A High-Level Query Language for Events

François Bry
University of Munich, Institute for Informatics
http://www.pms.ifi.lmu.de, bry@pms.ifi.lmu.de

Michael Eckert
University of Munich, Institute for Informatics
http://www.pms.ifi.lmu.de, eckert@pms.ifi.lmu.de

Abstract

Nowadays events are omnipresent and exchanged as messages over networks. Characteristic for applications involving advanced (or complex) event processing is the need to (1) utilize data contained in the events, (2) detect patterns made up of multiple events (so-called composite events), (3) reason about temporal and causal relationships of events, (4) accumulate events for negation and data aggregation.

This article describes a high-level language approach for expressing queries to events. Its foundations are: embedding of a query language for XML and other Web data formats, support for rule-based reasoning, and a complete coverage of the four dimensions mentioned above.

1. Introduction

In distributed computer systems, events are omnipresent and exchanged as messages over networks. Business applications use events to interact with other applications or human users. Increasing business demands and advances in technology make the current techniques for processing events insufficient. Amongst others, interest in complex event processing (CEP) is driven by:

- a need to understand the dynamic behavior in distributed large-scale information systems [20],
- increased generation of events from sensors due to drastic cost reductions in technology (e.g., RFID),
- a need to monitor log data generated in computer systems (e.g., fraud detection in credit card transaction logs)
- service-oriented architecture (e.g., accounting for on-demand services, synchronization of activities in business processes, monitoring of service level agreements),
- the emergence of event-driven architecture (e.g., detect and react to advantageous or dangerous situations) [12].

For many emerging applications it is not sufficient anymore to query and react to only single, atomic events (i.e., events signified by a single message). Instead, events have to be considered with their relationship to other events in a stream of events. Such events (or situations) that do not consist of one single atomic event but have to be inferred from some pattern of several events are called *composite events*.

Composite events as patterns of events do not exist explicity "by themselves" in an event stream (as single messages). Rather they are implicit and the patterns are conveniently specified by event queries.

Querying events has much in common with querying database data (using languages such as SQL or XQuery). In particular, events usually contain data and the messages used to transmit them are in a conventional data format such as XML; a key example of this are XML-based SOAP messages [18] used to communicate with Web Services. Processing of such message events comprises querying data in events and constructing new events or data.

There are, however, also important differences between querying database data and events:

- Events are received over time in a stream-like manner, while in a database all facts are available at once and stored on disk.
- Event streams are unbounded, potentially infinite, into the future, whereas databases are finite. This has especially consequences for non-monotonic query features such as negation or aggregation.
- Relationships between events such as temporal order or causality play an important role for querying events. In databases, relationships between facts are usually part of the data (e.g., functional dependencies)
- Timing of answers has to be considered when querying events: event queries are evaluated continously against the event stream and generate answers at different times. These answers may trigger actions such as updates to a database. Typically actions are sensitive to ordering; hence it is important *when* an answer to an event query is detected.
- Query evaluation and optimization for event streams requires different methods than for databases. In event streams a large number of (standing) queries are evaluated against small pieces of incoming data (events). Evaluation is thus usually data-driven, rather than query-driven. Optimization relies on exploiting similarities between queries rather than indexing data.

This article describes work in progress on a high-level event query language, focusing in particular on the language design. Our language is well-suited for use in enterprise applications built upon Web Service standards. In particular, it

supports events that are received as SOAP (or other XML) messages. It can be deployed as a stand-alone event mediation component in an event-driven architecture [12], as event component of the framework for reactivity and evolution on the Web described in [21] and [22], and as part (sub-language) of the reactive Web language XChange [4, 8].[1]

Our new approach to an event query language is motivated by previous work on XChange, a language employing Event-Condition-Action rules to program distributed, reactive Web applications and services. XChange includes composite event query capabilities [8, 7] that are similar to those found in active databases [17, 16, 11, 10, 1]. Our experience in XChange has taught us that there is a considerable gap between the requirements posed by applications and the expressivity of current event query languages.

There are (at least) the following four complementary dimensions that need to be considered in designing a sufficiently expressive event query language:

Data extraction: Events contain data that is relevant to whether and how to react. For events that are received as SOAP messages (or in other XML formats), the data can be structured quite complex (semi-structured). The data of events must be extracted and provided (typically as bindings for variables) to construct new events or trigger reactions (e.g., database updates).

Event composition: To support composite events, i.e., events that are made up out of several events, event queries must support composition constructs such as the conjunction and disjunction of events (or more precisely of event queries, since composite events can only be defined through composite event queries).

Temporal (and causal) relationships: Time plays an important role in event-driven applications. Event queries must be able to express temporal conditions such as "events A and B happen within 1 hour, and A happens before B." For some applications, it is also interesting to look at causal relationships, e.g., to express queries such as "events A and B happen, and A has caused B." (In this article we concentrate only on temporal relationships, causality is future work)

Event accumulation: Event queries must be able to accumulate events to support non-monotonic features such as negation (absence) of events, aggregation of data, or repetitive events. The reason for this is that the event stream is (in contrast to extensional data in a database) infinite; one therefore has to define a scope (e.g., a time interval) over which events are accumulated when aggregating data or querying the absence of events. Application examples where event accumulation is required are manifold. A business activity monitoring application might watch out for situations where "a customer's order has *not* been fulfilled within 2 days" (negation). A stock market application might require notification if "the *average* of the reported stock prices over the last hour raises by 5%" (aggregation).

[1] The new event query language described in this article is considered as a replacement of the previous composite event query language [7] of XChange.

The foundations of the high-level event query language described in this article are:

- It embeds the XML query language Xcerpt to specify classes of relevant events, extract data (in the form of variable bindings) from them, and construct new events.
- It supports rules as an abstraction and reasoning mechanism, for the same reasons and with the same benefits of views in traditional database systems.
- Its syntax enforces a separation of the four querying dimensions described above, yielding a clear language design, making queries easy to read and understand, and giving programmers the benefit of a separation of concerns. Even more importantly, this separation allows to argue that the language reaches a certain degree of expressive completeness.

Our language is in the combination of these foundations quite different from previous composite event query languages. We improve on previous work on composite event query languages in the following ways: It is a high-level language with a clear design that is easy to use and provides the appropriate abstractions for querying events. It emphasizes the necessity to query data in events, in particular semi-structured XML messages. Being targeted towards SOAP and Web Services, it is particularly suitable for use in business applications domains. We make an attempt towards expressive completeness by completely covering all four query dimensions explained earlier. Finally, with the separation of concerns, we hope to avoid misinterpretations of queries, as they can happen easily with other languages (see the discussions in [27, 15, 1]).

Using the example of a stock market application, we introduce our event query language incrementally, starting from queries to single (atomic) events (Section 2). We add complexity and expressivity with deductive and reactive rules for events (Section 3), the composition of (several) events (Section 4), temporal conditions on events (Section 5), and event accumulation (Section 6). We end with a short discussion of semantics and evaluation methods (Section 7), and conclusions and future work (Section 8).

2. Querying Atomic Events

Application level events in distributed enterprise systems are nowadays often represented as XML messages, especially as SOAP messages. Where this is not the case (e.g., sensor events in proprietary data formats), events can still conveniently be represented and queried as XML.[2]

In our stock market example we will be using four atomic events: stock buys, stock sells, and orders to buy or sell stocks. Involved applications could also generate further events; these would not affect our example queries. The left side of Figure 1 depicts the buy and buy order events in XML

[2] It is not necessary to actually materialize the XML representation for every incoming non-XML event, of course: instead queries can be translated for this non-XML format, much like one can rewrite queries to database views instead of materializing views.

```
<order>
  <orderId>4711</orderId>
  <customer>John</customer>
  <buy>
    <stock>IBM</stock>
    <limit>3.14</limit>
    <volume>40S00</volume>
  </buy>
</order>
```

```
order [
  orderId { 4711 },
  customer { "John" },
  buy [
    stock { "IBM" },
    limit { 3.14 },
    volume { 4000 }
  ]
]
```

```
<buy>
  <orderId>4711</orderId>
  <tradeId>4242</tradeId>
  <customer>John</customer>
  <stock>IBM</stock>
  <price>2.71</price>
  <volume>4000</volume>
</buy>
```

```
buy [
  orderId { 4711 },
  tradeId { 4242 },
  customer { "John" },
  stock { "IBM" },
  price { 2.71 },
  volume { 4000 }
]
```

Fig. 1. XML and term representation of an event

```
buy {{
  tradeId { var I },
  customer { var C },
  stock { var S },
  price { var P },
  volume { var V }
}}
where { var P * var V >= 10000 }
```

Fig. 2. Atomic event query

(sell and sell order are analogous). Details such as SOAP envelopes are skipped in this article.

For conciseness and human readability, we use a "term syntax" for data, queries, and construction of data. The right side of Figure 1 depicts the XML events as (data) terms. The term syntax is slightly more general than XML, indicating whether the order of children is relevant (square brackets `[]`), or not (curly braces {}).

Querying such single event messages is a two-fold task: one has to (1) specify a class of relevant events (e.g., all buy events) and (2) extract data from the events (e.g., the price). We embed the XML query language Xcerpt to both specify classes of relevant events and extract data from the events. Figure 2 shows an event query that recognizes buy events with a price total of $\$10\,000$ or more.

Xcerpt queries describe a pattern that is matched against the data. Query terms can be partial (indicated by double brackets or braces), meaning that a matching data term can contain subterms not specified in the query, or total (indicated by single brackets or braces). The queries can contain variables (keyword `var`), which will be bound to the matching data, and a `where`-clause can be attached to specify non-structural (e.g., arithmetic) conditions.

In this article, we will stick to simple queries as above. Note however that Xcerpt supports more advanced features such as subterm negation, optional subterm specification, subterms at arbitrary depth, and queries to graph-shaped data such as RDF [14]. An introduction to Xcerpt is given in [25].

The result of evaluating an Xcerpt query on an event message is the set Σ of all possible substitutions for the free variables in the query (non-matching is signified by $\Sigma = \emptyset$). Our example query does not match the order event from Figure 1, but matches the buy event with $\Sigma = \{\sigma_1\}$, $\sigma_1 = \{I \mapsto \texttt{4242}, C \mapsto \texttt{John}, S \mapsto \texttt{IBM}, P \mapsto \texttt{2.71}, V \mapsto \texttt{4000}\}$.

```
timer:datetime {{
    date { "2006-09-18" },
    time { "9:00" }
}}
```

```
catts:tradingDay {{
  dayOfWeek { var D }
}}
```

Fig. 3. Queries for absolute timer events

Xcerpt has the following advantages over other XML query languages such as XQuery [5] for querying events: (1) The notion of an event matching or not matching a query gives a straight-forward notion for defining classes of relevant events using queries. (2) Unlike XQuery, Xcerpt has a clear separation of querying (selecting) data and constructing new data. So far, we have only used the query-part of Xcerpt. (3) Due to the separation, Xcerpt provides clear semantics for the variable bindings using the concept of substitution sets; this is especially convenient when composing events. In comparison, in XQuery a variable takes different values at different steps (e.g., iterations of a FOR-loop).

In addition to event messages, event queries can query for timer events. Absolute timer events are time points or intervals defined without reference to the occurrence time of some other event. They are specified just like queries to event messages. The left of Figure 3 shows a query for a given time and date. Restrictions apply on the structure and on the places where variables can be used. For example, it is not allowed to skip `time` in the `datetime` event query or use a variable inside it — otherwise the timer would fire continously. However, one may skip the `date` specification, yielding a periodic event.

Various calendar systems such as CaTTS [9] can be used to define application-specific calendar events that are more complex than the simple `datetime` event. An example is the query for trading days on the right of Figure 3. This timer event stretches over a whole time interval of a trading day (e.g., 9am to 5pm); the event is detected at the end of the interval.

Relative timer events, i.e., time points or intervals defined in relation to some other event will be looked at in Section 4 on event composition.

3. Reactive and Deductive Rules for Events

Our language uses two kinds of rules: deductive rules and reactive rules. Deductive rules allow to define new, "virtual" events from the events that are received. They have no side effects and are analogous to the definition of views for database data. The arguments why rules for events are required are basically the same as why views are required in databases: (1) Rules serve as an abstraction mechanism, making query programs more readable (especially important when event queries get long, as they do for SOAP messages). (2) Rules allow to define higher-level application events from lower-level (e.g., database or network) events. (3) Different rules can pro-

```
DETECT bigbuy {
         tradeId { var I },
         customer { var C },
         stock { var S }  }
ON buy {{
     tradeId { var I },
     customer { var C },
     stock { var S },
     price { var P },
     volume { var V }
   }} where { var P * var V >= 10000 }
END
```

Fig. 4. Deductive rule

```
RAISE to(recipient="http://auditor.com",
         transport="http://.../soap/bindings/HTTP/"){
        var B
      }
ON var B -> bigbuy {{ }}
END
```

Fig. 5. Reactive rule

vide different perspectives (e.g., of end-user, system administrator, corporate management) on the same (event-driven) system. (4) Rules allow to mediate between different schemas for event data.

Additionally, rules can be beneficial when reasoning about causal relationships of events [20].

Figure 4 shows a deductive rule deriving a new event "bigbuy" from buy events satisfying the earlier event query of Figure 2. Deductive rules follow the syntax `DETECT` *event construction* `ON` *event query* `END`. The event construction in the rule head is simply a data term augmented with variables which will be replaced by their values obtained from evaluating the event query in the rule body. The event construction is also called a construct term; more involved construction will be seen in Section 6 when we look at aggregation of data.

At present we do not allow recursive rules and rule sets; relaxations of this restriction (e.g., stratified rule sets) are a matter for future work.

Reactive rules are used for specifying a reaction to the occurrence of an event. The usual (re)action is constructing a new event message (as with deductive rules) and use it to call some Web Service. For tasks involving accessing and updating persistent data, the event queries can be used in the Event-Condition-Action rules of the reactive language XChange.

An example for a reactive rule is in Figure 5; it forwards every bigbuy event (as derived in Figure 4) to a Web Service `http://auditor.com` using SOAP's HTTP transport binding. The syntax for reactive rules is similar to deductive rules, only they start with the keyword `RAISE`, and in the rule head `to()` is used together with `recipient` and `transport` to specify *where* the message goes and *how*. Alternatively to `to()`, addressing information can be specified in the header of a SOAP message using WS-Addressing [6].

The distinction between deductive and reactive rules is important. While it is possible to "abuse" reactive rules to simulate deductive rules (by sending oneself the result), this is undesirable: it is misleading for programmers, less efficient in the evaluation, and by allowing recursion and cycles risky.

```
DETECT fees {
         customer { var C },
         amount { 0.01 * var P * var V }  }
ON or {
     buy {{
       customer { var C },
       price { var P },
       volume { var V }  }},
     sell {{
       customer { var C },
       price { var P },
       volume { var V }  }}  }
END
```

Fig. 6. Disjunction of event queries

4. Composition of Events

So far, we have only been looking at queries to single events. Since temporal conditions are dealt with separately, only two operators, `or` and `and`, are necessary to compose event queries into *composite event queries*. (Negation falls under event accumulation, see Section 6.) Both composition operators are multi-ary, allowing to compose any (positive) number of event queries, and written in prefix notation.

When event queries are composed with `or`, every answer to one of the constituent queries is also an answer to the composite query. The rule in Figure 6 gives an example: every time it recognizes a buy or sell event, it generates a new event signaling the fees (1% of the total), the customer has to pay for the transaction.

Disjunctions are not strictly necessary: Instead of one rule $a \leftarrow b \vee c$, one could simply write two rules $a \leftarrow b$ and $a \leftarrow c$. As the example shows however, they are quite convenient and avoid redundancy.

Conjunctions on the other hand do increase the expressivity. When two event queries are composed with `and`, an answer to the composite event query is generated for every pair of answers to the constituent queries. If the constituent queries share free variables, only pairs with "compatible" variable assignments are considered.

Figure 7 illustrates the use of the `and` operator. The "orderfulfilled" event is detected for every corresponding pair of buy order and buy event as well as for every corresponding pair of sell order and sell event. The events have to agree on variables O (the `orderId`) and T (which is bound to an XML element name being either `buy` or `sell`). The occurrence time of the detected "orderfulfilled" event is (by default) the time interval enclosing the respective constituent events.

Event queries composed with `or` and `and` can be nested. Since the operators are associative, the multi-ary generalizations are obvious. For example, `and{a, b, c}` can be understood as `and{and{a, b}, c}`.

Composition of events gives rise to defining relative timer events, i.e., time points or intervals defined in relation to the occurrence time of some other event. Figure 8 shows a composite event query querying for an order event and a timer covering the whole time interval between the order event and one minute after the order event. We will see this timer event be used later in Section 6 when querying for the absence of a

```
DETECT orderfulfilled {
         orderId { var O },
         tradeId { var I },
         stock { var S },
         type { var T } }
ON and {
     order {
       orderId { var O },
       var T {{
         stock { var S } }} },
     var T {{
       orderId { var O },
       tradeId { var I } }} }
END
```

Fig. 7. Conjunction of event queries

```
and {
  event o: order {{ orderId { var O } }},
  event t: extend[event o, 1 min] }
```

Fig. 8. Composition with relative timer event

corresponding buy event in this time interval.

An event identifier (o) is given to the left of the event query after the keyword `event`. It is then used in the definition of the relative timer `extend[event o, 1 min]` which specifies a time interval one minute longer than the occurrence interval of o. (The time point at which o occurs is understood for this purpose as a degenerated time interval of zero length.) The event identifier t is not necessary here, but can be specified anyway. Event identifiers will also be used in temporal conditions and event accumulation (Sections 5 and 6).

The following constructors for relative timers are currently supported: `extend[e,d]` (adding the duration d to the end of e's time interval), `shorten[e,d]` (subtracting d from the end of e), `extend-begin[e,d]`, `shorten-begin[e,d]` (adding/subtracting d from the begin of e), `shift-forward[e,d]`, `shift-backward[e,d]` (moving e forward/backward by d).

5. Temporal Conditions

Temporal conditions on events and causal relationships between events play an important role in querying events. We concentrate in this paper on temporal conditions, though the approach generalizes to causal relationships.

Reasoning about occurrence times of events and their temporal order is less demanding both in requirements posed on event sources and computation resources than reasoning about causality. Temporal relationships between events can be determined only by looking at the occurrence times of events, which come "for free" with the event stream. Causal relationships require assistance by or intimate knowledge about the applications reacting to and generating events (event sources). They either have to be maintained extensionally (e.g., as tables in a database or as part of the event data) or defined intensionally (e.g., by defining causal relationships again as some form of composite event queries).

The event identifiers that we have introduced in the previous section are also used when specifying temporal conditions on events. Just like conditions on event data, temporal conditions are specified in the `where`-clause of an event query.

```
DETECT earlyResellWithLoss {
         customer { var C },
         stock { var S }
       }
ON and {
     event b: buy {{
       customer { var C },
       stock { var S },
       price { var P1 } }},
     event s: sell {{
       customer { var C },
       stock { var S },
       price { var P2 } }}
   } where {b before s, timeDiff(b,s)<1hour, var P1>var P2}
END
```

Fig. 9. Event query with temporal conditions

An example of an event query involving temporal conditions is given in Figure 9. It detects situations where a customer first buys stocks and then sells them again within a short time (less than 1 hour) at a lower price. The query shows that both qualitative conditions (`b before s`) and quantitative (or metric) conditions (`timeDiff(b,s) < 1 hour`) are required. In addition, the query also includes a data condition for the price (`var P1 > var P2`).

In principle, various external calendar and time reasoning systems could be used to specify and evaluate temporal conditions. However, many optimizations for the evaluation of event queries require knowledge about temporal conditions. In the example above, using the condition `b before s` allows to (1) completely avoid evaluating the sell query until a buy event is received and (2) use the values for variables C and S obtained from buy events when evaluating the sell query.

Our language deals with non-periodic time intervals (time points are treated as degenerated intervals of zero length), periodic time intervals (i.e., sequences of non-periodic intervals), and durations (lengths of time) and provides the following built-in constructs for specifying temporal conditions:

- Event identifiers (`event e`). They give as value the occurrence time (a non-periodic time interval) of the event they are bound to.
- Constructors for absolute time points and time intervals such as `datetime("2006-09-18T09:00")`. Periodic intervals are allowed and application-dependent constructors can be specified externally, e.g., `catts:tradingDay()`.[3]
- Constructors for durations such as `3 min 14 sec`.
- Functions for creating durations from time intervals such as `timeDiff(i,j)` and `length(i)`.
- Functions for manipulating time intervals such as `extend(i,d)`, `shift-forward(i,d)`.
- Relations for durations: $>$, $<$, $<=$, $>=$, $=$, e.g., as in `timeDiff(i,j) < 1 hour`.

[3] The requirement for externally defined periodic time intervals is that an iterator delivers the individual intervals in system time, ordered by their starting time. Thus a periodic interval like `tradingday` can be understood as a function $t : N \rightarrow T \times T$ with $\text{begin}(t(i)) < \text{begin}(t(i+1))$.

```
and {
  event s: sell {{ }},
  event t: timer:datetime {{
    date { "2006-09-18" },
    time { "9:00" }
  }}
} where { s before t }
```

```
event s: sell {{ }}
where {
  s before datetime(
    "2006-09-18T09:00")
}
```

Fig. 10. Timer events vs. time constructors

- Allen's 13 relations for time intervals [3] such as `before`, `after`, `during`, `contains`, `overlaps`.[4] When comparing two time intervals at least one of them has to be non-periodic, and the relations `before` and `after` should not be used for periodic intervals at all.

Note that there is an important difference between timer events used in queries and references to time as part of `where`-conditions. Timer events have to happen for the event query to yield an answer (i.e., they are waited for), while time references in conditions can lie in the future and only restrict the possible answers to an event query. In Figure 10, all answers to the event query on the left are detected at the same time (`2006-09-18T09:00`), while the answers to the event query on the right occur at different times (whenever a sell event is received).

Our language differs significantly from most other event query languages[5] in the respect that temporal relationships between events are specified as temporal conditions separately from the event composition itself and that thus only the two composition operators `and` and `or` are needed. Previous work on event query languages tended to have an "algebraic" flavor with lots of different event composition operators (including a sequence operator). Apart from giving a separation of concerns and being easily extensible to application-dependent calendars, our approach thus avoids some problems with the semantics and intuitive understanding of many composition operators.

To illustrate the last point, consider an event query asking for events a, b, c to happen, where a happens before c and b happens before c. Using the temporal condition in our approach, this is straight-forward. Using a sequence composition operator (usually denoted ;) as well as an (temporal) conjunction operator (with the same semantics as our `and`-operator; often denoted with $\wedge$ or $\triangle$), one might be tempted to write an event query $(a; c) \triangle (b; c)$. This however does not yield the intended result since *different* c-events can be used in answering the query. (A correct way to write the query would be $(a \triangle b); c$.) Similar examples are in [27, 15, 1].

From this toy example, it might seem that for small composite event queries (detecting only two or three events, say), the separation makes queries a bit lengthy. However in real-life, already atomic event queries are much longer (usually several lines) than the `where`-clause and event identifiers.

[4] "Exact" relationships such as `starts` or `equals` are less useful since different events rarely begin or end at exactly the same time [27], but are included for the sake of completeness.

[5] Cf. references [17, 16, 11, 23, 28, 27, 20, 19, 2, 1, 7]

```
DETECT andthen [var X, var Y]
ON and {
    event x: var X,
    event y: var Y
  } where {x before y}
END
```

Fig. 11. Rules simulating a sequence operator

```
DETECT buyOrderOverdue { orderId { var I } }
ON and {
    event o: order {{
      orderId { var I }  }},
    event t: extend(o, 1 min),
    while t: not buy {
      orderId { var I }  }
  }
END
```

Fig. 12. Event accumulation for negation

Further, deductive rules can be used to define syntactic sugar for common cases as illustrated in Figure 11.

6. Event Accumulation

Event querying displays its differences to database querying most perspicuously in non-monotonic query features such as negation or aggregation. For database queries, the data to be considered for negation or aggregation is readily available in the database and this database is *finite*.[6] In contrast, events are received over time in an event stream which is potentially infinite. To be able to query with negation or aggregation, one first has to restrict the infinite event stream to a finite extract. Once such a restriction (window or scope) is made, negation and aggregation of events can be applied to the events accumulated in this window and differ not much from their database counterparts.[7]

Such an accumulation window has to be of finite temporal extent, i.e., be given by a finite interval. It should be possible to determine this window dynamically depending on the event stream received so far. Typical examples are the time windows "from event a until event b," "one minute until event b," "from event a for one minute," and (since events can occur over time intervals, not just time points) "while event c." The last example of a window subsumes the first three since they can in turn be defined as composite events (using relative timers, see Section 4). We hence content ourselves with only looking at this case. (Syntactic sugar for the simpler but common cases will be defined as the language matures.)

Negation is supported by applying the `not` operator to an event query. A time window is specified with the keyword `while` and the event identifier of the event defining the window. The meaning is as one might expect: the negated event query `while` t`: not` q is successful if and only if no event satisfying q occurs during the time interval given by t. An

[6] Recursive rules or views may allow to define infinite databases intensionally. However, the extensional data (the "base facts") is still finite.

[7] Keep in mind that accumulation here refers to the way we specify queries, not the way evaluation is actually performed. Keeping all events in the accumulation windows in memory is generally neither desirable nor necessary for query evaluation.

```
RAISE to(recipient="http://example.com",
         transport="http://.../soap/bindings/HTTP/"){
        reportOfDailyAverages {
          all entry {
            stock { var S },
            avgPrice { avg( all var P ) }
          } group-by var S  }
      }
ON and {
     event t: catts:tradingDay{{ }},
     while t: collect sell {
         stock { var S }
         price { var P }  }
END
```

Fig. 13. Event accumulation for aggregation

example can be seen in Figure 12: it detects stocks orders that are overdue, i.e., where no matching buy or sell transaction has taken place within one minute after placing the order. The accumulation window is specified by the event query t, which is a timer relative to the order event. Observe that the negated query can contain variables (I here) that are also used outside the negation; the example reveals the strong need to support this.[8]

Let us now turn to aggregation. As common in rule-based languages, aggregation constructs are used in the *head* of a rule, since it is related to the construction of new data. The task of the *body* is only *collecting* the necessary data or events. Collecting events in the body of a rule is similar to the negation and indicated by the keyword `collect`. The rule in Figure 13 shows in the body an event query collecting sell events over a full trading day.

The actual aggregation takes place in the head of the rule, where all sales prices (P) for the same stock (S) are averaged and a report containing one entry for each stock is generated. The report is sent at the end of the trading day; this is reflected in the syntax by the fact that `catts:tradingDay{{ }}` is written as an event, i.e., has to actually occur.

The aggregation in the head of the rule follows the syntax and semantics of the Web query language Xcerpt, again showing that it is beneficial to base an event query language on a data query language. The keyword `all` indicates a structural aggregation, generating an `entry` element for each distinct value of the variable S (indicated with `group-by`). Inside the `entry`-element an aggregation function `avg` is used to compute the average price for each individual stock. For a full account of aggregation syntax, see work on Xcerpt [24].

We have seen in this section how negation and aggregation in event queries can be treated just like in database queries in our approach, once a window or scope has been introduced (with the keyword `while`) limiting the event stream to a finite extract. The window is specified by a (composite) event, making our approach more general than negation operators in many related works, where the window is given by a starting and an ending event. Aggregation has rarely been considered in work on composite events. A notable exception is [23], which however applies only to relational data (not semi-structured or XML) and does not have the benefits of a separation of the query dimensions as our approach.

[8] Such variables occurring in negated and non-negated form make evaluation slightly more complicated and less efficient, especially if the negated query applies to a time window *before* the variable is bound in some other event query. See work on the `without`-operator in XChange [7] for an account how the situation can be treated.

7. Approaches for Semantics and Evaluation

Formal and declarative semantics, especially model-theoretic semantics, are as desirable for an event query language as they are for data query languages. Formal semantics of languages provide a reference to implementors and help greatly in standardization efforts. They allow to prove correctness of evaluation methods, which is particularly important for research on optimization. They give rise to proofs about properties of the language in general, certain classes of queries, or individual queries. An example for such a property is the "bounded lifespan property" for the class of "legal event queries" defined and proven in [8]. Finally, an easy to understand and "mathematically aesthetic" model theory is an indication of a good language design and helps identifying design flaws.

A model-theoretic semantics for our language can be defined along the lines of the model theory of the data query language Xcerpt [24] by extending it with temporal features. This is however outside the scope of this paper, where we concentrate on the general design of the language and syntax.

Related work on semantics for event queries usually has an "algebraic flavor" (as the languages themselves do), where the semantics for operators are given as functions between sequences (or histories or traces) of events, e.g., [28, 19]. Algebras can be argued to be less declarative than model theories, expressing *how* an event is to be detected rather than *what* event is to be detected. They are however a very valuable step towards evaluation and optimization.

Evaluation of event queries differs strongly from database queries. Query evaluation in databases is usually query-driven. In contrast, evaluation of composite event queries on an event stream should be data-driven (or "event-driven") and incremental for efficency reasons. Data-driven approaches used in the past include finite automata [17], special petri nets [16], and event trees or graphs with inner nodes storing "semi-composed" events and a bottom-up flow of events [11].

For event queries putting an emphasis on data in the form of variable bindings, an event graph approach seems most fitting. Inner nodes in the graph need to perform joins on variables shared between different constituent queries of a composite query [7]. This makes event graphs much similar to rete [13], which is primarily used for production rules systems. Scalability of such approaches is indicated in [2].

8. Conclusions and Future Work

In this article, we have introduced a high-level event query language. It deviates from previous languages in a separation of the query dimensions data extraction, event compo-

sition, temporal and other relationships, and event accumulation. This separation allows a complete coverage of each of the dimensions, yielding a language that can be argued to have reached a degree of expressive completeness.

We have put emphasis on queries to events represented in XML and other Web formats, making our event query language suited for use in service-oriented and event-driven architectures based on Web Services. Important for practical use, rules are supported as an abstraction mechanism. Queries and rules for events are also relevant in efforts to bring rules, including reactive rules, to the (Semantic) Web [26].

A limitation of the language so far is that it does not support event instance selection and event instance consumption [28]. It could be argued that they are of less importance for business-level events, where similar effects can be achieved by considering the data contained in the events. However, instance selection can, e.g., be quite important for sensor events.

Investigation of this limitation is planned for the future. We are also considering restrictions on queries keeping demands on event storage constant (see also [8]). Causal relationships between events have only been touched on briefly in this article and will be treated in more detail in the future.

Implementation of our language in the scope of XChange is ongoing work. (The current prototype of XChange [29] still uses the event language described in [7], but the evaluation method used is similar to the one outlined in Section 7.) Optimization methods for evaluating large numbers of event queries are also being explored.

Acknowledgments

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (`http://rewerse.net`).

References

[1] R. Adaikkalavan and S. Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. *Data and Knowledge Engineering*, 2005. In press.

[2] A. Adi and O. Etzion. Amit — the situation manager. *Int. J. on Very Large Data Bases*, 13(2), 2004.

[3] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11), 1983.

[4] J. Bailey, F. Bry, M. Eckert, and P.-L. Pătrânjan. Flavours of XChange, a rule-based reactive language for the (Semantic) Web. In *Proc. Intl. Conf. on Rules and Rule Markup Languages for the Semantic Web*. Springer, 2005.

[5] S. Boag et al. XQuery 1.0: An XML query language. W3C candidate recommendation, 2005.

[6] D. Box et al. Web services addressing (WS-Addressing). W3C member submission, 2004.

[7] F. Bry, M. Eckert, and P.-L. Pătrânjan. Querying composite events for reactivity on the Web. In *Proc. Int. Workshop on XML Research and Applications*. Springer, 2006.

[8] F. Bry, M. Eckert, and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and applications of the language XChange. *J. of Web Engineering*, 5(1), 2006.

[9] F. Bry, F.-A. Rieß, and S. Spranger. CaTTS: Calendar types and constraints for Web applications. In *Proc. Int. World Wide Web Conf.* ACM Press, 2005.

[10] A. P. Buchmann, J. Zimmermann, J. A. Blakeley, and D. L. Wells. Building an integrated active OODBMS: Requirements, architecture, and design decisions. In *Proc. Int. Conf. on Data Engineering*. IEEE, 1995.

[11] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proc. Int. Conf. on Very Large Data Bases*, 1994.

[12] O. Etzion. Towards an event-driven architecture: An infrastructure for event processing (position paper). In *Proc. Intl. Conf. on Rules and Rule Markup Languages for the Semantic Web*. Springer, 2005.

[13] C. L. Forgy. A fast algorithm for the many pattern/many object pattern match problem. *Artif. Intelligence*, 19(1), 1982.

[14] T. Furche, F. Bry, and O. Bolzer. Marriages of convenience: Triples and graphs, RDF and XML in Web querying. In *Int. Workshop on Principles and Practice of Semantic Web Reasoning*. Springer, 2005.

[15] A. Galton and J. C. Augusto. Two approaches to event definition. In *Proc. Int. Conf. on Database and Expert Systems Applications*. Springer, 2002.

[16] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Workshop on Rules in Database Systems*. Springer, 1993.

[17] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Int. Conf. on Very Large Data Bases*, 1992.

[18] M. Gudgin et al. SOAP 1.2. W3C recommendation, 2003.

[19] A. Hinze and A. Voisard. A parameterized algebra for event notification services. In *Proc. Int. Symp. on Temporal Representation and Reasoning*. IEEE, 2002.

[20] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[21] W. May, J. J. Alferes, and R. Amador. Active rules in the Semantic Web: Dealing with language heterogeneity. In *Proc. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web*. Springer, 2005.

[22] W. May, J. J. Alferes, and R. Amador. Ontology- and resources-based approach to evolution and reactivity in the Semantic Web. In *Proc. Int. Conf. on Ontologies, Databases, and Applications of Semantics*. Springer, 2005.

[23] I. Motakis and C. Zaniolo. Temporal aggregation in active database rules. In *Proc. Int. Conf. on Management of Data (SIGMOD)*. ACM Press, 1997.

[24] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, Inst. f. Informatics, U. of Munich, 2004.

[25] S. Schaffert and F. Bry. Querying the Web reconsidered: A practical introduction to Xcerpt. In *Proc. Extreme Markup Languages*, 2004.

[26] W3C. Rule interchange format working group charter. `http://www.w3.org/2005/rules/wg/charter`.

[27] D. Zhu and A. S. Sethi. SEL, a new event pattern specification language for event correlation. In *Int. Conf. on Computer Communications and Networks*. IEEE, 2001.

[28] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *Proc. Int. Conf. on Data Engineering*. IEEE, 1999.

[29] `http://www.reactiveweb.org/xchange`.