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

GIANPAOLO CUGOLA and ALESSANDRO MARGARA, Politecnico di Milano

A large number of distributed applications requires continuous and timely processing of information as it flows from the periphery to the center of the system. Examples include 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 online analysis of stock prices to identify trends and forecast future values.

Traditional DBMSs, which need to store and index data before processing it, can hardly fulfill the requirements of timeliness coming from such domains. Accordingly, during the last decade, different research communities 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 article, we survey these systems to help researchers, who often come from different backgrounds, in understanding how the various approaches they adopt 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 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 Literature**]: Introductory and Survey

General Terms: Design, Documentation

Additional Key Words and Phrases: Complex event processing, publish-subscribe, stream processing

ACM Reference Format:
Cugola, G. and Margara, A. 2012. Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. 44, 3, Article 15 (June 2012), 62 pages.
DOI = 10.1145/2187671.2187677 http://doi.acm.org/10.1145/2187671.2187677

1. INTRODUCTION

An increasing number of distributed applications requires processing continuously flowing data from geographically distributed sources at unpredictable rates 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 the *information flow processing (IFP) domain*. Likewise we call the *information flow processing (IFP) engine* a tool capable of timely processing large amounts of information as it flows from the peripheral to the center of the system.

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.
Author's Address: G. Cugola and A. Margara, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy; email (corresponding author): margara@elet.polimi.it.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 0360-0300/2012/06-ART15 $10.00
DOI 10.1145/2187671.2187677 http://doi.acm.org/10.1145/2187671.2187677

The concepts of timeliness and flow processing are crucial for justifying the need for a new class of systems. Indeed, traditional DBMSs (i) require data to be (persistently) stored and indexed before it could be processed and (ii) process data only when explicitly asked by the users, that is, asynchronously with respect to its arrival. Both aspects contrast with the requirements of IFP applications. As an example, consider the need for detecting fire in a building by using temperature and smoke sensors. On the one hand, a fire alert has to be notified as soon as the relevant data becomes available. On the other, there is no need to store sensor readings if they are not relevant to fire detection, while the relevant data can be discarded as soon as the fire is detected, since all the information they carry (like the area where the fire occurred) should be part of the fire alert if relevant for the application.

These requirements have 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 having a common goal, these systems differ in a wide range of aspects, including architecture, data models, rule languages, and processing mechanisms. In part, this is due to 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 to 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 competing today: 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 processing streams of data coming from different sources to produce new data streams as output and views this problem as an evolution of traditional data processing, as supported by DBMSs. Accordingly, *data stream management systems (DSMSs)* have their roots in DBMSs but present substantial differences. 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 all the operators defined in general by relational algebra.

Conversely, the complex event processing model views flowing information items as notifications of events happening in the external world, which have to be filtered and combined to understand 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 that represent the higher-level events whose occurrence has to be notified to the interested parties. The contributions to this model come from different communities, including distributed information systems, business process automation, control systems, network monitoring, sensor networks, and middleware, in general. The origins of this approach may be traced back to the publish-subscribe domain [Eugster et al. 2003]. Indeed, while traditional publish-subscribe systems consider each event separately from the others and filter 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 subscription language to consider complex event patterns that involve the occurrence of multiple, related events.

In this article, we claim that neither of the two aforementioned models 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, Section 2 describes the IFP domain in more detail, provides an initial description of the different technologies that have been developed to support it, and explains the need for combining the best of different worlds to fully support IFP applications. Section 3 describes 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 runtime 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, Section 6 reviews related work, while Section 7 provides some conclusive remarks and a list of open issues.

2. BACKGROUND AND MOTIVATION

In this section, we characterize the application domain that we call Information Flow Processing and motivate why we introduce a new term. We describe the different technologies that have been proposed to support such a domain and 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 for processing 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 need to process data coming from sensors deployed in the field to acquire information about the observed world, detect anomalies, or predict disasters as soon as possible [Broda et al. 2009; Event Zero 2010a]. Similarly, several financial applications require a continuous analysis of stocks to identify trends [Demers et al. 2006]. 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 have to analyze network traffic in real-time, generating alerts when something unexpected happens [Debar and Wespi 2001]. RFID-based inventory management performs continuous analysis of RFID readings to track valid paths of shipments and to capture irregularities [Wang and Liu 2005]. Manufacturing control systems often require anomalies to be detected and signalled by looking at the information that describe how the system behaves [Lin and Zhou 1999; Park et al. 2002].

Common to all these examples is the need for processing information as it flows from the periphery to the center of the system without requiring, at least in principle, the information to be persistently stored. Once the flowing data has been processed, thereby producing new information, it can be discarded while the newly produced information leaves the system as output.[1]

The broad spectrum of applications domains that require information flow processing in this sense explains why several research communities focused their attention to the IFP domain, each bringing its own expertise and points of view, but also its own vocabulary. The result is a typical Tower of Babel syndrome, which generates misunderstandings among researchers that negatively impact the spirit of collaboration required to advance the state of the art [Etzion 2007]. This explains why in this article,

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

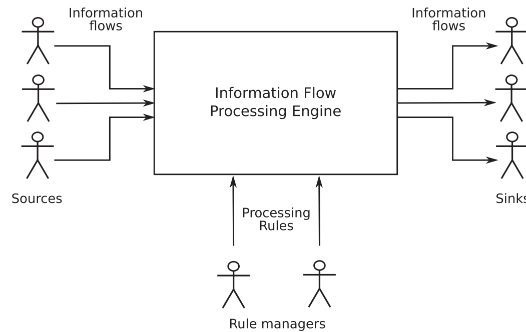


Fig. 1. The high-level view of an IFP system.

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.” In so doing, we neither have the ambition to propose a new standard terminology nor do we want to contribute to “raising the tower.” Our only desire is to help the reader look at the field with a mind clear from any bias toward the meaning of the various terms used.

This goal leads us to model an IFP system, as in Figure 1. The *IFP engine* is 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 as outputs. The entities entering the IFP engine that create the information flows are called *information sources*. Examples of such information are notifications about events happening in the observed world or, more generally, 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 information chunks with semantics and relationships (including total or partial ordering, in some cases). The IFP engine takes such information items as input and processes them, as soon as they are available, according to a set of processing rules which 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 *rule managers* are the entities in charge of adding or removing processing rules. 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 this definition is general enough to capture every IFP system, we are interested in classifying it to encompass some key points that characterize the domain and come directly from the requirements of the applications we just mentioned. First, it needs to perform real-time or quasi real-time processing of incoming information to produce new knowledge (i.e., outgoing information). Second, it needs an expressive language to describe how incoming information has 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, it needs scalability to effectively cope with situations in which a very large number of geographically distributed information sources and sinks have to cooperate.

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 for developing systems that are 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 interaction model, it is not possible to ask the system to send notifications when predefined situations 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 DBMS.

There are several tools classified as active database systems with different software architectures and functionality and that are 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 of three parts.

—*Event.* The event part 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.
—*Condition.* The condition part 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.
—*Action.* The action part identifies the set of tasks that should 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.* Detection of an event.
—*Triggering.* Association of an event with the set of rules defined for it.
—*Evaluation.* Evaluation of the conditional part for each triggered rule.
—*Scheduling.* Definition of an execution order between selected rules.
—*Execution.* Execution of all the actions associated to selected rules.

Active database systems are used in three contexts: 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, for example, to implement an automatic reaction to constraint violations. In closed database applications, active rules can support the semantics of the application, but external sources of events are not allowed. Finally, in open database applications, events may come both from inside the database and from external sources. This is the domain that is closer to IFP.

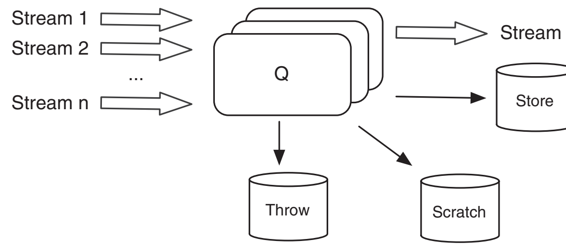


Fig. 2. The typical model of a DSMS.

2.2.2. Data Stream Management Systems. Even when taking into account external event sources, active database systems (like traditional DBMS), are built 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 exceeds a certain threshold or when the arrival rate of events (internal or external) is high. For most applications we classified as IFP, such limitations are unacceptable.

To overcome them, the database community developed a new class of systems oriented toward processing large streams of data in a timely way: *data stream management systems (DSMSs)*. DSMSs differ from conventional DBMSs 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*, that is, queries that are deployed once and continue to produce results until removed. Standing queries can be executed periodically or continuously as new stream items arrive. They introduce a new interaction model with respect to traditional DBMSs: users do not have to explicitly ask for updated information; rather, the system actively notifies it according to installed queries. This form of interaction is usually called *database-active human-passive (DAHP)* [Abadi et al. 2003].

Several implementations were proposed for DSMSs. They differ in the semantics they associate to standing queries. In particular, the 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 the required 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 $\boldsymbol{Q}$, one or more input streams, and four possible outputs.

—*Stream* is formed by all the elements of the answer that are produced once and never changed.
—*Store* is filled with parts of the answer that may be changed or removed at a certain point in the future. Stream and store together define the current answer to queries $\boldsymbol{Q}$.

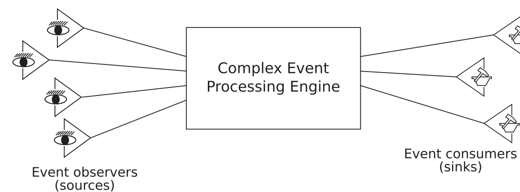


Fig. 3. The high-level view of a CEP system.

—*Scratch* represents the working memory of the system, that is, a repository where it is possible to store data that is not part of the answer but that may be useful for computing the answer.
—*Throw* is a sort of recycle bin used to throw away unneeded tuples.

To the best of our knowledge, this model is the most complete for to defining 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 preceding limitation originates from the same nature of DSMSs, which are generic systems that leave the responsibility of associating a semantics to the data being processed to their clients. *Complex event processing (CEP) systems* adopt the opposite approach. As shown in Figure 3, they associate a precise semantics to the information items being processed: they are notifications of events that happened in the external world and observed by sources. The CEP engine is responsible for filtering and combining such notifications to understand what is happening in terms of higher-level events (sometimes also called *composite events* or *situations*) to be notified to sinks, which act as *event consumers*.

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*, 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. These 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 sinks 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°C, within 10 s, one from the other.

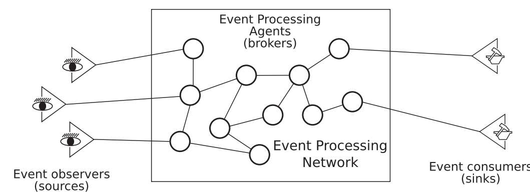


Fig. 4. CEP as a distributed service.

CEP systems put great emphasis on the issue that represents the main limitation of most DSMSs: the ability to detect complex patterns of incoming items involving sequencing and ordering relationships. Indeed, the CEP model relies on the ability to specify composite events through event patterns that match incoming event notifications on the basis of their content and on some ordering relationships on them.

This feature also has an impact on the architecture of CEP engines. In fact, these tools often have to interact with a large number of distributed and heterogeneous information sources and sinks which observe the external world and operate on it. This is typical of most CEP scenarios, such as environmental monitoring, business process automation, and control systems. This also suggests, 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 implements 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 are usually ignored in DSMSs.

CEP systems can thus be classified on the basis of the architecture of the CEP engine (centralized, hierarchical, acyclic, peer-to-peer) [Carzaniga et al. 2001; Eugster et al. 2003], the forwarding schemes adopted by brokers [Carzaniga and Wolf 2002], and 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 information flows according to their content and the relationships among its constituents. 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 large 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, leading to two classes of systems: active databases, lately subsumed by DSMSs on one side, and CEP engines on the other. DSMSs mainly focuses on flowing data and data transformations. Only few approaches allow the easy capture of sequences of data involving complex ordering relationships, not to mention taking into account the possibility of performing 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—both those developed as extensions of publish-subscribe middleware and those developed as totally new systems—define a quite different

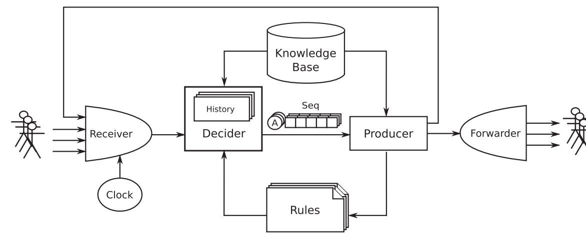


Fig. 5. The functional architecture of an IFP system.

model. They focus on processing event notifications with a special attention to their ordering relationships to capture complex event patterns. Moreover, they target the communication aspects involved in event processing, such as the ability to adapt to different network topologies, as well as to various processing devices, together with the ability to process information directly in-network, to distribute the load and reduce communication cost.

As we claimed in Section 1, IFP should focus both on effective data processing, including the ability to capture complex ordering relationships among data, and on efficient event delivery, including the ability to process data in a strongly distributed fashion. To pursue these goals, researchers must be able to understand what has been accomplished in the two areas, and how they complement each other in order to combine them. As we will see in Section 4, some of the most advanced proposals in both areas go in this direction. This work is a step in the same direction, as it provides an extensive framework to model an IFP system and uses it to classify and compare existing systems.

3. A MODELING 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 define 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.

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 the 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 it 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 yet fire, 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 inside the decider.

The detection phase ends by passing to the producer an action A and a sequence of information items *seq* for each triggered rule, that is, 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 for characterizing 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 for 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 the producer 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).

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

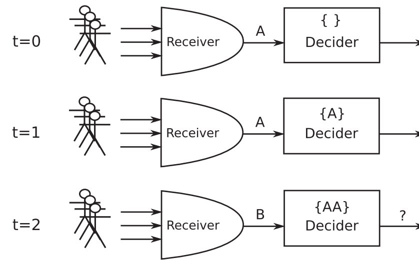


Fig. 6. Selection policy: an example.

In executing the 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, for example, 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

Together with the last information item entering the decider, the set of deployed rules, and the information stored in the history and in the knowledge base, three additional concepts concur to uniquely determine the output of a single detection-production cycle: the *selection*, *consumption*, and *load-shedding* policies adopted by the system.

Selection policy. In the 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 selected and sent to the producer.

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 for recognizing the pattern in the future. At $t = 1$, a new 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 is 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 this *multiple selection* policy allow each rule to fire more than once at each detection-production cycle. This means that in the preceding situation, 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 $t = 1$.

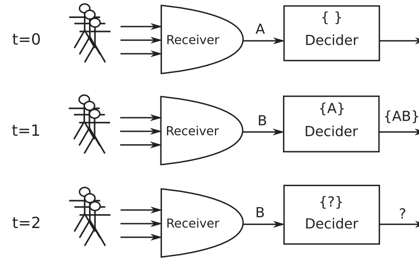


Fig. 7. Consumption policy: an example.

Conversely, systems that adopt a *single selection* policy allow rules to fire at most once at each detection-production cycle. In the same preceding situation, the decider of such systems would choose one of the two As 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.

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 preexisting 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 the 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 on the target application domain and on 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,

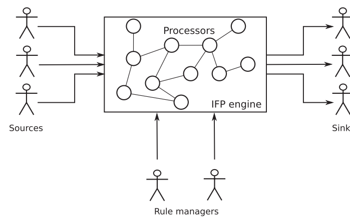


Fig. 8. The deployment architecture of an IFP engine.

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 predetermined 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—possibly dispersed over a 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, that is, how the components that implement the functional architecture in Figure 5 can be distributed over multiple nodes to achieve scalability.

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

With reference to Figure 8, we define an IFP engine as centralized when 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 for the further classification of distributed IFP engines. In a clustered engine, scalability is pursued by sharing the effort of processing incoming information flows among a cluster of strongly connected machines, usually part of the same local area network. In a clustered IFP engine, the links connecting processors among themselves perform much better than the links connecting sources and sinks with the cluster itself. Furthermore, the processors are in 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 to the IFP engine), while processors are widely distributed and usually run in different administrative domains. Some CEP systems adopt this architecture.

In summary, in seeking 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 form an IFP application. With reference to Figure 1,

we distinguish among three different submodels. The *observation model* refers to the interaction between information sources and the IFP engine. The *notification model* refers to the interaction between the engine and the sinks. The *forwarding model* defines the characteristics of the interaction among processors in the case of a distributed implementation of the engine.

We also distinguish between a *push* and a *pull* interaction style [Rosenblum and Wolf 1997; Cugola et al. 2001]. In a pull observation model, the IFP engine is the initiator of the interaction that brings information from sources to the engine; otherwise, we have a push observation model. Likewise, in a pull notification model, the sinks have the responsibility of pulling information relevant for them from the engine; otherwise, we have a push notification 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 in the observation model, notification model, and especially in the forwarding model, 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 manipulate 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, that is, their *format*. The main formats adopted by currently available IFP systems are *tuples*, either typed or untyped; *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 [Liu and Jacobsen 2004; Wasserkrug et al. 2008]. In many IFP scenarios, in fact, information received from sources has an associated degree of uncertainty. As an example, if sources were only able to provide rounded data [Wasserkrug et al. 2008], the system could associate such data with a level of uncertainty, instead of accepting it as a precise information.

Besides single information items, we classify IFP systems based on the nature of information flows, distinguishing between systems whose engine processes *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, for examples, 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.

In the other case, engines allow different types of information items in the same flow; for example, 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.

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) is characterized by the 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, 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 time stamps (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 one to select the set of items received in a given time span. After this step, the ordering is lost, and time stamps are not available for further processing. In particular, the ordering (and time stamps) of the output stream are conceptually separate from the ordering (and time stamps) 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 times tamp that represent an absolute time, usually interpreted as the time of occurrence of the related event. Hence, time stamps define a total ordering among items. In such systems, time stamps are fully exposed to the rule language, which usually provide advanced constructs based on them, like sequences (see Section 3.8). Notice how such kinds of times tamps 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 kind of label, which—while not representing an absolute instant in time—can define a partial ordering among items, usually reflecting some kind of causal relationship, that is, 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 systems that associate items with an interval, that is, two time stamps taken from a global time, usually representing the time when the related event started and 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 adopted rule language. However, we can roughly classify them into 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 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 flow of operators 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 which express rules using SQL-like statements.

As a final remark, we notice that 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 an explicit 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 databases, 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 [Liu and Jacobsen 2004; Wasserkrug et al. 2008; O'Keeffe and Bacon 2010]. In particular, some systems allow for distinguishing between *deterministic* and *probabilistic* rules. The former define their outputs deterministically from their inputs, while the latter allow a degree of uncertainty (or a confidence) to be associated with the outputs. Notice that the issue is only partially related with the ability to manage uncertain data (see Section 3.5). Indeed, a rule could introduce a certain degree of uncertainty, even in the 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 70ºC is detected. The input information is precise, but the rule introduces a certain degree of uncertainty of knowledge. (Other situations could explain the raising of temperature).

3.8. Language Model

The rule model described in the 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, that is, 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 which the information flows have to follow. Each primitive operator defines a transformation over its input. Usually, systems that adopt imperative languages offer visual tools to define rules.
—*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.

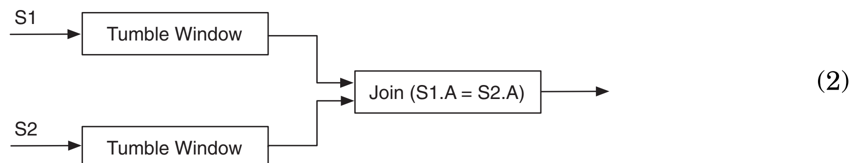
It is worth mentioning that existing systems sometime allow users to express rules using more than one paradigm. For example, many commercial systems offer both a declarative language for rule creation and a graphical tool for connecting defined rules in an imperative way. Moreover, some existing declarative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

As a representative example for declarative languages, let us consider CQL [Arasu et al. 2006], created within the Stream project [Arasu et al. 2003] and currently adopted by Oracle [Oracle 2010]. CQL 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 are used 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.

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

This rule isolates the last five elements of flow F1 and the last ten elements of flow F2 (using the stream-to-relation operator `[Rows n]`), then combines all elements having a common attribute A (using the relation-to-relation operator `Where`), 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 *boxes and arrows* [Abadi et al. 2003]. In Example (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 in such a language.

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

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 5–15 s) 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 here to build the examples we need.

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 previously defined, 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.

In general, there is no direct mapping between available operators and language 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 operator provided by existing IFP systems are single-item operators, that is, those processing information items one by one. Two classes of single-item operators exist.

—*Selection operators* filter items according to their content, discarding elements that do not satisfy a given constraint. As an example, they can be used to keep only the information items that contain temperature readings whose value is greater than 20°C.
—*Elaboration operators* transform information items. As an example, they can be used to change a temperature reading, converting from Celsius to Fahrenheit. Among elaboration operators, it is worth mentioning those coming from relational algebra.
  —*Projection* extracts only part 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 the name.
  —*Renaming* changes the name of a field in languages based on records.

Single-item operators are always present; declarative languages usually inherit 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 select those items that should be part of a complex pattern. Notably, they are the only operators available in publish-subscribe systems for choosing 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 next) in that they are *order independent*, that is, they define patterns that rely only on the detection (or non-detection) of information items and not on some specific ordering relation holding among them.

—A *conjunction* of items $I_1, I_2, \ldots I_n$ is satisfied when all the items $I_1, I_2, \ldots I_n$ have been detected.
—A *disjunction* of items $I_1, I_2, \ldots I_n$ is satisfied when at least one of the information items $I_1, I_2, \ldots I_n$ has been detected.
—A *repetition* of an information item I of degree $\langle m, n \rangle$ 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 other 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-item operators to dictate constraints both on the content of each element and on the relationships among them.

The use of logic operators allows for the definition of expressions whose values 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* (defined later).

Logic operators are always present in pattern-based languages, where they represent the typical way of combining 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).

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

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 data from both flows F1 and F2.

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

Sequences. Similar 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 $I_1, I_2, \ldots I_n$, which is satisfied when all the elements $I_1, I_2, \ldots I_n$ 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 time stamp field explicitly added to information items,

as in the following example.[3]

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

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 but, rather, implicitly using the iterating condition.

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
a.type = 'contaminated' and
b[1].from = a.site and                             (7)
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 the preceding one. Shipment information items need not 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*, next).

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. However, some work investigated the possibility of including 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, iterations are strictly related 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 often 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 as 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*, which need to read the whole input flows before producing results, and *non-blocking operators*, which can successfully return their results as items enter the system [Babcock et al. 2002]. An example of a blocking operator is negation, which

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

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, 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 in the window: for example, to limit the scope of an operator to the last ten elements arrived.

An orthogonal way of classifying 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, that is, 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 the lower and the 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 considers the items received in the last minute.

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

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

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 so doing, users are allowed to explicitly manage the part of the flow they want to consider, that is, 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 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.

```
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 : { }
}
```
(10)

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 it does not allow repetitions to include an upper bound. Similarly, CEDR [Barga et al. 2007] can express negation 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
WHEN UNLESS(A, B, 12 hours)
WHERE A.a < B.b
```
(11)

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

—*Join* operators are used to merge two flows of information as in a traditional DBMS. Being a blocking operator, the join is usually applied to portions of the input flows which are processed as standard relational tables. As an example, the CQL of Rule (12) uses the join operator to combine the last 1,000 items of flows $F1$ and

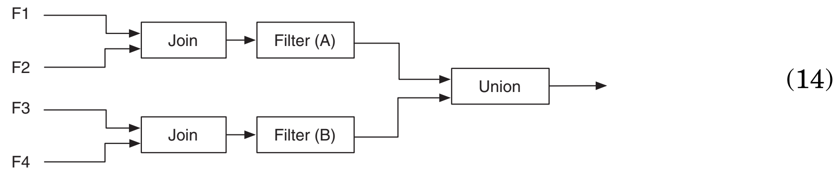
$F2$ by merging those items that have the same value in field A.

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

—*Bag operators* combine different flows of information, considering them as bags of items. In particular, we have the following bag operators.
- —*Union* merges two or more input flows of the same type, creating a new flow that includes all the items coming from them.
- —*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.
- —*Intersect* takes two or more input flows and outputs only the items included in all of them. It is a blocking operator.
- —*Remove-duplicate* removes all duplicates from an input flow.

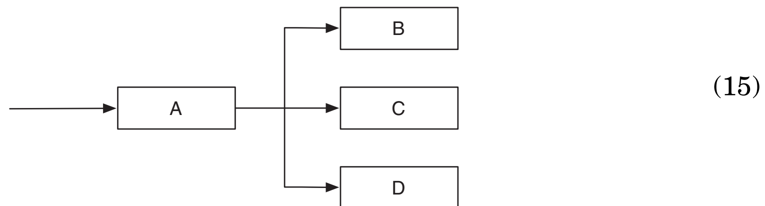
Rule (13) provides an example of using the union operator in CQL to merge together two flows of information. Similarly, in Example (14), we show the union operator, as provided by Aurora.

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



(14)

—*Duplicate* operators allow a single flow to be duplicated in order to use it as an input for different processing chains. Example (16) 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)

—*Group-by* operators are used to split information flows 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 the last 1,000 rows of flow $F1$ based on the value

of field B in order to count how may items exist for each value of B.

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

—*Order-by* operators are used to impose an ordering to the items of an input flow. They are blocking operators, so they are usually applied to well-defined portions of 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 joining the two information flows, that is, the one about room temperature and that about smoke, and imposing the room identifier to be the same. 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 feature, 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.

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

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

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

—Some have been designed to deal indifferently with information flows and with relational tables a la DBMS. In this case, flow creation operators can be used to create new flows from existing tables, for example, when new elements are added to the table.

—Other languages use windows as a way of transforming (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 for creating 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, and 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 removed from table T during the current processing cycle. Finally, in Example (20),

all items in T are put in the output stream independently from previous processing cycles.

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

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

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

Aggregates. Many IFP applications need to aggregate the content of multiple, incoming information items to produce new information, for 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 ten 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 online). Finally, some languages also offer facilities for creating *user-defined aggregates (UDAs)*. As an example of the latter, we have already shown the ESL in Rule (10), which defines an UDA for computing the smallest positive value among the received ones. 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, it analyzes if information items produced by fired rules may reenter the system (*recursion*), and if the rule set can be changed by fired actions at runtime. 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

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
TESLA/T-Rex	Present	Absent	Unbounded	Yes	No	Program.	Program.	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

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 of defining 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.

Within each class we do 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 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,

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.
TESLA/T-Rex	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

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 rules, 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 the 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 coupling mode between an event and a condition specifies when the condition is evaluated with respect to the transaction in which the triggering event is signaled. The coupling mode between a condition and an action specifies when the action is executed with respect to the transaction in which the condition is evaluated. HiPac introduces three

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
TESLA / T-Rex	Events	Tuples	No	Heterogeneous	Absolute	Detecting	No
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

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.

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			
TESLA/T-Rex	Det	X						X	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

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

Constraints are associated with a class definition and consist of a condition and an optional handler: all the objects of a class must satisfy its constraints. In case of violation, the handler is called: it is composed by a set of operations 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 aborts 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.

Triggers, like integrity constraints, are used to detect user-defined conditions on the database objects. Such conditions, however, do not represent consistency violations but capture generic situations of interest. Trigger, like a constraint, are specified in the class definition and consist of two parts: a condition and an action. The condition is evaluated on each object of the class defining it. When it becomes true, the corresponding action is executed. 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). Actions may include dynamic activation and deactivation of rules.

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; however, users can force actions to be executed when an active trigger has not fired within a given amount of time.

We can map the behavior of Ode to our functional and data model by considering public method calls to database objects as the information items that enter the system. Evaluation is performed by taking into account the database state (i.e., the knowledge base in our model).

So far, we have only mentioned processing involving single information items: support for multiple-items processing has been introduced in Ode [Gehani et al. 1992], allowing users to capture complex interactions with the database, involving multiple methods call. In particular, Ode offers logic operators and sequences for combining single calls: the centralized nature of the system makes it possible to define a complete ordering relation between calls (absolute time). The language of Ode derives from regular expressions with the addition of parameterization. As a consequence, negation (although present) can only refer to immediately subsequent events. 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. Gehani et al. [1992] also address the issue of defining precise processing policies; in particular, they propose a detection algorithm in which all valid set of events are captured (multiple selection policy), while the possibility of consuming 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 expressions (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 of considering *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 to deal with blocking operators (e.g., the negation); 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 for 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. As with 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 of specifying 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 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 of extending the semantics of the language to capture events that have a duration. This led to the definition of a new language, called SnoopIB [Adaikkalavan and Chakravarthy 2006], which offers all the operators and language construct of Snoop (including contexts) but uses interval-based time stamps 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, 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 that is 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 also designed 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 for retrieving information (in the form of XML datasets) 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 email) 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 come directly from Internet databases, they do not have an explicit associated time stamp.

NiagaraCQ can be seen as a sort of active database system where timer-based rules behave like traditional queries (the only exception being periodic evaluation), while change-based rules implement 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 the NiagaraCQ 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: an 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. Rules are expressed using an imperative language that 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 are used to specify the portions of input flows to take into account when performing aggregates. Since items do not have an explicit time stamp, 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 through 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 of storing a portion of each input flow inside the relational tables where processing takes place; 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. As already mentioned in Section 3.8, three relation-to-stream operators exist: `IStream`, `DStream`, and `RStream`. Notice that CQL is based on a time model that explicitly associates a time stamp to input items, but time stamps cannot be addressed from the language. As a consequence, 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 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 by 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 time stamp but are processed in their arrival order.

An interesting feature of Aurora is the possibility of including 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 loads between available resources; in these implementations, called Aurora* and Medusa, communication between processors takes 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, 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*, that is, an attribute that monotonically increases or decreases as items are produced by the source of the flow, for example a time stamp defined with respect to 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 rearrange 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, an 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 of creating custom aggregates in an imperative way directly within the language. To do so, the language offers the possibility of creating and managing custom tables which can be used as variables of a programming language during information flow processing. Processing itself is modeled as a 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 the 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 performs 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 the cost of transmission, as much as possible, 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 other events, 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 time stamped 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 sets of events matching a given pattern, which means that it applies multiple selection and zero consumption policies. Notice that complex events can be reused 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 on 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, the possibility of executing external routines and dynamically activating or deactivating 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 constraints. This way, it provides fixed, landmark, and sliding windows. All possible sets of events satisfying a rule are captured; so we can say that the system adopts 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. As with 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 of modeling 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 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. Barga et al. [2007] present the foundations of CEDR, defined as a general-purpose event streaming system. Different contributions are introduced. First, of 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 notifications of state updates; accordingly, the processing engine keeps a 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 patterns, 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 terms 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 databases:

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: indeed, 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. Similar 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, that is, the one that minimizes the usage of CPU resources and the processing delay.

PB-CED. Akdere et al. [2008], 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. 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 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 the 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 for computing 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. 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 monitors the rate of information items produced at each child; rare information is then asked in a push way (i.e., it is received immediately, as it is 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 time stamp 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 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 for programmable 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 in 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 for the 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 hand, this analysis enables for a direct comparison with other languages (e.g., traditional regular expressions, Cayuga language). On the other hand, 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; and *CREATE EVENT* and *SET*, which define the type of event to be generated and set the content of its attributes,

respectively. 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. In more detail, 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 reuse 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 of producing *partial events*, that is, 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 sources 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.

TESLA / T-Rex. T-Rex [Cugola and Margara 2010a] is a general-purpose complex event processing system designed to provide a balance between expressiveness and efficiency. The system exposes API to publish new events, subscribe to simple as well as complex events, and define new complex events through specification rules. Rules are expressed using TESLA [Cugola and Margara 2010b], a detecting language based on patterns. TESLA attempts to provide high expressiveness through a small set of operators to keep the language simple and compact: indeed, it includes only a selection of single events, time-bounded sequences, negations, and aggregates. Periodic rule execution is allowed through timers. In our functional model, they are captured by the presence of the clock. Moreover, TESLA offers completely programmable event selection and consumption policies. The authors show how the operators provided, together with the possibility of expressing hierarchies of events, can meet the requirements of a number of existing application domains. It is worth mentioning that TESLA is formally defined using a metric temporal logic: this provides a reference semantics that allows us to check for the correctness of a given implementation.

T-Rex has been implemented in C++. It performs processing incrementally, as new events enter the system. To do so, it adopts an algorithm based on detection automata. Cugola and Margara [2010a] present the algorithm as well as the implementation strategies in details. Since one of the goals of T-Rex is efficient detection, the authors define custom data structure explicitly conceived to reduce memory occupation (through data sharing) and to speed up processing through precompiled indexes. In the current implementation, T-Rex adopts a centralized infrastructure and relies upon an absolute time.

4.4. Commercial Systems

Aleri Streaming Platform. The Aleri Streaming Platform [Aleri 2010] is an IFP system that offers a simple, imperative, graphical language to define processing rules by combining a set of predefined operators. To increase a 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 time stamps. 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 for enabling 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 2010a, 2010b], despite its name, can be classified as a data stream system. Indeed, 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. As with 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 time stamp value if a time stamp 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.

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

StreamBase. Streambase [2010a] is a software platform that includes a data stream processing system, a set of adapters for gathering 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 2010b]. 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 for explicitly instructing 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 [2010] launched its event-driven architecture suite in 2006 and added BEA's WebLogic Event Server 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.

As 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 include technology for distributed processing in clustered environment, as well as tools for analysis of historical data.

Esper. Esper [2010] 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 of expressing 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 of explicitly programing event selection policies, exploiting the *every* and *every-distinct* modifiers.

Esper supports both centralized and clustered deployments; in fact, by 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.

Tibco Business Events. Tibco Business Events [Tibco 2010] 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, for 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 accepts rules specified using a declarative, SQL-like, language called SPADE [Gedik et al. 2008], which embeds all traditional SQL operators for filtering, joining, and aggregating data. Rules written in SPADE are compiled into one or more PEs. However, different from most of the other 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 for 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 machines 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 widely adopted commercial systems exist for which, unfortunately, documentation or evaluation copies are not available. We mention some of them here.

IBM WebSphere Business Events. IBM acquired CEP pioneer system AptSoft in 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 for the 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 [2010b], 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 2010] 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 systems that mainly focus on data processing and systems 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

[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.

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 means to capture the occurrence of relevant events.

The data and rule models summarized in Table III also show that data uncertainty and probabilistic rules have been rarely explored in existing IFP systems. Since these aspects are critical in many IFP applications, that is, 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 flexibility, 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. 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: time stamps, when present, are mainly used to partition input streams using windows and 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 are, instead not provided by stream processing systems (at least those coming from the research community). Table IV, in fact, shows that commercial systems adopt a peculiar approach. While they usually adopt the same stream processing paradigm as DSP research prototypes, they also embed pattern-based operators, including sequence and iteration. While, at first, this approach could appear to be the most effective way to combine processing abstractions coming from the two worlds, a closer look at 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. One gets the impression that several mechanisms were put together without investing much effort in their integration and without carefully studying the best and minimal set of operators to include in the language to offer both DSP and CEP processing support. In general, we can observe that this research area is new, and neither academia nor industry have found how to combine the two processing paradigms in a unifying proposal [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, that is, monitoring updates of webpages 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 a 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 for distributed processing adopt a push-based forwarding approach to send information from processor to processor.

The only exception is represented by Raced, which switches between push and pull dynamically for adapting 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. By looking at Table II, we notice a peculiarity that was anticipated in Section 2: the few systems that provide a networked implementation for performing filtering, correlation, and aggregation of data directly in network also focus on event detection. It is an open issue if this situation is an outcome of the history brought to the development of IFP systems from different communities with 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 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 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. Table I shows how half of the systems include such a (logical) component and consequently allow rules to fire periodically, while the remaining systems provide a purely reactive behavior. This feature 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 to the issue of agreeing QoS levels between sources, sinks, and the IFP system. While all CEP systems aim at maximizing 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 on the intrinsically different nature of data and event processing, or if it is a consequence of 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 in Table I, such a 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 for defining and managing 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 to adapt 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 feature can help us drawing a sharp distinction between traditional publish-subscribe systems and all other systems, which allow 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 are unbound. As an example, a system that allows for the combining (e.g., join) of 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 us 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 notice (Table I) that the ability to dynamically change the set of active rules, while available in most active databases developed years ago, has 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 ongoing or past research which aims at providing a more complete understanding of the IFP domain. In particular, we cover four different aspects: (i) we present work that studies 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 systems; (iv) we discuss existing attempts to create a standard for the IFP domain.

6.1. General Mechanisms for IFP

Many researchers have focused on developing general mechanisms for IFP by studying (i) rule 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 different rules coexist in IFP systems, 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, besides the definition of an optimal execution plan, a new problem emerges—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 in order 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., the possibility of replicating operators). 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 be 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 also has been proposed 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.

Recently, a few works coming from the event-based community have addressed the problem of QoS. In particular, some of them have studied the quality of event delivery with respect to different dimensions [Pandey et al. 2004; Xu et al. 2006; Roitman et al. 2010a]. Often, such dimensions are merged together to obtain a single variable; the objective is that of properly allocating system resources in order to maximize such a variable. Roitman et al. [2010b] study the quality of event delivery using a multi-objective approach in which the two competing dimensions of completeness of delivery and delay are considered together. The authors present an offline algorithm capable of finding the set of optimal solution to the multi-objective function, as well as a greedy approach that works online.

O'Keeffe and Bacon [2010] address the problem of defining a proper semantics for event detection when communication and synchromization errors may occur. Their proposal extends the rule definition language with policies so that the behavior of the system, in the presence of errors, can be customized by the users according to their needs. Several policies are presented, for example a *No-False-Positive* policy ensures the sinks that all received events have actually occurred, while a *Max-Delay* policy does not offer guarantees about the quality of results but, instead, ensures that the delay between event detection and event delivery does not overcome a given threshold.

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 different algorithms based on replication. Interestingly, Balazinska et al. [2008] also take 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 (incomplete) information it is still able to receive; such information would be updated as soon as the communication could 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 help 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 for reacting 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 flows of external events, the database community gave birth to 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 authors 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 2010; Streambase 2010a]) 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 2010].

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 Eugster et al. [2003] and Mühl et al. [2006]. Recently, several research efforts have been put in place to cope with rules involving multiple events: general models of CEP can be found in Luckham [2001] and Etzion and Niblett [2010] and Wright et al. [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 they deserve to be mentioned here. As an example, some papers 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 systems, 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 [Galton and Augusto 2002; Adaikkalavan and Chakravarthy 2006], distinguishing between a *detection* and an *occurrence* semantics for events. The former associates events with a single time stamp, representing the time in which they were detected; it has been shown that this model provides 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], 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 that could satisfy all those properties unless time is represented using time stamps of unbounded size. It has been also 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 systems.

6.3.1. Runtime Verification Tools. Runtime verification [Giannakopoulou and Havelund 2001; Havelund and Rosu 2002; Barringer et al. 2004; Baresi et al. 2007] defines techniques to check 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 specific goal they focus on 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 rule 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 can be defined using pattern-based languages, as

described in Section 3. In particular, they allow for composition of information through conjunctions, disjunctions, negations, and sequences of events. Additionally, much work is based on logics that also include timing constraints [Drusinsky 2000; Maler et al. 2006]; this introduces a flexible way to manage time, which can be compared to user-defined windows adopted by some IFP systems. Another aspect explored by some researchers is the possibility of expressing parameterized rules through variables [Stolz 2007], which are usually associated with universal and existential quantifiers.

A notable difference between IFP systems and runtime verification tools is that the former only deal with the history of past events to produce their output, while 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 work uses a three-valued logic, where the output of the verification can be *true*, *false*, or *inconclusive* [Bauer et al. 2006b, 2010]. Other work considers 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-valued logic, including the *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. Similar to IFP systems, runtime monitoring tools consider not only the satisfaction or violation of properties (as it happens in runtime verification) but also the 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 them in the list of IFP systems analyzed in Section 4. Others are more strongly bound to the domain for which they are studied: we describe them here.

A lot of effort recently has been put into runtime monitoring of service oriented architectures (SOAs) and, more specifically, Web services. The focus is on the analysis of the quality of service compositions—to detect bottlenecks and to eliminate them whenever possible. Different rule languages have been proposed for 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 for defining 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 2005a, 2005b; 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 nonoverlapping zones, organized hierarchically. When

information moves from layer, to layer, it is aggregated through user-defined functions, which progressively reduces 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 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 for the installation of 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 websites and blogs [Luckham 2010; Etzion 2010]. An *Event Processing Technical Society* [EPTS 2010] has been founded, as well, to promote understanding and advancement in the field of event processing and to develop standards.

All these efforts and discussions put a great emphasis on possible applications and uses of CEP systems, as well as on the integration with existing enterprise solutions. For these reasons, they have received a great deal of attention from the industry, which 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 for 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 article, 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 fit different domains (from data processing to event detection) and focus 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, it would be useful to be able to characterize the differences in term of expressiveness among the different rule languages. In fact, while our analysis allows for the drawing of a line between 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 contributes to the actual expressiveness of the language and which is the minimal set of operators needed to combine both full data processing and full event detection capabilities in a single proposal.

Related with the issue of expressiveness is the ability to support uncertainty in data and rule processing. As we have seen, a single system supports it among those we surveyed, but we feel this is an area that requires more investigation, since it is something useful in several domains.

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 is common in active databases but is rarely offered by more recent IFP systems. Again, we think that this is potentially very useful in several cases and 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, that is, looking at performance of the engine with an efficient centralized or clustered implementation, or they focus on minimizing communication costs by performing filtering, correlation, and aggregation of data directly in network. Both aspects are relevant for a system 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 although the IFP domain is now mature and many systems were developed by academia and industry, there is still space for new innovative approaches, possibly integrating in a smart way the different ideas already present in the various systems we surveyed.

ACKNOWLEDGMENT

We would like to thank Professor Carlo Ghezzi for the fundamental comments he provided on both the subject and the way to better present it. We are also in debt to the anonymous referees who provided many valuable comments and suggested several improvements.

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 *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*. ACM, New York, NY, 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 J. 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 *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR2005)*. ACM.

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. *VLDB J. 13,* 2, 177–203.

AGRAWAL, J., DIAO, Y., GYLLSTROM, D., AND IMMERMAN, N. 2008. Efficient pattern matching over event streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. ACM, New York, NY, 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 *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*. ACM, New York, NY, 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*. ACM, New York, NY, 882–884.

AHMAD, Y. AND ÇETINTEMEL, U. 2004. Network-aware query processing for stream-based applications. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB'04)*. 456–467.

AKDERE, M., ÇETINTEMEL, U., AND TATBUL, N. 2008. Plan-based complex event detection across distributed sources. *Proc. VLDB. 1,* 1, 66–77.

ALERI. 2010. `http://www.aleri.com/`. Last accessed 11/2010.

ALTINEL, M. AND FRANKLIN, M. J. 2000. Efficient filtering of xml documents for selective dissemination of information. In *Proceedings of the International Conference on Very Large Databases (VLDB'00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 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 *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms (DMSSP'06)*. ACM, New York, NY, 27–37.

AMINI, L., JAIN, N., SEHGAL, A., SILBER, J., AND VERSCHEURE, O. 2006. Adaptive control of extreme-scale stream processing systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE Computer Society, Los Alamitos, CA, 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 Eng. Bull. 26*.

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, Stanford, CA.

ARASU, A., BABU, S., AND WIDOM, J. 2006. The CQL continuous query language: Semantic foundations and query execution. *VLDB J. 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, Los Alamitos, CA.

AVNUR, R. AND HELLERSTEIN, J. M. 2000. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*. ACM, New York, NY, 261–272.

BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND WIDOM, J. 2002. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGMOD/PODS Symposium on Principles of Database Systems (PODS'02)*. ACM, New York, NY, 1–16.

BABCOCK, B., DATAR, M., AND MOTWANI, R. 2004. Load shedding for aggregation queries over data streams. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*. IEEE Computer Society, Los Alamitos, CA, 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 Trans. Knowl. Data Eng. 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 *Proceedings of the 15th ACM International Conference on Information and Knowledge Management (CIKM'06)*. ACM, New York, NY, 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 *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*. USENIX Association, 15.

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

BARBON, F., TRAVERSO, P., PISTORE, M., AND TRAINOTTI, M. 2006a. Run-time monitoring of instances and classes of Web service compositions. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06)*. IEEE Computer Society, Los Alamitos, CA, 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 16th International Conference on Automated Planning and Scheduling (ICAPS'06)*. 346–349.

BARESI, L., BIANCULLI, D., GHEZZI, C., GUINEA, S., AND SPOLETINI, P. 2007. Validation of Web service compositions. *IET Softw. 1,* 6, 219–232.

BARESI, L. AND GUINEA, S. 2005a. Dynamo: Dynamic monitoring of ws-bpel processes. In *Proceedings of the 3rd International Conference on Service-Oriented Computing (ICSOC'05)*. 478–483.

BARESI, L. AND GUINEA, S. 2005b. Towards dynamic monitoring of ws-bpel processes. In *Proceedings of the 3rd International Conference on Service-Oriented Computing (ICSOC'05)*. 269–282.

BARESI, L., GUINEA, S., KAZHAMIAKIN, R., AND PISTORE, M. 2008. An integrated approach for the runtime monitoring of bpel orchestrations. In *Proceedings of the 1st European Conference on Towards a Service-Based Internet*. Lecture Notes in Computer Science vol. 5377, Springer-Verlag, Berlin, 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 *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR'07)*. 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'04)*. 44–57.

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

BAUER, A., LEUCKER, M., AND SCHALLHART, C. 2006a. Model-based runtime analysis of distributed reactive systems. In *Proceedings of the 17th Australian Software Engineering Conference (ASWEC'06)*. 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.

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)*. Lecture Notes in Computer Science, vol. 5289, 126–138.

BAUER, A., LEUCKER, M., AND SCHALLHART, C. 2010. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol*. Forthcoming.

BEERI, C., EYAL, A., MILO, T., AND PILBERG, A. 2007. Monitoring business processes with queries. In *Proceedings of the 33rd International Conference on Very Large Databases (VLDB'07)*. 603–614.

BERTINO, E., FERRARI, E., AND GUERRINI, G. 1998. An approach to model and query event-based temporal data. In *Proceedings of the International Symposium on Temporal Representation and Reasoning*, 122.

BOZDAG, E., MESBAH, A., AND VAN DEURSEN, A. 2007. A comparison of push and pull techniques for AJAX. Tech. rep. Tud-SERG-2007-016a, Delft University of Technology, Delft, The Netherlands.

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 *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. ACM, New York, NY, 1100–1102.

BRODA, K., CLARK, K., MILLER, R., AND RUSSO, A. 2009. Sage: A logical agent-based environment monitoring and control system. In *Proceedings of the European Conference on Ambient Intelligence (AmI'09)*. 112–117.

BUCHMANN, A. P., DEUTSCH, A., ZIMMERMANN, J., AND HIGA, M. 1995. The reach active oodbms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*. ACM, New York, NY, 476.

CAMPAILLA, A., CHAKI, S., CLARKE, E., JHA, S., AND VEITH, H. 2001. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*. IEEE Computer Society, Los Alamitos, CA, 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 *Proceedings of the 28th International Conference on Very Large Databases (VLDB'02)*. 215–226.

CARZANIGA, A., ROSENBLUM, D., AND WOLF, A. 2001. Design and evaluation of a wide-area event notification service. *ACM Trans. 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 19th Annual ACM Symposium on Principles of Distributed Computing*. 219–227.

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

CARZANIGA, A. AND WOLF, A. L. 2003. Forwarding in a content-based network. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'03)*. ACM, New York, NY, 163–174.

CHAKRAVARTHY, S. AND ADAIKKALAVAN, R. 2008. Events and streams: Harnessing and unleashing their synergy! In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)*. ACM, New York, NY, 1–12.

CHAKRAVARTHY, S., ANWAR, E., MAUGIS, L., AND MISHRA, D. 1994. Design of sentinel: An object-oriented dmbs with event-based rules. *Inform. Softw. Technol. 36,* 9, 555–568.

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

CHAND, R. AND FELBER, P. 2004. Xnet: a Reliable content-based publish/subscribe system. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*. 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*. ACM, New York, NY, 668–668.

CHANDRASEKARAN, S. AND FRANKLIN, M. 2004. Remembrance of streams past: Overload-sensitive management of archived streams. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB'04)*. 348–359.

CHANDRASEKARAN, S. AND FRANKLIN, M. J. 2002. Streaming queries over streaming data. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB'02)*. 203–214.

CHAUDHURI, S. 1998. An overview of query optimization in relational systems. In *Proceedings of the 17th ACM SIGACT Symposium on Principles of Database Systems (PODS'98)*. ACM, New York, NY, 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 *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR'03)*. ACM. CA.

CHI, Y., WANG, H., AND YU, P. S. 2005. Loadstar: Load shedding in data stream mining. In *Proceedings of the 31st International Conference on Very Large Databases (VLDB'05)*, 1302–1305.

CLARKE, E. M. AND SCHLINGLOFF, B.-H. 2001. Model checking. In *Handbook of Automated Reasoning*, J. A. Robinson and A. Voronkov, Eds. The MIT Press, Cambridge, MA. 1635–1790.

CORAL8. 2010a. `http://www.coral8.com/`. Last accessed 11/2010.

CORAL8. 2010b. `http://www.aleri.com/WebHelp/coral8_documentation.htm`. Last assessed 11/2010.

CRANOR, C., GAO, Y., JOHNSON, T., SHKAPENYUK, V., AND SPATSCHECK, O. 2002. Gigascope: High performance network monitoring with an sql interface. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*. ACM, New York, NY, 623–623.

CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. 2003. Gigascope: A stream database for network applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*. ACM, New York, NY, 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 *Proceedings of the 8th International Workshop on Adaptive and Reflective Middleware (ARM'09)*. ACM, New York, NY, 1–6.

CUGOLA, G. AND MARGARA, A. 2010a. Complex event processing with t-rex. Tech. rep., Politecnico di Milano, Milano, Italy.

CUGOLA, G. AND MARGARA, A. 2010b. Tesla: A formally defined event specification language. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS'10)*. ACM, New York, NY, 50–61.

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 *Proceedings of the International Symposium on 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 *Proceedings of the International Conference on Extending Database Technology (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. Tech. rep. Hs-IDA-TR-97-010, University of Skörde, Sweden.

EPTS. 2010. `http://www.ep-ts.com/`. Last accessed 11/2010.

ESPER. 2010. `http://www.espertech.com/`. Last accessed 11/2010.

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. 2010. Event processing thinking. `http://epthinking.blogspot.com/`. Last accessed 11/2010.

ETZION, O. AND NIBLETT, P. 2010. *Event Processing in Action*. Manning Publications Co., Greenwich, CT.

EUGSTER, P., FELBER, P., GUERRAOUI, R., AND KERMARREC, A.-M. 2003. The many faces of publish/subscribe. *ACM Comput. Surv. 2,* 35.

EVENT ZERO. 2010a. `http://www.eventzero.com/solutions/environment.aspx`. Last accessed 11/2010.

EVENTZERO. 2010b. `http://www.eventzero.com/`. Last accessed 11/2010.

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 Intell. 19,* 1, 17–37.

GALTON, A. AND AUGUSTO, J. C. 2002. Two approaches to event definition. In *Proceedings of the 13th International Conference Database and Expert Systems Applications (DEXA'02)*. 547–556.

GATZIU, S. AND DITTRICH, K. 1993. Events in an active object-oriented database system. In *Proceedings of the International Workshop on Rules in Database Systems (RIDS)*, N. Paton and H. Williams, Eds. Workshops in Computing, Springer-Verlag, Edinburgh, U.K.

GATZIU, S., GEPPERT, A., AND DITTRICH, K. R. 1992. Integrating active concepts into an object-oriented database system. In *Proceedings of the 3rd International Workshop on Database Programming Languages : Bulk Types & Persistent Data (DBPL3)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. ACM, New York, NY, 1123–1134.

GEHANI, N. H. AND JAGADISH, H. V. 1991. Ode as an active database: Constraints and triggers. In *Proceedings of the 17th International Conference on Very Large Databases (VLDB'91)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 327–336.

GEHANI, N. H., JAGADISH, H. V., AND SHMUELI, O. 1992. Composite event specification in active databases: Model & implementation. In *Proceedings of the 18th International Conference on Very Large Databases (VLDB'92)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 327–338.

GIANNAKOPOULOU, D. AND HAVELUND, K. 2001. Runtime analysis of linear temporal logic specifications. Tech. rep., 01. 21, Research Institute for Advanced Computer Science, Mountain View, CA.

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 *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*. ACM, New York, NY, 658–669.

GUALTIERI, M. AND RYMER, J. 2009. The Forrester Wave^{TM}: Complex Event Processing (CEP) Platforms, Q3 2009. White paper.

GYLLSTROM, D., AGRAWAL, J., DIAO, Y., AND IMMERMAN, N. 2008. On supporting kleene closure over event streams. In *Proceedings of the IEEE 24th International Conference on Data Engineering (ICDE'08)*. IEEE Computer Society, Los Alamitos, CA, 1391–1393.

HAVELUND, K. AND ROSU, G. 2002. Synthesizing monitors for safety properties. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*. 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 *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*. IEEE Computer Society, Los Alamitos, CA, 779–790.

IBM. 2008. `http://www.ibm.com/software/integration/wbe/`. Last accessed 11/2010.

IBM. 2010. `http://www-935.ibm.com/services/us/index.wss`. Last accessed 11/2010.

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 *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*. ACM, New York, NY, 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 *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, 1–20.

KHOUSSAINOVA, N., BALAZINSKA, M., AND SUCIU, D. 2008. Probabilistic event extraction from rfid data. In *Proceedings of the IEEE 24th International Conference on Data Engineering (ICDE'08)*. IEEE Computer Society, Los Alamitos, CA, 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.

KUMAR, V., COOPER, B. F., CAI, Z., EISENHAUER, G., AND SCHWAN, K. 2005. Resource-aware distributed stream management using dynamic overlays. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*. IEEE Computer Society, Los Alamitos, CA, 783–792.

LAKSHMANAN, G. T., LI, Y., AND STROM, R. 2008. Placement strategies for internet-scale data stream systems. *IEEE Internet Comput. 12,* 6, 50–60.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. 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 *Proceedings of the 13th International Conference on Very Large Databases (VLDB'04)*. 492–503.

LEUCKER, M. AND SCHALLHART, C. 2009. A brief account of runtime verification. In *Proceedings of the 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07). J. Logic Algebra. Program. 78,* 5, 293–303.

LI, G. AND JACOBSEN, H.-A. 2005. Composite subscriptions in content-based publish/subscribe systems. In *Proceedings of the 6th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, 249–269.

LIEUWEN, D. F., GEHANI, N. H., AND ARLEIN, R. M. 1996. The ode active database: Trigger semantics and implementation. In *Proceedings of the 12th International Conference on Data Engineering (ICDE'96)*. IEEE Computer Society, Los Alamitos, CA, 412–420.

LIN, E. Y.-T. AND ZHOU, C. 1999. Modeling and analysis of message passing in distributed manufacturing systems. *IEEE Trans. Syst. Man, Cybernet. Part C 29,* 2, 250–262.

LIU, H. AND JACOBSEN, H.-A. 2004. Modeling uncertainties in publish/subscribe systems. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*. IEEE Computer Society, Los Alamitos, CA, 510.

LIU, L. AND PU, C. 1997. A dynamic query scheduling framework for distributed and evolving information systems. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*. IEEE Computer Society, Los Alamitos, CA, 474.

LIU, L., PU, C., AND TANG, W. 1999. Continual queries for internet scale event-driven information delivery. *IEEE Trans. Knowl. Data Eng. 11,* 4, 610–628.

LUCKHAM, D. 1996. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. Tech. rep. Stanford University, Stanford, CA.

LUCKHAM, D. 2010. `http://complexevents.com/`. Last accessed 11/2010.

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.

LUCKHAM, D. C. AND VERA, J. 1995. An event-based architecture definition language. *IEEE Trans. Softw. Eng. 21*, 717–734.

MADDEN, S., SHAH, M., HELLERSTEIN, J. M., AND RAMAN, V. 2002. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*. ACM, New York, NY, 49–60.

MAGEE, J., DULAY, N., AND KRAMER, J. 1994. Regis: A constructive development environment for distributed programs. *Distrib. Syst. Eng. 1,* 5, 304–312.

MALER, O., NICKOVIC, D., AND PNUELI, A. 2006. From mitl to timed automata. In *Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'06)*. 274–289.

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

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

MANSOURI-SAMANI, M. AND SLOMAN, M. 1997. Gem: A generalized event monitoring language for distributed systems. *Distrib. Syst. Eng. 4*, 96–108.

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.

O'KEEFFE, D. AND BACON, J. 2010. Reliable complex event detection for pervasive computing. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS'10)*. ACM, New York, NY, 73–84.

ORACLE. 2010. http://www.oracle.com/technologies/soa/complex-event-processing.html. Last accessed 11/2010.

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

PANDEY, S., DHAMDHERE, K., AND OLSTON, C. 2004. Wic: A general-purpose algorithm for monitoring web information sources. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB'04)*. 360–371.

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. Comput. Integ. Manuf. 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 *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*. IEEE Computer Society, Los Alamitos, CA, 49.

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

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

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

PROGRESS-APAMA. 2010. http://web.progress.com/it-need/complex-event-processing.html. Last accessed 11/2010.

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

REPANTIS, T., GU, X., AND KALOGERAKI, V. 2006. Synergy: Sharing-aware component composition for distributed stream processing systems. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, 322–341.

ROITMAN, H., GAL, A., AND RASCHID, L. 2008. Satisfying complex data needs using pull-based online monitoring of volatile data sources. In *Proceedings of the IEEE 24th International Conference on Data Engineering (ICDE'08)*. IEEE Computer Society, Los Alamitos, CA, 1465–1467.

ROITMAN, H., GAL, A., AND RASCHID, L. 2010a. A dual framework and algorithms for targeted online data delivery. *IEEE Trans. Knowl. Data Eng. 99,* Preprints.

ROITMAN, H., GAL, A., AND RASCHID, L. 2010b. On trade-offs in event delivery systems. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS'10)*. ACM, New York, NY, 116–127.

ROSENBLUM, D. AND WOLF, A. L. 1997. A design framework for internet-scale event observation and notification. In *Proceedings of the 6th European Software Engineering Conference (ESEC/FSE)*. Lacture Notes in Computer Science, Springer-Verlag, Berlin, 1301.

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 *Proceedings of the International Conference on Distributed Event-Based Systems (DEBS'09)*.

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 *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*. ACM, New York, NY, 827–838.

Shah, M. A., Hellerstein, J. M., and Brewer, E. 2004b. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*. ACM, New York, NY, 827–838.

Shah, M. A., Hellerstein, J. M., Chandrasekaran, S., and Franklin, M. J. 2003. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of the International Conference on Data Engineering*. 25.

Srivastava, U. and Widom, J. 2004. Memory-limited execution of windowed stream joins. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB'04)*. 324–335.

Stolz, V. 2007. Temporal assertions with parametrized propositions. In *Proceedings of the Runtime Verification Workshop (RV'07)*. 176–187.

Streambase. 2010a. `http://www.streambase.com/`. Last accessed 10/2010.

Streambase. 2010b. `http://streambase.com/developers/docs/latest/streamsql/index.html`. Last accessed 11/2010.

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. In *Proceedings of the International Symposium on Software Reliability Engineering*.

Sullivan, M. and Heybey, A. 1998. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC'98)*. USENIX Association, Berkeley, CA, 2–2.

Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., and Stonebraker, M. 2003. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB'03)*. 309–320.

Tatbul, N. and Zdonik, S. 2006a. Dealing with overload in distributed stream processing systems. In *Proceedings of the 22nd International Conference on Data Engineering Workshops (ICDEW'06)*. IEEE Computer Society, Los Alamitos, CA, 24.

Tatbul, N. and Zdonik, S. 2006b. Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32nd International Conference on Very Large Databases (VLDB'06)*, 799–810.

Tibco. 2010. `http://www.tibco.com/software/complex-event-processing/businessevents/default.jsp`. Last accessed 11/2010.

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 *Proceedings of the International Conference on Very Large Database (VLDB'05)* 1128–1139.

Wasserkrug, S., Gal, A., Etzion, O., and Turchin, Y. 2008. Complex event processing over uncertain data. In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)*. ACM, New York, NY, 253–264.

White, W., Riedewald, M., Gehrke, J., and Demers, A. 2007. What is "next" in event processing? In *Proceedings of the 26th ACM SIGMOD-PODS Symposium on Principles of Database Systems (PODS'07)*. ACM, New York, NY, 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 *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., New York, NY, 306–325.

Wright, M., Chodzko, J., and Luk, D. 2010. *Principles and Applications of Distributed Event-Based Systems*. IGI Global, 1–18.

Wu, E., Diao, Y., and Rizvi, S. 2006. High-performance complex event processing over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*. ACM, New York, NY, 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 multimodal stream analytic and monitoring application on system s. In *Proceedings of the 33rd International Conference on Very Large Databases (VLDB'07)*. 1185–1196.

XU, J., TANG, X., AND LEE, W.-C. 2006. Time-critical on-demand data broadcast: Algorithms, analysis, and performance evaluation. *IEEE Trans. Parallel Distrib. Syst. 17*, 3–14.

YALAGANDULA, P. AND DAHLIN, M. 2004. A scalable distributed information management system. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'04)*. ACM, New York, NY, 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 *Proceedings of the OTM Conferences*. 54–71.

ZIMMER, D. 1999. On the semantics of complex events in active database management systems. In *Proceedings of the 15th International Conference on Data Engineering (ICDE'99)*. IEEE Computer Society, Los Alamitos, CA, 392.

Received November 2009; revised March 2010, November 2010; accepted November 2010

Copyright of ACM Computing Surveys is the property of Association for Computing Machinery and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.