

Galton, A. and Augusto, J. C. 2002. Two approaches to event definition. In *Database and Expert Systems Applications, 13th International Conference, DEXA 2002, Aix-en-Provence, France, September 2-6, 2002, Proceedings*. 547–556.

Gatziu, S. and Dittrich, K. 1993. Events in an Active Object-Oriented Database System. In *Proc. 1st Intl. Workshop on Rules in Database Systems (RIDS)*, N. Paton and H. Williams, Eds. Springer-Verlag, Workshops in Computing, Edinburgh, UK.

Gatziu, S., Geppert, A., and Dittrich, K. R. 1992. Integrating active concepts into an object-oriented database system. In *DBPL3: Proceedings of the third international workshop on Database programming languages : bulk types & persistent data*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 399–415.

Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M. 2008. Spade: the system s declarative stream processing engine. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 1123–1134.

Gehani, N. H. and Jagadish, H. V. 1991. Ode as an active database: Constraints and triggers. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 327–336.

Gehani, N. H., Jagadish, H. V., and Shmueli, O. 1992. Composite event specification in active databases: Model & implementation. In *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 327–338.

Giannakopoulou, D. and Havelund, K. 2001. Runtime analysis of linear temporal logic specifications. Tech. rep.

Golab, L. and Özsu, M. T. 2003. Issues in data stream management. *SIGMOD Rec. 32,* 2, 5–14.

Golab, L. and Özsu, M. T. 2005. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 658–669.

Gualtieri, M. and Rymer, J. 2009. The Forrester Wave$^{TM}$: Complex Event Processing (CEP) Platforms, Q3 2009.

Gyllstrom, D., Agrawal, J., Diao, Y., and Immerman, N. 2008. On supporting kleene closure over event streams. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 1391–1393.

Havelund, K. and Rosu, G. 2002. Synthesizing monitors for safety properties. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, London, UK, 342–356.

Hwang, J.-H., Balazinska, M., Rasin, A., Cetintemel, U., Stonebraker, M., and Zdonik, S. 2005. High-availability algorithms for distributed stream processing. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 779–790.

IBM. 2008. Business event processing white paper, websphere software.

IBM. 2010. http://www-935.ibm.com/services/us/index.wss. Visited Feb. 2010.

Ioannidis, Y. E. 1996. Query optimization. *ACM Comput. Surv. 28,* 1, 121–123.

Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., and Venkatramani, C. 2006. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 431–442.

Jain, N., Mishra, S., Srinivasan, A., Gehrke, J., Widom, J., Balakrishnan, H., Çetintemel, U., Cherniack, M., Tibbetts, R., and Zdonik, S. 2008. Towards a streaming sql standard. *Proc. VLDB Endow. 1,* 2, 1379–1390.

Jarke, M. and Koch, J. 1984. Query optimization in database systems. *ACM Comput. Surv. 16,* 2, 111–152.

Khandekar, R., Hildrum, K., Parekh, S., Rajan, D., Wolf, J., Wu, K.-L., Andrade, H., and Gedik, B. 2009. Cola: optimizing stream processing applications via graph partitioning. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, USA, 1–20.

Khoussainova, N., Balazinska, M., and Suciu, D. 2008. Probabilistic event extraction from rfid data. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 1480–1482.

Konana, P., Liu, G., Lee, C.-G., and Woo, H. 2004. Specifying timing constraints and composite events: An application in the design of electronic brokerages. *IEEE Trans. Softw. Eng. 30,* 12, 841–858. Member-Mok, Aloysius K.

Kumar, V., Cooper, B. F., Cai, Z., Eisenhauer, G., and Schwan, K. 2005. Resource-aware distributed stream management using dynamic overlays. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society, Washington, DC, USA, 783–792.

Lakshmanan, G. T., Li, Y., and Strom, R. 2008. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing 12,* 6, 50–60.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21,* 7, 558–565.

Law, Y.-N., Wang, H., and Zaniolo, C. 2004. Query languages and data models for database sequences and data streams. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*. VLDB Endowment, 492–503.

Leucker, M. and Schallhart, C. 2009. A brief account of runtime verification. *Journal of Logic and Algebraic Programming 78,* 5, 293 – 303. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

Li, G. and Jacobsen, H.-A. 2005. Composite subscriptions in content-based publish/subscribe systems. In *Middleware '05: Proceedings of the 6th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., 249–269.

Lieuwen, D. F., Gehani, N. H., and Arlein, R. M. 1996. The ode active database: Trigger semantics and implementation. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 412–420.

Lin, E. Y.-T. and Zhou, C. 1999. Modeling and analysis of message passing in distributed manufacturing systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C 29,* 2, 250–262.

Liu, H. and Jacobsen, H.-A. 2004. Modeling uncertainties in publish/subscribe systems. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 510.

Liu, L. and Pu, C. 1997. A dynamic query scheduling framework for distributed and evolving information systems. In *ICDCS '97: Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*. IEEE Computer Society, Washington, DC, USA, 474.

Liu, L., Pu, C., and Tang, W. 1999. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowl. and Data Eng. 11,* 4, 610–628.

Luckham, D. 1996. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events.

LUCKHAM, D. 2009. http://complexevents.com/. Visited Sep. 2009.

LUCKHAM, D. C. 2001. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

LUCKHAM, D. C. AND VERA, J. 1995. An event-based architecture definition language. *IEEE Transactions on Software Engineering 21*, 717–734.

MADDEN, S., SHAH, M., HELLERSTEIN, J. M., AND RAMAN, V. 2002. Continuously adaptive continuous queries over streams. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 49–60.

MAGEE, J., DULAY, N., AND KRAMER, J. 1994. Regis: A constructive development environment for distributed programs.

MALER, O., NICKOVIC, D., AND PNUELI, A. 2006. From mitl to timed automata. In *FORMATS*. 274–289.

MANSOURI-SAMANI, M. AND SLOMAN, M. 1993. Monitoring distributed systems. *Network, IEEE 7,* 6, 20–30.

MANSOURI-SAMANI, M. AND SLOMAN, M. 1996. A configurable event service for distributed systems. In *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*. IEEE Computer Society, Washington, DC, USA, 210.

MANSOURI-SAMANI, M. AND SLOMAN, M. 1997. Gem: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering 4*, 96–108(13).

MCCARTHY, D. AND DAYAL, U. 1989. The architecture of an active database management system. *SIGMOD Rec. 18,* 2, 215–224.

MÜHL, G., FIEGE, L., AND PIETZUCH, P. 2006. *Distributed Event-Based Systems*. Springer.

ORACLE. 2009. http://www.oracle.com/technologies/soa/complex-event-processing.html. Visited Sep. 2009.

PALLICKARA, S. AND FOX, G. 2003. Naradabrokering: a distributed middleware framework and architecture for enabling durable peer-to-peer grids. In *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, USA, 41–61.

PARK, J., REVELIOTIS, S. A., BODNER, D. A., AND MCGINNIS, L. F. 2002. A distributed, event-driven control architecture for flexibly automated manufacturing systems. *Int. J. Computer Integrated Manufacturing 15,* 2, 109–126.

PATON, N. W. AND DÍAZ, O. 1999. Active database systems. *ACM Comput. Surv. 31,* 1, 63–103.

PIETZUCH, P., LEDLIE, J., SHNEIDMAN, J., ROUSSOPOULOS, M., WELSH, M., AND SELTZER, M. 2006. Network-aware operator placement for stream-processing systems. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 49.

PIETZUCH, P. R. AND BACON, J. 2002. Hermes: A distributed event-based middleware architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*. IEEE Computer Society, Washington, DC, USA, 611–618.

PIETZUCH, P. R., SHAND, B., AND BACON, J. 2003. A framework for event composition in distributed systems. In *In Proceedings of the 2003 International Middleware Conference*. Springer, 62–82.

PNUELI, A. 1997. The temporal logic of programs. Tech. rep., Jerusalem, Israel, Israel.

PROGRESS-APAMA. 2009. http://web.progress.com/it-need/complex-event-processing.html. Visited Sep. 2009.

RAMAN, V., RAMAN, B., AND HELLERSTEIN, J. M. 1999. Online dynamic reordering for interactive data processing. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 709–720.

REPANTIS, T., GU, X., AND KALOGERAKI, V. 2006. Synergy: sharing-aware component composition for distributed stream processing systems. In *Middleware '06: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, USA, 322–341.

ROITMAN, H., GAL, A., , AND RASCHID, L. 2008. Satisfying complex data needs using pull-based online monitoring of volatile data sources. Cancun, Mexico.

ROSENBLUM, D. AND WOLF, A. L. 1997. A design framework for internet-scale event observation and notification. In *Proc. of the 6$^{th}$ European Software Engineering Conf. (ESEC/FSE)*. LNCS 1301. Springer.

SADRI, R., ZANIOLO, C., ZARKESH, A., AND ADIBI, J. 2004. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst. 29,* 2, 282–318.

SCHULTZ-MOELLER, N. P., MIGLIAVACCA, M., AND PIETZUCH, P. 2009. Distributed complex event processing with query optimisation. In *International Conference on Distributed Event-Based Systems (DEBS'09)*. ACM, ACM, Nashville, TN, USA.

SELLIS, T. K. 1988. Multiple-query optimization. *ACM Trans. Database Syst. 13,* 1, 23–52.

SHAH, M. A., HELLERSTEIN, J. M., AND BREWER, E. 2004a. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 827–838.

SHAH, M. A., HELLERSTEIN, J. M., AND BREWER, E. 2004b. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 827–838.

SHAH, M. A., HELLERSTEIN, J. M., CHANDRASEKARAN, S., AND FRANKLIN, M. J. 2003. Flux: An adaptive partitioning operator for continuous query systems. *Data Engineering, International Conference on 0*, 25.

SRIVASTAVA, U. AND WIDOM, J. 2004. Memory-limited execution of windowed stream joins. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*. VLDB Endowment, 324–335.

STOLZ, V. 2007. Temporal assertions with parametrised propositions. In *RV*. 176–187.

STREAMBASE. 2009. http://www.streambase.com/. Visited Sep. 2009.

STREAMBASE. 2010. http://streambase.com/developers/docs/latest/streamsql/index.html. Visited Mar. 2010.

STROM, R. E., BANAVAR, G., CHANDRA, T. D., KAPLAN, M., MILLER, K., MUKHERJEE, B., STURMAN, D. C., AND WARD, M. 1998. Gryphon: An information flow based approach to message brokering. *CoRR cs.DC/9810019*.

SULLIVAN, M. AND HEYBEY, A. 1998. Tribeca: a system for managing large databases of network traffic. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 2–2.

TATBUL, N., ÇETINTEMEL, U., ZDONIK, S., CHERNIACK, M., AND STONEBRAKER, M. 2003. Load shedding in a data stream manager. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*. VLDB Endowment, 309–320.

TATBUL, N. AND ZDONIK, S. 2006a. Dealing with overload in distributed stream processing systems. In *ICDEW '06: Proceedings of the 22nd International Conference on Data Engineering Workshops*. IEEE Computer Society, Washington, DC, USA, 24.

TATBUL, N. AND ZDONIK, S. 2006b. Window-aware load shedding for aggregation queries over data streams. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 799–810.

TIBCO. 2009. http://www.tibco.com/software/complex-event-processing/businessevents/default.jsp. Visited Sep. 2009.

VAN RENESSE, R., BIRMAN, K. P., AND VOGELS, W. 2003. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst. 21,* 2, 164–206.

WANG, F. AND LIU, P. 2005. Temporal management of rfid data. In *VLDB*. VLDB Endowment, 1128–1139.

WASSERKRUG, S., GAL, A., ETZION, O., AND TURCHIN, Y. 2008. Complex event processing over uncertain data. In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*. ACM, New York, NY, USA, 253–264.

White, W., Riedewald, M., Gehrke, J., and Demers, A. 2007. What is "next" in event processing? In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, New York, NY, USA, 263–272.

Widom, J. and Ceri, S. 1996. Introduction to active database systems. In *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1–41.

Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.-L., and Fleischer, L. 2008. Soda: an optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, USA, 306–325.

Wu, E., Diao, Y., and Rizvi, S. 2006. High-performance complex event processing over streams. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 407–418.

Wu, K.-L., Hildrum, K. W., Fan, W., Yu, P. S., Aggarwal, C. C., George, D. A., Gedik, B., Bouillet, E., Gu, X., Luo, G., and Wang, H. 2007. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on system s. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 1185–1196.

Yalagandula, P. and Dahlin, M. 2004. A scalable distributed information management system. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, New York, NY, USA, 379–390.

Zhou, Y., Ooi, B. C., Tan, K.-L., and Wu, J. 2006. Efficient dynamic operator placement in a locally distributed continuous query system. In *OTM Conferences (1)*. 54–71.

Zimmer, D. 1999. On the semantics of complex events in active database management systems. In *ICDE '99: Proceedings of the 15th International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 392.

Received Nov 2009; revised Apr 2010;