Acknowledgements

This research has been founded in part by the European Commission within the the project “EMILI — Emergency Management in Large Infrastructures” under grant agreement number 242438 and by the German Research Foundation (Deutsche Forschungsgemeinschaft) within the project “QONCEPT — Query Optimization in Complex Event Processing Technologies” under reference number BR 2355/1-1.

Contents

A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed

Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann

Abstract Complex Event Processing (CEP) denotes algorithmic methods for making sense of events by deriving higher-level knowledge, or complex events, from lower-level events in a timely fashion and permanently. At the core of CEP are queries continuously monitoring the incoming stream of "simple" events and recognizing "complex" events from these simple events. Event queries monitoring incoming streams of simple events serve as specification of situations that manifest themselves as certain combinations of simple events occurring, or not occurring, over time and that cannot be detected solely from one or parts of the single events involved.
Special purpose Event Query Languages (EQLs) have been developed for the expression of the complex events in a convenient, concise, effective and maintainable manner. This chapter identifies five language styles for CEP, namely *composition operators*, *data stream query languages*, *production rules*, *timed state machines*, and *logic languages*, describes their main traits, illustrates them on a sensor network use case and discusses suitable application areas of each language style.

1 Introduction

Event-driven information systems demand a systematic and automatic processing of events. Complex Event Processing (CEP) encompasses methods, techniques, and tools for processing events *while they occur*, i.e., in a continuous and timely fashion. CEP derives valuable higher-level knowledge from lower-level events; this knowl-

Michael Eckert
TIBCO Software, Balanstr. 49, 81669 Munich, Germany,
e-mail: `meckert@tibco.com`

François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann
Institute for Informatics, University of Munich, Oettingenstr. 67, 80538 Munich, Germany,
e-mail: `{bry,brodt,poppe,hausmann}@pms.ifi.lmu.de`

edge takes the form of so called complex events, that is, situations that can only be recognized as combinations of several events.

The term Complex Event Processing was popularized in [45]; however, CEP has many independent roots in different research fields, including discrete event simulation, active databases, network management, and temporal reasoning. Only in recent years, CEP has emerged as a discipline of its own and as an important trend in the industry. The founding of the Event Processing Technical Society [20] in early 2008 underlines this development.

Important application areas of CEP are the following:

Business activity monitoring aims at identifying problems and opportunities in early stages by monitoring business processes and other critical resources. To this end, it summarizes events into so-called key performance indicators such as, e.g., the average run time of a process.

Sensor networks transmit measured data from the physical world to, e.g., Supervisory Control and Data Acquisition systems that are used for monitoring of industrial facilities. To minimize measurement and other errors, data of multiple sensors has to be combined frequently. Further, higher-level situations (e.g., fire) usually have to be derived from raw numerical measurements (e.g., temperature, smoke).

Market data such as stock or commodity prices can also be considered as events. They have to be analyzed in a continuous and timely fashion in order to recognize trends early and to react to them automatically, for example, in algorithmic trading.

The situations (specified as complex events) that need to be detected in these applications and the information associated with these situations are distributed over several events. Thus CEP can only derive such situations from a number of correlated (simple) events. To this end many different languages and formalisms for querying events, the so called Event Query Languages (EQLs), have been developed in the past.

There are also some surveys in the realm of CEP. For example, in [55, 54], rule-based approaches for reactive event processing are classified according to their origins. In [11], EQLs are divided into groups depending on the kind of system architecture they are used in. The survey of EQLs described in [63] distinguishes between a non-logic and logic-based view on handling the event triggered reactivity. There are also comparisons of different single CEP products, e.g., [32]. Both the multitude of EQLs and the diversity of surveys on event processing and reactivity can be attributed in part to the fact that CEP has many different roots and is only now recognized as an independent field.

To the best of our knowledge, there are no comprehensive surveys so far that (1) classify different EQLs into groups according to the language “style” or “flavor” and (2) compare the groups by means of the same example queries with respect to their expressivity, ease of use and readability, formal semantics, success in the industry and some other features. This chapter surveys the state of the art in CEP regarding these two points. Since CEP is a field that is very broad and without clear-cut boundaries, this chapter focuses strongly on querying events. It concentrates on EQLs that are known and specified at the outset. Other, less developed aspects of

CEP such as detecting unknown events using approaches like machine learning and data mining on event streams, are not discussed here.

The contributions of this chapter are:

1. Identification and abstract description of five language styles, namely *composition operators, data stream query languages, production rules, timed state machines,* and *logic languages*
2. Illustration of each language style on a sensor network use case
3. Discussion on suitable application areas of each language style
4. Abstract description of some of the combined approaches

2 Terminology

Since CEP has evolved from many different research areas, a standard terminology has not yet established and found broad adoption. For example, what is called a (complex) event query might also be called a complex event type, an event profile, or an event pattern, depending on the context. We will therefore devote this section to the basic notions and our informal definitions of them.

An **event** is a message indicating that something of interest happens, or is contemplated as happening. Events can be represented in different data formats such as relational tuples, XML documents or objects of an object-oriented programming language (e.g., Java).

In this chapter, we use the following presentation of events: $event\ type\ (attribute\ name_1\ (attribute\ value_1), \ldots, attribute\ name_n\ (attribute\ value_n))$. An **event type** specifies an event structure, similar to a relational database schema specifying the structure of tuples of a relation. For example, *high_temp(area)* is an event type of an event *high_temp(area(a))* indicating high temperature in area *a*. In this event, *area* is an attribute and *a* is its value. (In the following, capital letters denote variables and small letters denote literals.) An **event attribute** is a component of the structure of an event. It can be an entry of a tuple, an XML fragment, or a field of an object, depending on the event representation. The set of attribute values of an event is called **event data**.

The formalism introduced here is by no means compelling. One could prefer to use unnamed perspective identifying attribute values by their positions or use any alternative event representation instead. Since in all languages proposed so far, events are flat or structured records or tuples, the formalism retained for this chapter is no restriction.

Since events happen at particular time which is essential for event processing, all events must have a possibly implicit attribute called event occurrence time. An **event occurrence time** is a time point or time interval indicating when this event happens. A time interval is described by two timestamps indicating its bounds. A time point is described by a single timestamp. We shall see below that using time points or time intervals has far reaching consequences for event processing.

Timing and event order are difficult issues of distributed systems. Each node (computer, device, etc.) in a distributed system has its own local clock and the clocks of different nodes are hard to be synchronized perfectly [17]. Furthermore the transmission time of messages varies depending on sender and receiver, routing, network load, and other factors. Therefore the reception order of some events may differ from their emission order [42]. These issues are ignored in this chapter for the sake of simplicity.

Another characteristical feature of events is event identification. For example, one can assign an identifier *t* to the event *high_temp(area(a))*, written in the following $t : high\_temp\ (area(a))$. We will see the advantages of this feature below.

Events are sent by event producers (e.g., sensors) to event consumers (e.g., Supervisory Control and Data Acquisition system) on so called **event streams**.

In order to react to an event *e* (e.g., turn on air conditioning in an area if an event indicating high temperature in the area arrives) or to derive a new event from another event *e* (e.g., derive an event indicating high temperature from an event containing a temperature measurement if the measurement is considered to be high), an event query which matches *e* is specified in an event query language. An **Event Query Language (EQL)** is a high level programming language (possibly of limited expressivity) for querying events. A **simple event query** is a specification of a certain kind of *single* events by means of an event query language. A **complex event query** is a specification of a certain *combination* of events using multiple simple event queries and conditions describing the correlation of the queried events.

A **simple event** is either an event arriving on the event stream or an event derived by a simple event query (i.e., from a *single* event). A **complex event** is an event derived by a complex event query (i.e., from a certain *combination* of at least two events occurring or not occurring over time). In EPTS Glossary [20] many other kinds of events are defined, such as composite event, virtual event, derived event, raw event and some others.

Note that the occurrence time of a complex event *e* comprises the occurrence time of all events *e* it has been derived from. For example, a complex event *f:fire(area(a))* indicating fire can be derived from two simple events *s:smoke(area(a))* and *t: high_temp(area(a))* indicating smoke and high temperature respectively. *f* begins as soon as *s* or *t* begins and it ends as soon as both simple events are over.

The derivation of complex events is called Complex Event Processing. **Complex Event Processing (CEP)** denotes algorithmic methods for making sense of base events (low-level knowledge) by deriving complex events (high-level knowledge) from them in a timely fashion and over periods of time.

These are the most important notions in the field of event processing. In this chapter we will also need some other notions which will be informally introduced before using.

3 Identification of Language Styles

To bring some order into the multitude of EQLs, we try to group languages with a similar “style” or “flavor” together. We will focus on the general style of the languages and the variations within a style, rather than discussing each language and its constructs separately. It turns out most approaches for querying events fall into one of the following five categories:

1. languages based on composition operators (sometimes also called composite event algebras or event pattern languages),
2. data stream query languages (usually based on SQL),
3. production rules,
4. timed (finite) state machines, and
5. logic languages.

As we will see, the first, the second and the fifth approaches are languages explicitly developed for specifying event queries, while the third one is only a clever way to use the existing technologies of production rules to implement event queries. Similarly, the fourth approach is the use of an established technology to model event queries in a graphical way.

In Sections 4–8, we will describe each language style individually, mentioning the respective important languages from the research and industry. We will also discuss the strengths and weaknesses of each style and illustrate them on a sensor network use case which can be implemented using, e.g., TinyDB [46]. Section 9 summarizes the comparison by a discussion on suitable application areas of each language style. It is further worth mentioning that many industry products follow approaches where several languages of different flavors are supported or a single language combines aspects of several flavors. Section 10 will therefore be devoted to hybrid approaches. Section 11 concludes this chapter.

4 Composition Operators

4.1 General Idea

The first group of languages that we discuss builds complex event queries from simple event queries using composition operators. Historically, these languages have their roots primarily in Active Database Systems [58], though newer systems like Amit [3] run independently from a database. Some examples include: the COMPOSE language of the Ode active database [29, 30, 31], the composite event detection language of the SAMOS active database [27, 28], Snoop [16] and its successor SnoopIB [1, 2], GEM [47], SEL [68], CEDR [6], ruleCore [65, 51], the SASE Event Language [67], the original event specification language of XChange [18, 12, 13], and the unnamed languages proposed in the following papers: [59], [49], [34], [14], [7], [61, 60].

Complex event queries are expressed by composing single events using different composition operators. Typical operators are conjunction of events (all events must happen, possibly at different times), sequence (all events happen in the specified order), and negation within a sequence (an event does not happen in the time between two other events). Consider the use of the operators in the sensor network use case below.

4.2 Sensor Network Use Case

Since different composition-operator-based EQLs have very different and rather unreadable syntax we formulate the example queries in pseudo code in Figure 1. The pseudo code illustrates the idea of this kind of EQLs but it does not mean that each of the queries in Figure 1 can be analogously formulated in every composition-operator-based EQL.

Composition	*fire(area(A))* = $(\ smoke(area(A)) \wedge high\_temp(area(A))\ )_{1\ min}$
Sequence	*fire(area(A))* = $(\ smoke(area(A));\ high\_temp(area(A))\ )_{1\ min}$ or *fire(area(A)) = s:smoke(area(A)); high_temp(area(A)); s+1 min*
Negation	*failure(sensor(S)) = t:temp(sensor(S)); not temp(sensor(S)); t+12 sec*
Aggregation	–

Fig. 1 Example queries in pseudo code for composition operators

The first query in Figure 1 triggers fire alarm for an area when smoke and high temperature are both detected in the area within 1 minute, in other words the query derives a complex event *fire(area(A))* from the two events *smoke(area(A))* and *high_temp(area(A))*. The events *smoke(area(A))* and *high_temp(area(A))* are joined on variable *A*. Their order does not matter but it is important that both events appear within 1 minute indicated by the time window specification $(\ldots)_{1\ min}$. This is a typical example of event composition realized by the conjunction operator $\wedge$ and a time window specification.

The second example is similar to the first one but the events in the event query are connected by the sequence operator *;* denoting that the order of events is important, i.e., the event *smoke(area(A))* must appear before the event *high_temp(area(A))*. Only if the events appear within 1 minute and in the right order the complex event *fire(area(A))* is derived.

Alternatively if a composition-operator-based EQL supports event identification and relative timer events, this query can be formulated by means of the event identifier *s* for the event *smoke(area(A))* and a relative timer event *s+1 min*. In this case an EQL must decide whether the complex event $fire(area(A))$ is derived after the event $high\_temp(area(A))$ or after the event $s+1$ *min*.

Sequence operator is not as intuitive as it seems at first sight. Let A, B and C be simple event queries. Under time point semantics $(A;B);C$ is not equivalent to $A;(B;C)$, i.e., both queries do not yield the same answers. Let b, a, c be events arriving in this order and matching *B*, *A*, and *C*, respectively. They yield an answer for the query $A;(B;C)$ since *b* and *c* satisfy $(B;C)$ with the occurrence time (point) of *c* which is later than that of *a*. *b* happens before *a* which is not allowed by the query $(A;B);C$.

Under time interval semantics $A;(B;C)$ and $(A;B);C$ are equivalent. They both match events a, b, c arriving only in this order. $(B;C)$ matches b, c and has the occurence time interval starting as soon as *b* begins and ending as soon as *c* ends. Furthermore $A;(B;C)$ requires that *a* is over before *b* begins. This query matches a, b, c arriving exclusively in this order. Analogously $(A;B)$ matches a, b and has the time interval described by two time points, namely the begin of *a* and the end of *b*. Consequently $(A;B);C$ requires that *c* begins after *b* is over. This query can also match only the events a, b, c arriving in this order. Hence, using time points or time intervals has far reaching consequences [25].

The third example in Figure 1 shows how negation can be expressed by means of composition operators. The query uses event identification and relative timer events. It demonstrates the necessity for event identification if two events of the same type are used within one query and it has to be distinguished between them.

Assume all sensors of our network send temperature measurements every 12 seconds. The third query detects a failure of a sensor when its measurement is missing, i.e., the query derives a complex event *failure(sensor(S))* when there is an event *temp(sensor(S))* which is not followed by another event *temp(sensor(S))* within 12 seconds.

Another feature which must be supported by an EQL is aggregation. Aggregation means collection of data satisfying certain conditions, analysis of the data and construction of new data containing the result of the analysis. An example of aggregation in our use case is the computation of the average temperature reported by a sensor during the last minute every time a temperature measurement from the sensor arrives. Such a query is unfortunately not expressible by means of composition operators (compare Figure 1).

Nesting of expressions makes it possible to specify more complicated queries but we restrict ourselves to simple examples which should illustrate the main ideas of the language styles without embracing their whole expressivity.

4.3 Summary

Many composition-operator-based EQLs support restrictions on which events should be considered for the composition of a complex event. Event instance selection, for example, allows selection of only the first or last event of a particular type [69, 3, 34]. Event instance consumption prevents the reuse of an event for further complex events if it has already been used in another, earlier complex event [28, 69].

Composition operators offer a compact and intuitive way to specify complex events. Particularly temporal relationships and negation are well-supported. Event instance selection and consumption are features that are not present in the other approaches. Yet, there are hidden problems with the intuitive understanding of operators sometimes, e.g., several variants of the interpretation of a sequence (amongst others, interleaved with other events or not). Further, event data (i.e., access to the attribute values of an event) is often neglected in languages of this style, in particular regarding composition and aggregation.

Currently only very few CEP products are based on composition operators, among them IBM Active Middleware Technology (Amit) [3] and ruleCore [65, 51].

5 Data Stream Query Languages

5.1 General Idea

The second style of languages has been developed in the context of relational data stream management systems. Data stream management systems are targeted at situations where loading data into a traditional database management system would consume too much time. They are particularly targeted at nearly real-time applications where a reaction to the incoming data would already be useless after the time it takes to store it in a database. A typical example of data stream query languages is the Continuous Query Language (CQL) that is used in the STREAM systems [5]. The general ideas behind CQL apply to a number of open-source and commercial languages and systems including Esper [21], the CEP and CQL component of the Oracle Fusion Middleware [52], and Coral8 [50]. See also [38, 44] for the recent research in the field of data stream query languages.

Data stream query languages are based on the database query language SQL and the following general idea: Data streams carry events represented as tuples. Each data stream corresponds to exactly one event type. The streams are converted into relations which essentially contain (parts of) the tuples received so far. On these relations a (almost) regular SQL query is evaluated. The result (another relation) is then converted back into a data stream. Conceptually, this process is done at every point of time. Note that this implies a discrete time axis. (See however [36] for variations.)

For the conversion of streams into relations, stream-to-relation operators like time windows such as “all events of the last hour” or “the last 10 events” are used. For the conversion of the result relation back into a stream there are three options: “Istream” stands for “insert stream” and contains the tuples that have been added to the relation compared to the previous state of the relation, “Dstream” stands for “delete stream” and contains the tuples that have been removed from the relation compared to its previous state, or “Rstream” stands for “relation stream” and contains simply every tuple of the relation. In the following we only use “Istream”.

5.2 Sensor Network Use Case

Figure 2 shows equivalent example queries as Figure 1 but in Continuous Query Language (CQL). A CQL query is very similar to an SQL query. The FROM part of a CQL query is a cross product of relations, the optional WHERE part defines selection conditions, and the SELECT part is a usual projection.

For example, the FROM part of the first query in Figure 2 joins two relations *smoke* and *high_temp* which were generated out of event streams of type *smoke* and *high_temp* respectively by means of time windows. Generally there are several types of time windows. For the sake of brevity only two of them are explained here.

The first one is a simple sliding window. The resulting relation contains all stream tuples of a particular type between *now–d* and *now* where *now* is the current time point and *d* is a duration such as “1 Minute” or “12 Seconds”. The syntax for a sliding window of duration *d* is *T [Range d]* where *T* is an event type and the name of the resulting relation. For example, the notation *smoke [Range 1 Minute]* produces the relation *smoke* containing tuple representations of all events of type *smoke* which happened in the last minute.

The second time window that we explain here is a now window. The resulting relation contains only the stream tuples of a particular type with the occurrence time *now* where *now* denotes the current time point. The syntax for this window is *T [Now]* where *T* is the event type and the name of the resulting relation. For example, the result of the expression *high_temp [Now]* is the relation *high_temp* containing tuple representations of all events of type *high_temp* which happened at the current moment. Note that *T [Range 0 Minutes]* is equivalent to *T [Now]*.

Remember that the first query triggers fire alarm for an area when smoke and high temperature were both detected in the area within one minute. This temporal condition can be intuitively formulated by means of 1 minute-long simple sliding windows restricting the *smoke* and the *high_temp* streams. The join condition is specified in the WHERE block of the query. Consider the first example in Figure 2.

When the order of queried events is important the same query becomes less intuitive. Consider the second example in the figure. The query triggers fire alarm for an area when high temperature is being measured in the area now and smoke has been detected in the same area during the last minute.

Composition

```
SELECT Istream s.area
FROM smoke [Range 1 Minute] s,
     high_temp [Range 1 Minute] t
WHERE s.area = t.area
```

Sequence

```
SELECT Istream s.area
FROM smoke [Range 1 Minute] s,
     high_temp [Now] t
WHERE s.area = t.area
```

Negation

```
SELECT Istream t1.sensor
FROM temp [Now] t1
WHERE NOT EXISTS ( SELECT *
                   FROM temp [Range 12 Seconds] t2
                   WHERE t1.sensor = t2.sensor )
```

Aggregation

```
SELECT Istream t1.sensor, avg(t1.value)
FROM temp [Range 1 Minute] t1,
     temp [Now] t2
WHERE t1.sensor = t2.sensor
```

Fig. 2 Example queries in Continuous Query Language

The definition of correct time windows is essential as it has semantic consequences such as differentiation between an unordered composition and a sequence. Observe that a sequence of more than two events can only be expressed by means of rule chaining. E.g., the sequence of three events e_1, e_2, e_3 can be expressed in the following way: The first query guarantees that e_1 happens before e_2 and generates a complex event e as an intermediate result. The second rule queries events e and e_3 in this order and derives the resulting events.

Negation is hard to express in CQL (as well as in SQL) because the negated tuples have to be queried by an auxiliary query which is nested in the WHERE block of the main query and must be empty to let the main query produce an answer. For example, the third rule in the figure reports a failure of a sensor when it does not send a temperature measurement every 12 seconds.

Aggregation is well supported by the language as shown by the last example in Figure 2. Every time a temperature measurement from a sensor arrives the query computes the average temperature reported by the sensor during the last minute.

5.3 Summary

Data stream query languages are very suitable for aggregation of event data, as particularly necessary for market data, and offer a good integration with databases. Expressing negation and temporal relationships, on the other hand, is often cumbersome. The conversion from streams to relations and back may be considered somewhat unnatural and as may the prerequisite of a discrete time axis.

SQL-based data stream query languages are currently the most successful approach commercially and are supported in several efficient and scalable industry products. The better known ones are Oracle CEP, Coral8, StreamBase, Aleri and the open-source project Esper. However, there are big differences between the various projects and there also exist important extensions that go beyond the general idea that has been discussed here.

6 Production Rules

6.1 General Idea

Production rules are not an event query language as such, however they offer a fairly convenient and very flexible way of implementing event queries. The first successful production rule engine has been OPS [23], in particular in the incarnation OPS5 [22]. Since then, many others have been developed in the research and industry, including systems like Drools (also called JBoss Rules) [37], ILOG JRules [35], and Jess [62]. While the general ideas of production rules will be explained here, we refer the reader to [8] for a deeper introduction.

Production rules, which nowadays are mainly used in business rule management systems like Drools or ILOG JRules, are not EQLs in the narrower sense. The rules are usually tightly coupled with a host programming language (e.g., Java) and specify actions to be executed when certain states are entered [8]. The states are expressed as conditions over objects in the so-called working memory. These objects are also called facts.

Besides their use in business rule management systems that are not focused on events, production rules are also an integral part of the CEP product TIBCO Business Events, which also offers more CEP-specific features such as support for temporal aspects or modelling of event types and data.

The incremental evaluation (e.g., with Rete [24]) of production rules makes them also suitable for CEP. Whenever an event occurs, a corresponding fact must be created. Event queries are then expressed as conditions over these facts. In doing so, the programmer has much freedom but little guideline.

6.2 Sensor Network Use Case

Figure 3 contains our four example queries in the open source production rule system Drools. In Drools all events are represented as Java objects. Every time an event arrives some Java method has to convert it into an object, insert the object into the working memory, and call the rule engine to perform the rule evaluation (more precisely, fire all rules until no rule can fire). Note that in CEP-tailored systems such as TIBCO Business Events this happens automatically. If a complex event is derived by a rule it is also saved as an object in the working memory. We assume that in this case the *insert*-method sets the occurrence time of a complex event.

The occurrence time is a usual attribute of an object. This is actually a problem because every method can change every occurrence time inadvertently. This in turn leads to incorrect answers.

For the sake of simplicity we use time point semantics, assume that timestamps are given in seconds since the epoch (i.e., since the midnight of January 1, 1970) and we do not perform any garbage collection (i.e., deletion of events). These assumptions are not suitable for real-life applications but they help to keep the examples simple. Under the above assumptions we can express the temporal relations between events as simple comparisons of numbers. In real-life applications temporal relations would have to be programmed as Java methods that are called in Drools rules.

A Drools rule consists of two parts. The WHEN part is an event query, it specifies both, the types of queried events and conditions on the events. The THEN part derives an object representing the complex event, sets its occurrence time, and saves the object into the working memory. This newly asserted object can then also activate further rules.

Remember that the first rule detects fire in an area when smoke and high temperature are both detected in this area within one minute (consider the first rule in Figure 3). These conditions are coded into the specification of a *High_temp* object. Its attribute values are compared with the respective attribute values of a *Smoke* object *s*. In particular a *High_temp* event may happen at most one minute before or after a *Smoke* event.

In the second rule of the figure the order of the queried events is relevant. Smoke appears before high temperature is measured in the area. This is expressed by changing one of the conditions on the occurrence time of a *High_temp* object.

Negation is supported in Drools as shown by the third query. Recall that the query reports a failure of a sensor when the sensor does not send a temperature measurement every 12 seconds.

Aggregation of events is also supported. Consider the last rule in Figure 3. Every time a sensor sends a temperature measurement the query computes the average temperature reported by the sensor during the last minute. As this example illustrates aggregation is hard to express in Drools because the result of aggregation must be represented as an object in the WHEN part of a rule (an *Avg()* object in this case) to be used as a parameter of an object representing the complex event in the THEN part of a rule (an *Avg_temp()* object in this case).

Composition

```
when s: Smoke()
     High_temp(area == s.area &&
               timestamp >= (s.timestamp - 60) &&
               timestamp <= (s.timestamp + 60))
then insert(new Fire(s.area));
```

Sequence

```
when s: Smoke()
     High_temp(area == s.area &&
               timestamp > s.timestamp &&
               timestamp <= (s.timestamp + 60))
then insert(new Fire(s.area));
```

Negation

```
when t: Temp()
     not(exists(Temp(sensor == t.sensor &&
                     timestamp >= t.timestamp &&
                     timestamp <= (t.timestamp + 12))))
then insert(new Failure(t.sensor));
```

Aggregation

```
when t: Temp()
     a: Avg() from accumulate(
                      Temp(sensor == t.sensor &&
                           timestamp >= (t.timestamp - 60) &&
                           timestamp <= t.timestamp &&
                           v: value),
                      average(v))
then insert(new Avg_temp(t.sensor, a));
```

Fig. 3 Example queries in Drools

As the examples show all relations between events must be programmed manually and even simple temporal conditions (already in our strongly simplified time model) require low-level code which is hard to read.

6.3 Summary

CEP with production rules is very flexible and well integrated with existing programming languages. However, it entails working on a low abstraction level that is — since it is primarily state and not event oriented — somewhat different from other EQLs. Especially aggregation and negation are therefore hard to express. Garbage collection, i.e., the removal of events from the working memory, has to be programmed manually. (See however [66] for work towards an automatic garbage collection.) Production rules are considered to be less efficient than data stream query languages; this is however tied to the flexibility they add in terms of combining queries (in rule conditions) and reactions (in rule actions).

7 Timed State Machines

7.1 General Idea

State machines are usually used to model the behavior of a stateful system that reacts to events. The system is modelled as a directed graph. The nodes of the graph represent the possible states of the system. Directed edges are labeled with events and temporal conditions on them. The edges specify the transitions between states that occur in reaction to in-coming events.

State machines are founded formally on deterministic or non-deterministic finite automata (DFAs or NFAs). Since states in a state machine are reached by particular sequences of multiple events occurring over time, they implicitly define complex events. Timed Büchi Automata (TBA) [4] were the first attempt to extend automata to temporal aspects for modelling real-time systems. In a TBA each transition between states depends not only on the type of arriving events but also on their occurrence time. For this, temporal conditions are added to transitions. Other examples of this kind of EQLs are UML state diagrams and regular real-time languages [33]. Many representatives of this language style were developed to achieve a particular task or solve a problem of real-time distributed systems, examples are Timed abstract state machine language for real-time system engineering [53], Timed automata approach to real time distributed system verification [41], Timed-constrained automata for reasoning about time in concurrent systems [48].

7.2 Sensor Network Use Case

In this chapter we do not describe different kinds of real-time automata but explain their common principle. Figure 4 contains our example queries in a pseudo code for timed state machines. The pseudo code is an extension of Timed Büchi Automata [4]. The first extension is the consideration of event data. The second extension is the representation of complex events as automata in such a way that only if the end state of an automaton is reached the respective complex event is derived. A complex event can determinate a transition between states of another automaton so that arbitrary levels of abstraction can be achieved.

Remember that our first example derives a complex event *fire(area(A))* out of two events *smoke(area(A))* and *high_temp(area(A))* if these events happen within one minute. Their order does not matter. Since an automaton implicitly describes an ordered sequence we have to specify both acceptable orders of queried events.

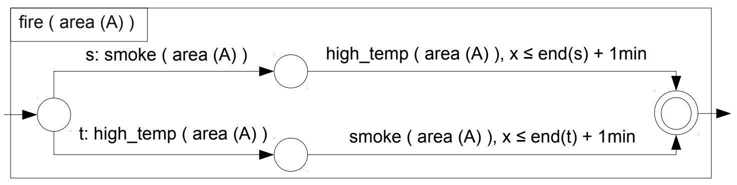
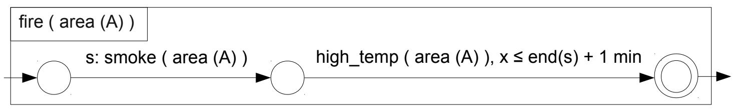
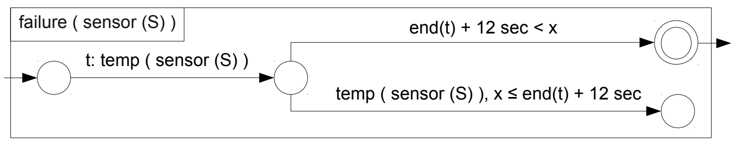
Composition	 fire (area (A)) s: smoke (area (A)) high_temp (area (A)), x ≤ end(s) + 1min t: high_temp (area (A)) smoke (area (A)), x ≤ end(t) + 1min
Sequence	 fire (area (A)) s: smoke (area (A)) high_temp (area (A)), x ≤ end(s) + 1 min
Negation	 failure (sensor (S)) end(t) + 12 sec < x t: temp (sensor (S)) temp (sensor (S)), x ≤ end(t) + 12 sec
Aggregation	–

Fig. 4 Example queries in pseudo code for timed state machines

Consider the first query in Figure 4. The longer the composition of events the more acceptable orders (all possible permutations of events) must be considered by the machine, i.e., a simple composition query provokes a complicated automaton (exponential blow-up).

The events *smoke(area(A))* and *high_temp(area(A))* must happen within one minute. This condition is expressed using event identifiers, an auxiliary function *end(i)* which returns the end timestamp of event *i* and a global clock *x*. (As mentioned above, we do not consider such problems as clock synchronization in this chapter and refer the reader to [43].) Note that both events *smoke(area(A))* and *high_temp(area(A))* are joined upon the value of attribute *area*. If the end state of the state machine is reached the complex event *fire(area(A))* is derived.

The second query describes the sequence of events *smoke(area(A))* and *high_temp (area(A))*. The latter must happen at most one minute after the former to let the automaton reach its end state, i.e., to derive the complex event *fire(area(A))*. This is a very intuitive presentation.

Aggregation is not supported by timed state machines. Negation is not supported also but can be simulated by a failure state without outgoing edges and with an incoming edge which is labeled by a temporal condition and an event which should not arrive for the query to return an answer. For example, the third machine in Figure 4 detects a failure of a sensor when it does not send a temperature measurement every 12 seconds. If a temperature measurement comes within 12 seconds after the last measurement the state machine goes into the failure state, meaning that the end state is unreachable and the complex event *failure(sensor(S))* cannot be derived anymore. If 12 seconds since the last temperature measurement are over (consider the temporal condition of the incomimg edge of the end state) and no new measurement has arrived during this time, the state machine goes into the end state and derives the complex event *failure(sensor(S))*.

7.3 Summary

Though timed state machines provide intuitive visualization of complex events their expressivity is limited. They do not support aggregation. Negation and even composition of events are cumbersome. Conditions on the event data which are more complex than equi-joins (e.g., an attribute value must grow) cannot be expressed.

To overcome deficits of the theoretical automata, state machines are usually combined with languages of other styles. An example of this is the combination of state machines with production rules in TIBCO Business Events. There, a transition between two states is specified with a production rule. The condition of the production rule expresses when the transition is activated. Frequently reactions to the complex events that are implicit in a state machine are desirable. These can be specified for a transition (in the action part of the production rule) as well as for the entry or exit of states.

8 Logic Languages

8.1 General Idea

Logic languages express event queries in logic-style formulas. An early representative of this language style is the event calculus [39]. While event calculus is not an event query language per se, it has been used to model event querying and reasoning tasks in logic programming languages such as Prolog or Prova [40]. The latter combines the benefits of declarative and object-oriented programming by merging the syntaxes of Prolog and Java. Prova is used as a rule-based backbone for distributed Web applications in biomedical data integration. One of the key advantages of Prova is its separation of logic, data access, and computation.

XChange$^{\text{EQ}}$ [9, 19] also adopts some ideas from event calculus-like approaches, but extends and tailors them to the needs of an expressive high-level event query language. XChange$^{\text{EQ}}$ identifies and supports the following four complementary dimensions (or aspects) of event queries: data extraction, event composition, temporal (and other) relationships between events, and event accumulation. Its language design enforces a separation of the four querying dimensions.

A further example of this language style is Reaction RuleML [57, 56] combining derivation rules, reaction rules and other rule types such as integrity constraints into the general framework of logic programming.

8.2 Sensor Network Use Case

Figure 5 contains our four example queries in XChange^{EQ}. An XChange^{EQ} rule consists of two parts. The ON part, i.e., the rule body, is a complex event query which is a conjunction or disjunction of simple or complex event queries and an optional WHERE block containing temporal and other conditions on the queried events. The DETECT part, i.e., the rule head, is a construction of a complex event using the variable bindings returned by the respective event query.

Note that events are neither converted to relational tuples nor to objects of an object-oriented programming language. Furthermore, it is not possible to manipulate event timestamps neither consciously nor unwittingly. Finally, relative timer events are supported by XChange^{EQ}.

Event query specifications are very intuitive and flexible in XChange^{EQ}. There are four types of event queries charaterized by different kinds of brackets. Single brackets denote a complete event query, i.e., the query matches only those events which do not have attributes other than the ones specified in the query. In contrast double brackets denote an incolmplete event query, i.e., events matched by the query may have additional attributes. Curly brackets denote an unordered query, i.e., the order of attributes does not matter. Square brackets denote an ordered event query. Hence, there are four possible combinations of brackets, i.e., four types of event queries (ordered complete, unordered complete and so on).

Consider the first rule in Figure 5. Its complex event query is a conjunction of two simple incomplete and unordered event queries *event s: smoke*{{ *area*{{ *var A* }} }} and *event t: high_temp*{{ *area*{{ *var A* }} }} where variable *A* is bound to the value of attribute *area*. Since the same variable is used in both queries the queried events are joined on the value of this variable.

The WHERE block of the first rule in Figure 5 contains the additional temporal condition that both events, i.e., smoke and high temperature, appear within one minute. Note the use of event identifiers *s* and *t*. Note also that the temporal conditions (like *before* and *within*) are built-in into the language and must not be manually programmed.

The second query contains the additional temporal condition that the smoke event must appear before the high temperature event. The effect that the *additional* temporal condition is mapped to an *additional* statement in the query is an outstanding feature of XChange^{EQ}.

Negation and aggregation of events are supported as shown by the last two examples in Figure 5. Both negation and aggregation are restricted to finite time intervals.

Composition

```
DETECT   fire { area { var A } }
ON and { event s: smoke {{ area {{ var A }} }},
         event t: high_temp {{ area {{ var A }} }}
   } where { {s,t} within 1 min }
END
```

Sequence

```
DETECT   fire { area{ var A } }
ON and { event s: smoke {{ area {{ var A }} }},
         event t: high_temp {{ area {{ var A }} }}
   } where { s before t, {s,t} within 1 min }
END
```

Negation

```
DETECT   failure { sensor { var S } }
ON and { event t: temp {{ sensor {{ var S }} }},
         event i: timer:from-end [ event t, 12 sec ],
         while i: not temp {{ sensor {{ var S }} }} }
END
```

Aggregation

```
DETECT   avg_temp { sensor{ var S }, value { avg(all var T) } }
ON and { event t: temp {{ sensor {{ var S }} }},
         event i: timer:from-start-backward [ event t, 1 min ],
         while i: collect temp {{ sensor {{ var S }},
                                 value {{ var T }} }} }
END
```

Fig. 5 Example queries in XChange^{EQ}

In the examples, the time intervals are given by relative timer events which are defined as follows:

- *timer:from-end[event e, d]* the relative timer *t* extends over the length of duration *d* starting at the end of *e*, i.e., *begin(t):=end(e), end(t):=end(e)+d*
- *timer:from-start-backward[event e, d]* the relative timer *t* extends over the length of duration *d* ending at the start of *e*, i.e., *begin(t):=begin(e)–d, end(t):=begin(e)*

In the above we write *begin(t)* and *end(t)* to denote the beginning and the end of event *t* respectively. There are of course many other relative timer events which are not discussed here, see [19].

Recall that the third example detects a failure of a sensor when it does not send a temperature measurement every 12 seconds, i.e., the query derives a complex event *failure*{ *sensor*{ *var S* } } when there is an event *temp*{{ *sensor*{{ *var S* }} }} which is not followed by another *temp*{{ *sensor*{{ *var S* }} }} event within 12 seconds.

The last query of the figure computes average temperature reported by a sensor during the last minute every time the sensor sends a temperature measurement. More precisely, every time an *event t: temp*{{ *sensor*{{ *var S* }} }} arrives, a relative timer event *i* denoting the time interval of one minute before *t*, is defined, all events happening during *i* and matched by the query *temp*{{ *sensor*{{ *var S* }}, *value*{{ *var T* }} }} are collected and a complex event *avg_temp*{ *sensor*{ *var S* }, *value*{ *avg(all var T)* } } containing the average temperature from the sensor *S*, is derived.

8.3 Summary

As the simple examples above demonstrate, logic languages offer a natural and convenient way to specify event queries. The main advantage of logic languages is their strong formal foundation, an issue which is neglected by many languages of other styles. (Chapter "Two Semantics for CEP, no Double Talk", in this volume describes a general, easily transferable approach for defining both, the declarative and operational semantics of an EQL). Thanks to the separation of different dimensions of event processing, logic languages are highly expressive, extensible and easy to learn and use. Some languages of this style, e.g., XChange^{EQ} supports an automatic garbage collection of events [10].

9 Application Areas of the Language Styles

Having described the strengths and weaknesses of the five language styles, we summarize the comparison by a discussion on suitable application areas of each language style.

Composition operators allow an intuitive specification of event patterns. This makes them attractive in scenarios, where business users should be allowed to define event patters such as real-time promotions and upselling (e.g., send three text messages within one hour to receive a free ringtone).

Data stream query languages are very suitable for aggregation of event data, as particularly necessary for applications involving market data (e.g., average price over 21 day sliding window) such as algorithmic trading. They also usually offer a good integration with databases, sharing in particular the common basis of SQL.

Production rules are very flexible and well integrated with existing programming languages. Since they allow the specification of actions to be executed when certain states are reached, they are particularly useful for applications involving tracking of stateful objects such as track and trace in logistics (maintain and react upon changes of the state of packages, containers, etc.) or monitoring of business processes and objects (also called Business Activity Monitoring). Due to their wide-spread use in business rules management systems, production rules often offer some support for exposing part of the logic to business users such as decision tables or trees.

Timed state machines also offer an easy and convenient way to maintain the current state. However they are limited to a finite set of states (e.g., "shipped", "delivered"). This makes them suitable, e.g., for monitoring of processes (which typically have a well-defined, finite number of states), but not suitable for applications involving infinite state spaces (e.g., a temperature control system where the temperature is a numeric value).

Logic languages have strong formal foundations, allow an intuitive specification of complex temporal conditions and account for event data. They could be successfully used in medical applications or emergency management in critical infrastructures.

Combination of different language styles in one approach allows to benefit from their strengths. This is the main reason why hybrid approaches are most successful in the industry. The next section is devoted to the combined approaches.

10 Combination of Different Language Styles

A comparison of the different language styles shows that so far there is no one-fits-all approach to querying events. Hence particularly industry products trend towards hybrid approaches, where several languages of different styles are supported or aspects of different styles are combined within one language. Hybrid approaches include the introduction of pattern matching into data stream query languages as in Oracle CEP [52], Esper [21], and some CQL dialects like the one used in [64], the use of composition operators on top of data stream queries [26, 15], the addition of composition operators to production rules [66], the combination of production rules and state machines, e.g., in TIBCO Business Events (see Section 7), the decoupled use of different languages (and possibly evaluation engines) that communicate only by means of exchanging events (derived as answers to queries).

11 Conclusion

CEP is an industrial growth market as well as an important research area that is emerging from coalescing branches of other research fields.

Even though the prevalent event query languages can be categorized roughly into five families as done in this article, there are significant differences between the individual languages of a family. Whether a convergence to a single, dominant query language for CEP is possible and advisable is currently in no way agreed upon.

Efforts towards a standard for a SQL-based data stream query language are on the way [36], but not yet within an official standardization body. A standardized XML syntax for production rules is being developed by the W3C as part of the Rule Interchange Format (RIF); however, the special requirements of CEP are not considered there yet. The same applies to the Production Rule Representation (PRR) by the OMG.

Activities of the Event Processing Technical Society (EPTS) [20] aim at a coordination and harmonization with the work on a glossary of CEP notions, the interoperability analysis of Event Processing systems from different vendors, a common reference architecture or framework of architectures, that handles current and envisioned Event Processing architectures, the analysis of the application areas of CEP, and the creation of a business value for a user in order to increase the adoption of Event Processing in the business and industry. The EPTS has also a working group for the analysis of EQLs.

References

1. R. Adaikkalavan and S. Chakravarthy. Formalization and detection of events using interval-based semantics. In *Proc. Int. Conf. on Management of Data (COMAD)*, pages 58–69. Computer Society of India, 2005.
2. R. Adaikkalavan and S. Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. *Data and Knowledge Engineering*, 1(59):139–165, 2006.
3. A. Adi and O. Etzion. Amit — the situation manager. *The VLDB Journal*, 13(2):177–203, 2004.
4. R. Alur and D. Dill. Automata for modeling real-time systems. In *Proc. Int. Colloquium on Automata, Languages and Programming*, volume 443 of *LNCS*, pages 322–335. Springer, 1990.
5. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
6. R. S. Barga and H. Caituiro-Monge. Event correlation and pattern detection in CEDR. In *Proc. Int. Workshop Reactivity on the Web*, volume 4254 of *LNCS*, pages 919–930. Springer, 2006.
7. M. Bernauer, G. Kappel, and G. Kramler. Composite events for XML. In *Proc. Int. Conf. on World Wide Web*, pages 175–183. ACM, 2004.
8. B. Berstel, P. Bonnard, F. Bry, M. Eckert, and P.-L. Pătrânjan. Reactive rules on the Web. In *Reasoning Web, Int. Summer School*, volume 4636 of *LNCS*, pages 183–239. Springer, 2007.
9. F. Bry and M. Eckert. Rule-Based Composite Event Queries: The Language XChange$^{\mathrm{EQ}}$ and its Semantics. In *Proc. Int. Conf. on Web Reasoning and Rule Systems*, volume 4524 of *LNCS*, pages 16–30. Springer, 2007.
10. F. Bry and M. Eckert. On static determination of temporal relevance for incremental evaluation of complex event queries. In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 289–300. ACM, 2008.
11. F. Bry, M. Eckert, O. Etzion, A. Paschke, and J. Riecke. Event processing language tutorial. In *3rd ACM Int. Conf. on Distributed Event-Based Systems*. ACM, 2009.
12. F. Bry, M. Eckert, and P.-L. Pătrânjan. Querying composite events for reactivity on the Web. In *Proc. Int. Workshop on XML Research and Applications*, volume 3842 of *LNCS*, pages 38–47. Springer, 2006.
13. F. Bry, M. Eckert, and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and applications of the language XChange. *J. of Web Engineering*, 5(1):3–24, 2006.
14. J. Carlson and B. Lisper. An event detection algebra for reactive systems. In *Proc. ACM Int. Conf. On Embedded Software*, pages 147–154. ACM, 2004.
15. S. Chakravarthy and R. Adaikkalavan. Events and streams: Harnessing and unleashing their synergy! In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 1–12. ACM, 2008.
16. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proc. Int. Conf. on Very Large Data Bases*, pages 606–617. Morgan Kaufmann, 1994.
17. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, third edition, 2001.
18. M. Eckert. Reactivity on the Web: Event Queries and Composite Event Detection in XChange. Master's thesis (Diplomarbeit), Institute for Informatics, University of Munich, 2005.
19. M. Eckert. *Complex Event Processing with XChange^{EQ}: Language Design, Formal Semantics and Incremental Evaluation for Querying Events*. PhD thesis, Institute for Informatics, University of Munich, 2008.
20. Event Processing Technical Society (EPTS). `http://www.ep-ts.com`.
21. EsperTech Inc. Event stream intelligence: Esper & NEsper. `http://esper.codehaus.org`.
22. C. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, 1981.

23. C. Forgy and J. P. McDermott. OPS, a domain-independent production system language. In *Proc. Int. Joint Conf. on Artificial Intelligence*, pages 933–939. William Kaufmann, 1977.
24. C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.
25. A. Galton and J. C. Augusto. Two approaches to event definition. In *Proc. Int. Conf. on Database and Expert Systems Applications*, volume 2453 of *LNCS*, pages 547–556. Springer, 2002.
26. V. Garg, R. Adaikkalavan, and S. Chakravarthy. Extensions to stream processing architecture for supporting event processing. In *Proc. Int. Conf. on Database and Expert Systems Applications*, volume 4080 of *LNCS*, pages 945–955. Springer, 2006.
27. S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Proc. Int. Workshop on Rules in Database Systems*, pages 23–39. Springer, 1993.
28. S. Gatziu and K. R. Dittrich. Detecting composite events in active database systems using petri nets. In *Proc. Int. Workshop on Research Issues in Data Engineering: Active Database Systems*, pages 2–9. IEEE, 1994.
29. N. H. Gehani, H. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *Proc. Int. ACM Conf. on Management of Data (SIGMOD)*, pages 81–90. ACM, 1992.
30. N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proc. Int. Conf. on Very Large Data Bases*, pages 327–338. Morgan Kaufmann, 1992.
31. N. H. Gehani, H. V. Jagadish, and O. Shmueli. Compose: A system for composite specification and detection. In *Advanced Database Systems*, LNCS, pages 3–15. Springer, 1993.
32. M. Gualtieri and J. R. Rymer. The Forrester Wave^{TM}: Complex Event Procecessing (CEP) Platforms. `http://www.forrester.com/rb/Research/wave%26trade%3B_complex_event_processing_cep_platforms%2C_q3/q/id/48084/t/2`, 2009.
33. T. A. Henzinger, J.-F. Raskin, and P.-Y. Schobbens. The regular real-time languages. In *In Proc. 25th Int. Coll. Automata, Languages, and Programming (ICALP'98*, pages 580–591. Springer, 1998.
34. A. Hinze and A. Voisard. A parameterized algebra for event notification services. In *Proc. Int. Symp. on Temporal Representation and Reasoning*, pages 61–65. IEEE, 2002.
35. ILOG. ILOG JRules. `http://www.ilog.com/products/jrules`.
36. N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. Towards a streaming SQL standard. In *Proc. Int. Conf. on Very Large Data Bases*, volume 1, pages 1379–1390. VLDB Endowment, 2008.
37. JBoss.org. Drools. `http://www.jboss.org/drools`.
38. M. Kersten, E. Liarou, and R. Goncalves. A query language for a data refinery cell. In *Proc. Int. Workshop on Event-Driven Architecture, Processing and Systems*, 2007.
39. R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Computuing*, 4(1):67–95, 1986.
40. A. Kozlenkov, R. Penaloza, V. Nigam, L. Royer, G. Dawelbait, and M. Schroeder. Prova: Rule-based Java scripting for distributed web applications: A case study in bioinformatics. In *Current Trends in Database Technology (EDBT)*, volume 4254 of *LNCS*, pages 899–908. Springer, 2006.
41. J. Krákora, L. Waszniowski, and Z. Hanzálek. Timed automata approach to real time distributed system verification. In *In Proc. of IEEE Int. Workshop on Factory Communication Systems (WFCS)*, pages 407–410, 2004.
42. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
43. Q. Li and D. Rus. Global clock synchronization in sensor networks. *IEEE Transactions on Computers*, 55(2):214–226, 2006.
44. E. Liarou, R. Goncalves, and S. Idreos. Exploiting the power of relational databases for efficient stream processing. In *Int. Conf. on Extending Database Technology (EDBT)*, volume 360, pages 323–334. ACM, 2009.

45. D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.
46. S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.
47. M. Mansouri-Samani and M. Sloman. GEM: A generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.
48. M. Merritt, F. Modugno, and M. R. Tuttle. Time-constrained automata. In *CONCUR '91: 2nd Int. Conf. on Concurrency Theory*, volume 527 of *LNCS*, pages 408–423. Springer, 1991.
49. D. Moreto and M. Endler. Evaluating composite events using shared trees. *IEE Proceedings — Software*, 148(1):1–10, 2001.
50. J. Morrell and S. D. Vidich. Complex Event Processing with Coral8. White Paper. `http://www.coral8.com/system/files/assets/pdf/Complex_Event_Processing_with_Coral8.pdf`, 2007.
51. MS Analog Software. ruleCore(R) Complex Event Processing (CEP) Server. `http://www.rulecore.com`.
52. Oracle Inc. Complex Event Processing in the real world. White Paper. `http://www.oracle.com/technologies/soa/docs/oracle-complex-event-processing.pdf`.
53. M. Ouimet and K. Lundqvist. The timed abstract state machine language: Abstract state machines for real-time system engineering. *Journal of Universal Computer Science*, 14(12):2007–2033, 2008.
54. A. Paschke and H. Boley. Rules capturing events and reactivity. In *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, pages 215–252. IGI Global, 2009.
55. A. Paschke and A. Kozlenkov. Rule-based event processing and reaction rules. In *Rule Interchange and Applications*, volume 5858 of *LNCS*, pages 53–66. Springer, 2009.
56. A. Paschke, A. Kozlenkov, and H. Boley. A homogenous reaction rule language for Complex Event Processing. In *In Proc. 2nd Int. Workshop on Event Drive Architecture and Event Processing Systems*, 2007.
57. A. Paschke, A. Kozlenkov, H. Boley, S. Tabet, M. Kifer, and M. Dean. Reaction RuleML. `http://ibis.in.tum.de/research/ReactionRuleML/`, 2007.
58. N. W. Paton, editor. *Active Rules in Database Systems*. Springer, 1998.
59. C. Roncancio. Toward duration-based, constrained and dynamic event types. In *Proc. Int. Workshop on Active, Real-Time, and Temporal Database Systems*, volume 1553 of *LNCS*, pages 176–193. Springer, 1997.
60. C. Sánchez, S. Sankaranarayanan, H. Sipma, T. Zhang, D. L. Dill, and Z. Manna. Event correlation: Language and semantics. In *Proc. Int. Conf. on Embedded Software*, volume 2855 of *LNCS*, pages 323–339. Springer, 2003.
61. C. Sánchez, M. Slanina, H. B. Sipma, and Z. Manna. Expressive completeness of an event-pattern reactive programming language. In *Int. Conf. on Formal Techniques for Networked and Distributed Systems*, volume 3731 of *LNCS*, pages 529–532. Springer, 2005.
62. Sandia National Laboratories. Jess, the rule engine for the Java(TM) platform. `http://herzberg.ca.sandia.gov/`.
63. K.-U. Schmidt, D. Anicic, and R. Stühmer. Event-driven reactivity: A survey and requirements analysis. In *SBPM2008: 3rd Int. Workshop on Semantic Business Process Management in Conjunction with the 5th European Semantic Web Conf. (ESWC'08)*. CEUR Workshop Proceedings, 2008.
64. B. Seeger. Kontinuierliche kontrolle. *IX: Magazin für Professionelle Informationstechnik*, 2, 2010.
65. M. Seiriö and M. Berndtsson. Design and implementation of an ECA rule markup language. In *Proc. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web*, volume 3791 of *LNCS*, pages 98–112. Springer, 2005.

66. K. Walzer, T. Breddin, and M. Groch. Relative temporal constraints in the Rete algorithm for complex event detection. In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 147–155. ACM, 2008.
67. E. Wu, Y. Diao, and S. Rizvi. High-performance Complex Event Processing over streams. In *Proc. Int. ACM Conf. on Management of Data (SIGMOD)*, pages 407–418. ACM, 2006.
68. D. Zhu and A. S. Sethi. SEL, a new event pattern specification language for event correlation. In *Proc. Int. Conf. on Computer Communications and Networks*, pages 586–589. IEEE, 2001.
69. D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *Proc. Int. Conf. on Data Engineering*, pages 392–399. IEEE, 1999.

Index