Rule-Based Composite Event Queries: The Language XChange$^{\mathbf{EQ}}$ and its Semantics

Michael Eckert and François Bry

Institute for Informatics, University of Munich
Oettingenstr. 67, 80538 München, Germany
`eckert@pms.ifi.lmu.de`, `bry@lmu.de`
`http://www.pms.ifi.lmu.de`

Abstract. Reactive Web systems, Web services, and Web-based publish/subscribe systems communicate events as XML messages, and in many cases require composite event detection: it is not sufficient to react to single event messages, but events have to be considered in relation to other events that are received over time. This entails a need for expressive, high-level languages for querying composite events.
Emphasizing language design and formal semantics, we describe the rule-based composite event query language XChange$^{\mathrm{EQ}}$. XChange$^{\mathrm{EQ}}$ is designed to completely cover and integrate the four complementary querying dimensions: event data, event composition, temporal relationships, and event accumulation. Semantics are provided as a model theory with accompanying fixpoint theory, an approach that is established for rule languages but has not been applied to event queries so far. Because they are highly declarative, thus easy to understand and well-suited for query optimization, such semantics are desirable for event queries.

1 Introduction

Emerging Web technologies such as reactive Web systems [16, 9, 14, 33], Web-based publish/subscribe systems [35, 23], and Web services communicate by exchanging messages. These messages usually come in an XML format such as SOAP [29] or Common Base Event (CBE) [22] and signify some application-level event, e.g., an update on a Web document, publication of new information, a request for some service, or a response to a request.

For many applications it is not sufficient 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*.

Examples for such composite events are omnipresent. An application for student administration might require notification when “a student has both handed in her thesis and given the defense talk.” A library application might send a monition when “a book has been borrowed and not returned or extended within one month.” A stock market application might require notification if “the average of the reported stock prices over the last hour raises by 5%.”

Development of applications that involve detection of composite events in a way that is cost-efficient, maintainable, and allows easy programming and automatic optimization entails a need for an expressive, high-level event query language. This article describes work on the rule-based high-level event query language XChange$^{\mathrm{EQ}}$ for the Web, focusing on language design and formal semantics. XChange$^{\mathrm{EQ}}$ has first been introduced in [8]. This work develops formal semantics in the form of a model theory and a fixpoint theory for stratified programs. XChange$^{\mathrm{EQ}}$ is developed as a part (sub-language) of the reactive, rule-based Web language XChange [16].[1] Its design also allows deployment as a stand-alone event mediation component in an event-driven architecture [25] or use in Semantic Web ECA frameworks [33, 4].

The contributions of this article are as follows. (1) We discuss language design issues of event query languages for the Web (Section 2). We identify four complementary dimensions that need to be considered for querying events. While they might have been implicit in some works on composite event queries, we are not aware of any works stating them explicitly before.

(2) We shortly introduce XChange$^{\mathrm{EQ}}$ (Section 3). XChange$^{\mathrm{EQ}}$ is significantly more high-level and expressive than previous (composite) event query languages. To the best of our knowledge, XChange$^{\mathrm{EQ}}$ is the first language to deal with complex structured data in event messages, support rules as an abstraction and reasoning mechanism for events, and build on a separation of concerns that gives it ease-of-use and a certain degree of expressive completeness.

(3) We provide formal semantics for XChange$^{\mathrm{EQ}}$ in the form of model and fixpoint theories (Section 4). While this approach is well-explored in the world of rule-based and logic programming, its application to event queries is novel and should be quite beneficial for research on composite event queries: semantics of earlier event query languages often have been somewhat ad hoc, generally with an algebraic and less declarative flavor, and did not accommodate rules. Declarative semantics with support for rules like ours are highly desirable because they are formal and unambiguous, easy to understand, and well-suited as a basis for query optimization (e.g., development of evaluation-oriented algebras [11, 12]). In our discussion, we highlight where we deviate from traditional model theories to accommodate the temporal notions required by event queries.

(4) We compare XChange$^{\mathrm{EQ}}$ with the traditional composition operator based approach for querying events and illustrate the advantages of XChange$^{\mathrm{EQ}}$’s language design in terms of expressiveness and ease of use (Section 5).

This article is an extended version of [10]. Major additions include the formal definition of stratification (Section 4.2), proofs for the theorems of Section 4.4 (Appendix), and the comparison of XChange$^{\mathrm{EQ}}$ with composition operator based approaches (Section 5).

---

[1] Accordingly, the superscript EQ stands for **E**vent **Q**ueries. XChange$^{\mathrm{EQ}}$ replaces the original composite event query constructs [15] of XChange. It has a different design and is an improvement both in expressivity and ease-of-use.

2 Expressive Event Query Languages

Our work on XChange$^{\mathrm{EQ}}$ is motivated by previous work on XChange [16], a language employing Event-Condition-Action rules to program distributed, reactive Web applications. Similar to composite event detection facilities found in active databases [28, 27, 21, 20, 2], XChange provides composition operators such as event conjunction, sequence, repetition, or negation. Our experiences with programming in XChange [17, 14] has taught us that there is a considerable gap between the requirements posed by applications and the expressivity of composition operators. Further, event querying based on composition operators is prone to misinterpretations as discussions in the literature show [40, 26, 2]. This experience has lead us to reconsider and analyze the requirements for event query languages, which we present here, and to the development of XChange$^{\mathrm{EQ}}$.

A sufficiently expressive event query language should cover (at least) the following four complementary dimensions. How well an event query language covers each of these dimensions gives a practical measure for its expressiveness.

Data extraction: Events contain data that is relevant for applications to decide whether and how to react to them. For events that are received as XML messages, the structure of the data can be quite complex (semi-structured). The data of events must be extracted and provided (typically as bindings for variables) to test conditions (e.g., arithmetic expressions) inside the query, construct new events, or trigger reactions (e.g., database updates).

Event composition: To support composite events, i.e., events that consist out of several events, event queries must support composition constructs such as the conjunction and disjunction of events (more precisely, of event queries). Composition must be sensitive to event data, which is often used to correlate and filter events (e.g., consider only stock transactions from the *same* customer for composition). Since reactions to events are usually sensitive to timing and order, an important question for composite events is *when* they are detected. In a well-designed language, it should be possible to recognize when reactions to a given event query are triggered without difficulty.

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 since causal relationships can be queried in essentially the same manner.[2]

Event accumulation: Event queries must be able to accumulate events to support non-monotonic features such as negation of events (understood as their absence) or aggregation of data from multiple events over time. The reason for this is that the event stream is (in contrast to extensional data in a database)

---

[2] While temporality and causality can be treated similarly in queries, causality raises interesting questions about how causal relationships can be *defined* and *maintained*. Investigation of these issues is planned for the future.

unbounded (or “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).

Further, a language should allow the use of (deductive) rules to define new events from the answers to a composite event query, similar to the way views are used in database systems.

Rules: (Deductive) rules allow to define new, “virtual” events from the existing one (i.e., those that are received in the incoming event stream), much in the same fashion as one uses view (or rules) in databases to define new, derived data from existing base data. Only very few event languages support such purely deductive rules, even though support is highly desirable for a number of reasons: Rules serve as an abstraction mechanism, making query programs more readable. They allow to define higher-level application events from lower-level (e.g., database or network) events. Different rules can provide different perspectives (e.g., of end-user, system administrator, corporate management) on the same (event-driven) system. Rules allow to mediate between different schemas for event data. Additionally, rules can be beneficial when reasoning about causal relationships of events [32].

In addition to deductive rules, event-based systems usually also require reactive rules, typically Event-Condition-Action (ECA) rules, to specify reactions to the occurrences of certain events. While deductive rules can be, and often are, implemented using reactive rules, we argue that deductive (event) rules are inherently different from reactive rules because they aim at expressing “virtual events,” not additional actions. Accordingly and importantly, deductive rules are free of side-effects. Implementing deductive rules using reactive rules blurs this distinction and often strongly restricts optimization: techniques that are applicable for deductive rules, such as backward chaining or program rewriting, are not generally applicable to reactive rules.

3 The Language XChange$^{\mathbf{EQ}}$

XChange$^{\mathrm{EQ}}$ is designed on the following foundations.

(1) 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 experience, stemming from attempts to express queries with existing event query languages, shows us that without such a separation not all dimensions are fully covered. We will illustrate the benefits of the separation with a comparison to other approaches in Section 5.

(2) It embeds the Web and Semantic Web query language Xcerpt [39] to specify classes of relevant events, extract data (in the form of variable bindings) from them, and construct new events.

(3) It supports rules as an abstraction and reasoning mechanism for events, with the same motivation and benefits of views in traditional database systems (cf. Section 2).

These foundations lead to improvements on previous work on composite event query languages in the following ways: XChange$^{\mathrm{EQ}}$ 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, which has been neglected or over-simplified earlier. Being targeted for semi-structured XML messages as required for CBE, SOAP, and Web Services, it is particularly suitable for use in business applications domains. We make an attempt towards expressive completeness by fully covering all four query dimensions explained earlier using a separation of concerns in XChange$^{\mathrm{EQ}}$. Arguably, in previous languages that do not use such a separation, some (usually simple) queries might be expressed more compactly. This compactness then however leads easily to misinterpretations (as discussed in [40, 26, 2]) and comes in previous work at the price of a serious lack in expressiveness (incomplete coverage of the four dimensions), where less simple queries cannot be expressed or only be expressed in a cumbersome, long winded way. We revisit these topics in more detail later in Section 5, after XChange$^{\mathrm{EQ}}$ and its semantics have been introduced.

Using the example of a stock market application, we now introduce the syntax of our event query language XChange$^{\mathrm{EQ}}$.

3.1 Querying Atomic Events

Application-level events are nowadays often represented as XML, especially in the formats Common Base Event [22] and SOAP [29]. Skipping details of such formats for the sake of brevity, we will be using four atomic events in our stock market example: stock *buy*s, stock *sell*s, and *order*s to buy or sell stocks. Involved applications may also generate further events without affecting our examples.

The left side of Figure 1 depicts a *buy order* event in XML. For conciseness and human readability, we use a "term syntax" for data, queries, and construction of data instead of the normal tag-based XML syntax. The right side of Figure 1 depicts the XML event as a (data) term. 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). XChange$^{\mathrm{EQ}}$ embeds the XML query language Xcerpt [39] for both. Figure 2 shows an exemplary *buy* event (left) and an event query that recognizes such *buy* events with a price total of \$10 000 or more (right).

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

```
<order>
  <orderId>4711</orderId>
  <customer>John</customer>
  <buy>    <stock>IBM</stock>
           <limit>3.14</limit>
           <volume>4000</volume>
</buy> </order>
```

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

Fig. 1. XML and term representation of an event

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

```
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

(indicated by single brackets or braces). 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 constructs for (subterm) negation, incompleteness in breadth and depth, and queries to graph-shaped data such as RDF. An introduction to Xcerpt is given in [39].

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 on the left of Figure 2 with $\Sigma = \{\sigma_1\}$, $\sigma_1 = \{I \mapsto \texttt{4242}, C \mapsto \texttt{John}, S \mapsto \texttt{IBM}, P \mapsto \texttt{2.71}, V \mapsto \texttt{4000}\}$.

For querying atomic events in XChange$^{\mathrm{EQ}}$, Xcerpt has the following advantages over other XML query languages such as XQuery [7]: (1) The notion of an event matching or not matching a query gives a straightforward 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 this separation, Xcerpt provides an intuitive way to deal with variable bindings using the concept of substitution sets; this is especially convenient when composing events. In comparison, in XQuery a variable might be understood as taking different values at different steps (e.g., iterations of a FOR-loop) and composition of events would typically require using nested FOR-loops.

In addition to event messages, XChange$^{\mathrm{EQ}}$ event queries can query for timer events. Absolute timer events are time points or intervals (possibly periodic) defined without reference to the occurrence time of some other event. They are specified in a similar way as queries to event messages and we refer to [8] for details. Relative timer events, i.e., time points or intervals defined in relation to some other event, will be looked at in Section 3.3 on event composition.

```
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
```

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

Fig. 3. Deductive rule (left) and reactive rule (right)

3.2 Reactive and Deductive Rules for Events

XChange$^{\mathrm{EQ}}$ 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. Figure 3 (left) shows a deductive rule deriving a new *bigbuy* events 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 are replaced during construction by their values obtained from evaluating the event query in the rule body. Several variables bindings will lead to the construction of several events if no grouping or aggregation constructs are used.) The event construction is also called a construct term; more involved construction will be seen in Section 3.5 when we look at aggregation of data.

As usual in rule-based query languages, all variables in the head must occur at least once in the body in a defining position, i.e., outside a negation like `without` (“range restriction”). Recursion of rules is restricted to stratifiable programs, see Section 4.2 for a deeper discussion.

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. Note that this new event leaves the system and that it is up to the receiver to decide on the occurrence time (typically such events are considered to happen only at the time *point* when the corresponding message is received). For tasks involving accessing and updating persistent data, our 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 3 (right); it forwards every *bigbuy* event (as derived by the deductive rule on the left) 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`; in the rule head `to()` is used to indicate recipient and transport.

The distinction between deductive and reactive rules is important. While it is possible to “abuse” reactive rules to simulate or implement deductive rules (by sending oneself the result), this is undesirable: it is difficult with events that

```
DETECT buyorderfulfilled {  orderId { var O },
                            tradeId { var I },
                            stock { var S } }
ON and {
      order  {  orderId { var O },
                buy {{  stock { var S }  }}  },
      buy   {{  orderId { var O },
                tradeId { var I }  }}  }
END
```

Fig. 4. Conjunction of event queries

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

Fig. 5. Composition with relative timer event

have a duration, misleading for programmers, less efficient for evaluation, and could allow arbitrary recursion (leading, e.g., to non-terminating programs or non-stratified use of negation).

3.3 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 3.5.) Both composition operators are multi-ary, allowing to compose any (positive) number of event queries (without need for nesting), and written in prefix notation. Disjunctions are a convenience in practical programming but not strictly necessary: a rule with a (binary) disjunction can be written as two rules. We therefore concentrate on conjunctions here.

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 bindings are considered. (This generalizes to composition of three and more event queries in the obvious manner.) Figure 4 illustrates the use of the `and` operator. The *buy order fulfilled* event is detected for every corresponding pair of *buy order* and *buy* event. The events have to agree on variable O (the `orderId`). The occurrence time of the detected *order fulfilled* event is the time interval enclosing the respective constituent events.

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 5 shows a composite event query asking for an *order* event and a timer covering the whole time interval from the *order* event until one minute after. This timer event will be used later in Section 3.5 when querying for the absence of a 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[o, 1 min]`

```
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. 6. Event query with temporal conditions

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 3.4 and 3.5).

Further constructors for relative timers are: `shorten[e,d]` (subtracting *d* from the end of *e*), `extend-begin[e,d]`, `shorten-begin[e,d]` (adding or subtracting *d* at the begin of *e*), `shift-forward[e,d]`, `shift-backward[e,d]` (moving *e* forward or backward by *d*).

3.4 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. Just like conditions on event data, temporal conditions are specified in the `where`-clause of an event query and make use of the event identifiers introduced above.

The event query in Figure 6 involves temporal conditions. 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 illustrates that typical applications require both qualitative conditions (`b before s`) and quantitative (or metric) conditions (`timeDiff(b,s) < 1 hour`). 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. See [11, 12] for an initial discussion of temporal optimizations.

XChange$^{\text{EQ}}$ deals supports (convex) 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). The following constructs for specifying temporal conditions are built-in:

- Event identifiers (declared with `event b:` in the query). In temporal conditions (such as `b before s`) they stand for 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.[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`.
- Allen's 13 relations for time intervals [5] 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.

XChange$^{\text{EQ}}$ differs significantly from most other event query languages 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 (see Section 5).

3.5 Event Accumulation

Event querying displays its differences to traditional querying most perspicuously in non-monotonic query features such as negation or aggregation. For traditional database queries, the data to be considered for negation or aggregation is readily available in the database and this database is *finite*.[5] In contrast, events are

---

[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 can be understood as a function $t : N \rightarrow T \times T$ with $\text{begin}(t(i)) < \text{begin}(t(i+1))$.

[4] "Exact" relationships such as `starts` or `equals` are less useful since different events rarely begin or end at exactly the same time [40], but are included for the sake of completeness.

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

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

```
RAISE to(...) {
  reportOfDailyAverages {
    all entry {
      stock { var S },
      avgPrice { avg(all var P) }
    } group-by var S  }  }
ON and {
     event t: tradingDay{{ }},
     while t: collect sell {
         stock { var S },
         price { var P }  }
END
```

Fig. 7. Event accumulation for negation (left) and aggregation (right)

received over time in an event stream which is unbounded, i.e., potentially infinite. Applying negation or aggregation on such a (temporally) infinite event stream would imply that one has to wait "forever" for an answer because events received at a later time might always change the current answer. We therefore need a way to restrict the event stream to a finite temporal extent (i.e., a finite time interval) and apply negation and aggregation only to the events collected in this accumulation window.[6]

It should be possible to determine the accumulation window dynamically depending on the event stream received so far. Typical cases of such accumulation windows are: "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*." Here we only look at the last case because it subsumes the first three (they can be defined as composite events).

Negation is supported by applying the `not` operator to an event query. The 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 no event satisfying *q* occurs during the time interval given by *t*. An example can be seen in Figure 7 (left): it detects buy orders that are overdue, i.e., where no matching buy 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 that are also used outside the negation; the example reveals the strong need to support this.

Following the design of the embedded query language Xcerpt, aggregation constructs are used in the *head* of a rule, since they are 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 negation and indicated by the keyword `collect`. The rule in Figure 7 (right) has 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

---

[6] 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.

and a report containing one entry for each stock is generated. The report is sent at the end of each trading day; this is reflected in the syntax by the fact that `tradingDay{{ }}` must be written as an event, i.e., must actually occur.

Aggregation follows the syntax and semantics of Xcerpt (see [38] for a full account), 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.

Aggregation has rarely been considered in work on composite events, though it is clearly needed in many applications, including our stock market example. A notable exception is [34], 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 XChange$^{\mathrm{EQ}}$.

4 Formal Semantics

Having introduced XChange$^{\mathrm{EQ}}$ informally above, we now supply formal, declarative semantics for stratified programs in the form of model and fixpoint theories. While this is a well-established approach for rule-based languages [31, 6], including traditional database query languages supporting views or deductive rules, it has not been applied to event query languages before. 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., [41, 30]. Further, these approaches often neglect *data* in events (especially semi-structured data) and it is not clear how they could be extended to support deductive *rules* (or views) over events.

In addition to accommodating both rules and data, the model theoretic approach presented here can be argued to be more declarative than previous algebraic approaches, expressing *how* an event is to be to detected rather than *what* event is to be detected, making programs easier to understand and optimize.

The following specifics of querying events as opposed to pure (database) data have to be arranged for in our semantics and make it novel compared their counterpart in the logic programming literature [31, 6]: (1) in addition to normal variables, event identifiers are accommodated, (2) answers to composite event queries have an occurrence time, (3) temporal relations have a fixed interpretation. Finally, the model theory must be (4) sensible for potentially *infinite* streams of events (this also entails that negation and aggregation of events must be "scoped" over a time window as we have seen earlier in Section 3.5).

4.1 Model Theory

Our model theory is Tarskian-style [18], i.e., it uses a valuation function for free variables and defines an entailment relation between an interpretation and sentences (rules and queries) from the language *recursively over the structure* of the sentences. This recursive definition over the structure allows to consider sub-formulas of a formula in isolation, which is beneficial for both understanding and evaluation. Tarskian model theories have the advantage of being highly declarative, theoretically well-understood, and relatively easy to understand.

An **event** happens over a given time interval and has a representation as message (as data term). Formally it is a tuple of a (closed and convex) time interval t and a data term e, written e^t. The set of all events is denoted *Events*.

Time is assumed to be a linearly[7] ordered set of time points $(\mathbb{T}, <)$. The time intervals over which events happen are closed and convex, i.e., have the form $t = [b, e] = \{p \mid b \leq p \leq e\}$ (where $b \in \mathbb{T}$ and $e \in \mathbb{T}$). For convenience we define: $begin([b, e]) = b$, $end([b, e]) = e$, $[b_1, e_1] \sqcup [b_2, e_2] = [min\{b_1, b_2\}, max\{e_1, e_2\}]$, and $[b_1, e_1] \sqsubseteq [b_2, e_2]$ iff $b_2 \leq b_1$ and $e_1 \leq e_2$.

Matching of **Atomic Event Queries** against single incoming events is based on a non-standard unification that is especially designed for the variations and incompleteness in semi-structured data. Atomic Event Queries are single query terms q that match only for the data term part e of events e^t; this does not involve time or multiple events. Note that the query terms usually contain free variables. The matching of query terms and data terms is based on **Simulation**, which is a relation between ground terms, denoted $\preceq$. Intuitively, $q \preceq d$ means that the nodes and structure of q can be found in d. Simulation naturally extends to a non-ground query term q' by asking whether there is a (grounding) substitution σ for the free variables in q' such that the ground query term $q = \sigma(q')$ obtained by applying the substitution σ to q' simulates with the given data term d. Further details can be found in [38]; they are not important for understanding the presented model theory and thus not discussed here.

Substitution sets Σ rather than single substitutions σ are used in our model theory to accommodate grouping and aggregation in the construction in rule heads. Application $\Sigma(c)$ of Σ to a construct term c results in a set of data terms. For convenience we also define the application to query terms q with $\Sigma(q) = \{\sigma(q) \mid \sigma \in \Sigma\}$.

An **interpretation** for a given XChange$^{\mathrm{EQ}}$ query, rule, or program is a 3-tuple $M = (I, \Sigma, \tau)$, where (1) $I \subseteq Events$ is the set of events e^t that "happen," i.e., are either in the stream of incoming events or derived by some deductive rule. (2) $\Sigma \neq \emptyset$ is a grounding substitution set containing substitutions for the "normal" variables (i.e., data variables, but not event identifiers). (3) τ is a substitution for the event identifiers, i.e., a mapping from event variables to *Events*. The substitution τ for event identifiers (cf. Section 3.3) is the first unusual features of our model theory. Since τ signifies the events that contributed to the answer of some query, we also call it an "event trace."

The **satisfaction** $M \models F^t$ of an XChange$^{\mathrm{EQ}}$ expression F over an occurrence time t in an interpretation M is defined recursively in Figure 8. The time stamping of expressions is the second unusual feature of our model theory.

---

[7] Linear time is chosen because we are interested in event that actually happened, not in potential futures (where a branching time would be more apt).

$$
\begin{array}{ll}
I, \Sigma, \tau \models (\texttt{event } i : q)^t & \text{iff exists } e^{t'} \in I \text{ with } \tau(i) = e^{t'},\ t' = t, \\
 & \text{and for all } e' \in \Sigma(q) \text{ we have } e' \preceq e \\
I, \Sigma, \tau \models (\texttt{event } i : \texttt{extends}[j, d])^t & \text{iff exists } e^{t'} \text{ with } \tau(j) = e^{t'},\ \tau(i) = e^t,\ t = t' + d \\
\ldots & \text{(Definitions for other temporal events are similar and skipped.)} \\
M \models (q_1 \wedge q_2)^t & \text{iff } M \models q_1^{t_1} \text{ and } M \models q_2^{t_2} \text{ and } t = t_1 \sqcup t_2 \\
M \models (q_1 \vee q_2)^t & \text{iff } M \models q_1^t \text{ or } M \models q_2^t \\
I, \Sigma, \tau \models (Q \texttt{ where } C)^t & \text{iff } I, \Sigma, \tau \models Q^t \text{ and } W_{\Sigma,\tau}(C) = \mathit{true} \\
I, \Sigma, \tau \models (\texttt{while } j : \texttt{ not } q)^t & \text{iff exists } e^{t'} \text{ with } \tau(j) = e^{t'},\ t' = t, \\
 & \text{and for all } t'' \sqsubseteq t \text{ we have } I, \Sigma, \tau \not\models q^{t''} \\
I, \Sigma, \tau \models (\texttt{while } j : \texttt{ collect } q)^t & \text{iff exists } e^{t'} \text{ with } \tau(j) = e^{t'},\ t' = t, \text{ and exist } n \geq 0, \\
 & \Sigma_1, \ldots \Sigma_n,\ t_1 \sqsubseteq t, \ldots t_n \sqsubseteq t \text{ with } \Sigma = \bigcup_{i=1..n} \Sigma_i, \\
 & \text{and for all } i = 1..n \text{ we have } I, \Sigma_i, \tau \models q^{t_i} \\
I, \Sigma, \tau \models (c \leftarrow Q)^t & \text{iff (1) } \Sigma'(c)^t \subseteq I \text{ for } \Sigma' \text{ maximal (w.r.t. } \mathit{FreeVars}(Q)) \text{ and } \tau' \\
 & \text{such that } I, \Sigma', \tau' \models Q^t, \text{ or (2) } I, \Sigma', \tau' \not\models Q^t \text{ for all } \Sigma', \tau'
\end{array}
$$

$$
\begin{array}{ll}
W_{\Sigma,\tau}(i \texttt{ before } j) = \mathbf{true} & \text{iff } \mathit{end}(\tau(i)) < \mathit{begin}(\tau(j)) \\
W_{\Sigma,\tau}(i \texttt{ during } j) = \mathbf{true} & \text{iff } \mathit{begin}(\tau(j)) < \mathit{begin}(\tau(i)) \text{ and } \mathit{end}(\tau(i)) < \mathit{end}(\tau(j)) \\
W_{\Sigma,\tau}(i \texttt{ overlaps } j) = \mathbf{true} & \text{iff } \mathit{begin}(\tau(j)) < \mathit{begin}(\tau(i)) < \mathit{end}(\tau(j)) < \mathit{end}(\tau(i))
\end{array}
$$

Fig. 8. Model Theory for XChange$^{\mathrm{EQ}}$

Given an XChange$^{\mathrm{EQ}}$ program P and a stream of incoming events E, we call an interpretation $M = (I, \Sigma, \tau)$ a **model** of P under E if (1) M satisfies all rules $(c \leftarrow Q) \in P$ for all time intervals t and (2) contains the stream of incoming events, i.e., $E \subseteq I$. Note that here the event stream simply corresponds to the notion of base facts or extensional data found of traditional model theories.

The satisfaction relation uses a fixed interpretation W for all conditions that can occur in the `where`-clause of a query. This includes the temporal relations like `before` and is the third unusual feature of our model theory. W is a function that maps a substitution set Σ, an event trace τ, and an atomic condition C to true or false; we usually write Σ and τ in the index. $W_{\Sigma,\tau}$ extends straightforwardly to boolean formulas of conditions. The definition of W is left outside the "core model theory" to make it more modular and allow to easily integrate different temporal reasoners. In Figure 8, we have given only the definitions for `before`, `during`, and `overlaps`; the other cases are analogous and the full model theory can be found in [24].

Our fourth requirement on the model theory was that it is sensible on (potentially) infinite streams of events. The basic idea for this is that to evaluate a program P over a time interval t, we only have to consider events happening during t. We will state this formally as a theorem later.

4.2 Stratified XChange$^{\mathrm{EQ}}$-Programs

Non-monotonic features such as negation and aggregation introduce well-known issues when they are combined with recursion. Consider a program consisting of the following two rules:

$$
\begin{array}{l}
\mathrm{p}[x] \leftarrow \texttt{event } w : \mathrm{s}[x] \ \wedge \ \texttt{while } w : \ \texttt{not } \mathrm{q}[x] \\
\mathrm{q}[x] \leftarrow \texttt{event } w : \mathrm{s}[x] \ \wedge \ \texttt{while } w : \ \texttt{not } \mathrm{p}[x]
\end{array}
$$

It is not clear what the intended semantics of this program are. For example under the event stream $E = \{s(1)^{[1,3]}\}$ both $M_1 = \{s(1)^{[1,3]}, p(1)^{[1,3]}\}$ and $M_2 = \{s(1)^{[1,3]}, q(1)^{[1,3]}\}$ are models and, since they are symmetric, none is preferable. This is a common and inherent difficulty when rules and negation are combined. (In fact it is an adaption of the standard (non-event and non-temporal) example $p \leftarrow \neg q, q \leftarrow \neg p$ from logic programming and deductive databases.) A simple and established solution is to avoid such situations by requiring programs to be stratifiable.

Stratification restricts the use of recursion in rules by ordering the rules of a program P into so-called strata (sets P_i of rules with $P = P_1 \uplus \cdots \uplus P_n$) such that a rule in a given stratum can only depend on (i.e., access results from) rules in lower strata (or the same stratum, in some cases).

Three types of stratification are required: (1) Negation stratification, i.e., events that are negated in the query of a rule may only be constructed by rules in lower strata, events that occur positively may only be constructed by rules in lower strata or the same stratum. (2) Grouping stratification, i.e., rules using grouping constructs like `all` in the construction may only query for events constructed in lower strata. (3) Temporal stratification, i.e., if a rule queries a relative temporal event like `extends[i, 1min]` then the anchoring event (here: `i`) may only be constructed in lower strata. While negation and grouping stratification are fairly standard, temporal stratification is a requirement specific to complex event query programs like those expressible in XChange$^{\mathrm{EQ}}$. We are not aware of former consideration of the notion of temporal stratification.

We say that some rule r depends on another rule $'$ if r (potentially) queries events that have been constructed by r'. We distinguish different kinds of dependency:

- $r = c \leftarrow Q \in P$ *depends temporally* on $r' = c' \leftarrow Q' \in P$ if there exists a query term q in Q with an event identifier j attached to it such that j is used elsewhere in Q to define a relative temporal event and $q \preceq c'$.[8]
- $r = c \leftarrow Q \in P$ *grouping depends* on $r' = c' \leftarrow Q' \in P$ if c contains grouping constructs (such as `all`) and there exists a query term q in Q such that $q \preceq c'$.
- $r = c \leftarrow Q \in P$ *depends negatively* on $r' = c' \leftarrow Q' \in P$ if there exists a query term q that occurs negated (i.e., within a `not`) in Q such that $q \preceq c'$.
- $r = c \leftarrow Q \in P$ *depends positively* on $r' = c' \leftarrow Q' \in P$ if r does not depend on r' by the previous dependencies and there exists a query term q in Q such that $q \preceq c'$.

A stratification $P = P_1 \uplus \cdots \uplus P_n$ for an XChange$^{\mathrm{EQ}}$ program P is a partitioning of the rules of P such that for each pair of rules $r = c \leftarrow Q \in P_i$, $r' = c' \leftarrow Q' \in P_j$:

- if r depends temporally on r' or r grouping depends on r' or r depends negatively on r', then $i < j$.

---

[8] In slight abuse of notation, we write $q \preceq c'$ for $\exists \Sigma \exists e \in \Sigma(q) \exists e' \in \Sigma(c). e \preceq e'$.

– if r depends positively on r', then $i \geq j$.

An XChange$^{\text{EQ}}$ program is called stratifiable, if there exists a stratification for it.

The restriction to stratifiable programs is necessary for the fixpoint semantics given in the next section. Note that this is not a very severe restriction in the domain of event queries, and in fact it might be desirable to restrict programs further to so-called hierarchical programs, which do not allow any recursion, to obtain more efficient evaluation algorithms. However, this does not make any difference to the formal semantics, so we treat the more general case of stratifiable programs. It would also be conceivable to (partially) lifted the restriction to stratifiable programs at the cost of a more involved semantics and evaluation. Extending semantics beyond stratified programs is possible (e.g., with established approaches from logic programming), but outside the scope of this paper.

4.3 Fixpoint Theory

A model theory, such as the one presented above, has the issue of allowing many models for a given program, even for stratifiable programs. A common and convenient way to obtain a unique model is to define it as the solution of a fixpoint equation (which is based on the model theory). A fixpoint theory also describes an abstract, simple, forward-chaining evaluation method, which can easily be extended to work incrementally as is required for event queries [9].

The **fixpoint operator** T_P for an XChange$^{\text{EQ}}$-Program P is defined as:

$$\begin{aligned} T_P(I) = I \cup \{e^t \mid & \text{ there exist a rule } c \leftarrow Q \in P, \text{ a maximal substitution set } \Sigma, \\ & \text{and a substitution } \tau \text{ such that } I, \Sigma, \tau \models Q^t \text{ and } e \in \Sigma(c)\} \end{aligned}$$

The repeated application of T_P until a fixpoint is reached is denoted T_P^ω.

The **fixpoint interpretation**[9] $M_{P,E}$ of a program P with stratification $P = P_1 \uplus \cdots \uplus P_n$ under and event stream E is defined by computing fixpoints stratum by stratum: $M_0 = E = T_\emptyset^\omega(E)$, $M_1 = T_{\overline{P_1}}^\omega(M_0)$ …, $M_{P,E} = M_n = T_{\overline{P_n}}^\omega(M_{n-1})$. Here, $\overline{P_i} = \bigcup_{j \leq j} P_j$ denotes the set of all rules in strata P_i and lower.

4.4 Theorems

We now give a theorem that shows that the declarative semantics for programs in our rule language provided by the fixpoint interpretation are well-defined and unambiguous.

Theorem 1 justifies our definition as usual for fixpoint semantics: For a stratifiable program P and an event stream E, $M_{P,E}$ is a minimal model of P under E. Further, $M_{P,E}$ is independent of the stratification of P.

---

[9] Since we consider whole programs P now, only the set I of events that happen is relevant for the fixpoint interpretation of P; Σ and τ are thus skipped from now on.

More interestingly, we can show that the model theory and fixpoint semantics are sensible on unbounded (or "infinite") event streams, which is the last feature of our semantics that is peculiar for event queries. The next theorem justifies a streaming evaluation, where answers to composite event queries are generated "online" and we never have to wait for the stream to end (which it will not if infinite). This is especially important, since event streams can conceptually be infinite and thus not end at all.

In particular it ensures an event e^t can be detected at the time point $end(t)$ since no knowledge about any events in the future of $end(t)$ is required. Ensuring that evaluation methods are not expected to "crystal gaze" is of course an important requirement and one example where we can use the declarative semantics to prove interesting statements about a (composite) event query language.

Theorem 2: Let $E \mid t$ denote the restriction of an event stream E to a time interval t, i.e., $E \mid t = \{e^{t'} \in E \mid t' \sqsubseteq t\}$. Similarly, let $M \mid t$ denote the restriction of an interpretation M to t. Then the result of applying the fixpoint procedure to $E \mid t$ is the same as applying it to E for the time interval t, i.e., $M_{P,E|t} \mid t = M_{P,E} \mid t$. In other words to evaluate a program over a time interval t, we do not have to consider any events happening outside of t.

Proofs for both theorems are presented in the appendix. The proof for theorem 1 is an adoption of a proof in [31].

5 Language Design Revisited

Having introduced the syntax (Section 3) and semantics (Section 4) of XChange$^{\mathrm{EQ}}$, we now revisit its language design and illustrate its advantages over the traditional approach of using a rather large number of composition operators for querying composite events. A more in-depth discussion can be found in [24]

Event query languages based on composition operators (see, e.g., [28, 27, 21, 34, 36, 41, 40, 32, 30, 3, 2, 15] build up composite event queries from atomic event queries (in this context often also called an event types) and composition operators. Such languages are also often called event algebras, since they consist of a set (the event types) and operations on it (composition operators). Composition operators can be understood as functions whose input and output are streams of events. For example, binary composition operators such as conjunction (often written $A \wedge B$ or $A \triangle B$) or sequence (often written $A; B$) take as input two event streams, the results of the queries that are their arguments, and produce as output another event stream containing the composite events. Negation is usually expressed as a ternary variation of the sequence operator, e.g., $A; \neg B; C$, meaning that an event of type A is followed by an event of type C, and no event of type B is happening in-between. There is a wide spectrum of further operators supported by different languages, but for our purpose of comparing XChange$^{\mathrm{EQ}}$ to the algebraic approach conjunction, sequence, and negation will suffice.

Consider now 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 XChange$^{\mathrm{EQ}}$, this is straight-forward:

```
and { a: A, b: B, c: C }
where { a before c, b before c }
```

In an event algebra using a sequence composition operator, one might be tempted to write such an event query as $(A;C)\triangle(B;C)$. 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 also in [40, 26, 2].

As a further example, consider an event query asking for events A, B, C and D to happen, with the constraints that A happens before B, A also happens before C, and C before D. The XChange$^{\mathrm{EQ}}$ query is analogous to this natural language description. Similar to the previous example, note that the query cannot be expressed in an event algebra as $(A;B)\triangle(A;C)\triangle(C;D)$ because this would allow different instance of A and C to be used. A correct way to express the query would be $A;(B\triangle(C;D))$. Consider now, what happens if we only add an additional constraint that B happens before D. In the XChange$^{\mathrm{EQ}}$ query we simply have to add this statement to the `where`-clause. In an event algebra, however, the new query bears only little resemblance to the old: $A;(B\triangle C);D$. In fact, even though we *added* a constraint in our specification, the number of operators in the query stays *same* for the event algebra![10]. This is quite unnatural and might easily cause programming errors.

For a final example, consider queries that involve also metric temporal constraints such as "event A and B happen within 1 hour." Many event algebras support this by offering extended operators with temporal constraints. For example, $(A;B)_t$ would denote that B happens within t time units after A, and $(A\triangle B)_t$ would denote that A and B happen within t time units (regardless of their order). The XChange$^{\mathrm{EQ}}$ query

```
and { a: A, b: B, c: C }
where { a before b, b before c,
        timeDiff(b,a) < 1 hour, timeDiff(c,b) < 1 hour }
```

specifying that A happens, then B happens within 1 hour of that A, and then C happens within 1 hour of that B, cannot be expressed in an event algebra using a (binary) temporally constrained sequence operator. Note that the expression $((A;B)_{1h};C)_{1h}$ would require C to happen within 1 hour of A, not of B (symmetrically for $(A;(B;C)_{1h})_{1h}$).[11] Of course, what can or cannot be expressed in a given event algebra always depends on the operators it offers. For example, an event algebra could also offer a sequence operator of higher arity with temporal constraints and thus be able to express the query (e.g., as $A;_{1h}B;_{1h}C$).

---

[10] In some event algebra languages that support n-ary versions of the sequence operator (such as `andthen` in [15, 16]), it could even be said to be one operator *less*

[11] Note that we assume a sequence operator based on time intervals, not time points, here [40, 26].

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

Fig. 9. Using rules to provide a sequence operator as syntactic sugar

Composition operators mix the event querying dimensions explained earlier (e.g., in the case of a sequence operator, event composition and temporal relationships are mixed). It can be argued that this leads to the exemplified difficulties in expressing and understanding queries and also to a certain incompleteness in the expressiveness of typical event algebras. Due to its separated treatment of the event query dimensions, XChange$^{\text{EQ}}$ does not have these difficulties and can easily be extended, e.g., with external temporal reasoners that support application-specific calendar types such as "business day" or "teaching term" [19].

From the toy examples above, it might seem that for small composite event queries (detecting only two or three events, say), XChange$^{\text{EQ}}$'s separation of querying dimensions makes queries a bit lengthy. However when atomic events are in an XML format, already atomic event queries are usually much longer (typically stretching several lines) than the `where`-clause and event identifiers, as the examples in Section 3 show. This can actually make more compact approaches such as event algebras harder to read, since the composition operators "hide" somewhere between the atomic event queries, and therefore more error prone.

Further, individual queries can be made concise (and easier to maintain) in XChange$^{\text{EQ}}$by introducing appropriate deductive rules. Figure 9 illustrates how rules can be used to define syntactic sugar for a common case like the sequence of two events (reminiscent of a sequence operator). The need for deductive rules in an event query language and a clear distinction between deductive and reactive rules has already been emphasized in Section 2.

6 Conclusions and Future Work

This article has introduced the high-level event query language XChange$^{\text{EQ}}$, emphasizing language design and formal semantics. XChange$^{\text{EQ}}$ deviates from previous event query languages in a separation of the query dimensions data extraction, event composition, temporal 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.

The ability to query events represented in XML and other Web formats, makes XChange$^{\text{EQ}}$ suited for use in service-oriented and event-driven architectures based on Web Services. Important for practical use, rules are supported as an abstraction and reasoning mechanism for events. Rule-based reasoning about events is also expected to become relevant in efforts to bring rules, including reactive rules, to the (Semantic) Web [37, 9].

Efficient evaluation methods and a prototype implementation for XChange$^{\mathrm{EQ}}$ have been developed [24], but are beyond of the scope of this article. The evaluation of XChange$^{\mathrm{EQ}}$ is based on a tailored form of relational algebra called Complex Event Relational Algebra (CERA). An important aspect in the evaluation is that events have to be stored for some time. In order avoid that memory demands grow at least linearly over time (w.r.t. to the length of the event stream received so far), a garbage collection mechanism is necessary. In [13], such a garbage collection method is developed based on the notion of temporal relevance of an event, which can be determined statically (i.e., at query compile time rather than at run time). Evaluation methods that utilize temporal conditions and query optimization for large numbers of event queries have also been investigated [11, 12].

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. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
2. R. Adaikkalavan and S. Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. *Data and Knowledge Eng.*, 2005. In press.
3. A. Adi and O. Etzion. Amit — the situation manager. *Int. J. on Very Large Data Bases*, 13(2), 2004.
4. J. J. Alferes and R. Amador. r3: A foundational ontology for reactive rules. In *Proc. Int. Conf. on Ontologies, DataBases, and Applications of Semantics*, 2007.
5. J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11), 1983.
6. K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, 1988.
7. S. Boag et al. XQuery 1.0: An XML query language. W3C candidate recommendation, 2005.
8. F. Bry and M. Eckert. A high-level query language for events. In *Proc. Int. Workshop on Event-driven Architecture, Processing and Systems*, 2006.
9. F. Bry and M. Eckert. Twelve theses on reactive rules for the Web. In *Proc. Int. Workshop Reactivity on the Web*, 2006.
10. F. Bry and M. Eckert. Rule-Based Composite Event Queries: The Language XChange$^{\mathrm{EQ}}$ and its Semantics. In *Proc. Int. Conf. on Web Reasoning and Rule Systems*, 2007.

11. F. Bry and M. Eckert. Temporal order optimizations of incremental joins for composite event detection. In *Proc. Int. Conf. on Distributed Event-Based Systems*, 2007.
12. F. Bry and M. Eckert. Towards formal foundations of event queries and rules. In *Proc. Int. Workshop on Event-Driven Architecture, Processing and Systems*, 2007.
13. F. Bry and M. Eckert. On static determination of temporal relevance for incremental evaluation of complex event queries. In *Proc. Int. Conf. on Distributed Event-Based Systems*. ACM, 2008.
14. F. Bry, M. Eckert, H. Grallert, and P.-L. Pătrânjan. Evolution of distributed Web data: An application of the reactive language XChange. In *Proc. Int. Conf. on Data Engineering (Demonstrations)*, 2006.
15. 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*, 2006.
16. 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.
17. F. Bry, M. Eckert, P.-L. Pătrânjan, and I. Romanenko. Realizing business processes with ECA rules: Benefits, challenges, limits. In *Proc. Int. Workshop on Principles and Practice of Semantic Web*, 2006.
18. F. Bry and M. Marchiori. Ten theses on logic languages for the Semantic Web. In *Int. Workshop on Principles and Practice of Semantic Web Reasoning*, 2005.
19. F. Bry, F.-A. Rieß, and S. Spranger. CaTTS: Calendar types and constraints for Web applications. In *Proc. Int. World Wide Web Conf.*, 2005.
20. 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*, 1995.
21. 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.
22. Common Base Event. www.ibm.com/developerworks/webservices/library/ws-cbe.
23. Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale XML dissemination service. In *Proc. Int. Conf. on Very Large Data Bases*, 2004.
24. M. Eckert. *Complex Event Processing with XChange$^{\mathrm{EQ}}$: Language Design, Formal Semantics, and Incremental Evaluation for Querying Events.* PhD thesis, Institute for Informatics, University of Munich, 2008. http://edoc.ub.uni-muenchen.de/9405/.
25. O. Etzion. Towards an event-driven architecture: An infrastructure for event processing (position paper). In *Proc. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web*, 2005.
26. A. Galton and J. C. Augusto. Two approaches to event definition. In *Proc. Int. Conf. on Database and Expert Systems Applications*, 2002.
27. S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Proc. Int. Workshop on Rules in Database Systems*, 1993.
28. N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proc. Int. Conf. on Very Large Data Bases*, 1992.
29. M. Gudgin et al. SOAP 1.2. W3C recommendation, 2003.
30. A. Hinze and A. Voisard. A parameterized algebra for event notification services. In *Proc. Int. Symp. on Temporal Representation and Reasoning*, 2002.
31. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1993.
32. D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley, 2002.

33. 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*, 2005.
34. I. Motakis and C. Zaniolo. Temporal aggregation in active database rules. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1997.
35. J. Pereira, F. Fabret, H.-A. Jacobsen, F. Llirbat, and D. Shasha. WebFilter: A high-throughput XML-based publish and subscribe system. In *Proc. Int. Conf. on Very Large Databases*, 2001.
36. C. Roncancio. Toward duration-based, constrained and dynamic event types. In *Proc. Int. Workshop on Active, Real-Time, and Temporal Database Systems*, 1997.
37. Rule Interchange Format WG Charter. www.w3.org/2005/rules/wg/charter.
38. S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web.* PhD thesis, Inst. f. Informatics, U. of Munich, 2004.
39. S. Schaffert and F. Bry. Querying the Web reconsidered: A practical introduction to Xcerpt. In *Proc. Extreme Markup Languages*, 2004.
40. D. Zhu and A. S. Sethi. SEL, a new event pattern specification language for event correlation. In *Proc. Int. Conf. on Computer Communications and Networks*, 2001.
41. D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *Proc. Int. Conf. on Data Engineering*, 1999.

A Proof of Theorem 1

A.1 Minimal Model

We want to prove that the fixpoint interpretation $M_{P,E}$ of a stratified XChange$^{\mathrm{EQ}}$ program P under an event stream E is a minimal model, i.e., there is no model M of P under E with $M \subsetneq M_{P,E}$. An analogous statement for stratified logic programs is well-established [6]. For our proof in the world of XChange$^{\mathrm{EQ}}$, we adapt the proof from [31].[12] The most important hurdle in transferring this proof is that the stratification of XChange$^{\mathrm{EQ}}$ programs is defined differently: in XChange$^{\mathrm{EQ}}$ strata consist of rules, while in logic programming strata consist of predicate symbols.

Before starting with our proof, we repeat a well-known result on fixpoints due to Tarski (as presented in [31]): Let L be a complete lattice with order $\leq$ and $T : L \rightarrow L$ monotonic. Then T has a least fixpoint, $lfp(T)$, that is the greatest lower bound (glb) of the set of all fixpoints of T as well as of the set of all prefixed points: $lfp(T) = glb\{x \mid T(x) = x\} = glb\{x \mid T(x) \leq x\}$.

Let $P = P_1 \uplus \cdots \uplus P_n$ be a stratified XChange$^{\mathrm{EQ}}$ program and E an event stream. We show by induction on the number n of strata that $M_{P,E} = M_n$ is a minimal fixpoint of T_P with $E \subseteq M_{P,E}$. For the induction base $n = 0$ this is obvious since then $P = \emptyset$ and thus T_P is simply the identity transformation.

For the induction step $n-1 \rightarrow n$, we will make use of Tarski's theorem from above. Of course we cannot apply the theorem to T_P directly since T_P is, in

---

[12] As we will see, the proof in [31] uses the existence of least fixpoints for monotonic operators on complete lattices, a well-known result established by Knaster and Tarski. The earlier proof in [6] works without this result. We base our proof on [31] because it is shorter and seems more intuitive and easier to understand than [6].

general, not monotonous. However we can apply it to a certain restriction. Let Λ be the following complete lattice:

$$\Lambda = \{M_{n-1} \cup S \mid S \subset \bigcup_{r \in P} gi(r)\}$$

Here $gi(r)$ denotes the set of all events a rule $r = c \leftarrow Q$ can query or construct (speaking in the language of logic programming, the "*g*round *i*nstances" of the terms occurring in the rule[13]). Formally:

$$\begin{aligned} gi(c \leftarrow Q) = \{e^t \mid t \in \mathbb{T} \text{ and } \exists \Sigma . e \in \Sigma(c)\} \cup \\ \{e^t \mid t \in \mathbb{T} \text{ and } \exists \text{query term } q \text{ in } Q\ \exists \sigma . e \preceq \sigma(q)\} \end{aligned}$$

Lemma 1 The restriction $T_P \mid_\Lambda$ of T_P to the complete lattice Λ is well-defined (i.e., application of T_P to an element of Λ yields an element that is again in Λ) and monotonous (i.e., $I \subseteq J$ implies $T_P \mid_\Lambda (I) \subseteq T_P \mid_\Lambda (J)$). To avoid distraction from the main proof, we give the proof for this lemma later.

With Tarski's theorem from above, this lemma gives us that $T_P \mid_\Lambda$ has a least fixpoint M:

$$\begin{aligned} M = lfp(T_P \mid_\Lambda) &= glb\{I \in \Lambda \mid T_P(I) = I\} \\ &= glb\{I \in \Lambda \mid T_P(I) \subseteq I\} \end{aligned}$$

Note that $M = M_{P,E}$ simply by definition of $M_{P,E} = T_P^\omega(M_{n-1})$. Further, M is a model for P under E due to the following Lemma.

Lemma 2 An interpretation I is a model for an XChange$^{\text{EQ}}$ program P ("$I \models P$") if and only if $T_P(I) \subseteq I$. Again, we delay the proof for this lemma.

We now show that M is minimal, i.e., if some $M' \subseteq M$ is a model of P and $E \subseteq M'$ then $M' = M$. By induction hypothesis, we have that $M_{n-1} \subseteq M'$ (since M_{n-1} is a minimal model for $P_1 \uplus \cdots \uplus P_{n-1}$). By definition of Λ this gives us that $M' \in \Lambda$. Lemma 2 and M being also the least pre-fixed point of $T_P \mid_\Lambda$ yield $M' \subseteq M$.

A.2 Proof of Lemma 1

To show that $T \mid_\Lambda$ is well-defined, let $I \in \Lambda$. By definition of Λ, $I = M_{k-1} \uplus S_I$ for some $S_I \subseteq \bigcup_{r \in P} git(r)$. Now $T_P(I) = I \cup \{e^t \mid e^t \text{generated by some rule } r \in P\}$ and the right side of the union is a subset of $\bigcup_{r \in P} git(r)$. This gives us that $T_P(I) = M_{k-1} \uplus S_T$ for some $S_T \subseteq \bigcup_{r \in P} git(r)$ (S_T is the union of $\{e^t \mid \dots\}$ and S_I) and thus $T_P(I) \in \Lambda$.

For $T_P \mid_\Lambda$ monotonic, let $I \in \Lambda$, $J \in \Lambda$, $I \subseteq J$ and $e^t \in T_P(I)$. What we want to show is that $e^t \in T_P(J)$.

If $e^t \in I$ we immediately have $e^t \in T_J(P)$ by $I \subseteq J$ and the definition of T_P. Otherwise there is a rule $r = c \leftarrow Q \in P$ and a maximal Σ and a

---

[13] To avoid confusion: note that we talk about the ground instances of the *terms* occurring in the rule, not about ground instances of the rule itself.

τ with $I, \Sigma, \tau \models Q$ and $e^t \in \Sigma(c)$. If $r \in \overline{P_{n-1}}$, then $e^t \in M_{n-1}$ and thus also $e^t \in T_P(J)$ since $M_{n-1} \subseteq T_P(J)$ (remember that $T_P \mid \Lambda$ is well-defined). It remains to consider the case where $r \in P_n$, where we have to show that $e^t \in \Sigma(c)$.

We distinguish whether c is free of grouping constructs or not. In the former case it suffices to show that $J, \Sigma, \tau \models Q^t$. In the latter case we have to show additionally that Σ is maximal.

Case 1: c free of grouping constructs. By induction on Q we show that $J, \Sigma, \tau \models Q^t$. Besides the induction hypothesis (IH) and the definition of the model theory (Def$_\models$) from Figure 8, we have available that $I, \Sigma, \tau \models Q^t$ with Σ being maximal $(*)$ and that $I \subseteq J$ $(**)$.

Case 1.1: $Q =$ (event $i : q$). By (Def$_\models$) and $(*)$, we have $e^t \in I$ and by $(**)$ we get the $e^t \in J$.

Case 1.2: $Q =$ (event $i :$ extends$[j, d]$). Trivial, since (Def$_\models$) makes no reference to the interpretation I (J, respectively). As in Figure 8, we skip the other temporal events since they are analogous.

Case 1.3: $Q = (q_1 \wedge q_2)$. (Def$_\models$) and $(*)$ gives us $I, \Sigma, \tau \models q_1^{t_1}$ and $I, \Sigma, \tau \models q_2^{t_2}$ with $t = t_1 \sqcup t_2$. Applying (IH) we get $J, \Sigma, \tau \models q_1^{t_1}$ and $J, \Sigma, \tau \models q_2^{t_2}$ and can apply (Def$_\models$).

Case 1.4: $Q = (q_1 \vee q_2)$. Obvious application of (IH), see case 1.3.

Case 1.5: $Q = (Q'$ where $C)$. Obvious application of (IH).

Case 1.6: $Q =$ (while j : not q). $(*)$ gives us an e^t with $\tau(j) = e^t$ and $I, \Sigma, \tau \not\models q^{t''}$ for all $t'' \sqsubset t$. We have to show that also $J, \Sigma, \tau \not\models q^{t''}$ for all $t'' \sqsubset t$. Now, if there were a t'' such that $J, \Sigma, \tau \models q^{t''}$, then already $M_{n-1}, \Sigma, \tau \models q^{t''}$ due to the (negation) stratification. This however would imply $I, \Sigma, \tau \models q^{t''}$ in contradiction to our assumptions.

Case 1.7: $Q =$ (while j : collect q). Again a simple application of (IH).

Case 2: c contains grouping constructs. As in case 1 we get that $J, \Sigma, \tau \models Q^t$. It remains to show by induction on Q that Σ is in fact maximal, i.e., $J, \Sigma \cup \{\sigma\}, \tau \models Q^t$ with some $\sigma \notin \Sigma$ $(***)$ leads to a contradiction with the maximality in $(*)$.

Case 2.1: $Q =$ (event $i : q$). Suppose by (Def$_\models$) and the assumption $(* * *)$ that there is an $e^t \in J$ with $\sigma(q) \preceq e$. Due to stratification, $e^t \in M_{n-1}$ and thus $I, \Sigma \cup \{\sigma\}, \tau \models Q^t$ in contradiction to $(*)$.

Case 2.2: $Q =$ (event $i :$ extends$[j, d]$). Trivial since $I, \Sigma, \tau \models Q^t$ for any Σ, i.e., also for $\Sigma \cup \{\sigma\}$.

Case 2.3: $Q = (q_1 \wedge q_2)$. (Def$_\models$) and $(* * *)$ give $J, \Sigma \cup \{\sigma\}, \tau \models q_1^{t_1}$ and $J, \Sigma \cup \{\sigma\}, \tau \models q_2^{t_2}$ and $t = t_1 \sqcup t_2$. Application of (IH) leads to the contradiction.

Case 2.4: $Q = (q_1 \vee q_2)$. Obvious application of (IH), see case 1.3.

Case 2.5: $Q = (Q'$ where $C)$. Obvious application of (IH).

Case 2.6: $Q =$ (while :j : not q). (Def$_\models$) and $(***)$ give an e^t with $\tau(j) = e^t$ such that $J, \Sigma \cup \{\sigma\}, \tau \not\models q^{t''}$ for all $t'' \sqsubset t$. The maximality in $(*)$ however gives $I, \Sigma \cup \{\sigma\}, \tau \models q^{t''}$. By (IH) then the contradiction $J, \Sigma, \tau \models q^{t''}$.

Case 2.7: $Q = (\text{while } j : \text{collect } q)$. By $(\text{Def}_{\models})$ and $(***)$ there must exist an e^t with $\tau(j) = e^t$ such that there are Σ_i and $t_i \sqsubset t$ with $J, \Sigma_i \cup \{\sigma\}, \tau \models q^{t_i}$. Application of (IH) now gives the contradiction.

A.3 Proof of Lemma 2

We want to show that $M \models P$ if and only if $T_P(M) \subseteq M$. From right to left, suppose $T_P(M) \subseteq M$, but $M \not\models P$, i.e., there's a rule $r = c \leftarrow Q \in P$ with $M \not\models r$. Accordingly we must have a $t \in \mathbb{T}$, a τ, and a maximal Σ with $M, \Sigma, \tau \models Q^t$ but $\Sigma(c)^t \nsubseteq M$. I.e., there must be an $e \in \Sigma(c)$ such that $e \notin M$, which however is in contradiction to $e \in T_P(M) \subseteq M$.

From left to right, let $M \models P$ and $e^t \in T_P(M)$ and suppose $e^t \notin M$. Then e^t must have been generated by a rule $r = c \leftarrow Q \in P$. Accordingly we must have a τ and a maximal Σ such that $M, \Sigma, \tau \models Q^t$ and $e \in \Sigma(c)$. This however would mean that $e^t \in M$ since $M \models r$, which gives us the contradiction.

A.4 Independence from Stratification

To prove that $M_{P,E}$ is said to be independent of the given stratification of P, we show that any two possible stratifications of P are equivalent, i.e., the fixpoint procedure yields the same model. In the world of stratified logic programs, this is again a well-established result. In fact, proof of this statement for datalog$^{\neg}$ found in [1] (Theorem 15.2.10) transfers directly to XChange$^{\text{EQ}}$. Only two things need to be adapted to deal with XChange$^{\text{EQ}}$'s notion of stratification, which is defined over rules not predicate symbols: the notion of the precedence graph (sometimes this is also called a dependency graph) and a lemma that enables us to argue that if two strata that are independent of each other then they can be permuted in the fixpoint procedure. For space reasons we do not repeat the proof from [1] here, but give only the two adaptations just mentioned.

The **precedence graph** G_P for an XChange$^{\text{EQ}}$ program P is a directed graph with edges label either "+" (called positive edges) or "−" (called negative edges). The vertices of the graph are the rules of P. There is a positive edge from r to r' if r depends positively on r'. There is a negative edge from r to r' if r' depends on r' in any other way (negatively, temporally, or by grouping).[14]

We have to show the following **lemma** as a replacement for lemma 15.2.9 in [1] (note that this is the only point where the proof of [1] is specific to datalog$^{\neg}$): If P is a semi-positive XChange$^{\text{EQ}}$ program, i.e., it only contains positive dependencies, and $P = P_1 \uplus P_2$ is a stratification of P, then the fixpoint procedure yields the same model for P and for $P_1 \uplus P_2$: $T_P^{\omega}(E) = T_{\overline{P_2}}^{\omega}(T_{\overline{P_1}}^{\omega}(E))$ for all event streams E.

---

[14] Admittedly, we are a bit abusive of notation here, using "−" to label not only negative dependencies but also grouping and temporal dependencies. However, it is not necessary to distinguish negative, grouping, and temporal dependencies here, since also in the definition of a stratification the are treated the same (requiring the dependent rule to be in a strictly higher stratum).

Proof. Observe that $T_P = T_{\overline{P_2}}$ is monotonous (just like in Lemma 1 from above). With the inclusion $E \subseteq T^{\omega}_{P_1}(E) = T^{\omega}_{\overline{P_1}}(E)$ this yields $T^{\omega}_{P}(E) \subseteq T^{\omega}_{\overline{P_2}}(T^{\omega}_{\overline{P_1}}(E))$. On the other hand, the inclusion $T^{\omega}_{\overline{P_1}}(E) = T^{\omega}_{P_1}(E) \subseteq T^{\omega}_{P}(E)$ gives $T^{\omega}_{\overline{P_2}}(T^{\omega}_{\overline{P_1}}(E)) \subseteq T^{\omega}_{\overline{P_2}}(T^{\omega}_{P}(E)) = T^{\omega}_{P}(T^{\omega}_{P}(E)) = T^{\omega}_{P}(E)$.

B Proof of Theorem 2

We have to show that $M_{P,E|u} \mid u = M_{P,E} \mid u$ for an arbitrary time interval u. For this, we first make the following observation.

Lemma 3 Let t and u be time intervals with $t \sqsubseteq u$ and let Q be a query. We then have:

$$I \mid u, \Sigma, \tau \models Q^t \quad \text{iff} \quad I, \Sigma, \tau \models Q^t$$

The proof for this is by a trivial induction on Q.

With the definition of the fixpoint operator T_P, the above observation gives us that $T_P(I \mid u) \mid u = T_P(I) \mid u$ for all time intervals u and all programs P.

Further, the definition of T_P says that all events e^t constructed by a rule r "inherit" their occurrence time t from the rule's query. Thus it holds that $(T_P \mid u)^{\omega} = (T^{\omega}_P) \mid u$ and we get $M_{P,E|u} \mid u = M_{P,E} \mid u$.