LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK
LEHR- UND FORSCHUNGSEINHEIT FÜR
PROGRAMMIER- UND MODELLIERUNGSSPRACHEN

Institut für Informatik

Survey and Comparison of Event Query Languages Using Practical Examples

Hai-Lam Bui

Diplomarbeit

Beginn der Arbeit:	03. November 2008
Abgabe der Arbeit:	26. März 2009
Betreuer:	Prof. Dr. François Bry
	Dr. Michael Eckert

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 26. März 2009 Hai-Lam Bui

Zusammenfassung

Complex Event Processing (CEP) beginnt sich in der Geschäftswelt zu etablieren. Obwohl diese Technologie seit Jahrzehnten eingesetzt wird, wurde sie oft mit Hilfe von Standard-Programmiersprachen implementiert, was viel Programmierung auf einem übermäßig detaillierten Niveau benötigte, oder mit Hilfe spezialisierterer Sprachen, die jedoch außerhalb des für sie bestimmten Problembereichs kaum einsetzbar waren. Mächtigere Programmiersprachen sind erst in den letzten Jahren entstanden. In diesem Überblicksartikel betrachten wir neuere Sprachen für CEP; wir überprüfen die Einfachheit der Handhabung und ihre Möglichkeiten mit Hilfe von Beispielszenarien.

Abstract

Complex Event Processing (CEP) is gaining a foothold in the business domain. While this technology's benefits have been used for decades, it was often implemented using general-purpose languages, causing a lot of low-level coding, or using more specialized languages for event processing, however often having limited use outside of the problem domain they were designed for. More powerful languages have just begun to emerge. In this survey, we take a look at recent languages for CEP, examining their ease of use and capabilities using example scenarios.

Danksagung

Mein Dank gilt meiner Familie, die mich während des Studiums und selbstverständlich auch davor gut unterstützt hat. Ebenso danke ich dem Lehrpersonal beider Fächer, auch für ihre lebendige Wissensvermittlung in den fünf Jahren. Besonderer Dank gilt Frau Brigitte Kommerowski, deren freiwillige Mehrarbeit diese Arbeit möglicherweise erst ermöglicht hat.

Contents

1 Introduction **7**
- 1.1 Definitions . . . 8
- 1.2 Purpose of this Survey . . . 10

2 Example Scenarios and Event Queries **13**
- 2.1 Feature Groups . . . 13
- 2.2 Related Work . . . 14
- 2.3 Common Query Types: Features of Expressivity . . . 15
  - 2.3.1 Filtering by Event Type . . . 15
  - 2.3.2 Windows . . . 15
  - 2.3.3 Data Extraction and Aggregation . . . 16
  - 2.3.4 Conjunctions and Disjunctions . . . 18
  - 2.3.5 Temporal Relationships between Events . . . 18
  - 2.3.6 Causal Relationships . . . 19
  - 2.3.7 Negation and Counting . . . 19
  - 2.3.8 Event Instance Selection and Consumption . . . 20
  - 2.3.9 Integration of Event and Non-Event Data . . . 21
- 2.4 First Example Scenario: Online Store . . . 21
- 2.5 Second Example Scenario: Stock Markets . . . 24

3 Language Selection and Grouping **29**
- 3.1 General Ideas of Data Stream Query Languages . . . 29
- 3.2 General Ideas of Production Rule Languages as Event Query Languages . . . 30
- 3.3 General Ideas of Composition-Operator-Based Languages . . . 31
- 3.4 Excluded Languages . . . 32

4 Language Evaluation **33**
- 4.1 STREAM . . . 33
- 4.2 Borealis . . . 39
- 4.3 AMiT . . . 48
- 4.4 ruleCore . . . 55
- 4.5 SASE+ . . . 67
- 4.6 Esper . . . 71
- 4.7 Cayuga . . . 76
- 4.8 Drools . . . 81
- 4.9 XChange^{EQ} (plus XChange) . . . 89

5 Conclusion **99**
- 5.1 Results Put Together . . . 99
- 5.2 Comparison of Ease of Use and Implementations . . . 101
- 5.3 Beyond this Survey – Other Approaches and Features . . . 103
- 5.4 The Real World – Present and Future . . . 104

Bibliography **106**

1 Introduction

Technological advances in the last decade laid the foundation for systems comprised of communicating components (for example, sensors, computers, software components). While there is a lot of communication happening in the all those systems, some part of it is of special interest: the so-called *events*. We define the term "event" in more detail later; for the moment, it suffices to say that events are some type of message that are sent to notify another component that something has *happened*, such as a failed login attempt on a computer system, stock ticks, or simply a message from a web server that it is running. (The latter two can be considered events, although it might not be obvious what happened: in the stock tick case, it denotes a price fixing for a given kind of stock, in the server case, it denotes the sending of the message.)

While those events may be useful by themselves, it is possible to obtain additional, new information when we look at certain combinations of them. Numerous failed login attempts may be indicative of an intrusion attempt—however, it is not possible to infer this simply by looking at the single events, as each of them might be just a mistyped login code by the personnel. The web server's status messages might not be useful by themselves; however, if it would suddenly stop sending them, it would be wise to take a look at it—particularly if the web server serves an important purpose. Similarly, knowing the price of a stock at some moment is much less interesting compared to knowing that the price has just risen by a large amount in the last few minutes, or has crossed the 90-day average line.

As for all of these examples, the single events are of rather little use to those monitoring these systems, but they can be useful to *detect* a *situation*, which is sometimes hard to detect otherwise. As situations are abstractions of events, they are also called *complex events*; in the above examples, there are complex events whose meanings are a possible intrusion attempt, an offline web server, and a possible opportunity to buy the monitored stock, respectively. The complex events in this section have something in common: It would be advantageous to *react* as fast as possible, in the moment the data (simple events) enters the system. Examples of complex events with such requirements regarding detection time are common. The process of detecting complex events using continuously incoming events on a lower abstraction level is called *Complex Event Processing (CEP)*.

As we have seen, CEP surely has benefits. CEP on event streams is also quite similar to detecting patterns in databases—it is possible to query databases and insert new tuples representing more abstract data. The main differences between CEP on streams and processing databases are:

- In CEP, the queries are executed on the data *every time* the data changes; in databases, this is not necessarily the case. (In implementations, there usually are efficiency improvements that do not change query results.)

- CEP applications try to minimize detection time, that means they avoid to save data on and read data from slow devices, which is very common in standard database applications. This is especially evident when considering that all queries run at every data change, which may happen very often in the real world (10^6 events per second are conceivable [Esp08]).

As more people get aware of CEP's possibilities, a new problem arises: the implementation of CEP features. While it is possible to implement a CEP application in a general-purpose language such as C or Java, it is needed to implement low-level functionality, such as the CEP data structures and algorithms, *in addition* to the queries, causing additional work. Therefore, languages specialized in defining CEP queries have emerged, so-called *Event Query Languages*. They provide an abstraction over the details of organizing the data and evaluating the queries; similar to SQL for databases, it only has to be specified *what* to look for, not *how* to look for something. Several Event Query Languages have been created in the last years, some of them already have production-quality; this survey's focus is to take a look at their capabilities. For a more detailed overview over CEP's history, see Luckham (2007) [Luc07a, Luc07b].

1.1 Definitions

This section is meant to provide some more precise definitions for keywords used throughout this survey and also the languages' documentation documents. As seen in Luckham's History [Luc07a], CEP has several independent roots, resulting in differing terminology in some cases. The Event Processing Glossary [LS08] is meant to unify the vocabulary, which is also the primary source for this section, although there are other glossaries found in research papers and language documentations [AE04, Ae05]. For brevity, some definitions are slightly customized for this survey, as we don't discuss CEP in general, focusing on the event query languages.

Event (I) Happenings in the application domain. Examples include: A withdrawal at an ATM, a request of a web page, a take-off of an airplane.

Timestamp An element of an ordered, infinite set. In our case, this set is either "the set of wall-clock time instants", $\mathbb{N}$, or $\mathbb{R}^+$.

Time interval A pair of timestamps $[a, b]$, such that $a \leq b$. A timestamp c lies within a time interval $[a, b]$ if $a \leq c \leq b$.

Event (II) Also known as *event object*, *event message*, *event tuple*, among others, these are representations of events (as defined above) that can be processed by computers. Usually and especially in this work, we can just call them events and, unless stated otherwise, use them in this second meaning. Events are associated with: a timestamp or a time interval denoting its *occurrence time*, a *type*, and *data*. The data is a set of named and typed *attributes*; the names and types are determined by the event's type. If e is an event and *attrib* is a name of one of e's attributes, then let $e.attrib$ be the *value* of this attribute. Event objects often have similar characteristics to objects in object-oriented languages; they are instances of their event type (or class). For examples, see figures 1 and 2.

Detection time Each event has, in addition to an occurrence time, a detection time. Detection times are related to occurrence times; intuitively, the detection time of a complex event is the time point when the existence of that event can be assured without any uncertainties. For complex events having several members, for example, the detection time is the maximum of the members' detection times, since at any time instant before all members have been detected, it's not known whether the members actually occur.

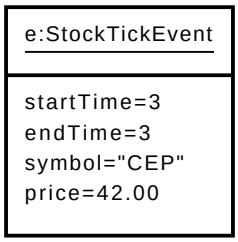

Figure 1: UML representation of an event object. In this example, e is of an event type that has a string attribute named "symbol" and a floating-point number attribute named "price". Further, $e.symbol = CEP$ and $e.price = 42$. It occurred at time 3 (actually, it is the time interval $[3, 3]$).

```
<stockTickEvent id="e">
  <startTime>3</startTime>
  <endTime>3</endTime>
  <symbol>CEP</symbol>
  <price>42.00</price>
</stockTickEvent>
```

Figure 2: XML representation of the event object e (from Figure 1).

Note that detection times are always time points, time intervals whose start and end time is the same.

Event stream A collection of events, not necessarily being of a single event type. *In this work, we require event streams to be ordered by detection time.* Event streams can (in theory) grow infinitely large; there is no size restriction. Event streams start in a component called the *event source*; it can add new events to the stream. Event streams can be read by an arbitrary number of *event processors*, whose only operation available on the stream is retrieving events from the stream; they cannot modify it. Sources and processors can be software components of any kind, including event processing engines. Since this is an abstract concept, implementations of event streams may have further restrictions, for example, events being only readable once, which is often the case if the stream transports events between hardware or software components, as opposed to shared data structures. Example: A SOAP stream used to send XML nodes denoting events from one system to another.

Complex event An event that is an abstraction of other events, called its *members*. In our context, we compute complex events from other events. Examples are sequences of events; more examples are given in Section 2.3, where we list complex events common in real-life applications.

Event query A semantical function that takes input streams as arguments and processes the events in the input streams, that means, possibly creating complex events, sending (received or created) events through output streams, or calling other functions (not necessarily event queries). Examples are a function that prints a message on the screen whenever it finds a new event in its input stream; as well as another function that

```
select * from Withdrawal.win:time(10 sec) where amount >= 200
```

Figure 3: Example of an event query written in Esper, retrieving all withdrawals of 200 units or more in the last 10 seconds.

sends an alert event whenever it receives three failed login attempt events in a short timespan.

Event query language A formal language that can be used to specify event queries. Each query in an event query language has its semantics determined by the language, corresponding to an event query as defined above. Usually, a programmer is given a natural-language description of the abstract event query, and it is the programmer's task to write a query in the event query language that has the desired behavior. An example of a query in the Esper language (a language that is included in the language evaluation in Section 4) is found in Figure 3.

Event processing engine A software component that is the endpoint of input streams to continuously process them, according to rules set by event queries formalized in an event query language. Usually, the rules are applied every time data enters through the input streams or is removed from them. See also Figure 4 for an conceptual example.

1.2 Purpose of this Survey

This survey focuses on the capabilities of the languages in question. We determine which types of complex events can be detected, which types of complex events can not be detected, and whether there are languages which can specify queries easier than others (as well as the reasons). It is *not* meant to be a comprehensive tutorial for any of the languages. We focus on those languages where event processing engines of reasonable quality are (publicly) available.

While the main audience of this survey are researchers who want a state-of-the-art survey of Event Query Languages, it can also be useful for students and researchers as an introduction to Complex Event Processing, due to the emphasis on examples seen through various approaches.

Most languages fall into one of several groups that have been developed independently. Only recently, they have been considered different approaches to the same problem. We believe this survey to be a useful comparison of the languages' ideas, strengths, and weaknesses, allowing language designers to combine the ideas (especially the strengths) to create more powerful languages for their needs. To a lesser extent, it shows how to approach the problem of formalizing event queries.

It should be noted that higher expressiveness of a language often comes with a price: lower performance [cay]. However, we do not deal in any way with performance, although in business use, this is a valid point that needs to be evaluated, as well.

This work's differences from related work are described in Section 2.2.

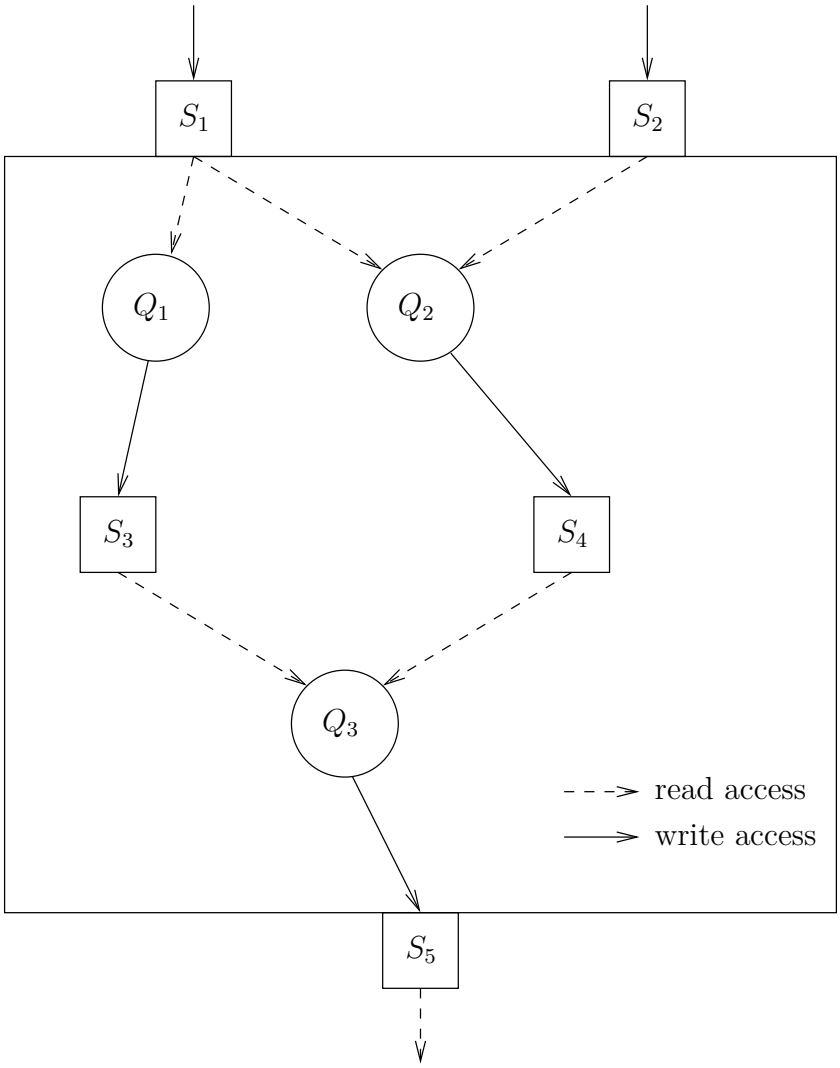

Figure 4: Example of an event processing engine, taking two input streams from outside and having one output stream. Streams are denoted by boxes, queries by circles. In this example, S_3 and S_4 would be implemented as simple data structures, such as sets, while S_1, S_2 and S_5 are inter-component connectors that have to be realized, for example, as sockets or middleware.

2 Example Scenarios and Event Queries

In this section, we develop a number of example scenarios in order to extract example queries. We try to find scenarios and queries that are relevant in the real world, at the same time striving to have a selection of queries containing as many important features as possible. The queries developed in this section are used to measure expressivity and usability of all languages considered in this survey. To this end, we keep the queries as simple as possible, so we can highlight features and shortcomings better. (Recall that queries in this work are abstract things; in order to implement them, they need to be specified in a language. However, queries are actually independent from any Event Query Language.)

To distinguish CEP queries from other queries, we first discuss the concepts specific to CEP. CEP has its roots in many different fields. Its concepts have been developed rather independently in the fields of Discrete Event Simulation, Network Management, Active Databases, and Middleware [Luc07a]. Only recently, sites such as `http://complexevents.com/` began to consider CEP as an independent field, to unify those approaches (especially terminology), and to promote wider use of CEP.

2.1 Feature Groups

This section systematically identifies the features we look for in the languages. After we make clear what CEP does, we see that there are several requirements on Event Query Languages; the most obvious one is expressivity, but there are others, such as integration with other components.

There is an overview over the concepts of CEP [Luc08] that we take as our foundation. It mentions events, patterns of events, complex events, event pattern triggered rules, event pattern constraints, event pattern abstractions, and hierarchies of events, as the most important CEP concepts. We now explain shortly what they are.

Events have been defined in Section 1.1, Definitions. As a basic operation, languages should be able to query streams for events of a certain type (streams are not always typed).

An *event pattern* is a relationship between events, possibly involving data. As an example, let A and B be event types. A has an attribute, y of type integer; B has an attribute p of type integer. Let R and S be streams that contain events of types A and B, respectively. Being able to detect many kinds of patterns of events is a desirable feature. Examples of patterns are:

- Conditions on data: "An event a of type A occurred in R and $a.y > 50$"
- Conjunctions: "An event of type A occurred in R and an event of type B occurred in S"
- Temporal relationship: "An event of type A occurred in R with occurrence time t and an event of type B occurred in S with occurrence time t' and $t \leq t'$"
- Of course, combinations of those are possible. For example: "An event a of type A occurred in R and an event b of type B occurred in S and $a.y = b.p$" (conjunction with data)

Another concept of CEP is, as the name suggests, *complex events*, *abstractions* of a set of events (as defined in Section 1.1). Should a query for a pattern succeed in an instance of that pattern, then it would be useful to have a representation of this instance as an event. For example, one query might process input streams for patterns and output a stream of complex events, which is in turn processed by another query. There is also the opposite of abstraction, *drill down*, which is retrieving the simpler events from a complex event.

Event pattern triggered rules, as defined in the overview [Luc08] consist of event queries and specifications of reactions to every answer to those queries (as defined in Section 1.1). They can react to every answer to their event query in an almost arbitrary way, including outputting complex events, or calling functions in other programming languages.

Event pattern constraints are patterns those occurrence is not desired. Typically, they are used in rules that react if they find an instance of these patterns, for example by issuing alerts.

As we see, there are some groups of features. The most important features of an Event Query Language are the patterns it can detect. Therefore, *expressivity of event queries* is a feature group that consists of:

- Events
- Event relationships (for example, temporal relationships)
- Event abstraction
- Representation of (complex) events
- Event pattern constraints
- Additional concepts: Aggregation of data, counting of events, and integration of event and non-event data (explained in Section 2.3)

The above points mostly deal with theoretical aspects of querying events; still, there is the aspect of *practical usability*. This group includes:

- Availability and maturity of implementations
- Existence of formal semantics
- Event pattern triggered rules
- Ease-of-use

2.2 Related Work

In contrast to existing and surely appearing future reviews of event processing engines, we give implementations less importance compared to expressivity. This is because those reviews usually target business users, who are primarily concerned with other questions, such as benefits *in their case* and cost of integration, measured in money.

While the overview [Luc08] lists CEP's core concepts, it is not detailed enough for our purposes and left out additional concepts that we consider useful, such as aggregation of data (full list in the previous section),

The only other comprehensive survey covering Event Query Languages is found in Eckert [Eck08], that led us to consider those additional concepts as important. It does, however, mention most language features without actually showing their usage, which is what we do in this survey. Also, the idea of grouping the languages into data stream languages, composition-operator-based languages, production rule languages and other approaches is taken from that survey, although we cover some languages it does not cover and vice versa.

Unlike both works, we identify concrete event query features by deriving them from example queries, which happens next.

2.3 Common Query Types: Features of Expressivity

In this section, we clarify how we measure "expressivity". Essentially, we measure expressivity by the number of example queries the language in question can formalize. Since there is an infinite number of event queries, we partition the query space to choose representative queries from each part. In other words, we find common features found in event queries, and test whether the languages support those features.

Languages and CEP engines usually come with some example queries they can express. We collect some of those example queries from different sources and identify common features. In some cases, we create queries on our own; these are not marked with a reference.

Note that the features in the following sections can not be seen as "dimensions"; that means that some of them overlap, have dependencies on other features, or are rarely used by themselves. For example, most queries in Section 2.3.4 can not be broken down into two subqueries, although they use both data extraction and conjunction features.

2.3.1 Filtering by Event Type

Recall that events are typed, while streams may contain events of various, unrelated types. Queries may filter streams by type, that means, output the events of a certain type found in its input stream. Examples include:

- Stock trading:
  - Inputs: Buy and sell orders.
  - Output: All buy orders.
- Sensors:
  - Inputs: Messages from heat sensors and smoke sensors.
  - Output: All heat sensor messages.

2.3.2 Windows

Often, one is not interested in all events that have been added to the stream (possibly since the system has been started). It is often desired to process only a subset of the events. We therefore need a function that retrieves a subset of the stream, containing events with certain timestamps, such as the 10 newest events (by using a *tuple window*), the events inserted

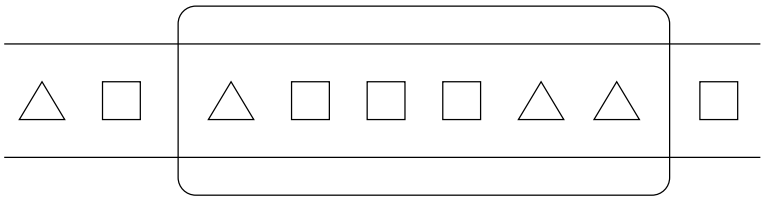

Figure 5: Window (rounded rectangle) applied to a stream. Triangles and squares are events in the stream, ordered by detection time.

during the last 8 seconds (by using a *time window*) or all events between two events (in theory, a window can have arbitrary boundaries, they are not necessarily restricted by events or the current time point). Note that window functions cannot select arbitrary subsets; they can only select all events whose detection times lie between two time points; they cannot exclude events in this time interval, or include events from outside (see Figure 5).

Clocks (such as the wall clock, system clock or the clock in formal semantics) can be seen as event sources continuously sending special clock events. The clock is most often involved in defining window boundaries. If we see time instants as events, then the concept of a window applied to a stream can be seen as a function with two event parameters, retrieving all events in that stream whose detection times lie between the parameters' detection times. As a simple example for a query using a window, we can take the stock trading example from above:

- Inputs: Buy and sell orders.
- Output: All buy orders during the last three hours.

More examples are found in the following sections.

Note that streams created by windows are not necessarily bounded. This may be the case if the start, but not the end event is found in the stream. If the start event is not found, the result is undefined (time and tuple windows start from the beginning, but this can not be applied to the general case).

2.3.3 Data Extraction and Aggregation

Events carry data according to their types. Accessing this data is very common in event queries. There are many uses conceivable, such as comparisons with constants and variables, and typecasting. For comparison purposes, it is usually needed to assign the extracted attribute value to a variable. Typecasting also includes event abstraction and drill down. Examples:

- Banking Monitor [Esp08]:
  - Input: ATM withdrawals.
  - Output: All large ATM withdrawals (for example, with an amount greater than 200 units).
- Online auctions [sqr]:

  - Input: Bids.
  - Output: All bids on a specified set of 5 items.
- Typecasting:
  - Input: TCP/IP requests received by a certain server.
  - Output: All TCP/IP requests received by this server, but only output the requester's IP address (the output stream is a stream of IP addresses instead of TCP/TP requests).
- Examples for abstraction and drill down are found in Section 2.3.4.

Data can also be processed with functions, such as the average, maximum, minimum of an attribute. Usually, this involves applying a window, so that the function can be applied to the subset of the stream one is interested in (as opposed to the whole stream). These functions do not work well with unbounded streams, though; their behavior is undefined in this case. Examples:

- Stock trading:
  - Input: Stock ticks of various stocks.
  - Output: The 200-day average of the "CEP" stock.
- Online shop:
  - Input: All sales (with volume).
  - Output: The sum of sales volumes in the last 24 hours.
- Sensors:
  - Input: Messages (containing the current measured value) of a certain sensor.
  - Output: The average value over the last 5 messages.
- Banking Monitor [IBM]:
  - Input: Withdrawals.
  - Output: An alert if the total value of all withdrawals in the last X days exceeds $\$Y$.

It is also possible to aggregate data over select events, for example, to aggregate data over all events with the same value of an attribute:

- E-mail address abuse detection:
  - Input: Sent E-mails (including sender).
  - Output: The customers that sent more than 1000 E-mails in the last 5 seconds.
- Online shop:
  - Input: Sales (including value and customer).
  - Output: The customers that bought items worth more than X in the last 3 months.
- Online auctions [sqr]:
  - Inputs: Persons and completed auctions.
  - Output: For each seller, maintain the average selling price over the last 10 items sold.

2.3.4 Conjunctions and Disjunctions

This category subsumes conjunctions and disjunctions of (possibly complex) events. A complex event denoting a conjunction represents the happening of all of its members, while a complex event denoting a disjunction represents the happening of one of its members. Conjunction and disjunction may involve data extraction by sharing variables over the member events (as seen in some of the following examples). Examples:

- Online shop [Esp08]:
  - Inputs: Customer orders, deliveries, and payment notifications (each having an ID denoting the order they belong to).
  - Output: Notify a successful trade if an order, a payment and a delivery event with the same order ID occur.
- Credit card fraud detection [IBM]:
  - Input: Purchases using a credit card (with location).
  - Output: A alert notification if two credit card purchases were made within one hour at a distance greater than 200 Km (using the same credit card number).
- Stock trading (abstraction):
  - Inputs: Buy and sell orders.
  - Output: An "Order" event encapsulating the input events.
- Stock trading (drill down):
  - Input: Order events created by the query above.
  - Output: The underlying "Buy" or "Sell" events from all "Order" events.

Note that queries abstracting several events as a complex event give rise to new problems: Occurrence and detection time of complex events, and *representation* of complex events. As for representation, complex events can hold references to its member events. Determining the times is rather easy with logical relationships: One might expect that the occurrence time of a conjunction spans the start time of the earliest event until the end time of the latest event, which is also the time point the event is detected. For disjunctions, the occurrence time should be the time interval of the first event, whose end point is also the detection time. In the next sections, we show that this is not that easy with some other relationships.

2.3.5 Temporal Relationships between Events

Recall that events have an occurrence time, which is generally a time interval (see timepoints as intervals whose start and end times are identical). There are 13 possible temporal relationships between two intervals (as described in Allen's Interval Algebra [All83]): `before`, `meets`, `overlaps`, `starts`, `during`, `finishes`, the inverse relationships of the six previously mentioned relations, and `equals`.

Now recall that we consider all event streams to be ordered (by detection time, see Section 1.1, Definitions). We argue that whether a language can handle unordered streams or not is an implementation detail, not part of expressivity. Further, while there are 13 possible

relations, such a level of detail is rarely needed in applications. Most important is the relation between the end points (which is usually the same as the detection time). For time points, there are only three relations, `before`, `after` and `equals`.

Here are some examples of queries involving temporal relationships between events:

- Stock trading:
  - Inputs: Buy and sell events (with stock and customer).
  - Output: Intraday trades: A buy event followed by the same event of the same stock and customer at the same day.
- Fraud detection (Banking) [Adi06]:
  - Inputs: All online banking events (Transactions, stock trades, failed logins, password changes, etc.).
  - Output: An alert if a password change is closely followed by a large transaction (for the same account).
- Monitoring new users in online auctions [sqr]:
  - Inputs: Registrations and auctions opened.
  - Output: IDs of persons who put up something for sale within 12 hours of registering to use the auction service.

As for occurrence and detection times, we see that binary temporal relations can be seen as conjunctions with additional constraints, therefore, the rules for conjunctions apply.

2.3.6 Causal Relationships

There is only one type of causal relationship. It is described in much more detail in Luckham [Luc01]; however, since no Event Query Language has causality in its core functionality yet, we only give an overview about this concept.

Usually, causality is, unlike temporal relationships, always relative to the event domain. That is, it is hard to infer it by just monitoring event streams. Causality can be encoded in events, though; if A caused B, then B might have a reference to A, enabling reconstruction of causality. A complex event would then be "A caused B". Whether such approaches are always applicable and sound is still subject to research.

2.3.7 Negation and Counting

These concepts revolve about counting events in a stream (usually preprocessed by a window and conditions on data). An event query looking for the absence of certain events produces a complex event if it does not find events of this kind (negation). Counting is a generalization of this concept; using counting, one can query for $\theta\ n$ instances of specific events, where n is a number and $\theta \in \{<, \leq, =, \geq, >\}$ (negation is then "at most 0"). Intuitively, these operators' results are not well-defined if the stream produced by the window is not bounded. Examples of queries involving counting are:

- Failed logins as possible intrusion attempt (from Section 1):

  - Input: Failed logins (at a system, with customer ID).
  - Output: An alert if a customer fails to log in at least three times in the last 30 seconds.
- Sudden absence of messages (from Section 1) [Esp08]:
  - Input: Messages sent by a web server on a regular basis (one per minute)
  - Output: An alert if the web server didn't send a ready message in the last 65 seconds.
- Service level agreements [IBM]:
  - Input: Customer orders (with customer).
  - Output: An alert if an order was sent for processing and no response was received within the time specified by the SLA.
- Online auctions [sqr]:
  - Input: Bids and items.
  - Output: Every minute, the items with the most bids in the last hour.

Detection time becomes an issue with patterns involving counting. Any condition with an upper bound, such as "at most n", has to wait until the end of the window, thus, the event can only be detected at the end of the window; at any time instant before, it is not known whether more events happen until the end. The other query types may detect patterns involving counting once they found enough events to satisfy their lower bound.

For representation, it is clear that events denoting the absence of another event must have a different structure than in the above cases—they can not hold references to absent events, after all. How answers to event queries involving negation (or at-most counting) look like depends strongly on the language; generally, however, it should be useful to know the time window in which the event did not happen.

2.3.8 Event Instance Selection and Consumption

This feature is used to restrict reuse of events for pattern detection. As an example, consider a stream containing login (I) and logout (O) events. We use an event query to abstract login-logout pairs to sessions. Let the first four events be I, O, I, O, then the query detects three sessions instead of the correct answer, two: The first login-logout pair is one session, the second login and the second logout is one session, and the first login together with the second logout is considered a session, with the last one being erroneous.

A language may therefore support Event Instance Consumption, which prevents events to be used again in a pattern once they have been used (either for the query that used them or all queries); as well as Event Instance Selection, which considers only a set number of events of a certain type, for example, the last, or the last n, or the first. In our example, we can fix its problem if we consume any used login or logout event (considering them removed).

As for instance selection, consider a stock market application retrieving rise (R) and fall (F) events for a given stock that is interested in detecting "W" patterns (they are believed to indicate a rise in the near future): $F+$, $R+$, $F+$, R ($E+$ denotes E, one or more times).

Consider the event sequence F, F, R, F, R; without instance selection, this would yield two answers: one matching the whole sequence and one skipping the first F (the more correct match). We can exclude the first match by considering only the last event instance in the $E+$ subpatterns.

2.3.9 Integration of Event and Non-Event Data

As we have seen in Section 2.3.3 about data extraction, it is sometimes needed to compare data in events with data from outside. Above, we could only use constants, or variables that can only depend on event data, for comparison. However, we might also want to include other data in our queries, such as data from a database, which we consider being another language feature. Examples include:

- Service level agreements [IBM]:
  - Input: Customer orders (with customer ID).
  - Output: An alert if three orders from the same platinum customer were rejected for the same reason during a 24-hour period (With platinum status being stored in a customer database.).
- Sensors:
  - Inputs: Heat sensor and smoke sensor messages (with sensor ID).
  - Output: An alert if at least two heat sensors and at least two smoke sensors that are all near each other, send alert. (With locations being stored in a database.)

2.4 First Example Scenario: Online Store

After we mentioned the query features, we now draw up two scenarios where they can be found, leading to queries to be expressed in the Event Query Languages. The first example is an online store that sells various items. We assume that:

- some of the items sold are physical items which need to be delivered by mail (in contrast to, for example, stores only selling software or music by download)
- the store has customers in the whole world and is large enough to get requested at least once per minute (this means there are no times, such as nights, where customer activity is significantly lower than usual)

In addition, our store provides alternative access methods (different from the web interface), easily allowing programs to check prices or order items. While this is possible, we assume that most customers will still use the web interface (that means, the web interface is requested at least once per minute).

Regardless of the access method, a successful trade consists of the following steps: *Login*, *order*, *delivery* and *payment*. Each of these steps creates an event of the corresponding type. It is not possible to order prior to logging in, but it is possible to pay before delivery. When using the web interface, logging in and ordering includes a *page request*. Note that these are not the only page requests. Other possible requests include searching items by name or

retrieving information about items. When not using the web interface, logins and orders are not accompanied by page requests.

In addition, we get notified about *failed login attempts*. In our example, we deal with processing the six emphasized kinds of events; they enter the event processing system from outside. We describe their meanings and data fields our application is interested in in more detail (the events may have more data fields, but they are omitted):

Login A customer has logged on using the web interface or other interfaces. Our application does not use any data from these events.

Order A customer ordered one or more items. Every order has a unique ID, which is carried by events of this type. In addition, these events carry the ID of the customer they were issued from, and the value of the items ordered.

Delivery All items of an order have been successfully delivered (that means, they arrived at the customer). Events of this type carry the ID of their order.

Payment All items of an order have been paid by the customer. Events of this type carry the ID of their order.

Page request A web page of the online store was requested and successfully served.

Failed login attempt A customer tried to log into an existing account, but delivered a wrong password. The ID of the customer the account belongs to is included in this event.

Absence of Customer Activities

Query 1 *Abstracting logins, orders and page requests to customer activities (uses filtering by type (Section 2.3.1) and disjunction (Section 2.3.4)).*

- *Inputs: Logins, orders, deliveries, payments, page requests, and failed login attempts.*
- *Output: A "customer activity" event for every login, order, and page request.*

Query 2 *Detecting absence of customer activities (uses time windows (Section 2.3.2) and negation (Section 2.3.7)).*

- *Input: Customer activities (from the previous query).*
- *Output: An alert (event) whenever no customer activity happened in the last 5 minutes.*

First, we want to detect inaccessibility of the whole system, that means, customers can not request pages, log in, or order. We therefore want to give an alert if there was no such event for 5 minutes. We use two queries to accomplish this. The first query is needed because when not using the web interface, a customer does not request pages when logging in or ordering (while he always does when using the web interface).

Note that when no customer activity happened for 5 minutes, and none happens for some more minutes, this query might generate many alerts, depending on how often the query is executed. To keep the query simple, we will allow this, although in practice, one wants to limit the number of alerts issued by this query.

Trade Event Abstraction

Query 3 *Abstract order, delivery and payment to trades (uses conjunction (Section 2.3.4 and data extraction (Section 2.3.3))).*

- *Inputs: Orders, deliveries and payments.*
- *Output: A "successful trade" event if an order, a delivery and a payment with the same order ID are found.*

Then, we want to abstract orders, deliveries and payments to successful trades. Note that we allow payments before delivery, therefore, delivery may happen before payment, or the other way round, so we use conjunction instead of a sequence. Subsequent queries interested in successful trades only could then work on the trades.

Note: This query is, in this form, not too realistic. Generally, one expects to limit the time that may pass between order and delivery, and order and payment. Such a limit has practical reasons: it increases processing efficiency, as we would not have to look infinitely for events that we no longer accept or expect, such as a payment to a (small) order from 2004. Implementations of this query may impose such a limit if the language does not support unbounded conjunction (conjunction without a window). Because we use seconds as the smallest time unit throughout or queries, we allow a (slightly unrealistic) limit of 60 seconds to be set (as 1 or 2 weeks, expressed in seconds, are quite an unwieldy number).

Accessing Customer Database

Query 4 *Search for orders by platinum customers (uses data extraction (Section 2.3.3) and integration of event and non-event data (Section 2.3.9)).*

- *Input: Orders.*
- *Output: Orders whose customer ID is of a platinum customer.*

Further similar queries involving conjunctions, time windows, and negation are conceivable. For example, one could query for orders without delivery for two days, if the customer is a platinum customer (for other customers, we allow longer delays). Platinum status is not included in the events, but found in the customer database. This query is used to determine which orders are from platinum customers.

Aggregating over Customers

Query 5 *Detect customers placing big orders (uses tuple windows (Section 2.3.2) and aggregation by group (Section 2.3.3)).*

- *Input: Orders.*
- *Output: A notification if a customer bought items worth more than 3000$ in his last 5 orders.*

We also want to detect "good" customer behavior, that is, if a customer tends to place big orders, we can consider to give him platinum status. We assume we have a stream containing the orders including their value.

Detecting Intrusion Attempts

Query 6 *Search for multiple failed login attempts (uses time windows (Section 2.3.2), counting (Section 2.3.7), and aggregation by group (Section 2.3.3)).*

- *Input: Failed login attempts.*
- *Output: An "intrusion attempt" event if there are at least three failed login attempts for an account in the last 5 minutes.*

For demonstrating counting, we use the failed login example. Note that although this is usually done by other application logic, we can imagine the use of a similar query to count intrusion attempts. This query might give more answers than necessary (for example, when four or more attempts occur), but for simplicity, we do not restrict reuse of event instances here.

More complex queries combining the above query features, such as the absence of delivery mentioned above, are possible, but can be expressed if the above queries can be expressed. So, for measuring expressivity, we do not need them.

2.5 Second Example Scenario: Stock Markets

In our second example scenario, we use an event processing system for technical analysis of stocks. Technical analysis is used to detect indicators in a stock's chart to predict the stock's future development, using only the chart, without using data (such as sales in the last quarter-year) about the company. We will use event queries to detect two types of patterns that are believed to indicate a rise in the near future: Outperformance and W pattern. We give a short explanation of stock market concepts used in this scenario. Further information is found in literature about technical stock analysis.

Index An index in the stock market measures the performance of a group of stocks. For example, an index over the 30 largest companies in the market performes well (rises), if these 30 companies' stocks perform well (and vice versa). Usually, not every stock has the same weight in an index. Stocks can be used in multiple indices. Examples of real-world indices are the Dow Jones Industrial Average (often called "Dow Jones" for short), the German DAX, or the Japanese Nikkei.

Performance The performance of a stock or index in a time span is the relative change in price at the end of the time span, compared with the price at the beginning. For example, if a stock had a price of 50 dollars, and has a price of 55 dollars one week later, it had a performance of +10% in that week.

Outperformance A stock that is listed in an index outperforms this index in a time span, if the stock had a better performance in that time span than the index (see Figure 6).

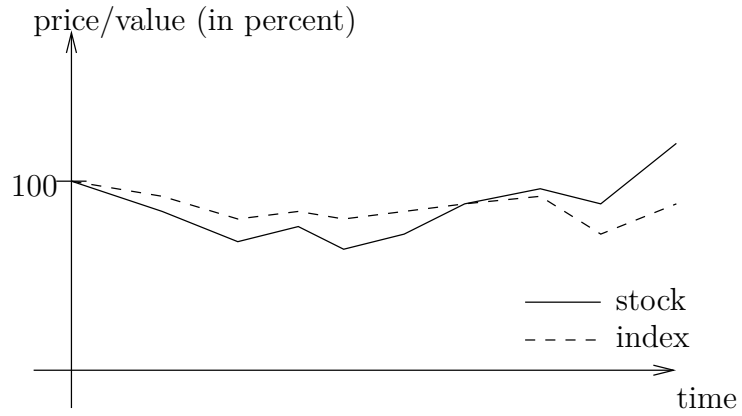

Figure 6: A stock outperforming its index.

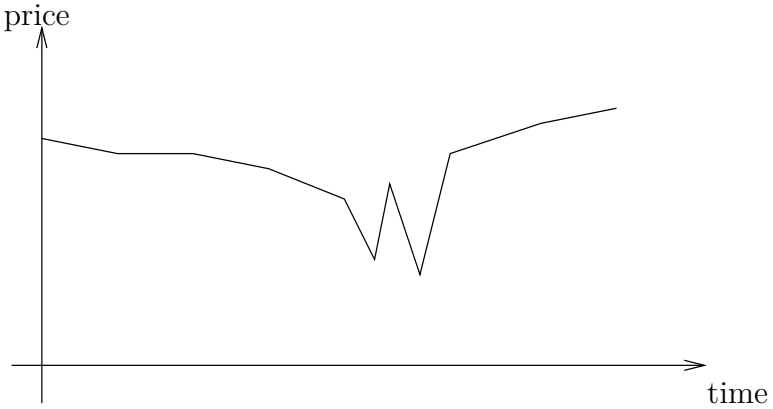

Figure 7: A W pattern in a stock's chart as the starting point for a rise.

V pattern A sequence of fall and rise in a stock's or index's chart.

W pattern Two V patterns, whose lower points are at approximately the same height (see Figure 7), occurring in a small time span. In our example, there may be other V patterns between the two V patterns.

In this scenario, we have two input streams; one contains only stock tick events for a certain stock (called `STK`), denoting a new price for this stock; therefore, each event has one data field, the new price. Note that the price does not necessarily have to be different from the price in the tick before.

The other input stream contains ticks for the index `STK` is in (called `IDX`). We also assume, departing from reality, that the stock market does not close under any circumstances, as well as active trading all the time, so that for both `STK` and `IDX`, there is at least one tick every 10 seconds, although there can be more.

Although analyzing just one stock might look artificial, analyzing a group of stocks at the same time does not add much to our measuring of expressivity, but greatly increases complexity, therefore, we left that out.

Detecting Outperformers

Query 7 *Compute the average index value over the last 30 seconds (uses time windows (Section 2.3.2) and data aggregation (Section 2.3.3)).*

- *Input:* `IDX` *values.*
- *Output: The average value of* `IDX` *in the last 30 seconds.*

On stock markets, the market participants' behavior may induce "spikes" in a stock's or even index's chart that is only the result of speculation, but is unrelated to any (major) news. In order to reduce the effect of such speculations on a chart, it is smoothed; that is, a new stream is produced, at each time instant carrying the average value over some fixed time interval. In our case, we compute the average value of the index in the last 30 seconds.

It is not exactly specified how often this query has to be executed. However, since the index stream contains new events on a steady basis, it should be at least every time a new index tick enters the stream.

Query 8 *Detect outperformers (uses time windows (Section 2.3.2), data extraction (Section 2.3.3), and event instance selection (Section 2.3.8)).*

- *Inputs:* `STK` *ticks, and smoothed index (from the previous query).*
- *Output: An "Outperform" event if* `STK` *outperformed the smoothed index in the last hour.*

We are now interested to be notified when `STK` outperformed its index in the last hour by more than two percent (Note that values such as these are not backed by the author's experience, and may differ in real applications.). This query will have to make use of event instance selection: We select the first and the last tick of both the index and the stock in the time window. Let the first tick of the stock and index be s and i, respectively, and the last tick be s' and i'. Then the stock outperformed the index if

$$\frac{s'}{s} - \frac{i'}{i} \geq 0.02$$

Similar to the absence of customer activities example, this query may report an outperformer multiple times, as it is likely to still outperform the index when the time window is moved a bit, but we do not restrict this.

Detecting W Patterns

Query 9 *Detect V patterns (uses data extraction (Section 2.3.3) and sequences (Section 2.3.5)).*

- *Input: Rises and falls (for* `STK`*). Both types carry the former and the new price in variables called* `lower` *and* `higher`*.*
- *Output: A "V" event for every fall, directly followed by a rise. Include the minimum value in the V event.*

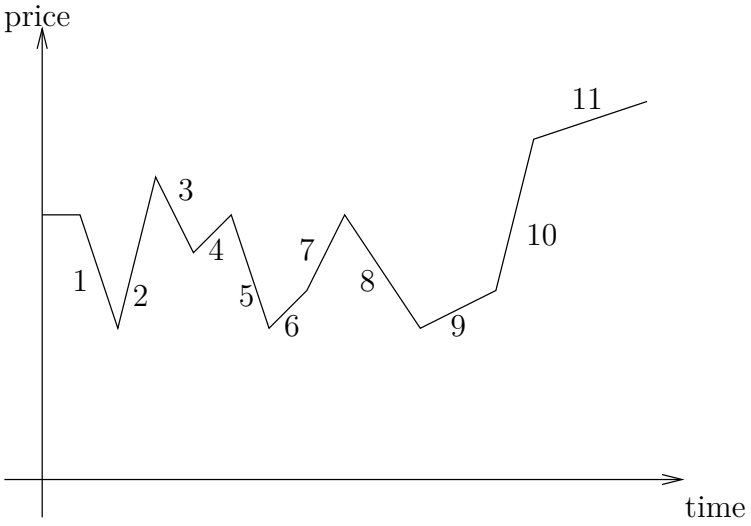

Figure 8: Chart of a stock. Rises and falls have been numbered. There are four V patterns in this chart, $(1, 2)$, $(3, 4)$, $(5, 6)$, and $(8, 9)$. $(3, 4)$ is the only V pattern with a different minimum. $(1, 2)$ forms a W pattern with $(5, 6)$, as well as $(5, 6)$ with $(8, 9)$. We use event instance consumption, so our query does not detect $(1, 2)$ with $(8, 9)$ as a W pattern, as it is not necessary.

To prepare W pattern detection, we first abstract falls followed by rises to V patterns (Assume that we already abstracted ticks to sequences of rises and falls; rises and falls carry their maximum und minimum value and are non-overlapping).

Query 10 *Detect W patterns using V patterns (uses data extraction (Section 2.3.3), sequences (Section 2.3.5), time windows (Section 2.3.2), and event instance consumption (Section 2.3.8)).*

- *Input: V patterns for* `STK` *(from the previous query).*
- *Output: W patterns for* `STK`*, that is, a V pattern followed by another V pattern (other V patterns in between are allowed). Additional constraints: the second V's minimum is* $100 \pm 1\%$ *of the first V's minimum; both V events must have occurred in the last hour; the first V may not be used in other answers to this query.*

As we mentioned, a W pattern is a sequence of two V patterns whose minima are approximately the same. As we allow other V patterns to occur inside the W, we can not just look for adjacent V patterns and instead have to use event instance consumption to avoid duplicate answers (see Figure 8). To define "approximately", we require the minimum of the second V to be $100 \pm 1\%$ of the first V's minimum.

As our queries used most of the features mentioned in Section 2.3, we can now go over to the languages.

3 Language Selection and Grouping

In this chapter, we present the groups the considered languages can be classified into, along with their roots and general ideas. We also give reasons why we exclude certain languages.

Many covered languages are related. For example, the STREAM language has strongly influenced commercial languages in use, like Esper. Related languages share many concepts; to avoid explaining them for every language, we give an abstract overview over those concepts in this section, and show how they can be used in the languages, as well as additional features, when we discuss each language in detail.

Generally, most languages have characteristics of one or more[1] of three groups: *data stream query languages*, *composition-operator-based languages*, and *production rule languages*. In this survey, the XChange^{EQ} language is the only exception; it uses a fourth approach to Complex Event Processing, based on logical formulas. As the only language in its group, the ideas of XChange^{EQ} are discussed in its evaluation (Section 4.9), but not in this section.

As we will see in Section 5.1, languages belonging to the same group will have similar expressivity for each query feature. For example, data stream languages are well suited for aggregating data, but are hard to use for finding event patterns.

3.1 General Ideas of Data Stream Query Languages

As the name implies, data stream query languages are about querying data streams; however, the data are usually events. Their main idea is to use a relational query language for querying streams. In most cases this is SQL, the industry standard language for querying databases. However, relational query languages are not capable of querying streams by themselves; they can only query relations. Streams change over time, but for every time instant τ, there is a set of events in the stream; this set can be converted into a relation. Data stream query languages therefore use, at every time instant, the following pattern:

1. Convert snapshots of the input streams to relations.
2. Apply the query formalized in the relational query language.
3. Convert the result (a relation) to a stream.

Therefore, data stream query languages have three types of operators: *stream-to-relation operators*, *relation-to-relation operators*, and *relation-to-stream operators*. (As a sidenote, implementations of the languages may apply techniques to avoid the stream-to-relation conversion, operating directly on the streams where possible, for efficiency reasons.)

Remember that streams are possibly continuously growing sets of events. For every time instant, there is a set of events present in the stream. Therefore, one can conceive an operator that takes a stream S and a time τ and returns all events that were present in S at that time. The resulting set of event objects can then be represented as a relation $R(\tau)$, thus converting a stream into a relation.

[1] Some languages have the characteristics of two groups, such as Esper and Cayuga.

Instead of retrieving all events from a stream at a given time, it is also possible to retrieve only select events by applying a window (as described in Section 2.3.2). Basic windows are tuple windows, retrieving only the last n events, and time windows, retrieving only the events that entered the stream in the last n time units. Other window concepts also exist, such as predicate windows [GAE06] that select all events that satisfy a condition.

Once relations have been obtained, they can be queried using relational query languages, such as SQL. The relations created by converting event streams (possibly using windows) can be queried like relations in databases, allowing operations such as selections, joins and aggregation. These operations are called relation-to-relation operations.

The relation created by the query in the relational query language is then converted to a stream; this can also happen in various ways, depending on the desired timestamping. Let $R'(\tau)$ be the relation created by the relation-to-relation operator from $R(\tau)$. The first possibility to convert $R'(\tau)$ to a stream is inserting all events in $R'(\tau)$ into the stream with timestamp τ, which is called *relation stream* of $R'(\tau)$.

In cases where $R'(\tau)$ is similar to its predecessor $R'(\tau-1)$, which is common, this will create undesired events in the output stream. These events differ from other events only in their timestamp, but not their data. For example, if we query the stream of all stock ticks for the ticks of a particular stock (without applying windows), every time the query is executed, all ticks of that stock are inserted into the stream again; however, in applications, we want each tick to appear only once in the output stream. In such cases, the *insert stream* is the correct set of events to insert into the output stream, that is, all events in $R'(\tau)$ not present in $R'(\tau-1)$. In some cases, we then need a similar operator, the *delete stream*, that returns all events in $R'(\tau-1)$ that are not present in $R'(\tau)$.

Note that accessing the predecessor $\tau-1$ of τ is only well-defined if the time domain is discrete, that means, the insert and delete stream operators can only be used if the time domain can be discretized. Usually, however, this is the case, as the time domain is a conceptual domain created by the languages and engines themselves. The exact nature of the time domain depends on the language [JMS+08]. Languages such as CQL increment the time whenever they evaluated all tuples of the newest timestamp, while languages such as Borealis increment the time for every tuple, even if there are more tuples with that timestamp. These different semantics sometimes cause different results for the same streams. The SPREAD operator [JMS+08] was designed as a more flexible operator that can be used to express both semantics, and also others.

3.2 General Ideas of Production Rule Languages as Event Query Languages

While production rule languages were not designed with CEP in mind, they have properties that reasonably make them usable as Event Query Languages. Production rule languages do not operate on streams, but on data structures called *working memories*: mutable sets of objects capable of carrying data, called *facts*. This definition sounds similar to that of streams; unlike streams, facts are not required to be timestamped; they are just arbitrary objects.

Production rules operate on facts in a working memory. Production rules consist of a

condition and an action; if a rule's condition becomes true, the action can be executed by *firing* the rule. Conditions are used to check the existing working memory, while actions can change the working memory by adding, removing, or altering facts. In production rule languages, adding facts is typically called *asserting*, while removing facts is called *retracting*.

A production rule language is used to express production rules that are then deployed in a production rule engine. The engine is then supplied with objects that are stored in the engine's working memory. While asserting facts, the condition for each rule may become true, in which case they activate and can be fired. The resulting changes in the working memory may then activate other rules. The evaluation algorithm for production rules is often a variant of the rete algorithm [For82] that increases condition-checking efficiency, for example, by avoiding unnecessary checks. Note that rules activate only if their condition *becomes* true, not every time it *is* true.

To use production rule languages for CEP, working memories must somehow be usable like streams. So, whenever an event would enter a stream, it is asserted into the working memory as a fact. These facts have to be timestamped, usually with an additional attribute. Therefore, time has no special role in these languages. Unfortunately, this is this language group's biggest disadvantage, as temporal relationships or time windows have to be coded into the rules, if possible; else, one would even need to use code in a general-purpose programming language. In other words, using time requires low-level coding, although Event Query Languages should actually avoid that. But still, the similarity of working memories and streams eases certain kinds of queries, especially when using non-event data sources. From a production rule language's point of view, non-event data sources differ only slightly from streams (only lacking the timestamp), and are accessed in the same way.

3.3 General Ideas of Composition-Operator-Based Languages

The general idea behind composition-operator-based languages is the composition of complex event queries using simpler event queries. The most basic event queries just query for events of a certain type; it is then possible, for example, to query for a conjunction of an event of type A and an event of type B. When this query is executed, it yields an answer if an event of type A and an event of type B is found. More generally, it yields an answer once both subqueries yield answers.

An important aspect of these languages are *query instances*. For example, if there is a sequence of A, B, B for the example above, there are actually two answers to the query, since there also is the conjunction of the A and the second B. To accomplish this, we need two instances of the query; both match the A, but only one matches the first B, while the other matches the second B.

The expressivity of these languages is determined by their supported operators, such as conjunction, disjunction, sequences, negation, counting, and applicability of constraints. Examples of constraints are:

- Relative temporal constraints: B happens within one hour of A
- Absolute temporal constraints: A and B happen at the same day
- Conditions on data: A and B agree on their ID attribute

It should be noted that the semantics of certain operators, especially sequence, may vary between languages. For example, some sequence operators allow events to happen between the operands (for example, in Esper), some do not; some sequence operators allow the second subquery's execution to start before the first finished, some do not. Further, some languages, such as Esper, require the occurrence time of events to be time points, while other languages also support time intervals. To this end, a language may have multiple sequence operators, differing in such details (for example, in AMiT).

3.4 Excluded Languages

While there are many engines and languages available, we excluded the following languages:

Commercial languages without publicly available documentation — such as from Aleri, INETCO, Oracle, Progress Apama, RTM, SENACTIVE, Syndera, TIBCO, and WestGlobal — are not covered at all. Documentation is even considered non-available if it only requires some kind of registration.

Data stream languages similar to covered languages, especially Esper, are excluded. Avaya Event Processor [ava], BEA (Oracle) Complex Event Processing [bea], Coral8 [cor], and StreamBase [strb] are commercial CEP engines of this kind; they are all are data stream languages extended with patterns, belonging to Esper's group of hybrid languages. The Intelligent Event Processor Module in OpenESB [ope] uses an Event Query Language that has a smaller operator set than Esper. TelegraphCQ [tel] is a research prototype built upon PostgreSQL, but has no new language features for CEP (in fact, the project's results influenced other languages).

Production rule languages can be covered by Drools. It is free software and contains most features found in other production rule languages.

4 Language Evaluation

This is this work's core; we take a close look at the languages in this chapter. We express the example queries, if possible, along with a description of design principles and availability of implementations. Also, we describe our personal experience with learning and using the nine languages we evaluated.

We express all queries in one language before continuing with the next. This avoids repeating a language's concepts at many queries. We begin with data stream languages, as we found them easy to understand, following with composition-operator-based languages and hybrid languages having features of both groups. We conclude this section with the production rule language Drools and the XChange^{EQ} language that fits in neither group.

4.1 STREAM

STREAM (STanford stREam datA Manager) was a research project at Stanford University. Its focus was to develop methods to manage and query data in data streams. In this process, a CEP engine, also called STREAM, was created.

STREAM proposed an Event Query Language, called Continuous Query Language (CQL), to query event streams. CQL has most features of data stream languages: stream-to-relation operators, relation-to-stream operators, and a subset of SQL as relation-to-relation operators. It plays an important role as the foundation for other data stream languages, such as Esper's EPL or Coral8's CCL, thus being a good language to begin the evaluation with.

The syntax of CQL resembles SQL very strongly. Queries usually have the form

```
Output(var1,...,varn):

  Select ...
  From ...
  Where ...
  Group By ...
  Having ...
```

where `Output` as well as the `var` elements are placeholders for names. Only the `Select` and `From` clauses are mandatory. The `From` clause specifies the streams and relations to be queried. Streams must have a window applied. The `Select` clause specifies the values to be returned by this query. These are expressions possibly depending on the input streams. The `Where` clause can be used to apply conditions on the events (or joined events if there is more than one input stream).

CQL provides several stream-to-relation and relation-to-stream operators. For converting streams to relations, it provides time windows and tuple windows. Time windows take the form `[Range N TimeUnits]`, where `N` is a number and `TimeUnits` is a time unit such as `Minutes`. Tuple windows take the form `[Rows N]`. There are also two special windows: `[Unbounded]`, the default window when no one is specified, takes the whole stream up to now into consideration, while `[Now]` only considers tuples whose timestamp is the same as the current time. Remember that CQL is a data stream language, converting streams

to relations at every time instant in a discrete time domain. As for relation-to-stream operators, `Istream`, `Rstream` and `Dstream` are available, denoting insert streams, relation streams and delete streams, respectively. The `Select` clause always contains a relation-to-stream operator; if it is not specified, it defaults to `Istream`.

The first line names the output stream, allowing it to be referred to by other queries.[2] The variable list in this line names all attribute names (also to be used by other queries). The variable list must have exactly the same number of names as the number of the values the `Select` clause yields.

The `Group By` clause, containing one or more variables from the input streams, is typically used with aggregation functions; aggregation functions are applied on groups of events, one for each distinct value of these variables, yielding one answer to the query for each group. For example, if our input stream contains the orders of customers carrying the volume of the order and the ID of the customer, we can sum up the total volume of all orders by the same customer, and do this for every customer. The `Having` clause can be used to apply conditions on these groups; as in SQL, this cannot be done in the `Where` clause.

As mentioned in Section 3.1, CQL executes queries only once for each timestamp in a stream. That means for example if two tuples with the same timestamp arrive in a stream, the queries involving this stream are only executed once, not twice.

The STREAM engine is more like a research prototype, and is not primarily designed to be used in complex systems such as business systems. While it has a manual, CQL is less well documented as languages designed for the real world, for example, Esper; we base our evaluation on papers describing the language design [ABW03, ABW06], as well as the manual [stra]. We also use features present in CQL, but not implemented in STREAM; whenever this happens, we note it.

Query 1 (Disjunction)

```
CustomerActivity(custId):

  (Select Istream(custId) From Login[Unbounded])
  Union
  (Select Istream(custId) From Order[Unbounded])
  Union
  (Select Istream(custId) From PageRequest[Unbounded])
```

The `Union` operator allows to combine multiple queries, as long as their output has the same schema. So we extract the same field from all three event types. We use the default operators, `Istream` and `Unbounded`; they are the default operators and left out in subsequent queries. The results are assigned to the `CustomerActivity` stream whose schema has one field. This output declaration is omitted in subsequent queries if we do not use the output stream ourselves.

The query can be rewritten as follows, yielding the same results but with possibly better performance:

[2] Output streams can be views or real output streams. Views are not sent outside the engine. STREAM does not handle possible cycles, such as using the same stream as both input and output stream.

```
CustomerActivity(custId):

  (Select Rstream(custId) From Login[Now])
  Union
  (Select Rstream(custId) From Order[Now])
  Union
  (Select Rstream(custId) From PageRequest[Now])
```

Query 2 (Negation, time windows)

This query is not fully expressible using language constructs, but can be approximated using low-level code.

```
Select true
From Timer
Where Not Exists
  (Select * From CustomerActivity[Range 5 Minutes])
```

This query uses a subquery, a feature from SQL also present in CQL. For every event in the input stream, when evaluating the `Where` clause, the subquery is executed.

In this example, it is tested whether the `CustomerActivity` stream contains events not older than 5 minutes. If this it not the case, this query yields an answer. Since the `Select` clause is mandatory, but the structure of the answers is not important, it was filled with the dummy value `true`.

In order for this to work, we must use an input stream that is guaranteed to contain events on a regular basis. This is because the subquery is only evaluated every time an event enters the input stream. We therefore use a new stream[3], `Timer`, that is filled with events on a regular basis by an external program, for example, every second. However, note that this external program has to be written by the user, cannot be optimized by the CEP engine, and may incur multithreading issues such as a dependency on a good scheduling.

Note that subqueries in the `Where` clause are proposed in CQL, but not supported by STREAM.

Query 3 (Conjunction, data extraction)

```
Select *
From Order As o, Payment As p, Delivery As d
Where o.orderId = p.orderId
  And o.orderId = d.orderId
```

This looks exactly like a standard SQL join, as no stream-specific features are used explicitly. This is because the standard operators are omitted.

This implementation of this query will likely exhaust memory over time, as events never become irrelevant (each stream has the implicit `Unbounded` window applied). In the previous

[3] Actually, it is mentioned in the revised paper [ABW06] that such streams sending so-called heartbeats are part of STREAM, however, the manual says that these heartbeats are not supported yet and does not describe their usage.

queries, events in the input streams could be discarded after processing, while in this query, they are hold in memory to be joined with others. We can prevent this by imposing the 60-seconds limit we allowed in the query specification, by replacing the `From` clause:

```
Select *
From Order[Range 60 Seconds] As o,
  Payment[Range 60 Seconds] As p,
  Delivery[Range 60 Seconds] As d
Where o.orderId = p.orderId
  And o.orderId = d.orderId
```

Query 4 (Using external data sources)

```
Select custId
From Customer
Where platinum_status = true

Select *
From Order[Unbounded] As o, PlatinumCustomerIDs As c
Where o.custId = c.custId
```

Input can be streams or relations. In this query, `Order` is a stream, while `Customer` and `PlatinumCustomerIDs` are relations. We do not omit the default `Unbounded` window to highlight the difference: Streams must be applied a window, while relations must not be applied a window. After applying a window, a stream becomes a relation.

Note that for external data sources to be usable inside the engine, the engine must be fed with the external data by a user-created program. This only works with sets of tuples following a schema, such as database tables. In this query, we assume that the engine is fed with the data of the external table `Customer`. STREAM only works with the internally stored `Customer` table, so features present in the external database system are not available.

In case the external data may change (in this example, upon new customers), STREAM must be used with low-level code to receive any changes. Once STREAM recognizes a change, it is treated like a new event in a stream; all queries involving the relation are re-executed.

Query 5 (Tuple windows, aggregation by group)

```
Select custId, SUM(value)
From Order[Partition By custId Rows 5]
where SUM(value) >= 3000
```

Using a tuple window, we select the last five orders for each customer. To accomplish this, we must use a partitioning window, or else we would only get the last five orders overall.

CQL inherits the `SUM` operator from SQL, providing the functionality we need for this query. It sums up the five orders. We then give an answer only if the sum is above the threshold.

Query 6 (Counting)

```
Select custId, count(*)
From FailedLoginAttempt[Range 5 Minutes]
```

```
Group By custId
Having count(*) >= 3
```

This query works similar to the one above, but using the SQL constructs `Group By` and `Having` instead. Unlike with tuple windows, these work with time windows. We group the failed login attempts for each customer account, count them and return the groups greater than the threshold.

Query 7 (Aggregation)

```
SmoothedIndex (price):

  Select Rstream(AVG(price))
  From IdxTick[Range 30 Seconds]
```

This is a straightforward implementation of the smoothing query. It is assigned the name `SmoothedIndex`, with one data field. We use `Rstream` to ensure that this query answers even if the average did not change from the last execution.

Query 8 (Event instance selection)

This query can not be fully expressed, but approximated.

```
SmoothedIndexLastHour(price):

  Select Dstream(price)
  From SmoothedIndex[Range 1 Hour]

StkLastHour(price):

  Select Dstream(price)
  From StkTick[Range 1 Hour]

Select true
From StkLastHour[Rows 1] As sd,
  SmoothedIndexLastHour[Rows 1] As id,
  StkTick[Rows 1] As s,
  SmoothedIndex[Rows 1] As i
Where (sd.price / s.price) - (id.price / i.price) >= 0.02
```

We found nothing about event instance selection, that is selecting the first event in a window in our case, in either the paper or the manual. However, some event instance selection features can be approximated.

By default, all streams are insert streams. At a given time point τ, they contain all events that are present now, but have not been present at the previous time point. However, if we specify a stream to be a delete stream (`Dstream`), the resulting stream contains all events that have been present at the previous time point, but are no longer present.

In this query, it is used to get the latest events outside the 1-hour window for both index and stock. Essentially, the delete streams are the same as the original streams, only that they lag behind the original streams by 1 hour. We get the latest event of a stream by using

a 1-row tuple window; it is applied to all four streams. After obtaining these four events, we apply the condition.

Note that while we try to select the first event in a window (which is not possible), we will only get the event *before* the first event in the window. For example, if we have a tick at 0:58, a tick at 1:01, and the final tick at 1:53, and execute the query at 2:00, the tick at 0:58 is considered, although we wanted the one at 1:01. So, the two events selected from this stream are the 0:58 tick and the 1:53 tick. We did not find a way to obtain the other tick.

Query 9 (Sequences)

This query is not expressible using the given streams. CQL does not know the concept of sequences. The only constructs affected by event timestamps are windows. This is not enough to detect sequences in the given streams of rises and falls.

In case the rises and falls are numbered by occurrence time, such as in Figure 8 of Section 2.5, thus having an additional ID attribute, it would work with the following join query:

```
Select f.lower, f.id
From Falls[Rows 1] As f, Rises[Rows 1] As r
Where r.id = f.id + 1
```

The ID is needed for the next query.

Query 10 (Event instance consumption)

This query is not expressible using the given streams. We found nothing about event instance consumption, that is preventing the query from reusing V patterns in our case, in the paper and the manual. In order to approximate the query, we would have to assume the numbering of the rises and falls and inclusion of the ID in the V pattern as in the above query, resulting in the following query:

```
Select true
From V[Range 1 hour] As v1, V[Range 1 hour] As v
Where v1.id < v.id And v.lower / v1.lower >= 0.99
  And v.lower / v1.lower <= 1.01
  And Not Exists (
    Select *
    From V[Range 1 hour] As v2
    Where v2.id < v.id And v2.lower / v1.lower >= 0.99
      And v2.lower / v1.lower <= 1.01
  )
```

This query works as follows: For every pair of V patterns, it is checked whether they are in the right order, whether they have approximately the same minimum, and whether there is no V pattern with a similar minimum in between.

Ease of Use and Implementation Status

STREAM is the product of a past research project. STREAM comes with a manual, although it refers readers to a research paper [ABW03] for the language features. As it is the usual case with research papers, there are only few examples, few reference material, even explanation of the syntax is shortened.

As for the implementation, it is quite complete, although some limitations exist, most notably that no subqueries may appear in the `Where` clause. It was buildable from source using the standard GNU procedure. STREAM has two modes of operation: it can be embedded into C++ applications, or run standalone, directed by configuration files (that have a custom format). We only used the latter mode. When trying to verify the syntax of our queries, some of them were rejected because of `Unknown error`s, followed by error codes. We did not receive any more helpful error messages, nor were the error codes documented.

Implementing the queries was more imitating and adapting examples than creating them from scratch. Since CQL is derived from the well-known SQL language, it was easy to learn, but also suffers from SQL's limitations, as seen in the negation query that had to be implemented using a subquery.

Implementations (engines): STREAM

Evaluated version (release date): STREAM 0.6.0 (unknown date, but before January 2006)

Web site: `http://infolab.stanford.edu/stream/`

4.2 Borealis

Borealis is a research CEP engine developed at Brandeis University, Brown University, and MIT. It combines the event processing capabilities of an earlier engine, Aurora, with functionality from the Medusa project, enabling Borealis to perform distributed event processing. Borealis extends both Aurora and Medusa, superseding them, resulting in its inclusion in this survey instead of Aurora. Borealis has an implementation released under the BSD license. Our evaluation is based on the Borealis Application Programmer's Guide [bor].

Borealis uses a "boxes-and-arrows" approach to CEP: Complex event queries can be described graphically with basic event queries as boxes and streams as arrows connecting boxes. Examples for basic event queries, thus boxes, are the Filter box, that produces an output stream from its input stream by filtering out undesired events, and the Aggregate box, that produces an output stream from its input stream containing values aggregated over the event data. This approach was originally developed in Aurora. While Borealis queries are written in XML, it still uses the boxes-and-arrows principle.

Other than engines using STREAM's approach, Borealis boxes do not convert streams to relations on every time point, but every time an event enters the box. The difference between these processing models is pointed out in an extra paper by another research group [JMS+08]. As mentioned in Section 3.1, the conceptual clock of Borealis is more similar to the StreamBase model mentioned in that paper, thus incrementing its conceptual clock for every tuple processed, and queries are executed for every incoming tuple. Also, the focus of the Borealis project was strongly on query evaluation, resulting in less abstract queries

than STREAM, but allowing the programmer to fine-tune evaluation aspects.

A Borealis query consists of one or more box declarations. A box must declare its input and output streams, its type (for example, `join`), and its parameters. A stream must define its structure by referring to a schema declaration.

Query 1 (Disjunction)

```
<schema name="Login">
  <field name="custId" type="int" />
</schema>

<schema name="Order">
  <field name="custId" type="int" />
  <field name="value" type="int" />
</schema>

<schema name="PageRequest">
  <field name="custId" type="int" />
  <field name="page" type="string" />
</schema>

<schema name="Intermediate">
  <field name="custId" type="int" />
</schema>

<input stream="Login" schema="Login" />
<input stream="Order" schema="Order" />
<input stream="PageRequest" schema="PageRequest" />
<output stream="Intermediate1" schema="Intermediate" />
<output stream="Intermediate2" schema="Intermediate" />
<output stream="Intermediate3" schema="Intermediate" />

<input stream="Intermediate1" schema="Intermediate" />
<input stream="Intermediate2" schema="Intermediate" />
<input stream="Intermediate3" schema="Intermediate" />
<output stream="CustomerActivity" schema="Intermediate" />

<box name="UnifyLogin" type="map">
    <in name="Login" />
    <out name="Intermediate1" />
    <parameter name="expression.0" value="custId" />
    <parameter name="output-field-name.0" value="custId" />
</box>

<box name="UnifyOrder" type="map">
    <in name="Order" />
    <out name="Intermediate2" />
    <parameter name="expression.0" value="custId" />
    <parameter name="output-field-name.0" value="custId" />
</box>
```

```
<box name="UnifyPageRequest" type="map">
    <in name="PageRequest" />
    <out name="Intermediate3" />
    <parameter name="expression.0" value="custId" />
    <parameter name="output-field-name.0" value="custId" />
</box>

<box name="Union" type="union">
    <in name="Intermediate1" />
    <in name="Intermediate2" />
    <in name="Intermediate3" />
    <out name="CustomerActivity" />
</box>
```

The `schema` tags are used to give a schema a name. A `stream` declaration can then reference a schema by its name to declare the structure of the data in the stream. In the queries below, we leave out most `schema` and `stream` declarations, as they can be inferred from the context.

Remember that boxes are the name for the basic event queries in Borealis. They can be used as building blocks for more complicated queries. The `union` box can combine the events from several streams into one single stream, provided the streams' schemas are the same. The `map` box can be used to apply functions on data of all events in a stream, but can also be used to change a stream's schema. Therefore, we use three `map` boxes to unify the schemas of logins, orders and page requests (essentially by just removing all attributes except the customer ID; at least one data field is required, although it is not used in the next query). The resulting streams are then combined by a `union` box.

Some boxes, such as `map`, have a variable number of parameters depending on the output stream's schema, not on the type of the box. For `map`, these are `expression` and `output-field-name`. There is one of these parameters for each data field of the resulting event. In our example, the events in the output stream of `map` have only one data field; its name is defined as the `output-field-name.0` parameter and its value is defined as the `expression.0` parameter. If a resulting event would have a second data field, its name would be defined as the `output-field-name.1` parameter and its value as the `expression.1` parameter. Expression parameters can contain various expressions, including arithmetic expressions.

This query is also graphically depicted in Figure 9.

Query 2 (Negation, time windows)

This query is not expressible for the following reasons. Recall that queries are executed every time an event enters the system. In this query, it may happen that no customer activities enter the system, so any query using the `CustomerActivity` stream as its only input stream is not executed in this case.

There is no negation box. An aggregation box exists, capable of counting; however, this box can have only one input stream. This has to be `CustomerActivity`, as this is the stream we have to apply counting on. However, if no event enters this stream, the counting query

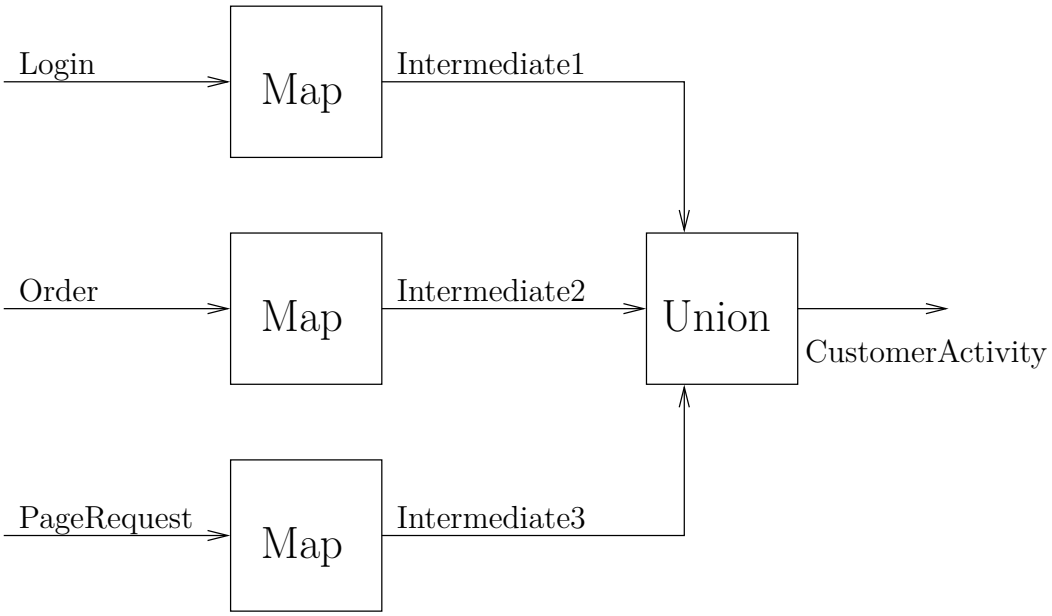

Figure 9: Query 1 in Borealis, graphically.

is never executed, thus not yielding a result that can be processed by other queries.

We also can not join the input stream with a timer stream (such as the one used in several STREAM queries above), as the join will not yield any results if the input stream is empty.

Therefore, negation is not expressible in Borealis. An example use of time windows is found in the next query.

Query 3 (Conjunction, data extraction)

Note: This query uses the schema definitions from Query 1.

```
<box name="Conjunction1" type="join">
  <in name="Order" />
  <in name="Payment" />
  <out name="OrderAndPayment" />
  <parameter name="predicate"
    value="left.orderId == right.orderId" />
  <parameter name="left-buffer-size" value="60" />
  <parameter name="right-buffer-size" value="60" />
  <parameter name="left-order-by" value="VALUES" />
  <parameter name="right-order-by" value="VALUES" />
  <parameter name="left-order-on-field" value="time" />
  <parameter name="right-order-on-field" value="time" />
</box>

<box name="Conjunction2" type="join">
  <in name="OrderAndPayment" />
  <in name="Delivery" />
```

```
  <out name="Trade" />
  <parameter name="predicate"
    value="left.orderId == right.orderId" />
  <parameter name="left-buffer-size" value="60" />
  <parameter name="right-buffer-size" value="60" />
  <parameter name="left-order-by" value="VALUES" />
  <parameter name="right-order-by" value="VALUES" />
  <parameter name="left-order-on-field" value="time" />
  <parameter name="right-order-on-field" value="time" />
</box>
```

This is a join in Borealis. It is not possible to join without using windows. This is how windows are implemented in Borealis:

Every schema contains a hidden field called `time`, an integer denoting the occurrence time in seconds. The user can order events (and apply windows) by any data field, and we use this time field. When using `VALUES` as value for the `window-size-by` parameter, and `time` as the field to order by, the window becomes a time window. The `window-size` parameter determines the size of the window, in seconds.

This query is split into two subqueries, as Borealis can not join more than two streams at the same time. The joined streams are referred to as `left` and `right` in a Join box, analogously to joins in relational algebra.

Query 4 (Using external data sources)

```
<box name="PlatinumCustomers" type="select">
  <in name="Order" />
  <out name="PlatinumCustomer" />
  <access table="Customer" />
  <parameter name="sql"
    value="select custId from Customer
      where platinum_status = true" />
  <parameter name="pass-on-no-results" value="0" />
  <parameter name="pass-result-sizes" value="0" />
</box>

<box name="PlatinumOrders" type="join">
  <in name="Order" />
  <in name="PlatinumCustomer" />
  <out name="PlatinumOrder" />
  <parameter name="predicate"
    value="left.custId == right.custId" />
  <parameter name="left-buffer-size" value="60" />
  <parameter name="right-buffer-size" value="1" />
  <parameter name="left-order-by" value="VALUES" />
  <parameter name="right-order-by" value="VALUES" />
  <parameter name="left-order-on-field" value="time" />
  <parameter name="right-order-on-field" value="time" />
</box>
```

This query would work if Borealis can somehow access databases outside the engine. This is not clear from its documentation; it is clear that it can create its own databases upon starting the engine, but it is not clear whether it can use databases already present, regardless of their format.

The database query in the `select` box is executed every time an order comes in, and outputs all platinum customers' IDs.[4] This is then used to join orders with platinum customer IDs to output all orders by platinum customers. We found no documentation about how to use event data in SQL queries, that is, keeping some variables in the SQL query and filling them with data from the event in the input stream. For example, it is conceivable to have a SQL query such as this:

```
select custId from Customer where custId = ?
  and platinum_status = true
```

When executing the query, we replace the `?` with the customer ID we are interested in, thus fetching only one result (if at all) from the database instead of all. However, it is not documented whether this is possible.

Query 5 (Tuple windows, aggregation by group)

```
<box name="OrderRows" type="aggregate" />
  <in name="OrderWithValue" />
  <out name="OrderRow" />
  <parameter name="aggregate-function.0" value="sum(price)" />
  <parameter name="aggregate-function-output-name.0"
    value="ordervolume" />
  <parameter name="order-by" value="FIELD" />
  <parameter name="order-on-field" value="time" />
  <parameter name="group-by" value="custId" />
  <parameter name="window-size" value="5" />
  <parameter name="window-size-by" value="TUPLES" />
  <parameter name="advance" value="1" />
  <parameter name="independent-window-alignment" value="1" />
</box>

<box name="GoodCustomers" type="filter">
  <in name="OrderRow" />
  <out name="GoodCustomer" />
  <parameter name="expression.0"
    value="ordervolume >= 3000" />
</box>
```

Aggregation and filtering can not be done in one box, so we split it up. The first box performs aggregation over a window.

In Borealis, a box may have multiple tuples in its input set. The `order-by` field determines the order they are processed. All tuples have a special field called `time` containing their timestamp. Using the `FIELD` called `time`, we process them in order of their timestamp.

[4] Unfortunately, there are no examples for accessing databases in the manual, so the `type` parameter of the box in the first query might be wrong.

The type and size of the window is determined by the next three parameters. We specify the size of the window to be 5 tuples. Using the `group-by` parameter, we make the window a partitioning window, such as the one found in the STREAM query. This means the window does not only span the last 5 orders overall, but the last 5 orders from each customer. The `advance` parameter determines how many tuples the window is moved after processing. The `independent-window-alignment` parameter is not important with an `advance` value of 1.

Finally, the output of the `aggregate` box is processed by a `filter` box, discarding all tuples not matching the condition.

Query 6 (Counting)

```
<box name="FailedLoginAttempts" type="aggregate" />
  <in name="FailedLoginAttempt" />
  <out name="FailedLoginAttemptRow" />
  <parameter name="aggregate-function.0" value="custId" />
  <parameter name="aggregate-function-output-name.0"
    value="custId" />
  <parameter name="aggregate-function.1" value="count(*)" />
  <parameter name="aggregate-function-output-name.1"
    value="attemptcount" />
  <parameter name="order-by" value="FIELD" />
  <parameter name="order-on-field" value="time" />
  <parameter name="group-by" value="custID" />
  <parameter name="window-size" value="300" />
  <parameter name="window-size-by" value="VALUES" />
  <parameter name="advance" value="1" />
  <parameter name="independent-window-alignment" value="1" />
</box>

<box name="IntrusionAttempts" type="filter">
  <in name="FailedLoginAttemptRow" />
  <out name="IntrusionAttempt" />
  <parameter name="expression.0"
    value="attemptcount >= 3" />
</box>
```

This is similar to the previous query. We count the number of events in the window and then apply a filter that creates an `IntrusionAttempt` event if we find enough `FailedLoginAttempt` events. In addition, we return the customer ID.

The difference between tuple and time windows is found in the `window-size-by` parameter. Time windows use `VALUES`; we order on the `time` field, so the window spans the next 300 values `time` can take. As each value is a second, it spans 5 minutes.

Query 7 (Aggregation)

```
<box name="SmoothIndex" type="aggregate" />
  <in name="IdxTick" />
  <out name="SmoothedIndex" />
  <parameter name="aggregate-function.0"
    value="average(price)" />
```

```
  <parameter name="aggregate-function-output-name.0"
    value="attemptcount" />
  <parameter name="order-by" value="FIELD" />
  <parameter name="order-on-field" value="time" />
  <parameter name="window-size" value="30" />
  <parameter name="window-size-by" value="VALUES" />
  <parameter name="advance" value="1" />
  <parameter name="independent-window-alignment" value="1" />
</box>
```

This is similar to Query 5, only omitting the `group-by` parameter (since there is no attribute to group by).

Query 8 (Event instance selection)

This query is not expressible. We found nothing about event instance selection, that is selecting the first event in a window in our case, in the manual. Delete streams such as found in STREAM are not present, either. Remember that in Borealis, queries are only executed for each tuple. Therefore, in Borealis, there can be no meaningful definition of a delete stream, as this requires a discrete time with well defined and evenly spaced time points.

Query 9 (Sequences)

This query is not fully expressible. The manual does not suggest whether it is possible to access the `time` field containing an event's timestamp (the concept of sequences is not mentioned at all). But even if this is the case, we cannot use this to detect direct sequences of a fall and a rise—that is, a fall and a rise and nothing in between. We also can not use negation (or counting in this case), as windows, thus aggregation, can only be defined over a fixed number of tuples or a fixed amount of time, but not over a start and an end event. However, it could be possible to detect sequences without this restriction by joining streams and applying a filter such as `a.time < b.time`. Also, if there would be a numbering of the falls and rises, it would be fully expressible, as in STREAM, using the following query:

```
<box name="JoinFallsAndRises" type="join">
  <in name="Fall" />
  <in name="Rise" />
  <out name="V" />
  <parameter name="predicate"
    value="left.id + 1 == right.id" />
  <parameter name="left-buffer-size" value="1" />
  <parameter name="right-buffer-size" value="1" />
  <parameter name="left-order-by" value="VALUES" />
  <parameter name="right-order-by" value="VALUES" />
  <parameter name="left-order-on-field" value="time" />
  <parameter name="right-order-on-field" value="time" />
  <parameter name="out-field.0" value="left.lower" />
  <parameter name="out-field-name.0" value="lower" />
</box>
```

Query 10 (Event instance consumption)

This query is not fully expressible. We found nothing about event instance consumption, that is preventing the query from reusing V patterns in our case, in the manual. Also, as in the above query, negation or counting can only be used with windows with a fixed size based on time or tuples, preventing an implementation that acts like event instance consumption in this specific case. This query can be implemented without the reuse restriction as follows:

```
<box name="DetectWs" type="join">
  <in name="V" />
  <in name="V" />
  <out name="W" />
  <parameter name="predicate"
    value="left.id &lt; right.id and left.lower / right.lower
      &lt;= 1.01 and left.lower / right.lower >= 0.99" />
  <parameter name="left-buffer-size" value="1" />
  <parameter name="right-buffer-size" value="1" />
  <parameter name="left-order-by" value="VALUES" />
  <parameter name="right-order-by" value="VALUES" />
  <parameter name="left-order-on-field" value="time" />
  <parameter name="right-order-on-field" value="time" />
</box>
```

Note that `<` is needed to encode < in XML.

Ease of Use and Implementation Status

Borealis is the product of a past research project. It comes with an application programmer's guide [bor]. This included installation and deployment instructions and a reference material. Unfortunately, although the reference material covered most of the high number of parameters, some documentation was incomplete, such as allowed expressions in `filter` boxes. Also, some information was wrong, such as the description of the output of an `aggregate` box.

While the implementation was available, building failed. Borealis depends on a library called NMSTL whose building failed. We did not find a way to fix this. This is the beginning of the error message (lines wrapped, replaced some characters not present in ASCII):

```
In file included from ../nmstl/ioevent:32,
                 from internal.cc:24:
../nmstl/callback:42: error: declaration of ‘typedef class
  nmstl::ptr<nmstl::callback<R, void, void, void, void>::func>
  nmstl::callback<R, void, void, void, void>::func::ptr’
../nmstl/ptr:184: error: changes meaning of ‘ptr’ from
  ‘class nmstl::ptr<nmstl::callback<R, void, void, void, void>::func>’
```

As seen in the evaluation, Borealis has a verbose XML syntax with lots of parameters. A number of them is there for efficiency reasons, although they were not helpful in just writing a query for testing expressivity without any aim of high efficiency. As it is usually the case with XML, it is better read and written by machines than by humans because of the sheer

length of the code. On top of it, it has its own approach to processing streams with its boxes and arrows, but it was intuitive enough to learn these concepts quickly.

Implementations (engines): Borealis

Evaluated version (release date): Version from Summer 2008. Application guide from May 2006.

Web site: `http://www.cs.brown.edu/research/borealis/public/`

4.3 AMiT

IBM Active Middleware Technology (AMiT) enables IBM middleware to become event-based. This technology is implemented in several products, most notably extending WebSphere Broker with CEP capabilities. As WebSphere is a commercial product, it is not freely available (requires registration); however, we have access to a research paper describing the capabilities of AMiT in detail [AE04] that we use for evaluation. It is the most detailed paper about how the CEP Detector Nodes' CEP facilities work. In addition, we use the Nodes' documentation [WSC], although it mostly concerns with administrational tasks, not with the programming. Particularly, it proposes a XML language for specifying events, based on composition operators.

AMiT queries generally have the following form:

```
<event name="...">
  <eventAttribute name="..." type="..." />
</event>
...
<event name="...">...</event>

<lifespan name="...">
  <initiator>...</initiator>
  <terminator>...</terminator>
</lifespan>

<situation name="..." lifespan="...">
  <operator>
    ...
  </operator>
  <situationAttribute name="..." type="..." />
  ...
  <situationAttribute name="..." type="..." />
</situation>
```

Basic events are declared with their attributes in `<event>` tags. *Lifespans* are windows defined by two events, an *initiator* and a *terminator* event. Lifespan types are therefore declared by referencing start and end event types. Whenever an event matching the initiator specification is detected, a new lifespan of this type is opened, and when an event matching the terminator specification is detected, the lifespan is closed. In both the paper and the manual, it is not explicitly mentioned whether the initiator and terminator events are included in the lifespan. Intuitively, it should be the case, so we assume it.

Complex events are called *situations*. A situation consists of at least one data attribute (it has to carry at least one kind of information), exactly one operator, and a lifespan type. Situations are only tried to be detected in lifespans of its type. A lifespan may be referenced by multiple situations. While the lifespan is open, events are processed to detect any of these situations.

Query 1 (Disjunction)

```
<event name="Login">
  <eventAttribute name="custId" type="string" />
</event>
<event name="Order">
  <eventAttribute name="custId" type="string" />
</event>
<event name="PageRequest">
  <eventAttribute name="custId" type="string" />
</event>

<lifespan name="Forever">
  <initiator><startup /></initiator>
  <terminator><noTerminator /></terminator>
</lifespan>

<situation name="CustomerActivity" lifespan="Forever">
  <operator>
    <nth quantity="1" detectionMode="immediate">
      <operandNth event="Login" />
      <operandNth event="Order" />
      <operandNth event="PageRequest" />
    </nth>
  </operator>
  <situationAttribute name="custId" type="string" />
</situation>
```

The unusually high length of this code is explained with AMiT's processing model, requiring basic events, lifespans, and situations to be fully declared. Basic events are omitted in our later queries.

This query uses one situation. The situation's lifespan is bounded by the special `startup` and `noTerminator` events, that means, the lifespan spans the whole stream. Its operator is `nth`. This operator assigns a weight to its operands and computes a sum over its operands. Each operand contributes 0 to this sum, if it is not detected, and its weight, if it is detected. `nth` fires if the sum is exactly the specified threshold. This matches the behavior we would expect from an "or" operator.

In this query, each event has the default weight of 1, and the threshold of `nth` is also 1, so the `CustomerActivity` event is detected every time one of the events is found. The basic events are then, by default, consumed, so they are not used in any further answers to this query. Generally, the weight can be any number. The threshold can likely be any expression (suggested by the fact it is specified as `CDATA` in the DTD for query documents, so it is not

restricted to numbers).

Once such a situation has been detected, the engine looks for further situations of this type, as the lifespan has not ended. However, the event used for this situation is (by default) consumed, so it does not raise the situation again.

This query is likely not complete, as the situation attribute's value is not specified. However, it is not documented how this can be done. There are no examples and no syntax definitions (other than `CDATA` in the syntax reference, which is essentially the same as no limitations). It is, however, likely that the value can include member events' data and arithmetic expressions.

Query 2 (Negation, time windows)

This query is only expressible using an additional timer stream, like the one in STREAM.

```
<lifespan name="FiveMinutes">
  <initiator><eventInitiator event="Timer" /></initiator>
  <terminator>
    <expirationInterval timeInterval="300" />
  </terminator>
</lifespan>

<situation name="NoCustomerActivity"
  lifespan="FiveMinutes">
  <operator>
    <not>
      <operandNot event="CustomerActivity" />
    </not>
  </operator>
  <situationAttribute name="time" type="string" />
</situation>
```

This query requires an additional `Timer` stream that sends timer events, for example, every minute. These events may then open a lifespan opening at the event's occurrence and closing 5 minutes later, unless such a lifespan is already open (specified with the default value of the `correlate` parameter we encounter later again). Note that in AMiT, time is to be defined in a domain-specific way; in our case, 1 unit of time is 1 wall-clock second, although it could also be 1 minute, or 1 trading day in other applications.

The timer stream is required because the customer activity stream might not contain any events at all. In order to open the lifespan in that case, we therefore need another stream. There is no such timer stream built in, so we have to supply one.

The timer events start the evaluation of the situation, looking for customer activities and only firing if nothing was found until the end of the lifespan. It will not fire before that, as negation, by default, waits until the end of the lifespan (unlike the default behavior of `nth`, for example). Customer activities may be used like basic events (mentioned in Section 2.3.4 of the paper, "Nested Situations").

Query 3 (Conjunction, data extraction)

```
<situation name="Trade" lifespan="Forever">
  <operator>
    <all where="o.orderId = d.orderId and o.orderId = p.orderId">
      <operandAll event="Order" as="o" />
      <operandAll event="Delivery" as="d" />
      <operandAll event="Payment" as="p" />
    </all>
  </operator>
  <situationAttribute name="orderId" type="string" />
</situation>
```

We abstract orders, deliveries and payments belonging to the same order into `Trade` events. This is done with the conjunction operator, `all`, that has a condition attached in its `where` attribute. The `all` operator requires all of its operands to be detected, and further the condition in the `where` clause to be met, to detect a situation.

With this implementation of the query, the machine will go out of memory if there are many order, delivery or payment events without forming a triple (for example, if there is no payment because the customer does not pay). These events would infinitely remain in the memory. On the other hand, events becoming part of `Trade` events are consumed. We can, however, impose the 60-seconds limit allowed by the specification by using this lifespan for this query:

```
<lifespan name="Trade60Seconds">
  <initiator>
    <eventInitiator event="Order" correlate="add" />
  </initiator>
  <terminator>
    <expirationInterval timeInterval="300" />
  </terminator>
</lifespan>
```

Whenever an order is detected, a new lifespan of this type opens. The `correlate` parameter is described in query 6.

Query 4 (Using external data sources)

Nothing is mentioned about using non-event data in the paper. In the manual for the CEP Detector Nodes, it is mentioned that the nodes can not read external data sources yet, but they can write certain kinds of information into such databases (although the exact nature is not further specified).

Query 5 (Tuple windows, aggregation by group)

Neither the paper nor the manual mention the presence of aggregation functions. Tuple windows cannot be expressed, either; there only exist windows extending a fixed time after the initiator event.

Query 6 (Counting)

```
<lifespan name="FLAFiveMinutes">
  <initiator>
    <eventInitiator event="FailedLoginAttempt" correlate="add" />
  </initiator>
  <terminator>
    <expirationInterval timeInterval="300" />
  </terminator>
</lifespan>

<key name="FLACustId">
  <eventKey event="FailedLoginAttempt" attribute="custId" />
</key>

<situation name="IntrusionAttempt" lifespan="FLAFiveMinutes">
  <operator>
    <atleast quantity="3">
      <operandAtLeast event="FailedLoginAttempt" />
      <keyBy name="FLACustId" />
    </atleast>
  </operator>
  <situationAttribute name="custId" type="string" />
</situation>
```

As in query 2, we use a 5-minute time window. Unlike in that query, lifespans are opened by `FailedLoginAttempt` events, not a timer. Also, they always open a new lifespan, regardless of whether there is already one of this type present (specified using the `correlate` attribute).[5] Since every `FailedLoginAttempt` event opens a new lifespan, thus some kind of thread, this may strain the machine the engine runs on if there are many failed login attempts. This, however, is needed; for example, if we would not open a new lifespan for each event, and events would occur at 0:01, 4:39, 5:04 and 5:17, the intrusion attempt consisting of the last three events would not be detected, as the first two events are only present in the first lifespan, and the 5:04 event opens a new one.

Attributes of event types can be declared *keys*. This expresses that events of this type semantically belong together if they agree on this attribute. We use the `atleast` operator with a key attribute so it counts only failed login attempts with the same customer ID; the `custId` attribute of the `FailedLoginAttempt` type has been declared a key referencing the customer. As the name suggests, this `atleast` fires if it finds at least 3 attempts belonging together.

Query 7 (Aggregation)

See Query 5.

[5] On a sidenote, this will reuse events quite often, even though events are removed from their lifespan if they are used to generate an answer. For example, if there are four attempts inside five minutes, there will be one answer for the first three (from the first lifespan), and there will be another answer for the last three (from the second lifespan). If there are even six attempts, there will be all six events in the first lifespan, resulting in two answers from the first lifespan in addition to all answers from the other lifespans.

Query 8 (Event instance selection)

As this query depends on the previous query, we think of the previous query to be expressible and creating `SmoothedIndex`.

```
<lifespan name="StkOneHour">
  <initiator>
    <eventInitiator event="StkTick" correlate="add" />
  </initiator>
  <terminator>
    <expirationInterval timeInterval="3600" />
  </terminator>
</lifespan>

<situation name="Outperformance" lifespan="StkOneHour">
  <operator>
    <all detectionMode="deferred" where=
      "(sd.price / s.price - (id.price / i.price) >= 0.02">
      <operandAll event="StkTick" as="sd"
        quantifierType="absolute" />
      <operandAll event="SmoothedIndex" as="id"
        quantifierType="absolute" />
      <operandAll event="StkTick" as="s" quantifier="last"
        quantifierType="absolute" />
      <operandAll event="SmoothedIndex" as="i" quantifier="last"
        quantifierType="absolute" />
    </all>
  </operator>
  <situationAttribute name="symbol" type="string" />
</situation>
```

Using `all` with a condition, we collect the events in the lifespan and take the first and last tick of the stock and the smoothed index, then check the condition.

The quantifier determines whether the first (or last) event in the timespan is taken. The default quantifier is `first`.

The quantifier type determines whether there is an answer at all if the first (or last) event does not fulfill the condition. As a new example, consider there are two A events whose only data field has the values 19 and 21, respectively, and the query has the condition that the data field has to be 20 or higher. When using the `first` quantifier, two quantifier types are conceivable: In the first one, there is no answer, because the first event's data does not fulfill the condition. In the second one, the first A fulfilling the condition (if present), that means, the second is selected. In our case, we select the first event in each stream, without checking the condition (first type of semantics).

Both semantics are available. The paper does, however, not specify which of these is `absolute`, and which of these is `relative`; in the explanation, it uses "strictly first" and "first" only, while in the syntax definition, it uses `absolute` and `relative`. We suggest the right one, "strictly first", to be `absolute`. The manual does not deal with this matter.

Query 9 (Sequences)

```
<lifespan name="FallStart">
  <initiator>
    <eventInitiator event="Fall" correlate="add" />
  </initiator>
  <terminator>
    <eventTerminator event="Fall" terminationType="discard"/>
    <eventTerminator event="Rise" />
  </terminator>
</lifespan>

<situation name="V" lifespan="FallStart">
  <operator>
    <sequence>
      <operandSequence event="Fall" />
      <operandSequence event="Rise" />
    </sequence>
  </operator>
  <situationAttribute name="lower" type="string"
    expression="Rise.lower" />
</situation>
```

The sequence is detected using the `sequence` operator. It requires its operands to occur in the specified order.

We disallow connecting non-adjacent falls and rises by using a special lifespan. A lifespan always begins with a fall; but if another fall follows, the current lifespan is terminated without creating any situation (using the `discard` option; note that the new fall initiates a new lifespan). If a rise follows instead, the situation is created as in the above queries. Also, we use a situation attribute this time, assigning to it the minimum of the V.

Query 10 (Event instance consumption)

```
<lifespan name="VOneHour">
  <initiator>
    <eventInitiator event="V" correlate="add" />
  </initiator>
  <terminator>
    <expirationInterval timeInterval="3600" />
  </terminator>
</lifespan>

<situation name="W" lifespan="VOneHour">
  <operator>
    <sequence repeatMode="once" where=
      "(v1.lower / v2.lower >= 0.99)
        or (v1.lower / v2.lower <= 1.01)">
      <operandSequence event="V" as="v1"
        quantifier="first" quantifierType="absolute" />
      <operandSequence event="V" as="v2"
        quantifier="last" quantifierType="relative" />
```

```
    </sequence>
  </operator>
  <situationAttribute name="minimum" type="number" />
</situation>
```

While AMiT does have event instance consumption features, they are not needed because of the lifespan semantics. This query works similar to the previous query, although we do not require the Vs to be adjacent, but apply conditions on their data instead. Each V is the initiator of a lifespan, it is also the first sequence operand (specified with the `first, absolute` quantifier), and if it can be used to detect a valid W, W detection stops in this lifespan (specified with the `once` repeat mode).

AMiT's event instance consumption features would allow us to reuse events in a lifespan if they are used for detecting a situation. If we detect a situation in a lifespan, we can select whether the situation's operands can be used in other situations in this lifespan. This feature is never used in our queries (it is specified using the `retain` option in `<operand...>` tags and defaults to `false`, thus consumption), but noteworthy.

Ease of Use and Implementation Status

AMiT is a technology employed in several IBM products. None of these products are publicly available. As such, a research paper [AE04] was most helpful in finding information about its approach to CEP. While it was helpful, it suffers from some inconsistencies between terminology used in the text (at the start of the paper) and the actual language constructs (described in an appendix). In some cases, there was no bridge provided; for example, we had to assume that "strictly first" is the same as setting `quantifier` to `first` and `quantifierType` to `absolute`.

Implementing the queries required reading the paper to understand the lifespan concept, the key concept, and some of the operators' semantics. Using this, it was easy to create the abstract structure of queries, however, due to the terminology inconsistencies, writing the queries in concrete syntax only works with the assumptions mentioned above. As with Borealis and ruleCore, the XML syntax caused even simple queries to span half of a page.

Implementations (engines): WebSphere Broker CEP Detector Nodes, among others

Evaluated version (release date): - (Research paper published in 2004.)

Web site: `http://www.haifa.il.ibm.com/dept/services/soms_ebs.html`

4.4 ruleCore

ruleCore is a CEP engine developed by Analog Software, building on research at the University of Skövde. As the name suggests, rules are the central concept of ruleCore. The ruleCore engine processes events using ECA (Event-Condition-Action) rules that consist of three parts: for every event (basic or complex), check a condition; if it is true, execute the action. ruleCore has two implementations; an open source variant called ruleCore, released under the terms of the GPL; as well as a commercial version called ruleCore CEP Server. Our evaluation is based on the user's guide found in the open source package, as well as the

paper [SB05]. As those two documents describe different versions of ruleCore, the user's guide is preferred over the paper.

We concentrate on the event detection capabilites, that is, we mainly deal with the E and C parts of the rules. ruleCore uses so-called *detector trees* for event detection. Leaf detector nodes (detector nodes without children) detect single events (they pick up events of their type). They are inactive until an event of their type is delivered to the rule (usually by entering the system, although exceptions are mentioned in the next paragraph), after which point they are always active. To detect complex events, a detector tree is built: the leaves detect simple events, and inner nodes detect complex events depending on whether its children detected events. For example, the `and` node activates when all of its children activated. A detector tree activates when its root activates. A rule may have several instances, each of which having its own detector tree. By using nodes and trees as main principles, this language is based on composition of queries using composition operators.

For each rule, there is an *event delivery* mode. Each rule has a sequence of active instances, each with its own detector tree. When an event enters the engine, the event is sent to the rule instances in their order. Depending on the event delivery mode, the event may be sent to every instance, to the first instance only, to the first instance that consumes it, or to all instances that have not received an event of this type yet.

In addition, each rule has a *instance creation* mode. A rule can declare event types to be initiators. A rule may be restricted to a fixed maximum number of instances, especially 1, or it may create a new instance when an event of an initiator type enters the system.

Unfortunately, the concrete syntax of event delivery and instance creation modes are not documented, so we note it in the explanations for each query instead if they deviate from the default values. Although not explicitly stated, we assume that by default events are delivered to all rule instances and initiator events create a new event instance.

ruleCore rules are created by using a GUI tool, but can be described in XML, too, using a composition-operator-based language called rCML. We will use the XML representation. A set of ruleCore queries has the following structure:

```
<rules>
  <rule ...>
    ...
    <event-ref ...>...</event-ref>
    <condition-ref ...>...</condition-ref>
    <action-ref ...>...</action-ref>
  </rule>
  ...
  <rule ...>...</rule>
</rules>

<event-defs>
  <event-def name="..." type="basic">
    <parameters>
      ...
    </parameters>
  </event-def>
  ...
```

```
  <event-def name="..." type="composite">
  ...
  </event-def>
</event-defs>

<condition-defs>
  <condition-def ...>...</condition-def>
  ...
  <condition-def ...>...</condition-def>
</condition-defs>

<action-defs>
  <action-def ...>...</action-def>
  ...
  <action-def ...>...</action-def>
</action-defs>
</rules>
```

So, a set of ruleCore rules defines basic events, complex events, conditions, actions, and rules. ECA rules are defined by referencing these definitions. There are some more XML nodes, especially on upper levels, but we will only state the `<rules>`, `<event-defs>`, and `<condition-defs>` subtrees; the others are not relevant for this work. For some language constructs, such as `<action-defs>` nodes, there is no sufficient documentation about generating events (it would be possible to generate rCML using the designer tool, we could not get it to run).

Query 1 (Disjunction)

```
<rules>
  <rule name="Rule 1" ...>
    <description></description>
    <event-ref enabled="yes">CustomerActivity</event-ref>
    <condition-ref enabled="no"></condition-ref>
    <action-ref enabled="no"></action-ref>
  </rule>
</rules>

<event-defs>
  <event-def name="Login" type="basic">
    <parameters />
  </event-def>
  <event-def name="Order" type="basic">
    <parameters />
  </event-def>
  <event-def name="PageRequest" type="basic">
    <parameters />
  </event-def>

  <event-def name="CustomerActivity" type="composite">
    <detector>
```

```
      <or>
        <event-ref>Login</event-ref>
        <event-ref>Order</event-ref>
        <event-ref>PageRequest</event-ref>
      </or>
    </detector>
  </event-def>
</event-defs>
```

Disjunctions are straightforward to use in ruleCore. The ECA rule detects any of the three simple events we are interested in. The complex event uses an `or` detector node for this purpose. It becomes active when one of the subnodes becomes active. There is no condition needed in this case. The detected `CustomerActivity` is sent back into the engine; this is the default behavior when the action is disabled.

Query 2 (Negation, time windows)

The `timeport` node is the only built-in node that can be used to express a sliding time window. The concrete syntax is unknown (lack of documentation for `<timeport>`), but it should be possible. Using a `timeport` node, we can use a periodic time specification so that every minute[6], a `timeport` node opens for five minutes, waiting for a customer activity. This `timeport` node then becomes the child of a `not` node, so the `not` node activates when no customer activity was found in the last 5 minutes (which is the case when no customer activity was found when the `timeport` node closes).

`timeport` nodes can not be opened by events, so we cannot use additional timer streams as in other languages.

Query 3 (Conjunction, data extraction)

```
<event-defs>
  <event-def name="Order" type="basic">
    <parameters>
      <parameter name="orderId" type="string" />
    </parameters>
  </event-def>
  <event-def name="Delivery" type="basic">
    <parameters>
      <parameter name="orderId" type="string" />
    </parameters>
  </event-def>
  <event-def name="Payment" type="basic">
    <parameters>
      <parameter name="orderId" type="string" />
    </parameters>
  </event-def>

  <event-def name="PossibleTrade" type="composite">
    <event-selector>Condition 1</event-selector>
```

[6]Finer granularities are not possible, resulting in less answers to this query than specified.

```
    <detector>
      <and>
        <event-ref>Order</event-ref>
        <event-ref>Delivery</event-ref>
        <event-ref>Payment</event-ref>
      </and>
    </detector>
  </event-def>
</event-defs>

<condition-defs>
  <condition-def always-true="no" composite="yes"
    name="Condition 1">
    <parameters>
      <parameter name="o">
        <event>
          <event-ref>basic(Order)</event-ref>
          <param-ref>orderId</param-ref>
          <instance>last</instance>
        </event>
      </parameter>
      <parameter name="d">
        <event>
          <event-ref>basic(Delivery)</event-ref>
          <param-ref>orderId</param-ref>
          <instance>last</instance>
        </event>
      </parameter>
      <parameter name="p">
        <event>
          <event-ref>basic(Payment)</event-ref>
          <param-ref>orderId</param-ref>
          <instance>last</instance>
        </event>
      </parameter>
    </parameters>
    <expressions>
      <expression name="c1">
        <lhs>
          <param-ref>o</param-ref>
        </lhs>
        <operator>equal</operator>
        <rhs>
          <param-ref>d</param-ref>
        </rhs>
      </expression>
      <expression name="c2">
        <lhs>
          <param-ref>o</param-ref>
        </lhs>
        <operator>equal</operator>
        <rhs>
```

```
          <param-ref>p</param-ref>
        </rhs>
      </expression>
    </expressions>
    <composite-condition>
      <and>
        <condition-ref>c1</condition-ref>
        <condition-ref>c2</condition-ref>
      </and>
    </composite-condition>
  </condition-def>
</condition-defs>
```

Similar to the disjunction query, although we now also use a condition to check whether the order, delivery and payment actually belong together. A condition consists of extracting data from events and performing comparisons.

The conjunction consists of two parts. The first, `c1`, tests whether the order and the delivery have the same order ID. In the `parameters` tag, the events and after that the desired data are selected. The collector tree only has three events to select from, one of each type, so we select them by their type[7] and assign a name to the data. In the `expressions` tag, the data is compared with each other. The second condition, `c2`, has the same structure. The two conditions are combined to form a composite condition (composed using an `and` operator).

As in other languages, this implementation of this query is likely to exhaust memory, even though events are consumed when detector trees complete detection. However, detector trees infinitely waiting for non-existent events will stay in memory. Imposing the 60-seconds limit we allowed in the specification is, however, difficult, as pointed out in the previous query; time windows are hard to express.

Query 4 (Using external data sources)

The version of ruleCore we evaluated can only be fed with events. It can not access non-event data such as data stored in relational databases, not even in the C part of a rule[8], so using external data sources for CEP is not possible in this language.

Query 5 (Tuple windows, aggregation by group)

Aggregation features are absent. Neither variables nor aggregation functions are available. Tuple windows are not present, either.

Query 6 (Counting)

Interestingly, the `count` node is not usable for this query. We want to count failed login attempts *for the same account*, thus using event data. However, the `count` node of rCML can

[7] `instance` is set to `last`, although does not make a difference compared to `first` as there is only one event. This is the event instance selection feature, used with event sets in queries 6 and 8.

[8] This deviates from other ECA languages, which can usually access databases in their C part.

not express this restriction. Due to lack of documentation for event selection in sequences, we do not know whether it is possible to express this query in the following way, however, we assume it.

```
<event-def name="FLARow" type="composite">
  <event-selector>FLASameAccount</event-selector>
  <sequence>
    <event-ref>FailedLoginAttempt</event-ref>
    <event-ref>FailedLoginAttempt</event-ref>
    <event-ref>FailedLoginAttempt</event-ref>
  </sequence>
</event-def>
...
<condition-defs>
  <condition-def always-true="no" composite="yes"
  name="FLASameAccount">
    <parameters>
      <parameter name="cid1">
        <event>
          <event-ref>basic(FailedLoginAttempt)</event-ref>
          <param-ref>custId</param-ref>
          <instance>1</instance>
        </event>
      </parameter>
      <parameter name="cid2">
        <event>
          <event-ref>basic(FailedLoginAttempt)</event-ref>
          <param-ref>custId</param-ref>
          <instance>2</instance>
        </event>
      </parameter>
      <parameter name="cid3">
        <event>
          <event-ref>basic(FailedLoginAttempt)</event-ref>
          <param-ref>custId</param-ref>
          <instance>3</instance>
        </event>
      </parameter>
    </parameters>
    <expressions>
      <expression name="c1">
        <lhs>
          <param-ref>cid1</param-ref>
        </lhs>
        <operator>equal</operator>
        <rhs>
          <param-ref>cid2</param-ref>
        </rhs>
      </expression>
      <expression name="c2">
        <lhs>
          <param-ref>cid1</param-ref>
```

```
        </lhs>
        <operator>equal</operator>
        <rhs>
          <param-ref>cid2</param-ref>
        </rhs>
      </expression>
    </expressions>
    <composite-condition>
      <and>
        <condition-ref>c1</condition-ref>
        <condition-ref>c2</condition-ref>
      </and>
    </composite-condition>
  </condition-def>
</condition-defs>
```

We use the `sequence` node instead of the `count` node. This node is then wrapped into a periodic `timeport` node (as in query 2) and a condition is added in the event's definition (similar to query 3). As in query 2, the lack of documentation for `timeport` prevents us from writing it up.

In the condition, we use the assumption that we can obtain the three member events using the `instance` parameter in this way. We read their customer IDs and compare them.

Query 7 (Aggregation)

See Query 5.

Query 8 (Event instance selection)

This query is only expressible when using timer streams. We also assume that there is a stream called `SmoothedIndex` with the results of the previous query.

```
<event-defs>
  <event-def name="Timer1" type="basic">...</event-def>
  <event-def name="Timer2" type="basic">...</event-def>

  <event-def name="Between1" type="composite">
    <event-selector>Outperformance</event-selector>
    <detector>
      <between>
        <event-ref>Timer1</event-ref>
        <event-ref>Timer2</event-ref>
      </between>
    </detector>
  </event-def>

  <event-def name="Between2" type="composite">
    <event-selector>Outperformance</event-selector>
    <detector>
```

```
      <between>
        <event-ref>Timer2</event-ref>
        <event-ref>Timer1</event-ref>
      </between>
    </detector>
  </event-def>
</event-defs>

<condition-defs>
  <condition-def name="Outperformance" type="composite">
    <parameters>
      <parameter name="id">
        <event>
          <event-ref>basic(SmoothedIndex)</event-ref>
          <param-ref>price</param-ref>
          <instance>first</instance>
        </event>
      </parameter>
      <parameter name="sd">
        <event>
          <event-ref>basic(StkTick)</event-ref>
          <param-ref>price</param-ref>
          <instance>first</instance>
        </event>
      </parameter>
      <parameter name="i">
        <event>
          <event-ref>basic(SmoothedIndex)</event-ref>
          <param-ref>price</param-ref>
          <instance>last</instance>
        </event>
      </parameter>
      <parameter name="s">
        <event>
          <event-ref>basic(StkTick)</event-ref>
          <param-ref>custId</param-ref>
          <instance>last</instance>
        </event>
      </parameter>
    </parameters>
    ...
  </condition-def>
</condition-defs>
```

ruleCore does have several event instance selection features. They are useful when using the `between` node. This node collects all events occurring between its two parameter events. It is then possible to obtain the first or last event out of the `between` event, for example, to access its data.

While the `between` node can be used to express this query, due to the lack of proper time windows, we require two additional streams. Both streams send an event into the engine

every minute, although the second stream is delayed by 30 seconds, so there is an event every 30 seconds[9] (for example, the first stream sends events at 0:00, 1:00, 2:00..., while the second stream sends events at 0:30, 1:30, 2:30...). Two events using a `between` node then collect the events between the event from one stream and the event from the other stream (the two types of events are needed; if we use only one type, both `between` parameter nodes will always accept the same event, resulting in a time interval of length 0). Each of these `between` using events is then processed by a condition in the event definition, obtaining the first and last ticks for the stock and the smoothed index, performing the calculations and comparison. Note that the calculations and comparison are left out, as these are not documented; we did not, for example, find information how to subtract one parameter from another, or what the concrete syntax of the "greater or equal" comparison operator is.

Query 9 (Sequences)

This query is not fully expressible.

```
<event-defs>
  <event-def name="Fall" type="basic">
     <parameters>
       <parameter name="lower" type="number" />
     </parameters>
  </event-def>
  <event-def name="Rise" type="basic">
     <parameters>
       <parameter name="lower" type="number" />
     </parameters>
  </event-def>

  <event-def name="NotFall" type="composite">
    <detector>
      <not>
        <event-ref>Fall</event-ref>
      </not>
    </detector>
  </event-def>

  <event-def name="V" type="composite">
    <detector>
      <sequence>
        <event-ref>Fall</event-ref>
        <event-ref>NotFall</event-ref>
        <event-ref>Rise</event-ref>
      </sequence>
    </detector>
  </event-def>
</event-defs>
```

[9] The interval of 30 seconds was chosen as this is the update interval of the smoothed index. Like in other queries, a smaller interval would increase the workload on the machine and detect more outperformances, although the latter increase would be rather small, considering the comparatively low number of events in the stock and index streams.

The `sequence` node requires its children to activate in the right order. If a child activates out of order, the `sequence` node will never become active; if the `sequence` node is the root of the detector tree, the rule instance ends. In this query, a rule instance is created for every fall, and fires when a rise follows.

This query requires a direct sequence, that is, a fall must be directly followed by a rise and may not have another fall in between. This aspect is possibly not expressible. Such an operator is not present, but maybe we can implement it differently by using a node looking for falls that is negated.

In the manual, it is not mentioned whether multiple leaves of a detector tree may pick up the same event. If they can, the negated node would pick up the same event as the first node, becoming permanently inactive[10] and thus always closing the detector tree. The sequence operator does not restrict children from processing events even if previous children have not yet activated.

On the other hand, if events are delivered to the basic event detection nodes in sequence and consumed by the first matching node, this aspect would be expressible.

There is another aspect that is not expressible at all. In the manual, complex events never have data. It is therefore not documented whether and how complex events can obtain data from their member events. Thus, we consider this query not fully expressible.

Query 10 (Event instance consumption)

This query is not fully expressible.

```
<event-defs>
  <event-def name="V" type="composite">
  ...
  </event-def>

  <event-def name="W" type="composite">
    <event-selector>Condition 10</event-selector>
    <detector>
      <sequence>
        <event-ref>V</event-ref>
        <event-ref>V</event-ref>
      </sequence>
    </detector>
  </event-def>
</event-defs>

<condition-defs>
  <condition-def name="Condition 10" type="composite">
    <parameters>
      <parameter name="v1">
        <event>
```

[10] Remember that basic event detection nodes are inactive until they find an event of their type, at which point they become permanently active. A negated basic event detector will therefore be active until it finds an event of its type, and then becoming permanently inactive.

```
          <event-ref>...</event-ref>
          <param-ref>lower</param-ref>
          <instance>first</instance>
        </event>
      </parameter>
      <parameter name="v2">
        <event>
          <event-ref>...</event-ref>
          <param-ref>lower</param-ref>
          <instance>last</instance>
        </event>
      </parameter>
    </parameters>
    ...
  </condition-def>
</condition-defs>
```

Again, we use a `sequence` node, this time looking for two V events. A condition is added, extracting the minima of both V events, and comparing them (extraction and comparison are omitted, as the syntax is not documented; see also queries 8 and 9).

As in query 9, the correctness of this query depends on whether two leaves can pick up the same event, which is not documented. If they can, this query is not expressible, as both leaves would always pick up the same event (and never different ones).

Further, the problem shown in Figure 8 from Section 2.5 is not solved correctly; there will be additional, wrong, answers. Event instance consumption features in the form of the event delivery modes are present, but not sufficient to express the needed restriction. Using the `consumption` parameter of the `rule` node, we can consume all or none of the events delivered to detector trees, but no subset (as needed in this query). Event instance consumption is configurated on a per-rule basis; a rule can not affect other rules. Finally, the usage of `not` nodes as in the previous query is not possible, as there is also a condition on the events in the sequence—however, it is not possible to express conditions on non-existent events in ruleCore.

Ease of Use and Implementation Status

The version of ruleCore we evaluated is a product of a research project conducted in cooperation with the industry. A fork of it is developed commercially, however, we have no access to it. The main sources of information were a paper [SB05] and the manual included in the open source package. Both described the processing model quite well, however, most of the concrete syntax was omitted.

The software was available in source form which was unhandy because of its dependencies on several non-mainstream libraries, and in a RPM package including all these libraries. We installed the latter, only to find out that it tries to use an old version of the standard C++ library that is already replaced with a newer version on our Ubuntu 8.10 system. So, we could not use the implementation, although it, especially its query designer (creation) tool, might have been helpful for learning more about the syntax. This is the error message

(lines wrapped):

```
  File "<string>", line 2, in ?
  File "./sbin/designer/main.py", line 6, in ?
  File "./lib/python/site-packages/qt.py", line 24, in ?
ImportError: libstdc++-libc6.2-2.so.3: cannot open shared
  object file: No such file or directory
```

ruleCore's query language is probably meant to be edited only by the designer, as its queries are the longest in all of the languages we evaluated. Even conditions have to be stated in their own XML tags, with each part of a condition in its own child. The incompleteness of the paper and the manual were not helpful in expressing the queries, either. While the general idea how a query might look like was easily conceivable, the exact implementation is sometimes unknown (especially if the query uses a time window, as the corresponding `timeport` node is inadequately documented).

Implementations (engines): ruleCore (open source) and ruleCore CEP Server (commercial)

Evaluated version (release date): ruleCore 1.0 (June 2004)

Web site: ruleCore (open source): `http://sourceforge.net/projects/rulecore/`
ruleCore CEP Server (commercial): `http://www.rulecore.com/`

4.5 SASE+

SASE+ is a CEP system developed at the University of Massachusetts, Amherst. It is an extension of the older SASE system. The system is designed for event streams with many events per time unit and also queries using large time windows, creating new issues regarding efficient query execution. The project's purpose is to devise techniques for high-performance querying of event streams, using a declarative, composition-operator-based language.

A SASE+ query has the following structure:

```
FROM ...
PATTERN ...
WHERE ...
WITHIN ...
HAVING ...
RETURN ...
```

Only the `PATTERN` clause is mandatory. The `FROM` clause specifies the input source, a collection of input streams in our sense. The input source is usually all input from the engine. It is omitted in all our examples, like in the report this evaluation is based on [DIG07].

The `PATTERN` clause applies a pattern on the input source, returning the complex events in the input source matching the pattern. Further restrictions not expressible by patterns alone can be included in the `WHERE` and `WITHIN` clauses, containing conditions on data and occurrence times, respectively. The `HAVING` clause is typically used when the pattern (meant as the combination of `PATTERN`, `WHERE`, and `WITHIN`) returns groups of events, allowing conditions to be expressed on these groups (including some not expressible in these other

clauses, such as aggregations). Finally, the `RETURN` clause specifies the exact structure of the output. When processing, the `PATTERN`, `WHERE`, and `WITHIN` clauses generate matches; all matches not fulfilling the `HAVING` clause are discarded, while the others are handled by the `RETURN` clause.

Our evaluation is based on information on the project homepage, a technical report [DIG07], building upon earlier work described in a paper [WDR06]. Whenever there is contradictory information between these sources, they are considered in this order. The first two sources describe SASE+, while the latter describes SASE.

Query 1 (Disjunction)

```
PATTERN ANY(LOGIN, ORDER, PAGEREQUEST)
RETURN  true
        AS CUSTOMERACTIVITY(b)
```

SASE+ streams can contain events of multiple types. The `ANY` operator from SASE returns all events of the specified event types.

The answer to this query is an event of the `CUSTOMERACTIVITY` type. It is implicitly defined to have a data attribute called `b`. The value of this attribute is `true`. The data attribute is only needed as it seems that events must have at least one data attribute.

Query 2 (Negation, time windows)

This query is only expressible using a timer stream like in other languages.

```
PATTERN SEQ(TIMER, ~(CUSTOMERACTIVITY))
WITHIN  5 minutes
```

`SEQ` is the sequence operator in SASE and SASE+. Negation on single events is presumably supported using the `~` operator.[11] Time windows can be expressed using the `WITHIN` clause, allowing a concise expression of this query. For every `TIMER`, a window of five minutes is opened and we look for `CUSTOMERACTIVITY` events, returning an answer only if we did not find one. The correctness of this query is ensured by the semantics of negation [WDR06].

As in other languages, this query's performance and frequency of answers on streams without customer activities is depending on the frequency at which timer events are sent.

Query 3 (Conjunction, data extraction)

```
PATTERN SEQ(ORDER o, DELIVERY d, PAYMENT p)
WHERE   o.orderId = d.orderId /\ o.orderId = p.orderId
RETURN  o.orderId
        AS TRADE(orderId)

PATTERN SEQ(ORDER o, PAYMENT p, DELIVERY d)
WHERE   o.orderId = p.orderId /\ o.orderId = d.orderId
```

[11] In SASE, the negation operator is `!`. However, this symbol is used in SASE+ for a different purpose, as explained in query 6, and negation is always expressed with `~`.

```
RETURN  o.orderId
        AS TRADE(o, d, p)
```

While SASE+ has a sequence operator, it can not express conjunctions directly. So, we use two subqueries covering all cases we are looking for (remember that orders always precede the other events). For large conjunctions, this would lead to very large queries.

Note that the concrete syntax for "and" is not described; $\wedge$ was used in the paper [DIG07], but certainly not in the language, as it is not part of ASCII. We used `/\` to denote "and". The other sources do not use this operator at all.

As in other languages, this implementation of this query will over time use all memory if parts of trades never happen. We can easily impose the 60-seconds limit allowed by the specification by adding the following `WITHIN` clause:

```
WITHIN  1 minute
```

Query 4 (Using external data sources)

The sources do not mention the possibility to use non-event data at all.

Query 5 (Tuple windows, aggregation by group)

```
PATTERN SEQ(ORDER+ o[ ])
WHERE   partition_contiguity(o[ ])
        { [custId] /\ o.LEN <= 5 }
HAVING  sum(o[].value) >= 3000
RETURN  o[1].custId
```

Now we consider a set of events in a window. The required pattern uses the feature added in the transition from SASE to SASE+: the Kleene closure, denoted by `+` after the event type. Using this construct, we look for sequences of orders fulfilling several conditions. First, they all must have the same customer ID, expressed by `[custId]`. Second, we restrict the sequence's maximum length to 5.

SASE+ allows to skip certain events when collecting events using the `+` operator by specifying an *event selection strategy*. `partition_contiguity` allows any non-order event and any order event with a different customer ID to occur in the sequence's timespan; however, if an order with the customer's ID falls within the sequence's timespan, it must be included.[12]

Now, the `HAVING` clause applies the condition that the sum of the orders' values must be at least 3000 to be an answer to this query. Note that the Kleene closure operator returns an array of events. The syntax (for example, `o[]`) follows the syntax of languages such as C.

Query 6 (Counting)

```
PATTERN SEQ(FAILEDLOGINATTEMPT+ a[ ])
WHERE   partition_contiguity(a[ ]!)
        { [custId] /\ a.LEN >= 3 }
```

[12] Other strategies, such as `skip_till_next_match`, would even allow orders by the customer to be skipped; in this case, we would search for *any* set of 5 orders by this customer with the minimum volume.

```
WITHIN  5 minutes
RETURN  a[1].custId
```

Similar to the above query, this time using a time window instead. Otherwise, since counting is an aggregation operation in SASE+, there is no new feature used in this query.

The `!` symbol in the `WHERE` clause specifies termination behavior: if it finds a sequence of failed login attempts fulfilling the conditions, no more events are added to the sequence. This means that a fourth attempt does not generate an answer to this query; however, if another sequence of three attempts follows, it would generate an answer.

Query 7 (Aggregation)

```
PATTERN SEQ(IDXTICK+ t[ ])
WITHIN  30 seconds
RETURN  avg(t[].price)
```

The first query in the stock market example can be implemented easily and concisely. According to the SASE+ semantics, this query returns an answer for every tick. Each answer averages over all ticks following the initiator event within 30 seconds.

Query 8 (Event instance selection)

We found this query to be not expressible in SASE+. To express this query, we have to find the first and last ticks for both the stock and the index and then applying a condition on these four events. Stock ticks and index ticks can be arbitrarily interleaved. Since these events have different types (`SMOOTHEDINDEX` and `STKTICK`), the Kleene closure is not applicable, since we are not looking for a sequence of index ticks followed by a sequence of stock ticks, but for an arbitrarily interleaved sequence of index and stock ticks. Now, as there is no pattern that can fetch all events in the window, the outperformance condition can not be applied, as conditions can only be applied on events from one pattern.

However, in general, selecting the first or last event in a window would be fairly easy to express in SASE+. Since the Kleene closure operator produces an array that also has a length attribute, we could access the first and last events in this array using `a[1]` and `a[a.LEN]`, respectively. (In SASE+, array elements are numbered starting with 1.)

Query 9 (Sequences)

```
PATTERN SEQ(FALL f, ~ANY(FALL f2, RISE r2), RISE r)
RETURN  f.lower
        AS V(minimum)
```

Since the sequence operator is always present in SASE+ patterns, it is not surprising that this query can be expressed. However, since the sequence operator allows other events to occur between the operands, including falls and rises, we restrict the pattern so we get only sequences of falls directly followed by rises. This excludes answers such as F_1, R_1 and F_2, R_2 for the fall-fall-rise-rise sequence F_1, F_2, R_1, R_2. This works as negated patterns are evaluated *after* positive patterns are matched, according to the semantics of SASE and

SASE+.

Query 10 (Event instance consumption)

This query is not fully expressible.

```
PATTERN SEQ(V v1, V v2)
WHERE   v2.minimum / v1.minimum >= 0.99
          /\ v2.minimum / v1.minimum <= 1.01
WITHIN  1 hour
```

We can not use any event instance consumption, since this feature is not present in SASE+. The restriction expressible by event instance consumption is not expressible otherwise in SASE+. Using the above code, we also return situations where a relevant V pattern (one with a relevant minimum) is between the V patterns matched by this sequence.

Interestingly, SASE+ allows negation to be applied to Kleene closure. This allows to detect sequences of events violating conditions. However, this type of negation has different semantics than non-presence of events. In this query, we are not interested in patterns violating conditions, but in patterns not violating conditions.

Ease of Use and Implementation Status

SASE+ is the product of a research project. No implementation was found. It is described on its project homepage (`http://sase.cs.umass.edu/`) and a paper [DIG07], building on earlier work [WDR06]. While all of these have examples and semantics, none of these comprehensively define the syntax.

As such, while the syntax is quite concise and easy to use after using the other composition-operator-based languages, it was sometimes not clear whether SASE+ actually has some features only described for SASE. This was especially the case with the `ANY` operator that was never mentioned in SASE+ material. However, under the assumption that all features in SASE are also in SASE+ (although sometimes using different keywords), the queries were easily expressible, especially since the semantics were provided.

Implementations (engines): -

Evaluated version (release date): - (Papers from 2006 and 2007.)

Web site: `http://sase.cs.umass.edu/`

4.6 Esper

Esper [Esp08] is an open-source CEP engine, developed by EsperTech Inc. and volunteers, released under the GNU General Public License (GPL v2). As stated on the official web site, it is designed for CEP and Event Stream Processing (ESP).

There are two implementations of Esper, Esper for Java and NEsper for .NET. Both supply an API to access the engine features, such as deploying queries, sending events into the engine and retrieving events out of the engine, in their respective language. Events are

objects in their respective language; for Esper, events can be instances of `java.util.Map`, `org.w3c.dom.Node` (Java representations of XML documents), or other Java objects.[13]

Regardless of the implementation language, queries are stated in a SQL-like language called Event Processing Language (EPL). While EPL has the usual features of a data stream language, it has been extended with pattern-matching constructs, inspired by composition-operator-based languages. Patterns in Esper work roughly like composition operators; applying the pattern to input streams produces an output stream that can be processed with the data stream constructs.

As mentioned in Section 3.1, there are different strategies regarding to how many executions of a query are triggered when there are multiple tuples with the same timestamp. In Esper, every tuple causes query execution, thus following the StreamBase model mentioned in [JMS+08]. In addition, queries are also executed whenever an event leaves a window.

In both example scenarios, we consider the event class names to use the event type names from Section 2, rewritten in CamelCase, as usual in Java. For example, a failed login attempt becomes a `FailedLoginAttempt`. An exception are orders, because `Order` is a reserved word in Esper; we use `CustomerOrder` instead.

Query 1 (Disjunction)

```
insert into CustomerActivity
select *
from pattern [every (Login or CustomerOrder or PageRequest)]
```

In Esper, every event type has a stream associated with it. This fact is used in many queries. In this query, we also use a pattern, a feature usually not present in data stream languages. In this query, we apply a disjunction pattern to get the three types of events we are interested in. This allows a concise implementation of this query.

Note that in Esper, streams produced by queries are insert streams by default, which is also the type of stream we need. Relation streams are not supported; delete streams can be created by adding `rstream` ("remove stream") after the `select`. `rstream` is just Esper's name for delete streams. The `insert` clause is optional and is only used to name the output stream (this feature can also be used to create a view on streams, similar to views in databases).

Esper does not have a `union` operator such as in STREAM.

Query 2 (Negation, time windows)

```
select true
from pattern [every timer:at(*, *, *, *, *, */30)
  -> (not (CustomerActivity where timer:within(5 min)))]
```

Here, the pattern waits 5 minutes for a customer activity, using the `timer:within` construct that acts like a stopwatch; if it finds one, the pattern returns nothing, else, it returns some internal event. We respond to this internal event by just acknowledging it.

[13] While instances of `Map` and `Node` are also Java objects, they receive a special treatment.

The pattern is evaluated every 30 seconds.[14] This is done using a timer stream, however, in Esper, it is built-in. The `timer:at` pattern atom triggers at the specified time, using its arguments denoting minutes, hours, days of month, months, days of week and seconds. For example, `timer:at(15, 20, 13, 2, 5, 0)` would be true at the next February 13th that is a Friday, at 8:15pm. Using `every`, the subpattern will be true at every such time point. In our example, we use wildcards so the subpattern will be true every 30 seconds (using the division operator `*/30` that actually says "every 30th value of this field is valid").

Without using patterns this query is not expressible without timer streams, as in other data stream languages.

Query 3 (Conjunction, data extraction)

```
select *
from pattern [every o=CustomerOrder -> every payment=Payment
  and every delivery=Delivery]
where o.orderId = payment.orderId
  and o.orderId = delivery.orderId
```

This query uses the pattern technique again, while also extracting and comparing some data. It would also have been possible to use a join, that is, just listing the three input streams in the `from` clause, similar to the implementation in STREAM.

To increase efficiency, we can impose the 60-seconds limit allowed by the specification by limiting the lifetime of a pattern instance:

```
select *
from pattern [every o=CustomerOrder -> (every payment=Payment
  and every delivery=Delivery) where timer:within(60 sec)]
where o.orderId = payment.orderId
  and o.orderId = delivery.orderId
```

Every time an order is encountered, all patterns of payments and deliveries are considered, but only if they happen inside 60 seconds of the occurrence of the order.

Query 4 (Using external data sources)

```
select *
from CustomerOrder as e,
  sql:StoreDB ["select custId from Customer
    where platinum_status = true and custId = ${e.custId}"] as d
where e.custId = d.custId
```

There are many kinds of external data sources that can be accessed with Esper. We assumed that the store has a database called `StoreDB` with a `Customer` table that tracks platinum status. We check for every order whether the customer behind the ID is a platinum customer. This is done by the SQL query that only gives a result if the customer is a platinum customer. Note the use of event data in the SQL query. Data from the order event is used in the SQL query. In order to use the database, it has to be made visible to Esper using Esper's API (in Java).

[14] As in other languages, this interval was arbitrarily chosen; small values increase the load on the executing machine, while large values may cause the query to miss answers.

Esper's integration into its host programming language also allows to use the return values of methods as a data source, for example, Java methods returning arrays or maps.

Query 5 (Tuple windows, aggregation by group)

```
select custId, sum(value)
from CustomerOrder.std:groupby(custId).win:length(5)
having sum(value) >= 3000
```

Windows in Esper are defined in the same way as in CQL, only with differing syntax. In Esper, tuple windows can be defined by adding `.win:length(...)` to a stream name, while time windows can be defined by adding `.win:time(...)`.

The grouping applied using `std:groupby`, and the tuple window applied in the `from` clause, first get the last 5 orders for each customer, and then compute the sum of those orders' values. The `having` clause then suppresses the output of customer IDs when the sum of the values is below the threshold of 3000.

Query 6 (Counting)

```
select custId, count(*)
from FailedLoginAttempt.win:time(5 min)
group by custId
having count(*) >= 3
```

We need to count failed login attempts for the same account, not over all accounts, so we have to use `group by`. The `having` clause then filters all accounts with fewer than three failed login attempts.

When using time windows, `std:groupby` and `group by` yield the same results, while yielding different results when a tuple window is applied afterwards. This is because `group by` is applied after the windows in the `from` clause, while `std:groupby` can be placed everywhere. Of course, this query can also be written with `std:groupby`, but it is harder to read. This difference is also present in STREAM with the `Partition By` window construct compared to the `Group By` clause.

Note that time windows in Esper are managed actively, so this query is also executed whenever an event leaves the window. In this case, this might lead to unwanted additional answers when four attempts are in the window and one leaves. When using the data stream constructs, this is not avoidable.

Query 7 (Aggregation)

```
insert into SmoothedIndex
select avg(price)
from IdxTick.win:time(30 sec)
```

The first query in the stock market example can be implemented easily and concisely.

Query 8 (Event instance selection)

```
insert into StkLastHour (price)
select prev(count(*)-1, price)
```

```
from StkTick.win:time(1 hour)

insert into SmoothedIndexLastHour (price)
select prev(count(*)-1, price)
from SmoothedIndex.win:time(1 hour)

select true
from StkLastHour.win:length(1) as sd,
  SmoothedIndexLastHour.win:length(1) as id,
  StkTick.win.length(1) as s,
  SmoothedIndex.win:length(1) as i
where (sd.price / s.price) - (id.price / i.price) >= 0.02
```

While event instance selection is present in Esper, it is somewhat unintuitive; this functionality is hidden in the `prev` function that actually retrieves data from a previous event. Recall that a query is evaluated every time an event comes in. `prev(1, price)` would then retrieve the `price` data field of the previous event in the stream. `prev(2, price)` would retrieve the `price` data field of the event before the previous in the stream.

To select the first tick in the last hour, we need to create a new stream that lags behind the real stream by one hour. This is done by continuously selecting the first event in the window using `prev(count(*)-1, price)` (since `count(*)` returns the number of events in the window, going back by `count(*)-1` events returns the first). Once we have the four streams we need, we choose the respective last event in each stream (using a tuple window) and extract the price.

Query 9 (Sequences)

```
insert into V (minimum)
select f.lower
from pattern [every f=Fall -> (not Fall) -> Rise]
```

This query requires a direct sequence, that is, a fall must be directly followed by a rise and may not have another fall in between. Since the `->` operator, the sequence operator, does not have this restriction, we want the pattern to stop (thus returning nothing) if after a fall was detected, another fall occurs before a rise. The `every` keyword ensures that for every fall, this pattern is evaluated: a new instance of the pattern is then created (like opening a lifespan in AMiT, or creating a new detector tree in ruleCore).

This works because of the semantics of the operators used. When we consider a pattern `A -> B`, the engine first looks for an `A`, and then looks for a `B`. This means that in our query, the fall used to match the first subpattern (`Fall`) is not used for the second subpattern (`not Fall`).

When considering the pattern `not A -> B`, the engine looks for a `B`, as long as no `A` is found. If an `A` is found before a `B` is found, evaluation of the current pattern instance is terminated.

Query 10 (Event instance consumption)

```
select true
from pattern [every v1=V -> V(lower / v1.lower >= 0.99,
  lower / v1.lower <= 1.01) where timer:within(1 hour)]
```

It is possible to use data in patterns; this is done with a so-called filter expression in Esper. This allows us to not match V patterns with a following V pattern whose minimum is out of range.

By omitting the `every` operator after the `->`, we avoid the unnecessary matches mentioned in Figure 8. Our pattern has the form `every A -> A(<condition>)`; every time it encounters an `A` event, it looks for the next (and only the next) `A` event that satisfies the condition. This contrasts with `every A -> every A(<condition>)`: Such a pattern would, every time it encounters an `A`, match for every following `A` satisfying the condition.

While this technique is used to implement event instance consumption in this case, it can only cover a subset of event instance consumption. For example, it is not possible to remove an event for other queries. In the Esper manual, the concept of event instance consumption is not mentioned at all.

Ease of Use and Implementation Status

Esper is free software, designed for purposes beyond research, including business. As such, it includes a comprehensive manual [Esp08] that allowed the language to be learned easily and also doubling as useful reference material.

Esper consists of some Java libraries. These are well documented and therefore, its API was easy to use. We used them for syntax-checking our queries.

In general, Esper was the most pleasant system to use, due to its good documentation and easy usable implementation. Its queries were easy to express after learning about both data stream and composition-operator-based languages. Even the event instance selection features could be found by taking a look into the documentation, although they were implemented in an unexpected way (through the `prev` function that returns older events).

Implementations (engines): Esper (Java) and NEsper (.NET)

Evaluated version (release date): Esper 2.3.0 (November 2008)

Web site: `http://esper.codehaus.org/`

4.7 Cayuga

Cayuga is a research CEP engine developed at Cornell University. It sets itself apart from other engines (like the ones above) in that it deliberately sacrifices expressivity for performance, targeting applications running large numbers of queries. It is free software, available under the terms of the BSD license.

Cayuga uses an Event Query Language called Cayuga Event Language (CEL). While its syntax resembles SQL, like many data stream languages, it also offers patterns, although using a different approach compared to Esper, inspired by regular expressions. It offers these operators to compute streams from other streams:

- The `filter` operator has two arguments: a stream and a condition. Its result is a stream containing only the events from the input stream fulfilling the condition.
- The `union` operator, inspired by the | (or) operator in regular expressions, returns all events from both input streams (similar to disjunction).

- The `next` operator, inspired by concatenation in regular expressions, works like a conditional sequence. It has three arguments: two streams and a condition. For every event in the first stream, it is joined with the temporally next event in the second stream that meets the condition. Note that while determining the event immediately following an event is easy if the occurrence time is a time point, it is not so easy with time intervals [WRGD07]. Let s spanning the time interval $[s_0, s_1]$ be the event in the first stream. t spanning the time interval $[t_0, t_1]$ is the next event in the second stream according to Cayuga's `next` operator's semantics if $s_1 < t_0$, t fulfills the specified condition, and there is no event r spanning the time interval $[r_0, r_1]$ such that r fulfills the specified condition and $s_1 < r_0 \leq r_1 < t_1$.
- The `fold` operator, inspired by the + operator in regular expressions, works like applying `next` repeatedly until a stop condition is fulfilled. It has five arguments: two streams, a relevance condition, a stopping condition and zero or more expressions. For every event in the first stream, for every temporally following event in the second stream satisfying the relevance condition, the expression is evaluated. Processing is then continued unless the stopping condition is true. The expression, resembling executable code in that it can assign new values to variables, often is some kind of accumulation. Example:

```
(select *, 0 as count from Subscription)
  fold{$1.sourceId = $2.sourceId,count < 5,
    $.value + $2.value as value, $.count + 1 as count}
B
```

  In this example, someone subscribes to some data source (that are identified by IDs), such as a sensor. All data sources publish on a stream called `B`; each event in this stream carries the ID of its source. This stream is computed as follows: for every subscription to a data source, it sums up the value of the next five temporally following events of the requested data source (`B` events whose source ID matches the subscription's source ID). In this example, the left stream is `Subscription`, the right stream is `B`, the relevance condition is `$1.sourceId = $2.sourceId`, the stopping condition is `count < 5`, and the expression is `$.value + $2.value as value, $.count + 1 as count`. These arguments can access special variables: `$1` denotes the event of the left stream, `$2` denotes the current event of the right stream, `$` denotes the previous event of the right stream (often used to store variables). In this example, evaluation of the expression would add the value of the current `B` to the value accumulator, and increments the counter of `B`s processed. The variable `count` is introduced with its initial value as an added attribute to the `Subscription` event.

Our evaluation is based on a paper describing the semantics [DGH+06] and one describing the concrete syntax [DGP07], as well as the examples on the home page.

Query 1 (Disjunction)

```
select custId
from Login
```

```
publish LoginCustId

select custId
from Order
publish OrderCustId

select custId
from PageRequest
publish PageRequestCustId

select custId
from LoginCustId union OrderCustId union PageRequestCustId
publish CustomerActivity
```

The first query resembles a SQL query, but uses the composition operator `union` instead of joining. In fact, joining as in SQL (`from A, B`) is not possible, as there is only one stream allowed in the `from` clause (but it may be the result from a composition operator). Since `union` requires all streams to have the same schema, the first three subqueries unify the schemas, and then the disjunction happens. The `publish` clause names the output stream.

Query 2 (Negation, time windows)

This query is not expressible. The `fold` construct can be used to aggregate, including counting of events, but `fold` requires at least one event in both input streams to start processing, thus yielding answers. One input stream has to be the customer activity stream. In case it contains no events, we could not detect the non-existence of events using `fold`; it would not start counting. `fold` is the only operator capable of counting. So this query cannot be expressed, even if we use timer streams.

Time windows are demonstrated in the next query.

Query 3 (Conjunction, data extraction)

```
select *
from Order next{$1.orderId = $2.orderId}(Delivery)
publish OrderAndDelivery

select *
from CustomerOrder next{$1.orderId = $2.orderId}(Payment)
publish OrderAndPayment

select *
from OrderAndDelivery next{$1.orderId_1 = $2.orderId}(Payment)
publish OrderAndDeliveryAndPayment

select *
from OrderAndPayment next{$1.orderId_1 = $2.orderId}(Delivery)
publish OrderAndPaymentAndDelivery

select *
from OrderAndDeliveryAndPayment
```

```
  union OrderAndPaymentAndDelivery)
publish Trade
```

Interestingly, while Cayuga has two sequence operators, it does not have a conjunction operator. The conjunction can be expressed using two sequences; one for a payment followed by a delivery, and one for a delivery followed by a payment. This covers all cases in our example, as orders always precede the other events.

The 60-seconds limit to improve efficiency can be added by using `filter`, demonstrated here for the first subquery:

```
select *
from filter{dur <= 60 seconds} (Order
  next{$1.orderId = $2.orderId}(Delivery))
publish OrderAndDelivery
```

Time windows are expressed by applying a `filter`. The special `dur` variable contains the time a complex event, constructed by `next` in this example, spans.

Query 4 (Using external data sources)

No information regarding accessing external data sources was found in the papers.

Query 5 (Tuple windows, aggregation by group)

```
select custId, count
from filter{volume >= 3000} ((select *, 1 as count from Order)
  fold{$2.custId = $1.custId,count < 5,
    $.volume + $2.volume as volume,
    $.count + 1 as count}
  (Order))
```

Using `fold`, whenever we encounter an order, we search the following 5 orders for those with the same customer account ID, counting them and summing up their values. The stopping condition checks the `count` variable; it is set to the number of orders considered, starting with 1, as the order from the left stream is considered, too. Processing stops when we considered 5 orders.

The stream created by `fold` is then processed by a `filter`, removing all sequences of orders by the same customer having a value smaller than 3000.

Query 6 (Counting)

```
select custId, count
from filter{dur = 5 minutes and count >= 3} ((select *, 1 as count
  from FailedLoginAttempt) fold{$2.custId = $1.custId,,
  $.count + 1 as count} (FailedLoginAttempt))
```

Using `fold`, whenever we encounter a failed login attempt, we search the following failed login attempts in the next 5 minutes for those with the same customer account ID, counting

them.

Query 7 (Aggregation)

```
select price / count as averagePrice
from filter{dur = 30 seconds} ((select *, 1 as count from IdxTick)
  fold{,,$.count+1 as count,
  $.price + $2.price as price}(IdxTick))
publish SmoothedIndex
```

We aggregate using the `fold` operator. Each event in the window established by `filter` is counted and its price added to aggregator variables. These are then used to compute the average. This is similar to the above query, except that even conditions on the events are non-present: all events in the stream are relevant.

Query 8 (Event instance selection)

No information was found regarding event instance selection in both papers. We did not find a way to obtain events one hour before or later. Also, Cayuga lacks delete streams, so we can not approximate the query like in STREAM.

Query 9 (Sequences)

This query is not fully expressible.

```
select minimum
from (select lower as minimum from Fall) next (Rise)
publish V
```

The only operators in Cayuga capable of expressing sequences are `next` and `fold`. The latter is, however, mainly a repeating version of `next`, whose additional functionality is not needed. So if the query would be expressible, it would use `next`.

The above query combines every fall with the next rise, hereby also obtaining the minimum of the V pattern. While it would find all correct V patterns, it would yield additional, wrong answers: if a fall is followed by another fall, the first fall may actually not be used as an answer to this query according to the specification. For example, wenn encountering the sequence F_1, F_2, R, R is the next rise for F_1, however, we cannot see that there is F_2 in between. We did not find a way express this restriction.

Query 10 (Event instance consumption)

```
select *
from filter{dur <= 1 hour} (V
  next{$2.lower / $1.lower >= 0.99 and $2.lower / $1.lower <= 1.01}
  (V))
```

No information was found regarding event instance consumption in both papers, but this query is still fully expressible using the `next` operator. This query finds every W; the wrong

Ws those V patterns enclose another V pattern with a similar minimum are excluded, because the `next` operator stops at the first join partner.

Ease of Use and Implementation Status

Cayuga is the product of a research project. It does not come with a manual, so two papers [DGH+06, DGP07] were used for learning the language instead. Overall, they contained all information we needed, although we sometimes had to look into the semantics to fully understand certain operators.

While an implementation is freely available, we failed in building it. When calling `make`, the compiler rejected the code, as the compiler version on our Ubuntu 8.10 system (GCC 4.3) is stricter than previous versions[15]. This behavior could not be changed by adding the `-fpermissive` flag. We considered the amount of work to fix this to be too large, as it meant adding lines to many files of the source code. This is the first error message (lines wrapped, replaced some characters not present in ASCII):

```
../../../extlib/inc/antlr/CharScanner.hpp: In member function
  'bool antlr::CharScannerLiteralsLess::operator()(
    const std::string&, const std::string&) const':
../../../extlib/inc/antlr/CharScanner.hpp:565: error:
  'strcasecmp' was not declared in this scope
```

Overall, Cayuga's operators are different from those found in the other languages. This required some time when expressing the queries, especially when it seemed that a query is not expressible. Understanding the operators was not too difficult, although taking a look into the semantics was sometimes needed.

Implementations (engines): Cayuga

Evaluated version (release date): Version of April 29, 2008.

Web site: `http://www.cs.cornell.edu/bigreddata/cayuga/`

4.8 Drools

Drools, also known as JBoss rules, is a production-quality business rule management system, including a production rule engine. It is free software, released under the Apache License.

The Drools engine is implemented in Java, as is JBoss, and is also controlled using that language. Initialization of the engine and deployment of rules is implemented in Java. Also, as unusual for production rule engines, rules never fire by themselves, but are issued to do so by the Java program that controls the engine.

In addition, Drools can be extended by defining so-called Domain Specific Languages. These are languages that may have a different syntax than the standard Drools syntax to write queries in. Rules in Domain Specific Languages are then translated into the Drools language when inserted into the engine. This eases the usage of Drools by non-technical staff, for example, by creating abstraction barriers. Domain Specific Languages are not used in our examples.

[15] `http://gcc.gnu.org/gcc-4.3/porting_to.html`

Drools rules have the following simple structure:

```
when
  ...
then
  ...
```

In the `when` part, fact patterns are listed. Fact patterns are similar to simple event queries, looking for facts of a certain type. In the `when` part, conditions can also be expressed, such as conditions on the data of a fact or between multiple facts. Rules become active if (and only if) facts on the `when` part are new or changed, as rules become active only when their condition *becomes* true, not every time they *are* true.

In the `then` part, the actions are specified, such as the insertion or retraction of facts or a Java method call. They are executed if the rule is fired. Only active rules can be fired.

In the `when` part, there may be variable assignments that can be used in both `when` and `then` parts. Through variables, we can obtain various objects, such as numbers and strings from data in facts, but also facts themselves. We will precede variables with `$`, as recommended in the manual, although it is not strictly required.

The evaluation of this language is based on its documentation [Dro]. The following queries are always written in the Java dialect (rules can also be written in the MVEL dialect). Since this is a production rule language, thus not primarily intended to do CEP, all event types get an additional integer attribute, `timestamp`, set upon insertion of the event into the engine. The `timestamp` is assumed to denote seconds, so that an event with timestamp 0 occurred one minute before an event with timestamp 60.

Also, we assume that the Java application uses the engine in an instance of a class called `Controller`. Since rules are not fired unless ordered to do so by `Controller`, we assume that all rules are fired when `Controller` inserts events, and also assume that `Controller` has a static `fireAllRules()` method that fires all rules. This is needed to fire all rules when we create an event inside a rule, allowing the new event to be immediately processed by other queries.

Query 1 (Disjunction)

```
when
  $o : Login($t : timestamp)
then
  insert(new CustomerActivity($t));
  Controller.fireAllRules();

when
  $o : Order($t : timestamp)
then
  insert(new CustomerActivity($t));
  Controller.fireAllRules();

when
  $o : PageRequest($t : timestamp)
```

```
then
  insert(new CustomerActivity($t));
  Controller.fireAllRules();
```

A stream in Drools is simulated by a working memory containing all events. So, since we want a customer activity event whenever one of three events happen, we just check for the presence of these events and then insert a new customer activity event. The new customer activity event has the same timestamp has the actual login, order or page request event; it is captured by binding the timestamp to a variable (using `:`) and using this variable on the rule's right-hand side.

Query 2 (Negation, time windows)

This query is only expressible using low-level code.

```
when
  Timer($t : timestamp)
  not(exists(CustomerActivity(timestamp <= $t
    && timestamp >= ($t-300))))
then
  insert(new NoCustomerActivity($t));
  Controller.fireAllRules();
```

While negation is not a problem for Drools, the time window has to be coded into the rule. In this query, the time window is not started by any event. We therefore, like in other language, need to create an artificial timer stream that is guaranteed to regularly containing events.

Time is simulated by introducting a new event type, `Timer`, with an integer attribute denoting the current time. We insert `Timer(0)` into new working memories, and start a daemon thread in the Java program, incrementing the time by one every second[16]:

```
final Timer timer = new Timer(0);
final FactHandle timer_h = workingMemory.insert( timer );

Thread timeThread = new Thread() {public void run() {
    while(true) {
        try {
            timer.setTimestamp(timer.getTimestamp() + 1);
            workingMemory.update(timer_h, timer);
            workingMemory.fireAllRules();
            Thread.sleep(1000);
        } catch(InterruptedException e) {
            // Nothing
        }
    }
    }
};

timeThread.setDaemon(true);
timeThread.start();
```

[16] As in other languages, this can be done less frequent for efficiency reasons.

The `Timer` event has the current time as its data. We can bind the time to a variable and check whether there is a customer activity whose timestamp is between now and 5 minutes before; if there is no such activity, we insert a `NoCustomerActivity`.

As the `Timer` object is modified every second, this rule is activated every second. It is also executed every second because of the Java code in the user program calling `fireAllRules()` whenever the time is updated. As a consequence, this query might answer quite often, but we are not concerned about that.

Note that on the left-hand side, there is an implicit `and` operator applied to all subconditions, the presence of the `Timer` and the nonpresence of the `CustomerActivity` in this case.

Query 3 (Conjunction, data extraction)

```
when
  $o : Order($ot : timestamp, $oid : orderId)
  $d : Delivery($dt : timestamp, $did : orderId)
  $p : Payment($pt : timestamp, $pid : orderId)
  eval($oid == $did && $oid == $pid)
then
  insert(new Trade(Math.max(Math.max($ot, $dt), $pt), $o, $d, $p));
  Controller.fireAllRules();
```

As in the query before, all lines of the left-hand side are combined by an implicit `and`. A new construct used is the `eval` function, a special function evaluating its expression that is Java code that can contain variables bound by the query, but also Java expressions such as method calls. Note that facts are nothing more than references to Java objects, so we can use them on constructing new events. The timestamp of the new event is the largest timestamp of its three member events, calculated by Java code embedded in the rule.

Like in other languages before, this implementation of this query will exhaust the machine's memory over time, as the orders, deliveries and payments are never removed. If one of these components of a trade is missing, then the other components will also never get irrelevant for this query, as the missing part might appear in the future. We can impose the 60-seconds limit we allowed in the specification by adding restrictions on the timestamps:

```
when
  $o : Order($ot : timestamp, $oid : orderId)
  $d : Delivery($dt : timestamp, timestamp < ($ot+60),
    $did : orderId)
  $p : Payment($pt : timestamp, timestamp < ($ot+60),
    $pid : orderId)
  eval($oid == $did && $oid == $pid)
then
  insert(new Trade(Math.max(Math.max($ot, $dt), $pt), $o, $d, $p));
  Controller.fireAllRules();
```

Components not able to be combined to a trade can be retracted from the memory. However, this has to be done using another rule, or Java code, but not automatically by the engine. As with all user-created auxiliary code, it would be more prone to bugs and efficiency issues than a built-in facility.

In addition, the removal of an event by retraction would remove it from all rules, even from rules with larger time windows the programmer might not be aware of.

On a sidenote, Drools allows objects to be logically inserted, that means, if the `when` part of the rule instance that inserted it becomes false again, the object is automatically removed again. In some cases, this can be used for garbage collection.

Query 4 (Using external data sources)

This query is only expressible using low-level code.

```
when
  $o : Order($t : timestamp, $c : custId)
  PlatinumCustomer(custId == $c)
    from Controller.obtainPlatinumCustomers()
then
  insert(new PlatinumOrder($t, $o));
  Controller.fireAllRules();
```

While the database is not accessed directly using the Drools language, it can call a custom Java method returning a collection of platinum customers retrieved from a database, done with the `from` keyword. This collection can then be accessed as it would be part of the working memory.

More general Java methods taking a SQL query as their argument and just executing it are conceivable, but still require to use considerable amounts of the Java database API, thus counting as low-level code. This is mainly due to the fact that a SQL query in Java returns a `ResultSet`, and has first to be processed to become a collection of the desired type.

Query 5 (Tuple windows, aggregation by group)

This query is easier to express using low-level code. Although it can be expressed without it, low-level code removes serious drawbacks.

As we have seen, there are no built-in time windows, and tuple windows are not present, either. To make things more difficult, tuple windows are also harder to express when not calling Java methods, if their size is not 1. To find the 5 newest orders of a certain customer on a given (possibly non-empty) working memory, we would have to search the whole working memory.

Unfortunately, this would not work with the aggregation functions, all of which require the use of the `accumulate` construct. This takes four types of input: a pattern (simple filtering by type and on data), an expression with `from` (allows input from various sources, including Java methods and accumulations), an expression with `collect`, or another aggregation. However, none of these are applicable without using a Java method returning the last 5 orders of a given customer, especially since the latter two types of input have the same restrictions as `accumulate`.

So, in the general case, we assume we have such a Java method `lastFiveOrders()` being present in the `Customer` class, in which case the query looks like this:

```
when
```

```
  Order($t : timestamp, $cid : custId)
  $c : Customer(custId == $cid)
  Number(intValue >= 3000) from accumulate(Order($value : value)
    from $c.lastFiveOrders(), sum($value))
then
  insert(new BigOrderSequence($t, $cid));
  Controller.fireAllRules();
```

Every time an order comes in, the Java method is called, retrieving the last five orders of the customer. Using `accumulate`, we can sum up the value of these five orders and check whether it is large enough.

In case we start with an empty working memory, one might think of a solution that does not use custom Java code: one keeps a list of the last orders for every customer. Whenever an order comes in, it is updated. This is one of the rules used in this solution:

```
when
  $o : Order($t : timestamp, $cid : custId)
  $c : Customer(custId == $cid)
  $m : Map()
  $l : Queue() from $m.get($c)
  eval($l.size() == 5)
then
  $l.remove();
  $l.add($o);
```

Unfortunately, even though not using custom Java code outside the query, this has some drawbacks. First, there are several typing issues. We would have to assume that there is only one `Map` in the working memory, especially none from other queries; we cannot mark a `Map` to belong to this query, and we cannot keep a handle. Second, if the number of customers increases over time, there would be a order list for every customer. This would cost memory, even if customers become inactive. As a result, we favor the first solution.

This query demonstrates the tight coupling of Drools with its implementation language. Facts are objects, allowing some methods to be called and their results to be used. Also, `accumulate` can process Java collections.

Query 6 (Counting)

```
when
  FailedLoginAttempt($t : timestamp, $cid : custId)
  Customer(custId == $cid)
  $l : ArrayList(size >= 3) from collect(FailedLoginAttempt(
    custId == $cid && timestamp <= $t
    && timestamp >= ($t-300)))
then
  insert(new IntrusionAttempt($t, $cid));
  Controller.fireAllRules();
```

The `collect` construct collects all events fulfilling the conditions given as its argument as a `java.util.Collection` or compatible subclasses, an `ArrayList` in this case. The resulting

collection can then be used as an event in the working memory; in our case, we apply a condition.

The number of `IntrusionAttempts` inserted by this query is quite high if there are many failed login attempts inside five minutes. This is because we reuse all events in that time window. In order to limit reuse, we could retract some of these events, for example the first three. However, this would remove them from other queries, too.

Query 7 (Aggregation)

```
when
  IdxTick($t : timestamp)
  $a : Number() from accumulate(IdxTick(timestamp <= $t,
    timestamp >= ($t-30), $price : price), average($price))
then
  insert(new SmoothedIndex($t, $a));
  Controller.fireAllRules();
```

This query computes the average with each tick. We apply the `accumulate` element that returns the average value over the selected ticks (a time window spanning the last 30 seconds). In the end, the sliding average object is updated by creating a new event.

Query 8 (Event instance selection)

```
when
  StkTick($t : timestamp, $s : price)
  not(StkTick(timestamp > $t))
  SmoothedIndex($idx_t : timestamp, $i : price, timestamp <= $t,
    timestamp >= ($t-3600))
  not(SmoothedIndex(timestamp <= $t, timestamp > $idx_t))
  StkTick($stkd_t : timestamp, $sd : price, timestamp <= $t,
    timestamp >= ($t-3600))
  not(StkTick(timestamp < $stkd_t, timestamp >= ($t-3600)))
  SmoothedIndex($idxd_t : timestamp, $id : price, timestamp <= $t,
    timestamp >= ($t-3600))
  not(SmoothedIndex(timestamp < $idxd_t, timestamp >= ($t-3600)))
  eval(($sd / $s) - ($id / $i) >= 0.02)
then
  insert(new Outperformance($t));
  Controller.fireAllRules();
```

The nature of production rule languages allows us to select events even when the conditions are not simple. As in the above queries, we have to define the time window manually. For selection of the last event in the window, we use for the `IDX` tick:

```
  SmoothedIndex($idx_t : timestamp, $i : price, timestamp <= $t,
    timestamp >= ($t-3600))
  not(SmoothedIndex(timestamp <= $t, timestamp > $idx_t))
```

The first line ensures that the event is in the window. The second line ensures that it is actually the last event in the window. Selecting the first event in the window is very similar:

```
  SmoothedIndex($idxd_t : timestamp, $id : price, timestamp <= $t,
```

```
    timestamp >= ($t-3600))
  not(SmoothedIndex(timestamp < $idxd_t, timestamp >= ($t-3600)))
```

As with the no customer activity query, this query has a tendency to answer multiple times if it answers, as if a stock outperforms, it will likely outperform at the next tick, too. This behavior does not violate the query specification, though.

Query 9 (Sequences)

```
when
  $f : Fall($f_t : timestamp, $l : lower)
  $r : Rise($r_t : timestamp)
  not(Fall(timestamp > $f_t, timestamp < $r_t))
  not(Rise(timestamp > $f_t, timestamp < $r_t))
  eval($f_t < $r_t)
then
  insert(new V($r_t, $l));
  Controller.fireAllRules();
```

This query can be expressed quite easily and resembles pattern matching. We take some `Fall` and some `Rise` and check whether there are other `Fall`s or `Rise`s in between; if this is not the case, it is a V pattern.

Query 10 (Event instance consumption)

```
when
  $v : V($v_t : timestamp, $l : lower)
  $v1 : V($v1_t : timestamp, timestamp > $v_t, lower >= (0.99*$l),
    lower <= (1.01*$l), timestamp >= ($v_t-3600))
  not(V(timestamp < $v1_t, timestamp > $v_t,
    lower >= (0.99*$l), lower <= (1.01*$l)))
then
  insert(new W($v1_t, $v, $v1));
  retract($v);
  Controller.fireAllRules();
```

Since it is a core feature of production rule languages to remove facts from their working memory, event instance consumption can be easily expressed. The pattern matching resembles the previous query, only with the conditions on the minima, and time window boilerplate code added. Since we retract the first V, this query surely does not yield any additional, wrong, answer.

Note that this consumption also affects other queries. In order to invalidate events for certain queries only, we would either have to mark the used facts using variables in the facts. Likely, such variables are not present, though. Another solution would be to keep a list of processed events, although one should keep this list from growing indefinitely.

In this case, the query would even work without the retraction, as it works the same as in some composition-operator-based languages. The retraction was added to demonstrate

event instance consumption.

Ease of Use and Implementation Status

Drools is free software. Like Esper, it is developed for productive use. As such, it comes with a comprehensive manual [Dro]. However, as Drools was not designed for CEP, we had to find out ourselves how to use it for this. In particular, no example in the manual dealt directly with events, that is, facts with timestamps.

Drools is found as a Java library that integrated into the Eclipse development environment. It was useful for testing some aspects of Drools, as well as syntax-checking all queries.

Overall, the main difficulty in using Drools for CEP was not Drools itself, but how to implement CEP features such as time and garbage collecting. As these have to be implemented by the user, this is Drools's biggest weakness.

Implementations (engines): Drools

Evaluated version (release date): Drools 4.0.7 (May 2008)

Web site: `http://www.jboss.org/drools/`

4.9 XChange^{EQ} (plus XChange)

XChange^{EQ} is a research Event Query Language. It is developed at the University of Munich and designed for automated reasoning on the Semantic Web. XChange^{EQ} introduces a new style of event querying. It separates event query features into four so-called dimensions: Data extraction, event composition, temporal relationships, and event accumulation. Most operators belong to exactly one of these dimensions. This was done to define clear semantics.

As XChange^{EQ} is designed for use on the Web, it works best at processing tree-structured events, such as XML messages. Queries are generally structured like the XML representations of the events queried. For querying simple events, it embeds the Xcerpt language [Sch04]. Xcerpt queries apply patterns to XML documents, similar to templates. A Xcerpt query starts with the name of the event type, for example, `login`, possibly followed by conditions on data fields or binding of data to variables. The following query obtains the number of a flight, but only if it starts from Munich (else, the query is not successful):

```
flight {{
  number { var N },
  from { "MUC" }
}}
```

The braces specify how the query should react to data fields found in the event, but not in the query. In XML documents, a node's children are ordered. When using square brackets `[ ]`, children in the event must appear in the order they appear in the query, while this is not required when using curly braces `{ }`. Also, when using double braces (`[[ ]]` or `{{ }}`), other children not mentioned in the query are allowed, else, the node must have exactly the specified children.

XChange^{EQ} complex queries have the following structure:

```
DETECT
```

```
  ...
ON
  ...
END
```

The `DETECT` clause specifies what is to be done when an event detected by the query in the `ON` clause is found. The `ON` clause contains Xcerpt queries, possibly composed using composition operators and possibly also having additional conditions applied.

XChange^{EQ} can be used in the context of XChange. XChange is a reactive programming language. Using Event-Condition-Action rules, it allows Web sites to react to changes at other Web sites, for example by updating its own data. This is done by exchanging XML messages; for example, a Web site might notify others about a content change; the other Web sites could then update their own content, possibly sending messages themselves, propagating changes automatically through the Web. Event-Condition-Action rules in XChange have the form of *Event query – Web query*[17] *– Action*. In order to accomplish these tasks, XChange can use XChange^{EQ} as its CEP facility.

We evaluate this system, that is, XChange [BEP06a, BEP06b] using XChange^{EQ} [Eck08, BE08, BE07, BE06] as its Event Query Language.

Query 1 (Disjunction)

```
DETECT
  customeractivity {}
ON
  or {
    login {{ }}, order {{ }}, pagerequest {{ }}
  }
END
```

This query consists of three Xcerpt queries combined with the `or` operator, allowing a concise query. The data in the events is irrelevant, so we use the double braces.

Query 2 (Negation, time windows)

```
DETECT
  nocustomeractivity {}
ON
  and {
    event now: timer:datetime {{ millisecond { 0 } }},
    event window: timer:extend-begin[now, 5 minutes],
    while window: not customeractivity {{ }}
  }
END
```

While XChange^{EQ} has sophisticated functions to extend or shorten time intervals, it does not have built-in windows starting at the current time point and extending in the past. So, every second, we obtain an internal timer event (a time point), and obtain the desired time

[17] Web queries are explained in Query 4 (Using external data sources).

interval by moving its start 5 minutes into the past. Finally, we check whether there is a customer activity in the interval.

Recall that double curly braces allow other data fields to be present in the event. Actually, `datatime` events have many fields such as `year`, `millisecond`, and `day-of-week`; by specifying only the millisecond, the pattern matches any time with 0 milliseconds, so it matches one event per second. As in other languages, we can execute this query less frequently to sacrifice precision for efficiency.

Query 3 (Conjunction, data extraction)

```
DETECT
  trade {
    tradeId { var I }
  }
ON
  and {
    order {{ orderId { var I } }},
    payment {{ orderId { var I } }},
    delivery {{ orderId { var I } }}
  }
END
```

This is much like the disjunction query, only with a conjunction and also using data. Resembling logical formulas, using the same variable name in the conjunction expresses the condition that order, payment and delivery must have the same order ID. The order ID is bound to the variable `I` that is used in the construction of the trade, thus obtaining the order ID from the other events. As in other languages, this implementation of this query is likely to exhaust the machine's resources, as some events here may never get irrelevant. We can easily impose the 60-seconds limit allowed by the specification by adding a condition:

```
DETECT
  trade {
    tradeId { var I }
  }
ON
  and {
    event o : order {{ orderId { var I } }},
    event p : payment {{ orderId { var I } }},
    event d : delivery {{ orderId { var I } }}
  }
  where { {o,p,d} within 1 minute }
END
```

Query 4 (Using external data sources)

```
DO
  platinumorder { orderId { var I } }
ON
  order {{ orderId { var I }, custId { var C } }}
FROM
  in { resource { "http://company.com/database" },
```

```
  customer {{
    custId { var C },
    platinum { "true" }
  }}
END
```

Using XChange, we can access Web resources such as XML data on the Web. This is done using a Web query, also written in Xcerpt, in the `FROM` clause. Since Xcerpt is also used for querying simple events, the syntax does not differ, only a parameter for `in`, specifying the source, has to be added. However, note that the `DO` clause is a *reaction* instead of a newly-constructed event (as usual for XChange). As such, further XChange^{EQ} queries treat it not as an event.

At the moment, XChange^{EQ} is not able to express this query by itself. However, future work [Eck08, Chap. 18.3.2] may enable this. The proposal, though not yet implemented, allows external data sources to be used similar to the way this has to be done with XChange in the moment:

```
DETECT
  platinumorder { orderId { var I } }
ON
  and {
    event o : order {{ orderId { var I }, custId { var C } }},
    at o : in { resource { "http://company.com/database" }
      customer {{
        custId { var C },
        platinum { "true" }
      }}
  }
END
```

Every time an order comes in, the resource is queried (`at o`). The resulting `platinumorder` would be a complex event and could be processed by another query.

Query 5 (Tuple windows, aggregation by group)

This query is partly expressible by coding tuple windows into the query.

```
DETECT
  OrderWindow { }
ON
  and {
    event o1 : Order {{ custId { var C } }},
    event o2 : Order {{ custId { var C } }},
  } where { o1 before o2 }
END

DETECT
  OrderSet {
    all entry {
      custId { var C },
      orderCount { count(all var V) },
      sumValue { sum(all var V) }
```

```
    } group-by var C
  }
ON
  event w : OrderWindow {{ }} {
  while w : collect Order {
      custId { var C },
      value { var V }
    }
  }
END

DETECT
  GoodCustomer {
    custId { var C }
  }
ON
  OrderSet {{
    custId { var C },
    orderCount { var N },
    sumValue { var V }
  }}
  where { var V >= 3000, var N = 5 }
END
```

There is no tuple window construct. For aggregation, we need a time window spanning the time in which the events are collected. In order to do this, we take two distinct[18] orders from the same customer and construct a complex event out of these. This event can be used in the `while` construct as a time window spanning these events.[19] We collect all orders in this time window and count them, as well as taking the sum of the values. If the number of events in the window is exactly 5, we assure that we got the five last orders of a customer (and no more). Also, we filter out any order sequences whose average value is under the threshold.

This works like a tuple window spanning the 5 most recent orders. XChange^{EQ} processes the first subquery as follows: When a new order comes in, it builds an `OrderWindow` event using this new order and any other event that can satisfy all conditions. In particular, no older 5-order sequences, that means, no order sequences without the latest order, are used, as the latest order is always considered. However, this implementation of this query is also very inefficient, as for every order, there is one window and thus one aggregation for every past order of the same customer. If a customer ordered 20 times (including the newest), there will be 20-1 = 19 aggregations, regardless of whether they are relevant. If a customer ordered 50 times, there will be 50-1 = 49 aggregations.

Using negation for such queries (in the sense of "we consider five orders which have no other orders of this customer in between") is currently not possible in XChange^{EQ}. This is

[18] We take two orders of this customer and require one of them to be before the other, thus, this requires them to be distinct.

[19] It is currently not possible to build a time window whose boundaries are events from the same rule. It is needed to construct a complex event from these events and using the new complex event for the time window.

because in the current XChange^{EQ} version, a window defined by two events always contains the two boundary events. Thus, a statement that no events of a certain type are in a window is always false if a boundary has that type—which is the case in this query. A proposed new feature allows windows not containing one or even both of the boundaries. This would allow using negation for this query:

```
  ...
  and {
    event window : OrderSequence {{ custId { var C } }}
    while(excl,excl) window : not Order {{ custId { var C } }}
  }
  ...
```

Query 6 (Counting)

```
DETECT
  FailedLoginAttemptSet {
    all entry {
      custId { var C },
      attemptCount { count(all var A) }
    } group-by var C
  }
ON
  and {
    event a : FailedLoginAttempt {{ custId { var C } }},
    event window : timer:extend-begin[t, 5 minutes],
    while window : collect FailedLoginAttempt {
      custId { var C },
      flaId { var A }
    }
  }
END

DETECT
  IntrusionAttempt {
    custId { var C }
  }
ON
  FailedLoginAttemptSet {{
    custId { var C },
    attemptCount { var A }
  }}
  where { var A >= 3 }
END
```

The first step is to collect failed login attempts over the last five minutes. Every time a failed login attempt is encountered, we construct a time window spanning the time from now (including the encountered attempt) to five minutes earlier and collect all failed login attempts in this time window. Aggregation always happens in the head of the rule, in this case grouping the found events by customer ID and counting them.

We then need a second rule to apply the condition that we raise an alarm only if we found at least three attempts in the last 5 minutes.

Query 7 (Aggregation)

```
DETECT
  SmoothedIndex {
    all entry {
      price { avg(all var P) }
    }
  }
ON
  and {
    event tick : IdxTick {{ }},
    event window : timer:extend-begin[tick, 30 seconds],
    while window : collect IdxTick {
      price { var P }
    }
  }
END
```

For each IdxTick, we construct a time window starting from and including this tick and ending 30 seconds before the tick. We then compute the average from all ticks falling into this window, as in the previous queries.

Query 8 (Event instance selection)

This query is not expressible. However, future features were suggested that would be used to code event instance selection into the query.

Suppose we build an 1-hour window at every stock tick (we name it `S`). When selecting the first stock tick (or index tick, does not matter in this context) in that window, we cannot use event instance selection, as these features are not present. We can, however, just require the presence of a stock tick (we name it `SD`) and add the condition that there is no stock tick between the beginning of the window and the stock tick. However, it is not possible to obtain this window, as we can only build windows using two events, or if we know the size in advance.

In case there are stock ticks before the time window, a "dirty" trick is possible: We take one of these stock ticks (we name it `SDD`). We add the condition the time window between `SDD` and `SD` overlaps the other window (the hour before `S`). Also, we require that there is no stock tick in the time window `SDD` and `SD`. The latter condition is not expressible in the current version of XChange^{EQ}, as the boundary events are always included in a window, but a suggested future feature allows that (as described in query 5).

The overlapping condition is slightly dangerous. If `SD` would be exactly one hour before `S`, it would still be part of the window "one hour before `S`". However, the window `SDD-SD` would *not* overlap the other window according to the definition of overlapping in XChange^{EQ}, because its end point is not after the start point of the other window. As a hypothetical alternative solution, if we could create windows by intersecting other windows (that is also not possible), we could obtain the window between the candidate event and the boundary

of the window "1 hour before S" quite easily without such subtle problems.

Selecting the last smoothed index tick can be done by obtaining the window between a smoothed index tick and S and requiring that there is no smoothed index tick inside that window—again using the currently non-present feature of considering a boundary event not to be in a window.

Query 9 (Sequences)

This query is not fully expressible.

```
DETECT
  V {
    lower { var L }
  }
ON
  and {
    event f : Fall {{ }},
    event r : {{ }},
  }
  where { f before r }
END
```

While many temporal relationships can be expressed in XChange^{EQ}, it does not have a version of the sequence operator that can express this query. This query would combine any fall with and rise, as long as the rise is after the fall. It can not be expressed that the rise has to be the very next event after the fall. Negation and counting can not be used, as the window, defined by a fall and a rise, always contains a fall—the boundary. As described in query 5, it cannot be excluded in the current version, but the language extension was already proposed. The additional query would look like this:

```
DETECT
  PossibleV {
    lower { var L }
  }
ON
  and {
    event f : Fall {{ }},
    event r : {{ }},
  }
  where { f before r }
END

DETECT
  V {
    lower { var L }
  }
ON
  and {
    event window : PossibleV {{ lower { var L } }}
    while(excl,incl) window : not Fall {{ }}
```

```
    while(incl,excl) window : not Rise {{ }}
  }
END
```

Query 10 (Event instance consumption)

```
DETECT
  W { }
ON
  and {
    event v1 : V {{ lower { var L1 } }}
    event v : V {{ lower { var L } }}
  }
  where { v1 before v, var L1 / var L >= 0.99,
    var L1 / var L <= 1.01, {v1, v} within 1 hour
  }
END
```

XChange^{EQ} does not have event instance consumption features. As in some queries above, the fact that windows always include their boundaries causes trouble when trying to implement this query. Essentially, the not expressible part of this query is similar to the previous query; it will match a V with any following V with a similar minimum, as long as both events are within one hour (note the use of the `within` construct; it can express the condition that all events in the set must happen within the specified amount of time).

Ease of Use and Implementation Status

XChange^{EQ} is a product of a research project. It is documented in several papers [BE08, BE07, BE06], but most comprehensively in the PhD thesis [Eck08]. Especially the latter was useful for acquiring knowledge about the knowledge, due to the length of its descriptions.

While XChange^{EQ} has an implementation, several of the features are missing, most notably aggregation of data. This is because it leverages the Xcerpt language, but at the time of writing, only supports a subset of the language.

XChange^{EQ} queries resemble logical formulas, and as such, the language principles were easy to learn, we have a background in logic. However, some language features, such as tuple windows, are absent and had to be implemented using other means, sometimes resulting in long and unintuitive code.

Implementations (engines): XChange^{EQ}

Evaluated version (release date): XChange^{EQ} 0.9.0 (unknown date)

Web site: See Chapter 16 in Michael Eckert's PhD thesis [Eck08].

5 Conclusion

In this final section, we compare the languages' capabilities regarding query features, as well as evaluating all languages as a whole technology, and giving an outlook on what might come in the future.

5.1 Results Put Together

Table 1 sums up how well each language was able to express the example queries. Language groups are abbreviated: DS = data stream language, CO = composition-operator-based language, PR = production rule language. In the table, we assess whether the language is capable of expressing the individual queries fully, partially, or not at all. We use the following symbols to denote levels of expressibility:

Symbol	Meaning
+	Fully expressible using desired features
⊕	Partially expressible using desired features (desired features insufficiently present)
⊖	Fully or partially expressible using other features (desired features not present, or requires not generally applicable "tricks" such as additional streams or low-level coding to work, or insufficient documentation)
−	Not expressible

Language	Group	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
STREAM	DS	+	⊖	+	⊖	+	+	+	⊖	⊖	⊖
Borealis	DS	+	−	+	⊖	+	+	+	−	⊖	−
AMiT	CO	+	⊖	+	−	−	+	−	+	+	+
ruleCore	CO	+	⊕	+	−	−	⊕	−	⊖	⊕	⊕
SASE+	CO	+	⊖	+	−	+	+	+	−	+	⊖
Esper	DS, CO	+	+	+	+	+	+	+	+	+	⊖
Cayuga	DS, CO	+	−	+	−	+	+	+	−	−	⊖
Drools	PR	+	⊖	+	⊖	⊖	+	+	+	+	+
XChange^{EQ}	Other	+	+	+	⊕	⊖	+	+	−	⊕	−

Table 1: Expressibility of each query by language.

As we can see from Table 1, there is no language capable of expressing all example queries without restrictions yet. Each language only supports a subset of the features listed in Section 2.3.

The queries can be roughly divided into three groups. Queries 1-3 deal with composing simple event queries to a complex event query using the disjunction, negation, and conjunction operators. Queries 4-7 mostly deal with processing the data found in the events, be it

enriching them with relational data, or aggregating them. Queries 8-10 are used to detect patterns in streams.

As we can see, some languages perform well in some groups, while performing less well in other groups. In some cases, it can even be said that some language groups favor certain query groups and have trouble with others. We take a look at the details and reasons by group.

Queries 1-3: Composition of Simple Event Queries

These queries are expressible in most languages. The disjunction query caused no trouble. Conjunction proved difficult in Borealis and Cayuga, where we had to build it using multiple sequence operators, as they lack a conjunction. Also, in Borealis, we were forced to apply a window with a fixed finite length. This means that the query is only expressible in Borealis if a limit on the temporal distance between order, delivery, and payment is set in advance.

The implementation of negation proved to be hard or not possible at all in some languages. In composition-operator-based languages, this is sometimes an operator that only needs to be applied. However, data stream languages often have a hard time with negation. As queries there are executed whenever an event enters the input stream, queries monitoring only the stream where we try to find absence of events never get executed. In some languages, we could use a built-in timer stream, or, if such a stream is not present, work around with a new timer stream from outside; in Borealis and Cayuga, not even this was possible.

Query 2 also features time windows. These are sliding time windows, so these queries have to be executed regularily. ruleCore's built-in timer is a bit unwieldy and is not capable of generating timer events more than once per minute. In Drools, we had to implement a large portion of the time mechanism ourselves: temporal conditions are conditions on the added timestamp fields in that language.

Queries 4-7: Processing Event Data

From the languages' perspective, query 4 was probably the hardest query. 4 languages did not support accessing non-event data at all. It is barely documented in Borealis. STREAM, Esper, and Drools can access data using the API of their implementation languages; the latter two can accept databases queries from event queries, while STREAM needs all database queries in the C++ code controlling the engine. XChange^{EQ} supports access to resources if it is deployed in a reactive web environment it (and XChange) was designed for. However, at this time, external resources cannot be used to create complex events; the type of output is different.

When looking at Table 1, we see that data stream languages perform very well when aggregating event data, while composition-operator-based languages do either not support it at all or poorly. SASE+ is an exception, due to the operations possible on Kleene closures that can be used similar to arrays in that language. Composition-operator-based languages have this property mainly because of their history. Their origin lies in Active Database Systems, where they are found in the E part of ECA (Event-Condition-Action) rules, while data is usually accessed in the C part.

On a side note, Drools and XChange^{EQ} completely lack tuple windows, while these are often part of data stream languages. Overall, both languages performed quite well in this query group.

Queries 8-10: Event Pattern Detection

In this group, data stream languages perform less well, while composition-operator-based groups are at an advantage. AMiT even expresses all of these queries easily.

The sequence operator found in queries 9 and 10 is often a basic composition operator. On the other hand, it is not present at all in STREAM and Borealis. Implementing the sequence by hand needed some changes in the environment in these languages.

Event instance selection and event instance consumption seem to be minor features, as most languages do not support these concepts. Both can be implemented using other means to an extent, although remember the fact that our example queries are kept simple; not any event instance selection/consumption example can be approximated in the way we did.

XChange^{EQ} does not claim to support event instance selection/consumption. While having sophisticated time-related facilities, it was not capable of expressing many restrictions appearing in these queries. Drools performed quite well in this group. In that language, the programmer can (and must) control the addition and removal of events from the working memory manually.

5.2 Comparison of Ease of Use and Implementations

Whether a language is easy to use is depending on several factors: Quality of the documentation, familiarity of the language's concepts, and how well the implementation assists the programmer. In this section, we sum up our personal opinion from the language evaluation, where we gave our opinion for each language separately.

The languages we selected have a wide range of purposes. Only three of them (ruleCore, Esper, Drools) were supposed to be used in production environments; the rest mainly serves research purposes. Note that the version of ruleCore that is freely available can also be considered a research prototype, as it seems to be abandoned since 2004.

The documentation tends to be better with systems designed for productive use. Esper and Drools have well-written, extensive documentation spanning hundreds of pages. On the other hand, research languages might only be documented in research papers of less than ten pages and with a different purpose, that is, explanation of the design for other scientists, not for end users. Most research languages we considered did not even fully mention the available syntax, with Borealis and XChange^{EQ} being notable exceptions.

However, in research languages, the language semantics are often given in full or are well described. Only ruleCore was hard to follow. In Drools, we had to test some things to truly understand the language (for example, whether and how rules activate if they make their own condition true again), while in Esper, the explanations of the semantics are often part of operator descriptions. Available semantics were usually very helpful. The negation query, for example, was usually easier to express if the language specified what exactly triggers the

execution of a query. In the negation query, we saw that all languages execute queries only if an event enters the system in a stream that is mentioned in the query. Most languages execute queries whenever an event enters, while CQL, the language of STREAM, tends to collect tuples with the same timestamp before execution.

We were familiar with SQL already, so STREAM's concept of windowing was easy to follow. Borealis's boxes, while being a new concept, were easy to understand, although the high number of required parameters for each box were confusing in the beginning. Composition operators, while being a simple concept, often come with subtle difficulties when trying to understand what they do after they have matched—for example, when it detected a sequence of three failed login attempts, what happens if a fourth comes in.[20] In addition, each composition-operator-based language in this survey came with its own approach (Lifespans, detector trees, added concepts from data stream languages, or completely different operators). The lifespan concept from AMiT needed the most reading in its design documents, particularly to understand why events actually need lifespans, and how events are reused in a lifespan. ruleCore's detector trees, while simple to understand, came with subtleties such as whether multiple leaf nodes can pick up the same event. Overall, poor documentation made the open-source version of ruleCore quite hard to use.

SASE+ has easy concepts, but some unfamiliar syntax such as the usage of `[attrib]` to require that events agree on an attribute, and the way it filters out event groups that are too large (for example, `a.LEN < 5`). Esper was easy to understand after we understood the languages before; it has a clear concept of combining composition operators with the data stream constructs. For Cayuga, while the operators have their semantics included in the documentation, they were also unfamiliar and posed challenges when expressing queries using only these operators. With Drools, the hardest aspect to learn was when a condition becomes true (especially that it does not become true for modified facts unless notified), otherwise, it was easy to learn. We think that XChange^{EQ} is more accessible if the learner has a background in logic, although it should not be too difficult to learn either. Note that we read a smaller survey before ([Eck08], Chapter 3), thus we had some knowledge about the general ideas of the language groups.

Some languages have more concise syntax than others. Borealis, AMiT and ruleCore use a very verbose XML syntax that is unpleasant to use unless using tools. All of these languages also share the fact that there are plenty of options, most of which have to be set, so the queries are less abstract than in the other languages, making them hard to use and to debug. The other languages, however, usually do not allow queries to be fine-tuned as in the three mentioned languages.

With implementations, quality varied greatly. Of course, research languages tend to have poorer implementations or none at all. Some implementations (Borealis, ruleCore, Cayuga) were no longer working after newer compilers broke backward compatibility, or after libraries were replaced with newer versions. These implementations therefore suffer from poor maintenance. Other implementations (STREAM, XChange^{EQ}) sometimes come with limitations on the usable language features, so that certain queries can not be used in the engine although they are part of the language specification. STREAM also suffered from

[20] Understanding in which ways older events are reused indeed was an issue with most languages, not only composition-operator-based languages.

poor error messages, consisting only of undocumented error numbers, which was not helpful in finding syntax errors in queries. Esper and Drools both provide an Eclipse plugin and were easy to install and use.

5.3 Beyond this Survey – Other Approaches and Features

Even the languages we evaluated had features that are useful, but did not appear in our queries. Also, some languages we did not evaluate have features that might become more common in the future. We describe some of the noteworthy ones.

Basket expressions

Recent research [KLG08] proposed another approach to Complex Event Processing. In this approach, streams are collections of events where events can be inserted *and* removed. These are called *baskets*. Baskets are not saved on storage devices; they are only found in the main memory. Built on top of SQL, CEP is done by only adding so-called *basket expressions*.

A basket expression queries database tables and another baskets, removing the events found from its source and inserting them into the destination basket. Streams from outside the engine, as well as streams leading outside, are modelled as baskets. Basket expressions have added syntax to distinguish them from standard SQL queries. A query is considered a continuous query iff it contains a basket expression.

One of the main advantages of this approach is that existing SQL engines can be used, instead of starting an engine from scratch. The existing functionality can be fully retained, while it is extended with CEP capabilities.

Exception Handling in Queries

Avaya EP [ava] is a data stream language. Its queries have the following structure:

```
ON ...
  WHEN ...
  THEN ...
  CATCH ...
```

The `CATCH` clause allow exception handling inside the query, thus inside the language. In other languages, exceptions generally have to be handled outside the language using low-level code. The `CATCH` clause can use any definition appearing in the `WHEN` and `THEN` clauses, as well as the exception thrown by the (Java) engine.

Expiry Time Windows

Avaya EP also allows sliding windows that detect events exiting the window even if no event enters the window. We have seen in the negation query that data stream languages usually need a stream different than the monitored one to detect the absence of events. When using expiry time windows instead, a different, although slower, algorithm is used.

However, queries using expiry time windows will also be evaluated when an event exits the window. In our example, if there are no customer activities for some time, the absence is detected once the last customer activity leaves the window. In essence, using expiry time windows, one can concisely express queries that are to be executed both when an event enters the stream *and* when an events exits the window. This is shorter than using two similar subqueries, one of which using an insert stream to cover incoming events and one of which using a delete stream to react to events exiting the window.

The advantage of using expiry time windows compared to the timer stream solution is that the mechanism does not have to be programmed by the user, and can have enhancements by the language providers. Note that Esper's time windows are expiry time windows by default and does not offer other modes.

Batch Windows

Batch windows are time or tuple windows found in Esper and Avaya EP, among other languages. These are sliding windows whose move behavior can be specified. When using standard time windows starting at the current time point, for example, at each evaluation, a window starting at that time point with the specified length is applied. Essentially, the movement of the window is specified by the engine.

Batch windows are windows that move only in steps specified by the user. For example, a batch window with size 1 day will select all events of the current day, regardless of the time of day the query is evaluated. In contrast, a standard time window would select all events of the last 24 hours, possibly including events of the previous day. In applications, batch windows can be used to process events inside time units, such as the current day, where data from the previous time unit is not desired.

In Esper and Avaya EP, the length of the window and the movement interval are the same, although use cases for windows where these are not the same might be conceivable.

5.4 The Real World – Present and Future

As we have seen, no language could fully express all our 10 queries (although Esper came very close). However, there were also only few problems some languages could not solve even with some help.

We have also shown that there are language groups, each with its own strengths and weaknesses. The language groups have long existed with few or no knowledge of the other language groups. This is due to their origins, being created in four different communities. [Luc07a] Only recently, language designers became aware of the language groups as different approaches to similar problems, incorporating results from multiple research communities.

In industry, we currently see the dominance of data stream languages extended with composition operators, created by companies such as Oracle, Coral8, StreamBase, to name a few. Indeed, these hybrid languages, represented by Esper and Cayuga, fared well in our evaluation, combining the power of data stream languages when processing event data with the power of composition-operator-based languages when detecting patterns in event streams.

Still, as we have seen, we can expect to see even more powerful languages to appear in the future. As Complex Event Processing is likely to become an important technology, some diversity might also occur. Cayuga is an example of a language trading expressivity for efficiency. In the future, more languages, including some from the industry, might distinguish themselves from others in aspects such as these.

References

[ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical report, VLDB Journal, 2003.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, 2006.

[Adi06] Asaf Adi. IBM Active Middleware Technology Overview, 2006. Whitepaper, Online only (`http://www.haifa.il.ibm.com/dept/services/papers/cep_Jan07_nc.pdf`).

[AE04] Asaf Adi and Opher Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, May 2004.

[Ae05] José J. Alferes and Wolfgang May (ed.). Specification of a Model, Language and Architecture for Reactivity and Evolution. Technical Report IST506779/Lisbon/I5-D4/D/PU/a1, REWERSE, 2005.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, November 1983.

[ava] Avaya home page. `http://www.avaya.com/`.

[BE06] François Bry and Michael Eckert. A High-Level Query Language for Events. In *SCW '06: Proceedings of the IEEE Services Computing Workshops*, pages 31–38, Washington, DC, USA, 2006. IEEE Computer Society.

[BE07] François Bry and Michael Eckert. Rule-Based Composite Event Queries: The Language XChange EQ and its Semantics. In *Proceedings of the International Conference on Web Reasoning and Rule Systems*. Springer, 2007.

[BE08] François Bry and Michael Eckert. On static determination of temporal relevance for incremental evaluation of complex event queries. In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, pages 289–300, New York, NY, USA, 2008. ACM.

[bea] Bea (Oracle) Complex Event Processing home page. `http://www.oracle.com/technologies/soa/complex-event-processing.html`.

[BEP06a] François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Querying Composite Events for Reactivity on the Web. In *LNCS 3842*, pages 38–47, 2006.

[BEP06b] François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. *Journal of Web Engineering*, 5(1):3–24, 2006.

[bor] Borealis Application Guide. `http://www.cs.brown.edu/research/borealis/public/publications/borealis_application_guide.pdf`.

[cay] Cayuga home page. `http://www.cs.cornell.edu/bigreddata/cayuga/`.

[cor] Coral8 home page. `http://www.coral8.com/`.

[DGH+06] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards Expressive Publish/Subscribe Systems. In *LNCS 3896*, pages 627–644, 2006.

[DGP07] Alan Demers, Johannes Gehrke, and Biswanath P. Cayuga: A general purpose event monitoring system. In *In CIDR*, pages 412–422, 2007.

[DIG07] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. Sase+: An agile language for kleene closure over event streams, 2007.

[Dro] JBoss Rules User Guide. `http://www.jboss.org/drools/documentation.html`.

[Eck08] Michael Eckert. *Complex Event Processing with XChange^{EQ}: Language Design, Formal Semantics, and Incremental Evaluation for Querying Events.* PhD thesis, University of Munich, 2008.

[Esp08] EsperTech Inc. Esper Reference Manual. Online only (`http://esper.codehaus.org/esper/documentation/documentation.html`), 2008.

[For82] Charles. L. Forgy. Rete A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[GAE06] Thanaa M. Ghanem, Walid G. Aref, and Ahmed K. Elmagarmid. Exploiting predicate-window semantics over data streams. *SIGMOD Record*, 35(1):3–8, 2006.

[IBM] IBM. What is IBM Active Middleware Technology? Online only (`http://www.haifa.il.ibm.com/dept/services/soms_ebs_tech.html`).

[JMS+08] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a Streaming SQL Standard. *Proceedings of the VLDB Endowment*, 1(2):1379–1390, 2008.

[KLG08] Martin L. Kersten, Erietta Liarou, and Romulo A. Goncalves. A Query Language For A Data Refinery Cell. 2008.

[LS08] David Luckham and Roy Schulte. Event Processing Glossary - Version 1.1. Online only (`http://complexevents.com/wp-content/uploads/2008/08/epts-glossary-v11.pdf`), 2008.

[Luc01] David Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Luc07a] David Luckham. A Short History of Complex Event Processing, Part 1: Beginnings. Online only (`http://complexevents.com/wp-content/uploads/2008/02/1-a-short-history-of-cep-part-1.pdf`), 2007.

[Luc07b] David Luckham. A Short History of Complex Event Processing, Part 2: the rise of CEP. Online only (`http://complexevents.com/wp-content/uploads/2008/07/2-final-a-short-history-of-cep-part-2.pdf`), 2007.

[Luc08] David Luckham. A Brief Overview of the Concepts of CEP. Online only (`http://complexevents.com/wp-content/uploads/2008/07/overview-of-concepts-of-cep.pdf`), 2008.

[ope] OpenESB home page. `http://open-esb.dev.java.net/`.

[SB05] Marco Seiriö and Mikael Berndtsson. Design and Implementation of an ECA Rule Markup Language. *LNCS*, 3791:98–112, 2005.

[Sch04] Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, University of Munich, 2004.

[sqr] STREAM Query Repository. `http://infolab.stanford.edu/stream/sqr/`.

[stra] STREAM: The Stanford Stream Data Manager. User Guide and Design Document. `http://www-db.stanford.edu/stream/code/user.pdf`.

[strb] StreamBase home page. `http://streambase.com/`.

[tel] TelegraphCQ home page. `http://telegraph.cs.berkeley.edu/`.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418, New York, NY, USA, 2006. ACM.

[WRGD07] Walker White, Mirek Riedewald, Johannes Gehrke, and Alan Demers. What is "next" in event processing? In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 263–272, New York, NY, USA, 2007. ACM.

[WSC] WebSphere Message Broker CEP Detector Nodes. `ftp://ftp.software.ibm.com/software/integration/support/supportpacs/individual/ia0s_v6.1.pdf`.