Chapter 6
The Complex Event Processing Paradigm

Gianpaolo Cugola and Alessandro Margara

6.1 Introduction

As mentioned in the previous chapters, pervasive systems require continuous processing of information collected from multiple sources deployed in the environment under analysis. Often, a large portion of such information encodes *notifications of events* that occurred within the pervasive system or in the environment where it operates. In such cases, the goal of the processing step is to detect *situations of interest* as soon as they occur by looking at the *primitive events* that have been observed.

On the one hand, this requires the ability to define situations of interest (often called *composite events*) as *patterns* of primitive events, joined by specific relationships. On the other hand, this also demands the capability of effectively exploiting such definitions to detect composite events at runtime, as soon as they occur. *Complex event processing* (CEP) languages and systems have been proposed by research and industry to satisfy these complementary needs.

The CEP paradigm shares some similarities with the data streaming approach described in Chap. 5. Nevertheless, several key differences exist. Specifically, data stream management systems (DSMSs) focus on *transforming* the incoming streams of information into new streams, e.g., by joining or aggregating data items. Conversely, CEP systems interpret incoming information as notifications of events and focus on *detecting* relevant patterns among such streaming notifications. This

G. Cugola (✉)
Dipartimento di Elettronica, Politecnico di Milano, Informazione e Bioingegneria (DEIB), Piazza L. da Vinci 32, Milan, Italy
e-mail: gianpaolo.cugola@polimi.it

A. Margara
Faculty of Informatics, Università della Svizzera Italiana, Via Buffi 13, Lugano, Switzerland
e-mail: alessandro.margara@usi.ch

F. Colace et al. (eds.), *Data Management in Pervasive Systems*, Data-Centric Systems and Applications, DOI 10.1007/978-3-319-20062-0_6

chapter focuses on this specific processing abstraction, describing the key features and issues that characterize it.

We discuss CEP languages in Sect. 6.2, while Sect. 6.3 focuses on CEP systems. The need of processing large amounts of event notifications coming from multiple sources pushed researchers to study how to distribute event processing; we discuss the issues behind this choice in Sect. 6.4. Finally, Sect. 6.5 discusses the most advanced topics in CEP, like management of uncertainty and automated learning of relevant event patterns, and Sect. 6.6 provides some final remarks.

6.2 CEP Languages

This section discusses the main data and processing models for CEP and concretely exemplifies their usage by referring to the TESLA language [7].

6.2.1 Event Model

Event notifications encode interesting occurrences in the domain of analysis [13, 20]. They are characterized by a *time annotation* and a *payload*. The former is used to define ordering relationships among events, while the latter contains relevant information about their occurrence. For example, if an event represents the reading of a vibration from an accelerometer on a painting, then the payload could include the actual acceleration measured, the location of the sensor, and its battery status.

While data stream management systems (DSMSs) assume *homogeneous* streams [10], in which all information items have the same structure, most CEP systems process *heterogeneous* streams of input events. Accordingly, most CEP systems assume event notifications to be annotated with a *type*, which implicitly defines a structure for the payload. Referring to the previous example, a system could define a type `Vibration` for all the vibration occurrences, prescribing a payload of four fields, the first one of type `double` (the acceleration value), the second and third ones of type `string` (the identifier of the room and painting), and the last one of type `int` (the battery percentage). Several *formats* have been proposed to encode the payload of events, ranging from tuples, to key value pairs, to structured XML documents, and to RDF triples [15].

Even more significantly, several *time models* have been discussed in the literature. First of all, the time model specifies the *semantics* of time annotations. There are two common semantics [10, 30]: time annotations may be assigned based on the time indicating when an event enters the CEP system or based on some application time when events have been generated. In the former case, it is easy to define a total ordering among events. The latter case presents more difficulties, since the information items can be received out of order due to unsynchronized application

clocks at sources, network latency, and non-order-preserving communication channels.

Furthermore, the time model specifies the *encoding* of time. Most systems adopt a single timestamp, which represents a unique point in time in which the event occurs. Other systems assume that events can have a duration and adopt an interval-based representation, where two timestamps are used, indicating the lower and upper bounds of the interval of occurrence. The use of time intervals enables the definition of a rich set of temporal relations. For example, an interval I_1 can follow an interval I_2, start or finish together with I_2, and overlap or include I_2. An extensive study of the relations between time intervals is present in the pioneering work of Allen [4]. As discussed in [33], different semantics can be provided to define the temporal relations between time intervals (e.g., to define the immediate successor of an item), each of them satisfying different properties (e.g., associativity).

As a concrete example, in the following we will refer to TESLA [7], the language of the T-Rex CEP system [8]. TESLA encodes the payload of events using attribute-value pairs. Each TESLA event is characterized by a type which defines the number, order, names, and types of the attributes that build the notification. Moreover, TESLA assumes that events occur instantaneously at some points in time and encode their time of occurrence into a single timestamp. Referring to the example above, TESLA would encode a vibration reading at time `10` from painting `P` located at room `R` as follows:

```
Vibration@10(value=4.6, room='R', painting='P', battery=85)
```

6.2.2 Processing Model

Although CEP is a relatively new area of research, several CEP systems have been developed in the last few years, each one proposing a different processing model. Nevertheless, it is possible to devise some key commonalities among such models. Indeed, at an abstract level, all systems are based on rules that define composite events starting from patterns of primitive events observed from the environment under analysis. Rules do not explicitly provide the processing steps to be performed; on the contrary, the computation is implicitly specified by the pattern.

The set of operators that allowed inside patterns changes from system to system, but some of them are common: *selection* of primitive events relevant for processing based on their payload; *combination* of multiple events based on their mutual relations in terms of payload and time; *negation*, to identify events that must *not* occur in order to satisfy the pattern; *aggregation* of payload data from multiple primitive events; and *production* of new (composite) events. In the remainder of this section, we present these operators in detail using TESLA as the reference language.

Selection and Combination Rule R1 below *selects* and *combines* two primitive events (`Vibration` and `PeopleNear`) to define a composite event `Touch`

that expresses the potential touch of a given painting. In TESLA the occurrence of a composite event is always bound to the occurrence of an observed event (`Vibration` in our example), which implicitly determines the time at which the new event is detected. This anchor point, called the *terminator* of the rule, is coupled with other events (`PeopleNear` in our example) through *combination* operators (`each-within` in our example).

Listing 6.1 Rule R1

```
define  Touch(room: string, painting: string)
from    Vibration(painting=\$p and value>3.0) and
        each PeopleNear(painting=\$p) within 2 min. from ↩
            Vibration
where   room=Vibration.room and painting=Vibration.painting
```

The *selection* operator restricts valid `Vibration` events based on their `value` attribute (which must be greater than `3.0`). The `combination` operator binds together `Vibration` and `PeopleNear` notifications that (1) refer to the same painting (through the *parameter* constraint `painting=$p`) and (2) occur within 2 min from each other, with `PeopleNear` preceding `Vibration`.

Rule R1 adopts the `each-within` combination operator, which potentially generates multiple, simultaneous composite events for each terminator (e.g., a `Touch` event for each `PeopleNear` event that matches the occurring `Vibration` event). TESLA also provides other composition operators, e.g., the `last-within` and `first-within` operators would generate a single `Touch` event for every `Vibration` event, binding it to the last or the first `PeopleNear` event that satisfies the prescribed time and content constraints. The `where` clause is used to define the values of the attributes for the produced `Touch` events.

It is worth noticing that different systems provide different trade-offs between expressiveness and processing efficiency. In particular, some systems define simpler semantics for combination, e.g., by restricting combination to contiguous events [5].

Negation Rule R2 below exemplifies the use of a *negation*, demanding a `StaffAt` event not to occur in order to distinguish between touches and potential theft.

Listing 6.2 Rule R2

```
define  Theft(room: string, painting: string)
from    Vibration(room=\$r and painting=\$p and value>3.0) and
        each PeopleNear(painting=\$p) within 5 min. from ↩
            Vibration
        and not StaffAt(room=\$r) within 1 min. from Vibration
where   room=Vibration.room and painting=Vibration.painting
```

Aggregation Rule R3 below exemplifies the use of an *aggregation*, which computes the average value over all the `Vibration` events received in the last 5 min to decide if a potential theft is occurring.

Listing 6.3 Rule R3

```
define  Theft(room: string, painting: string)
from    Vibration(room=\$r and painting=\$p and value>3.0) and
        v=Avg(Vibration(room=\$r and painting=\$p)
              within 5 min. from Vibration).value > 3.0
where   room=Vibration.room and painting=Vibration.painting
```

Hierarchies of Events As a final remark, we notice that several systems, including TESLA, enable composite events to take part of patterns that define other composite events. This allows users to build hierarchies of events, where (intermediate) composite events can be used to define other (higher-level) composite events. By allowing output events to reenter the system, this feature enables the definition of recursive rules. Other systems provide a similar expressivity through ad hoc operators that specify trends or unbounded repetitions of events (e.g., all vibration events having values that increase over time) [2].

6.3 Processing Algorithms

This section provides an overview of the main data structures and algorithms used in CEP systems. Due to lack of space, we mainly focus on combination operators, which constitute the core abstraction offered by CEP systems. The interested reader can find additional details in [8, 9, 21]. We present and compare two different approaches, one based on automata and incremental processing (Sect. 6.3.1) and one based on columns and delayed processing (Sect. 6.3.2). Furthermore, we describe how the latter approach can be accelerated by exploiting the parallelism offered by modern hardware architecture (Sect. 6.3.3).

6.3.1 Automata-Based Processing

The main goal of a processing algorithm is to detect patterns of events organized into temporal sequences and also satisfy additional constraints involving their payload (e.g., selection and parameter constraints). At a high level, this problem shares many similarities with the problem of detecting regular expressions into streams of characters, which is usually solved using automata. Because of this, several

existing systems, both from academia [1, 5, 29] and from industry,[1] adopt automata-based approaches to process events. In the following, we show the main concepts behind such approaches by describing the AIP (Automata Incremental Processing) algorithm used by T-Rex [8] to deal with TESLA rules.

When adopting the AIP algorithm, each TESLA rule is translated into an *automaton model*, which is composed of one or more *sequence models*. At the beginning, each sequence model is instantiated in a *sequence instance* (or simply *sequence*). Upon a new event arrival, three actions may be taken: (1) new sequences can be created by duplicating existing ones; (2) existing sequences can be moved from a state to the following one; (3) existing sequences can be deleted, either because they arrive to an accepting state or because they are unable to proceed any further.

Creation of Automata As described in Sect. 6.2, each TESLA rule filters event notifications according to their type and content and uses the `*-within` operators to define one or more sequences of events. Let us consider again Rule R1 to exemplify how automata are created. It defines a sequence in which a `PeopleNear` event precedes a `Vibration` event. Furthermore, it requires the value of `Vibration` to be greater than `3.0`, and it forces the two events to refer to the same painting.

Figure 6.1 shows the translation of Rule R1 into an automaton model. AIP creates one *sequence model* for each sequence in the rule (only one in the case of Rule R1). A sequence model is a linear, deterministic, finite state automaton. Each event in the sequence captured by R1 is mapped to a state in the sequence model, and a transition between two states s_1 and s_2 is labeled with the *type*, *content*, and *timing* constraints that an incoming event has to satisfy to trigger the transition. Additional constraints (e.g., parameters) are expressed using dashed lines connecting two states.

Detection Algorithm At the beginning, a single sequence is instantiated from each sequence model built from existing rules. When a new event e arrives, the algorithm reacts as follows: (1) it checks whether the type, content, and arrival time of e satisfy a transition for one of the existing sequences; if not, the event is immediately discarded. If a sequence `Seq` in a state s_1 can use e to move to its next state s_2, the algorithm (2) creates a copy `S'` of `Seq`, and (3) uses e to move `S'` to state s_2. Notice that the original sequence `Seq` remains in state s_1, waiting for further events. Sequences are deleted when it becomes impossible for them to proceed to the next

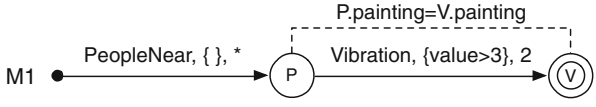

Fig. 6.1 Event detection automata for Rule R1

[1] Esper, http://esper.codehaus.org

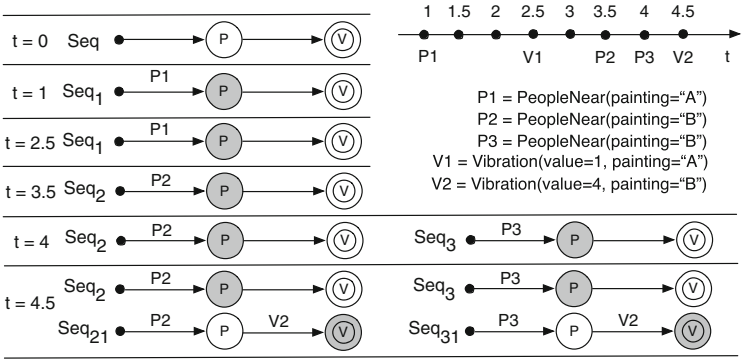

Fig. 6.2 An example of automata processing

state since the time limits for future transitions have already expired. A sequence in its initial state is never deleted as it cannot expire.

As an example of how processing of sequences works, consider Rule R1 and the corresponding model `M1`. Figure 6.2 shows, step by step, how the set of incoming events drawn at the upper right corner is processed. At time `t=0`, a single sequence `Seq` of model `M1` is present in the system, waiting in its initial state. Since `Seq` does not change with the arrival of new events, we omit it in the figure for `t>0`. At time `t=1`, an event `P1` of type `PeopleNear` is received. Since it matches type, content, and timing constraints for the next transition of `Seq`, we clone it by creating `Seq1`, which advances to state `P`. At `t=2.5`, a `Vibration` event arrives, but its value is too low to satisfy Rule R1, so it is immediately discarded. At time `t=3.5`, a new event `P2` enters the system. It generates a new sequence `Seq2` (cloning `Seq`) and advances it to state `P`. At the same time, `Seq1` is canceled, since the time limit for its next state transition has expired. At time `t=4`, event `P3` generates a new sequence `Seq3` and advances it to state `P`. At time `t=4.5`, an event `V2` is received. As we are considering a `each-within` operator, we use `V2` to advance every matching sequence in state `P`. Since `V2` satisfies the constraints of both `Seq2` and `Seq3`, it is used to duplicate both of them, generating and advancing two new sequences, `Seq21` and `Seq31`. `Seq21` and `Seq31` arrive at the accepting state: this means that two valid sequences, composed of events `P2`–`V2` and events `P3`–`V2`, have been recognized. This triggers the generation of two composite events. After that, sequences `Seq21` and `Seq31` are deleted, while `Seq2` and `Seq3` remain in the system, waiting for potential future events of type `Vibration`.

As a final remark, we observe that our example adopts the `each-within` operator. The presence of other composition operators (e.g., `last-within`) may enable early deletion of sequences, which optimizes the use of resources.

6.3.2 Columns-Based Processing

While the AIP algorithm processes rules incrementally, as new events enter the system, an alternative approach consists in collecting events until a terminator is found. As in the previous case, we present this approach by referring to the CDP (Column Delayed Processing) algorithm implemented into T-Rex, which organizes events into columns, one for each event type that appears in the sequence defined by R, until a terminator is found.

Creation of Columns As an example of how columns are created by the CDP algorithm, consider again Rule R1. For this rule, CDP creates two columns (see Fig. 6.3), each labeled with the type of the primitive events it stores and with the set of constraints on their content. The maximum time interval allowed between the events of a column and those of the previous one (i.e., the window expressed through the `*-within` operator) is modeled using a double arrow. Similarly, additional constraints coming from parameters are represented as dashed lines. Notice that the last column reserves space for a single event.

Detection Algorithm When a new event e enters the system, it is processed as follows. First, CDP checks whether it matches (i.e., satisfies type and payload constraints of) one or more columns. If this is the case, e is added on top of the matching columns; otherwise it is immediately discarded. If among the matched columns there is the terminating one (c_ℓ), the processing of the events stored so far starts. Processing is performed column by column, from the last one to the first one, creating *partial sequences* of increasing size at each step. More precisely:

- The timestamp of e is used to find the index i of the first valid element in column $c_{\ell-1}$, by looking at the time window.
- All events in column $c_{\ell-1}$ having an index $i' < i$ are deleted, since they have no chance to enter the window in the future.
- The operation is repeated for each column, considering the timestamp of the first event left in column c_k to delete old events from column c_{k-1}.
- e is combined with all the events stored in column $c_{\ell-1}$ that satisfy timing constraints, parameters, and selection policies, creating partial sequences of two events.

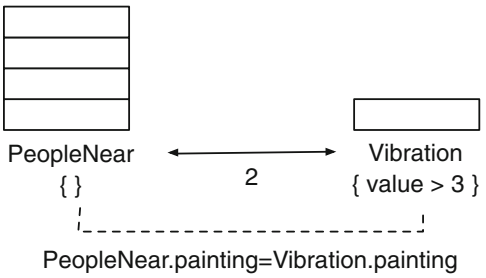

Fig. 6.3 Columns for Rule R1

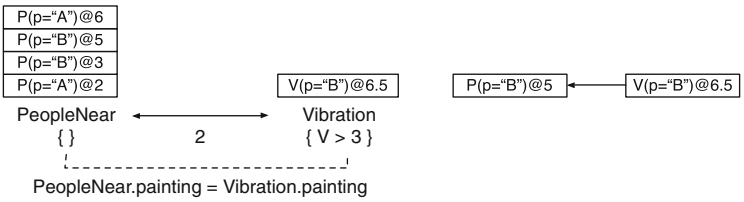

Fig. 6.4 An example of processing using columns

- Each partial sequence is used to select elements from the previous column $c_{\ell-2}$. The algorithm is repeated recursively until the first column is reached, generating zero, one, or more sequences including one selected event from each column.
- One composite event is generated for each sequence.

To better understand how the algorithm works, consider again Rule R1 and the situation in Fig. 6.4, where the events stored in each column are represented with their type, their value for the attribute `painting` (`p`), and their timestamp. The event `V(p='B')@6.5` was the last to enter the system. Since it is a terminator for Rule R1, it starts the processing algorithm. Its timestamp is used to permanently discard elements of the previous column that are too old to satisfy timing constraints. In particular, we discard all events having a timestamp lower than `4.5` (`6.5-2`). This operation is performed recursively in the presence of more than two columns. The remaining events (i.e., `P(p='B')@5` and `P(p='A')@6`) are evaluated to detect valid sequences. The former matches the constraint on the `painting` attribute and is used to create the valid sequence shown on the right of Fig. 6.4. On the contrary, the latter does not match the attribute constraint. Both events are kept in the leftmost column, waiting for the arrival of further `Vibration` events.

6.3.3 Exploiting Parallel Hardware

One of the key benefits of the CDP algorithm consists in the simple layout of its data structures, which makes it suitable for running on parallel hardware architectures. To exemplify this feature, we present an implementation of CDP on Compute Unified Device Architecture (CUDA) graphics processing units (GPUs).

CUDA CUDA is a parallel computing architecture introduced by Nvidia in 2006 to offer a new parallel programming model and instruction set for general-purpose programming on GPUs. CUDA has been exploited in several domains to speed up complex computations. However, attaining good performance with CUDA is challenging, and only some algorithms can be effectively ported to such parallel architecture. Next, we briefly discuss the main features of CUDA and how they affect the design of algorithms. While we focus on CUDA, most of the concepts

presented below apply in general to several modern parallel architectures (e.g., Xeon Phi) and programming API (e.g., OpenMP).

The CUDA programming model assumes that CUDA threads execute on a *device* (the GPU), which operates as a coprocessor to a *host* (the CPU) and has its own separate memory space. The programmer starts a new computation on the device by invoking a function, called *kernel*, which defines a single flow of execution. CUDA implements a Single Program Multiple Threads paradigm: the kernel is executed in parallel by a number of threads on different data. Threads can be combined in (multidimensional) groups. Inside the kernel, each thread is identified by a *groupId* and a *threadId* variable. Conditional statements involving these variables are the only way for a programmer to differentiate the execution flows of different threads.

There are two main factors to consider while programming in CUDA, which can severely impact on performance. (1) A modern GPU can run thousands of threads concurrently, but its hardware is designed for data parallel executions. Because of this, full efficiency is achieved only if all the threads agree on their execution path. If threads diverge via a data-dependent conditional branch, CUDA serially executes each branch path taken. (2) Often, retrieving data from memory constitutes the main bottleneck for CUDA. To alleviate this issue, it is necessary to design the data structures in such a way that threads with contiguous threadId access contiguous memory regions. This enables the hardware to fully exploit the available memory bandwidth.

Accelerating CDP with CUDA The complexity of AIP's data structures prevents the possibility of efficiently implementing it in CUDA. Accordingly, only CDP has been ported to CUDA.

In CUDA, the GPU memory is pre-allocated by the CPU and this operation has a non-negligible latency. Because of this, CDP implements each column as a statically allocated circular buffer. Furthermore, it keeps a copy of each column into the CPU memory, which allows to perform on the CPU all the operations that do not benefit from parallelization. In particular, when an event e enters the system and matches a state s for a rule r, it is added to the column for s in main memory. Then, a copy of the event to the GPU memory is issued asynchronously, such that the CPU does not need to wait for the copy to end. If e is a terminator for r, the CPU computes which events need to be considered for each column, starting from their timestamp and from the time windows in r. CDP performs this operation on the CPU since it requires a sequential analysis of the columns (from the last to the first one), and the computation inside each column can be efficiently performed by a standard CPU through a binary search. At the end of this phase, the CPU invokes the GPU to process the relevant events from each column.

When using the `each-within` operator to process a column c, each partial sequence generated at the previous step may be combined with more than one event in c. The algorithm performs the following steps:

- It allocates two arrays of sequences, called seq_{in} and seq_{out}, used to store the input and output results of each processing step. Sequences are represented as fixed-size arrays of events, one for each state defined in the rule.

- It allocates an integer *index* and sets it to 0.
- At the first step, seq_{in} contains a single sequence with only the last position occupied (by the received terminator).
- When processing a column c, a different thread t is executed for each event e in c and for each sequence seq in seq_{in}.
- t checks if e can be combined with seq, i.e., if it matches timing and parameter constraints of seq.
- If all constraints are satisfied, t uses a special CUDA operation to atomically read and increase the value of *index*. The read value k identifies the first free position in the seq_{out} array: the thread adds e to seq in position c and stores the result in position k of seq_{out}.
- When all threads have finished, the CPU copies the value of *index* into the main memory and reads it.
- If the value of *index* is greater than 0, it proceeds to the next column by resetting the value of *index* to 0 and swapping the pointers of seq_{in} and seq_{out} into the GPU memory.
- The algorithm continues until *index* becomes 0 or all the columns have been processed. In the first case, no valid sequence has been detected, while in the second case, all valid sequences are stored in seq_{out} and can be copied back to the CPU memory.

To better understand how the CDP algorithm works, consider the example in Fig. 6.5. It shows the processing of a rule R defining a sequence of three primitive events. Two columns have already been processed resulting in six partial sequences of two events each, while the last column c to be processed is shown in the figure. Since there are six sequences stored in seq_{in} and seven events in c, the computation requires 42 threads. Figure 6.5 shows one of them, thread T, which is in charge of processing the event E_3 and the partial sequence S_3. Now suppose that E_3 satisfies all the constraints of Rule R and thus can be combined with S_3. T copies E_3 into the first position of S_3; then, it reads the value of *index* (i.e., 3) and increases it. Since this operation is atomic, T is the only thread that can read 3 from *index*, thus

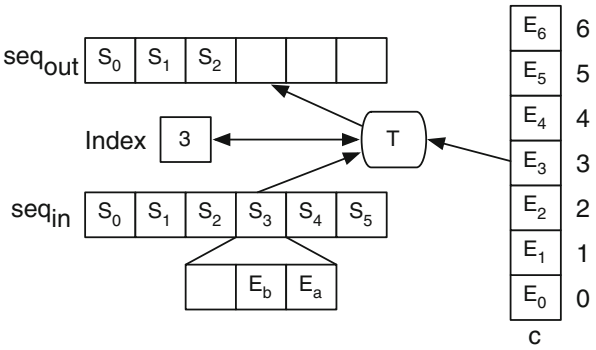

Fig. 6.5 CDP algorithm on CUDA (each-within operator)

avoiding memory clashes when it writes a copy of S_3 into the position of index 3 in seq_{out}.

In this implementation, threads with contiguous identifiers are used to process contiguous positions in a column: this increases the performance of memory access, since the hardware can combine operations issued by different threads.

6.3.4 Performance Analysis

This section provides an overview of the performance of the algorithms described above. All the results discussed below have been collected using a 2.8 GHz AMD Phenom II PC, with 6 cores and 8 GB of DDR3 RAM, running 64 bit Linux.

Figure 6.6 compares the performance of the CPU implementation of AIP and CDP. We deploy 1000 rules in the system, all of them defining a sequence composed of a variable number of events. Each incoming event is relevant for exactly 1 % of the deployed rules; each rule (and each state inside a rule) has the same probability to select incoming events. We consider an average window size of 15 s between two consecutive events in a sequence.

First, Fig. 6.6 highlights the efficiency of both the AIP and the CDP algorithms. We consider two scenarios: the first adopts the `each-within` operator to define sequences (Fig. 6.6a), while the second adopts the `last-within` operator (Fig. 6.6b). In both cases, the considered algorithms can easily process events in sub-millisecond time. Second, as expected, managing the `each-within` operator is more expensive—more primitive events need to be combined and more composite events are generated—and results in higher processing times for both algorithms. Third, Fig. 6.6 shows that a traditional approach based on automata is less efficient than the CDP algorithm.

As a second step, we performed an additional experiment to test the benefits of adopting massively parallel hardware. In this experiment, we deploy a single rule r defining a sequence of length 3. We repeat the measurement with different sizes of windows, which is probably the parameter that affects a GPU implementation the most. Larger window sizes increase the average number of events to be considered

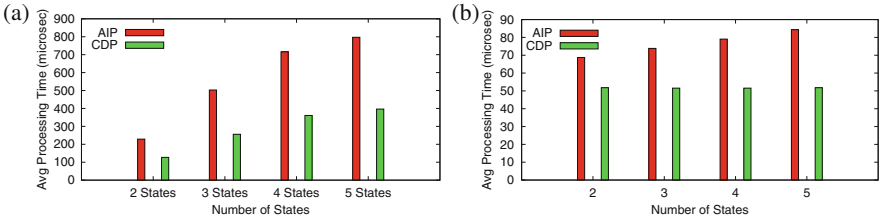

Fig. 6.6 Comparison of automata-based and column-based algorithms. (**a**) Each-within. (**b**) Last-within

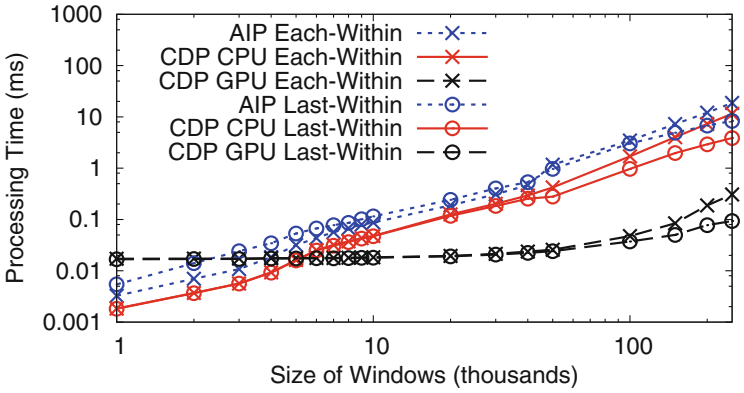

Fig. 6.7 Analysis of the benefits of parallel hardware

at each state. While the CPU processes those events sequentially, the GPU uses different threads running in parallel. Furthermore, using the GPU involves an (almost constant) overhead to transfer input data from the CPU to the GPU memory and to activate a CUDA kernel. This makes the usage of the GPU convenient only if a significant number of events to be processed is present at each state.

Figure 6.7 emphasizes this behavior: on one hand, the cost of the algorithms running on the CPU grows with the size of windows. On the other hand, the cost of the CDP algorithm running on the GPU is initially constant at 0.017 ms (it is dominated by the fixed overhead of CUDA), and it starts growing only when the number of available threads is not enough to compute events entirely in parallel. This growth is faster with the `each-within` operator, which uses more threads and produces more composite events to be transferred back to the main memory. In our test, the smallest size of windows that determines an advantage in using the GPU is 4000. With a sequence of three states, this results in considering 1333 events in each state, on average. This result isolates one of the most significant dimensions for deciding the hardware architecture to adopt. If an application involves rules that need to process a small number of events for each state, then the CPU represents a better choice. Otherwise, the advantages of parallel processing favor the use of a GPU.

6.4 Protocols for Distributed Event Detection

In the previous sections, we assumed that primitive events were delivered to a single node responsible for detecting composite events and delivering them to the interested recipients. However, several application domains may involve components dispersed over a wide geographical area. This is certainly the case for many pervasive systems. In these settings, centralizing the processing in a single

node is not ideal, since it prevents from exploiting the locality of information: even if a (small) group of nodes holds all the information required to detect a composite event, it still needs to rely on the centralized processing server and cannot operate autonomously. This increases the time for detecting composite events and the amount of data exchanged through the network, which, in turn, negatively impacts the lifetime of battery-powered devices. Finally, a centralized solution may hamper scalability when the processing resources of the processing node are not sufficient.

6.4.1 Distribution Strategies

To overcome the limitations described above, some systems proposed *distribution strategies* to enable a decentralized processing of rules. A distribution strategy involves (1) an algorithm to decide how the processing load is partitioned among different processing nodes and (2) a communication protocol that defines how the nodes interact and communicate with each other to produce the required results.

The first algorithm solves the *operator placement* problem: it searches for the best mapping of the operators defined in rules on the set of available nodes. Depending on the application, the operator placement may pursue different goals, e.g., reducing the latency to detect and notify composite events or minimizing the usage of network resources. Given the complexity of this problem, several works addressed it using approximated algorithms and heuristics [19]. They usually rely on a centralized decider, which collects information about the status of the nodes and locally computes a suitable deployment of operators. Only a few solutions considered decentralized algorithms [25]. Finally, most operator placement algorithms are studied for cluster infrastructures, in which all processing nodes are colocated and well connected [17, 34]: in this setting, the operator placement problem essentially translates into a load balancing problem.

Besides operator placement, a distribution strategy also requires a communication protocol to govern the interaction among processing nodes, specifying how rules are deployed and how primitive and composite events are forwarded. These issues are rarely considered by existing CEP systems: even when distributed processing is allowed, the communication among nodes requires manual configuration [3].

In the remainder of this section, we briefly introduce some concrete examples of distribution strategies for the T-Rex system [11].

6.4.2 A Concrete Example: Distribution Strategies for T-Rex

To illustrate possible distribution strategies for T-Rex, we consider a set of processing nodes P connected with each other in a physical network. To simplify the routing, our deployment strategies organize nodes into one or more *processing trees*

on top of the physical network. More precisely, they use a processing tree to collect primitive events from sources (the leaves) and to filter and (partially) process them as they move toward the root of the tree, where the composite events are generated. This enables incremental evaluation of rules at intermediate nodes, reducing the amount of information flowing along the tree and the processing load at the root node. The same tree adopted for detecting a composite event *ce* is also used to distribute *ce* to the interested clients. Since we want to minimize latency, we build *Shortest Path Trees* using the link delay as a cost metric.

Single Tree vs. Multiple Trees We consider two classes of deployment strategies. The first one organizes all nodes into a single processing tree. One node ℓ is elected as the network leader and all events move from the sources to ℓ going up along the tree rooted at ℓ (T_ℓ), while they get incrementally evaluated according to the rules deployed in the system. When they reach ℓ, the processing is complete, and the corresponding composite events are generated and delivered to the sinks along T_ℓ.

The second class of strategies builds one tree for each client interested in a composite event (for each *sink*). Primitive events flow from the sources along multiple trees. In particular, if a sink s is interested in a composite event *ce*, all primitive events that contribute to *ce* move from their sources up along T_s. Processing is performed incrementally on each tree. When a composite event reaches the root of a processing tree, there is no need to further distribute it, since the root of the tree coincides with the interested client. This approach removes the need for spreading composite events once they have been detected at the cost of duplicating primitive events that must be forwarded over multiple trees.

Partitioning TESLA Rules To enable incremental evaluation on a processing tree T, rules are recursively partitioned into *partial rules* moving from the root to the leaves of T. The goal is to push the processing of events as close as possible to the sources where events are generated. To do so, nodes first declare the type of events they will produce (e.g., an accelerometer will produce only `Vibration` events). This knowledge is used for partitioning: the parts of a rule responsible for detecting a certain event e are deployed as close as possible to the nodes that produce e.

Next, we offer an overview of the partitioning algorithm. The interested reader can refer to [6, 11] for further details. Let us consider Rule R4 and the processing tree T_1 (Fig. 6.8), rooted at 1 and containing three sources: 3 produces events of types `A` and `B`; 4 produces events of types `C` and `E`; 5 produces events of type `D`. This information is available to node 2. Similarly, node 1 knows that it can receive events of types `A`, `B`, `C`, `D`, and `E` from node 2.

Listing 6.4 Rule R4

```
define    CompEvent()
from      A() and last B() within 5 min. from A
          and last C() within 5 min. from B
          and last D() within 5 min. from C
          and last E() within 5 min. from D
```

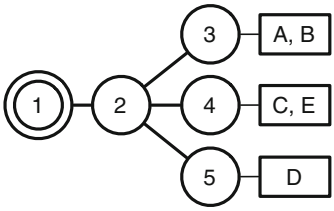

Fig. 6.8 Rule deployment: an example

Partitioning is performed as follows: 1 observes that 2 receives all the information necessary to correctly evaluate the rule. Accordingly, it entirely delegates the processing of *R*4 (including the generation of composite events) to 2. 2 observes that none of its children has enough information to process Rule R4. Accordingly, 2 remains responsible for producing composite events while it delegates only parts of the processing to 3, 4, and 5 in the form of partial rules.

As mentioned, partial rules are used to filter primitive events as close as possible to sources. A node p responsible for a partial rule r' forwards a set of primitive events to its parent in the processing tree only when this satisfies the pattern in r'. Let us consider node 3: its clients are the only sources of events `A` and `B`. To correctly process Rule R4, node 2 does not need to receive all events of types `A` and `B` but only events `A` that are preceded by an event `B` in the previous 5 min; moreover, only the last `B` event before each `A` is relevant (Rule R4 uses the `last-within` operator). Accordingly, 2 creates the following partial rule for 3:

```
A() and last B() within 5 min. from A
```

Similarly, 2 does not need to receive all `C` events but only those preceded by an event of type `E`. Accordingly, it creates and sends the following partial rule to 4:

```
C() and each E() within 10 min. from C
```

Notice that `C` and `E` are not contiguous elements in the sequence defined by Rule R4, but they are separated by event `D`. Because of this, the partial rule considers a window that sums the one between `C` and `D` and the one between `D` and `E`. Similarly, the local knowledge of node 4 is not sufficient to evaluate the single selection constraint on `E`; for this reason, the partial rule adopts the `each-within` operator, capturing all notifications of `E` followed by a `C` event within 10 min. Finally, node 5 receives a partial rule that simply asks for all events of type `D`.

The partitioning algorithm described above is applied recursively: partial rules are split into other partial rules until all sources have been reached.

Forwarding of Events Once detected, a primitive event p is forwarded along the relevant trees, i.e., those associated to rules that involve p (a decision taken by looking at the type of p). In some cases, this *push-based forwarding* strategy can be complemented by a *pull-based forwarding* strategy that tries to limit network traffic by pulling events of a certain type only when other events they are related to have been detected. As an example, considering Rule R1 described in Sect. 6.2,

it may be better to avoid forwarding `PeopleNear` events until a (less frequent) `Vibration` event is detected (for further details, see [11]).

6.4.3 *Performance Analysis*

In the following, we report some measurements conducted in a simulated environment to show the benefits of distributed processing on reducing the network traffic and the delay for delivering events. In particular, we consider three different strategies: `ST` performs distributed processing on a single tree; `MT` performs distributed processing on multiple trees; `Centr` exploits a single processor, which receives primitive events, processes them, and delivers composite events to interested sinks.

Tests are performed in a scenario that includes 20 nodes, each one connected with five others, on average. Sources produce 120 different types of primitive events with a generation rate between 1000 s and ten notifications per second, with exponential distribution. We deploy 100 TESLA rules, each one including a sequence of three events with time windows of 1 min, on the average. Each rule produces a different composite event, and a set of sinks connected with each processor is interested in ten of them, on average.

Figure 6.9a shows the delay for delivering events. First, we observe that `MT` strategies provide much lower delays with respect to `ST` strategies. Indeed, `MT` strategies do not need to deliver composite events after detection, thus eliminating the delay introduced in this phase. `ST` strategies perform similarly to a centralized scenario: both need to deliver composite event notifications after detection.

Network traffic is significantly higher in the `Centr` strategy (Fig. 6.9b). This means that distributed processing effectively enables the filtering of a large number of primitive events close to their sources. If we compare the two distributed strategies, we observe that `MT` generates more traffic than `ST`. While the former does not require the forwarding of composite events after detection, it demands for the forwarding of primitive events along multiple trees.

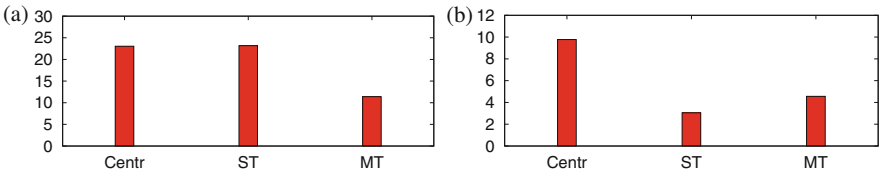

Fig. 6.9 A comparison of distribution strategies for T-Rex. (**a**) Average delay (ms). (**b**) Average traffic (KB/s)

6.5 Advanced Topics

This section introduces some advanced topics in the CEP domain that are currently investigated in research. It focuses on three topics relevant for pervasive systems: management of uncertainty, automated generation of CEP rules, and integration with programming languages.

Management of Uncertainty CEP technologies are used to model and capture phenomena of interest. In this context, the accuracy in modeling the domain of analysis is fundamental for a successful adoption of CEP. At the same time, human ability to define and capture a phenomenon is often affected by some form of uncertainty, which, if ignored, may lead to incomplete, inaccurate, or even incorrect decisions concerning the phenomenon itself.

A successful management of uncertainty in CEP involves three main steps: *identification* of the sources of uncertainty, *modeling* of uncertainty, and *propagation* of uncertainty across the systems, from primitive to composite events.

The identification step requires an analysis of the environment and of the specific phenomena to capture. In general, we can identify two main sources of uncertainty [12]: uncertainty in the input data and uncertainty in the CEP rules. The former refers to the presence of incomplete, incorrect, or imprecise information observed from the external environment. This is extremely relevant for pervasive systems. For example, in the case of input data coming from sensors, the value measured and propagated may be imprecise (because of the limited accuracy in the sensor) or incomplete (because of communication or battery-related issues). The latter refers to the imprecision of the rules in correctly formulating the causal relations between the primitive and the composite events. This may derive from a limited knowledge of the environment under analysis but also from the impossibility to observe all the aspects that may influence the occurrence of a phenomenon.

The modeling step aims at providing a sound mathematical foundation to represent uncertainty, let the CEP engine be aware of it, and manipulate it consistently. For example, probability theory can be used to model measurement errors, allowing the CEP engine to process uncertain values and combine them with other ones.

Finally, an uncertainty-aware CEP system should propagate the uncertainty by producing results that are annotated with uncertainty information consistent with the identified sources of uncertainty and the models adopted to represent them.

The recent literature presents several proposals for capturing uncertainty in CEP [12, 18, 27, 31, 32]. Most of these solutions rely on an explicit encoding and representation of uncertainty inside data items. For instance, [12] adopts random variables to represent received information, thus making it possible to encode measurement errors in sensor applications.

There are various metrics for evaluating a model for uncertainty, including expressiveness, precision, computational cost, and simplicity. While end consumers may be interested in receiving precise indications about the level of certainty associated to the results, this should not negatively impact the compactness and readability of information or the level of performance of the CEP system.

Automated Generation of Rules Our discussion of uncertainty highlights key issues of current CEP systems: while research and development efforts were mainly directed toward processing efficiency, a widespread adoption of CEP technologies depends on the capability to correctly and precisely model the phenomena under analysis. As our discussion in Sect. 6.2 shows, this task can be extremely difficult and involve several different aspects, including identification of the relevant primitive events, filtering of their content, and identification of their mutual relation in terms of content and time ordering.

To overcome this problem, researchers are currently focusing on defining techniques to support users in rule definition. In particular, some preliminary results have been achieved in the area of learning CEP rules from the available historical information about the environment under analysis [23]. These techniques adopt automated algorithms that analyze past occurrences of the phenomena of interest and suggest CEP rules for detecting them.

Language Integration As discussed so far, CEP represents a mainstream technology for promoting the interaction of components in complex software systems, typical in many scenarios of pervasive systems. In this context, CEP becomes a key component for *programming* the overall behavior of the software architecture. Because of this, some research efforts targeted the integration of event processing within programming languages and frameworks.

In this context, we can distinguish two main areas of investigation. On the one hand, some languages have been defined that provide primitives to send and receive events. In these proposals, events become first-class objects of the language and are fully integrated, e.g., with the type system. Furthermore, some CEP operators for event composition and pattern detection are offered as language construct. Examples of this approach are EventJava [14], Ptolemy [26], and EScala [16].

On the other hand, some systems are built on top of event processing to define a novel programming paradigm known as *reactive programming*. In reactive programming, developers can define *reactive* variables through an expression that involves other (reactive or imperative) variables. Whenever one variable changes, all dependent variables get automatically updated. While notifications of changes are implemented as events, they are not exposed to programmers, who only observe their effect (i.e., changes to the values of reactive variables). Examples of reactive systems are REScala [28], Flapjax [24], and DREAM [22].

6.6 Conclusions

This chapter provided an overview of CEP technology, considering both the operators it offers to applications and some key algorithms and implementation mechanisms used to achieve high-performance processing and scalability.

In the domain of pervasive systems, CEP represents an ideal solution to guide the interactions of a plethora of distributed components. Indeed, CEP enables domain

experts to focus on the modeling of the application domain and to entirely delegate to the CEP system the task of observing and responding to the stimuli of the application environment according to such modeling.

References

1. Adi, A., Etzion, O.: Amit - the situation manager. VLDB J. **13**(2), 177–203 (2004)
2. Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N.: Efficient pattern matching over event streams. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, pp. 147–160. ACM, New York (2008)
3. Ali, M.: An introduction to microsoft sql server streaminsight. In: Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research and Application, COM.Geo '10, pp. 66:1–66:1. ACM, New York (2010)
4. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM **26**(11), 832–843 (1983)
5. Brenna, L., Demers, A., Gehrke, J., Hong, M., Ossher, J., Panda, B., Riedewald, M., Thatte, M., White, W.: Cayuga: a high-performance event processing engine. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07, pp. 1100–1102. ACM, New York (2007)
6. Cugola, G., Margara, A.: Raced: an adaptive middleware for complex event detection. In: Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware, ARM '09, pp. 5:1–5:6. ACM, New York (2009)
7. Cugola, G., Margara, A.: Tesla: a formally defined event specification language. In: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS '10, pp. 50–61. ACM, New York (2010)
8. Cugola, G., Margara, A.: Complex event processing with t-rex. J. Syst. Softw. **85**(8), 1709–1728 (2012)
9. Cugola, G., Margara, A.: Low latency complex event processing on parallel hardware. J. Parallel Distrib. Comput. **72**(2), 205–218 (2012)
10. Cugola, G., Margara, A.: Processing flows of information: from data stream to complex event processing. ACM Comput. Surv. **44**(3), 15:1–15:62 (2012)
11. Cugola, G., Margara, A.: Deployment strategies for distributed complex event processing. Computing **95**(2), 129–156 (2013)
12. Cugola, G., Margara, A., Matteucci, M., Tamburrelli, G.: Introducing uncertainty in complex event processing: model, implementation, and validation. Computing **97**(2), 103–144 (2015)
13. Etzion, O., Niblett, P.: Event Processing in Action, 1st edn. Manning Publications Co., Greenwich (2010)
14. Eugster, P., Jayaram, K.: Eventjava: an extension of java for event correlation. In: Drossopoulou, S. (ed.) ECOOP 2009 – Object-Oriented Programming. Lecture Notes in Computer Science, vol. 5653, pp. 570–594. Springer, Berlin/Heidelberg (2009)
15. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. **35**, 114–131 (2003)
16. Gasiunas, V., Satabin, L., Mezini, M., Núñez, A., Noyé, J.: Escala: modular event-driven object interactions in scala. In: Proceedings of the Tenth International Conference on Aspect-Oriented Software Development, AOSD '11, pp. 227–240. ACM, New York (2011)
17. Khandekar, R., Hildrum, K., Parekh, S., Rajan, D., Wolf, J., Wu, K.L., Andrade, H., Gedik, B.: Cola: optimizing stream processing applications via graph partitioning. In: Middleware '09, pp. 1–20. Springer, New York (2009)

18. Kuka, C., Nicklas, D.: Quality matters: supporting quality-aware pervasive applications by probabilistic data stream management. In: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, pp. 1–12. ACM, New York (2014)
19. Lakshmanan, G.T., Li, Y., Strom, R.: Placement strategies for internet-scale data stream systems. IEEE Internet Comput. **12**(6), 50–60 (2008)
20. Luckham, D.: The power of events: an introduction to complex event processing in distributed enterprise systems. In: Bassiliades, N., Governatori, G., Paschke, A. (eds.) Rule Representation, Interchange and Reasoning on the Web. Lecture Notes in Computer Science, vol. 5321, pp. 3–3. Springer, Berlin/Heidelberg (2008)
21. Margara, A.: Combining expressiveness and efficiency in a complex event processing middleware. Ph.D. thesis, Politecnico di Milano (2012)
22. Margara, A., Salvaneschi, G.: We have a dream: distributed reactive programming with consistency guarantees. In: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, pp. 142–153. ACM, New York (2014)
23. Margara, A., Cugola, G., Tamburrelli, G.: Learning from the past: automated rule generation for complex event processing. In: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, pp. 47–58. ACM, New York (2014)
24. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: a programming language for ajax applications. In: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09, pp. 1–20. ACM, New York (2009)
25. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: Proceedings of the 22nd International Conference on Data Engineering, ICDE '06, p. 49. IEEE Computer Society, Washington (2006)
26. Rajan, H., Leavens, G.: Ptolemy: a language with quantified, typed events. In: Vitek, J. (ed.) ECOOP 2008 – Object-Oriented Programming. Lecture Notes in Computer Science, vol. 5142, pp. 155–179. Springer, Berlin/Heidelberg (2008)
27. Ré, C., Letchner, J., Balazinksa, M., Suciu, D.: Event queries on correlated probabilistic streams. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, pp. 715–728. ACM, New York (2008)
28. Salvaneschi, G., Hintz, G., Mezini, M.: Rescala: bridging between object-oriented and functional style in reactive applications. In: Proceedings of the 13th International Conference on Aspect-Oriented Software Development, AOSD, vol. 14 (2014)
29. Schultz-Møller, N.P., Migliavacca, M., Pietzuch, P.: Distributed complex event processing with query rewriting. In: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS '09, pp. 4:1–4:12. ACM, New York (2009)
30. Srivastava, U., Widom, J.: Flexible time management in data stream systems. In: Proceedings of the 23rd ACM Symposium on Principles of Database Systems, pp. 263–274. ACM, New York (2004)
31. Wasserkrug, S., Gal, A., Etzion, O., Turchin, Y.: Complex event processing over uncertain data. In: Proceedings of the Second International Conference on Distributed Event-Based Systems, DEBS '08, pp. 253–264. ACM, New York (2008)
32. Wasserkrug, S., Gal, A., Etzion, O., Turchin, Y.: Efficient processing of uncertain events in rule-based systems. IEEE Trans. Knowl. Data Eng. **24**(1), 45–58 (2012)
33. White, W., Riedewald, M., Gehrke, J., Demers, A.: What is "next" in event processing? In: PODS, pp. 263–272. ACM, New York (2007)
34. Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.L., Fleischer, L.: Soda: an optimizing scheduler for large-scale stream-based distributed computer systems. In: Middleware '08, pp. 306–325. Springer, New York (2008)