Reactivity on the Web: Event Queries in XChange

James Bailey [a] Francois Bry [b] Michael Eckert [b]
Paula-Lavinia Pătrânjan [b]

[a] *Department of Computer Science, University of Melbourne*
[b] *Institute for Informatics, University of Munich*

Abstract

Reactivity, the ability to detect simple and composite events and respond in a timely manner, is an essential requirement in many present-day information systems. With the emergence of new, dynamic Web applications, reactivity on the Web is receiving increasing attention. Reactive Web-based systems need to detect and react not only to simple events but also to complex, real-life situations. This paper introduces *XChange*, a language for programming reactive behaviour on the Web, emphasising the querying of event data and detection of composite events.

Key words: World Wide Web, Reactivity, Reactive languages,
Event-Condition-Action rules, Composite events

1 Introduction

Reactivity on the Web —the ability of sites on the Web to detect happenings or events of interest and to react to them automatically through reactive programs— is gaining importance as part of the solution of many business applications.

A key issue in supporting reactivity is the ability to detect and respond to events in a timely fashion. Events are now beginning to play an increasingly

Email addresses: `jbailey@cs.mu.oz.au` (James Bailey), `francois.bry@ifi.lmu.de` (Francois Bry), `michael.eckert@stud.ifi.lmu.de` (Michael Eckert), `paula.patranjan@ifi.lmu.de` (Paula-Lavinia Pătrânjan).

important role within business strategy on the Web and event driven applications are being more widely deployed [17]. Terms such as *zero latency enterprise*, the *real-time enterprise* and *on-demand computing* are being used to describe a vision in which events recognized anywhere within a business, can immediately activate appropriate actions across the entire enterprise and beyond. Businesses that are able to react to events quickly and take appropriate decisions are likely to have a competitive advantage

For example, consider a tourism Web service responsible for providing timely information about flights to its subscribers, who can receive information in real-time by their PDAs. Events of interest might include delays or cancellations of flights, or the appearance of new discounts for flights offered by a particular airline. Reactions to such events include notifying subscribers and their colleagues about delays or automatically searching for and booking alternative flights. Recognising and acting on such events rapidly is crucial for the business to maintain its customer base.

Businesses on the Web are now also demanding the ability to recognise composite, as well simple events. Complex events go beyond simple (atomic) happenings and allow the specification of situations which aggregate together several simple events. For example, the tourism Web service may wish to detect flights for which at least twenty different customers have cancelled their bookings in the last week. This could imply that a competitor is offering a better rate and so offering a cheaper rate may be appropriate.

Most current reactive languages for the Web are oriented towards reacting only to simple events and do not provide constructs for detecting and querying composite events. The issue of reacting to *composite events*, i.e. (possibly time-related) combinations of events, has received considerable attention in the field of active databases (e.g. [13,18]). Useful concepts can be "borrowed" from active databases when investigating reactivity on the Web. However, differences between (generally centralised) active databases and the Web, where a global synchronised time and central management are missing, necessitate new approaches. One such approach is proposed by the language *XChange* [2,4,5] and is presented in this article.

The remainder of this paper is structured as follows: We first give a high-level overview of the declarative reactive language XChange and its introduce its its design philosophy (Section 2). Next, we introduce event queries, the part of XChange used to specify (composite) events that require a reaction (Section 3). A declarative semantics and an algorithm for evaluation of composite event queries follow in (Section 4). Finally, we give a discussion of related work and a summary in (Section 5 and 6).

2 XChange: Rule-based Reactivity

We begin by introducing some high level design aspects of XChange.

2.1 Local Programs, Global Behaviour

An XChange program runs locally at some Web site — called an XChange-aware Web site. It can access and modify local and remote data (Web resources) in reaction to events. Typical events include updates of data, timer events, but can also be high-level application-dependent events, such as the cancellation of a flight. The notion of locally running programs is coherent with the Web's decentralised architecture. Programs exhibit global behaviour by reacting to changes at (remote) Web sites. In turn, these reactions can trigger further reactions at other Web sites.

2.2 Event-Based Communication

XChange programs running at different Web sites can communicate with each other by sending event messages, i.e. messages containing information about events that have occurred. They are represented in XML format, giving great flexibility for coping with different kinds of applications and levels of event abstraction, ranging from the low-level, e.g. insertion of a data item, to high-level, e.g. cancellation of a certain flight.

For communicating or propagating events on the Web, two approaches are conceivable: the *push*-manner, where a Web site informs possibly interested Web sites about events, and the *pull*-manner, where interested Web sites periodically query (poll) persistent data found at other Web sites, in order to determine changes. For propagating events (i.e. communicating data about events), the push-approach has several important advantages: it allows faster reaction, causes usually less network traffic, and saves local resources. Hence XChange uses push-communication for events.

Communication of XChange-aware Web sites follows a *peer-to-peer* model. All parties have the same capabilities and every party can initiate communication. Event messages are directly communicated between Web sites; no broadcasting of events is allowed and XChange assumes no central instance responsible for controlling (e.g. synchronising) communication over the Web.

2.3 Separation of Volatile and Persistent Data

In a reactive Web, one can distinguish *volatile data* —data about events, communicated in a push-manner— and *persistent data* —data data Web resources, retrievable in a pull-manner. There is a key difference between the two. Persistent data is modifiable; metaphorically it is like (computer-)*written text*. It can be revised by directly executing updates. Volatile data is not modifiable; it is like a stream of *speech*. To correct, complete, or invalidate former volatile data, new *event messages* have to be communicated.

Due to this essential difference in nature, access to volatile and persistent data should not be mixed. Indeed, XChange clearly separates the querying (volatile) event data from the querying of (persistent) Web data: event queries specify situations requiring a reaction, Web queries are only evaluated as a reaction.

Event queries also need to be defined in such a way that no data on any event needs to be kept forever in memory, i.e. the event lifespan must be bounded. By design, XChange event queries are such that volatile data remains volatile. If, for some applications, it is necessary to make some of the volatile data persistent, then the applications should turn events into persistent Web data by explicitly saving the relevant events.

2.4 Event-Condition-Action Rules

Event-Condition-Action (ECA) rules are a natural candidate to implement reactive functionality. An XChange program consists of one or more ECA rules of the form *Event Query – Web Query – Action*. They specify that an *Action* should be automatically executed as an automatic response to the occurrence of a situation specified by an *Event Query*, provided the *Web Query* can be evaluated successfully.

Actions comprise raising events (composing and sending event messages), as well as updates to Web resources (persistent data). XChange supports transactions for executing actions in an all-or-nothing manner. There are two kinds of ECA rules in XChange: *event-raising rules* (specifying events to be raised) and *transaction rules* (specifying transactions to be executed).

Figure 1 shows an example of an XChange rule. It represents the rule of a (reactive) personalised organiser, that detects cancellations of flights his owner has booked (code following the keyword `ON`). As a reaction, it searches for (code following the keyword `FROM`) and books another suitable flight (code following the keyword `TRANSACTION`).

```
TRANSACTION
    in { resource { "http://airline.com/reservations/" },
         reservations {{
             insert reservation { var F, name { "Christina Smith" } }
         }}
    }
ON
    xchange:event {{
         xchange:sender { "http://airline.com" },
         cancellation {{
             flight-number { "AI2021" },
             date { var D }
         }}
    }}
FROM
    in { resource { "http://airline.com" },
         flights {{
             var F -> flight {{
                 from { "Paris" }, to { "Munich" }, date { var D }
             }}
         }}
    }
END
```

Fig. 1. An XChange ECA-rule

```
flights {
    flight {
        number { "AI2011" },
        from { "Paris" },
        to { "Munich" },
        date { "2005-08-21" },
    },
    flight {
        number { "AI2021" },
        from { "Paris" },
        to { "Munich" },
        date { "2005-08-21" },
    },
  ...
}
```

(a) Data term representation

```
<flights>
    <flight>
        <number> AI2011 </number>
        <from>   Paris  </from>
        <to>     Munich </to>
        <date> 2005-08-21 </date>
    </flight>
    <flight>
        <number> AI2021 </number>
        <from>   Paris  </from>
        <to>     Munich </to>
        <date> 2005-08-21 </date>
    </flight>
    ...
</flights>
```

(b) XML representation

Fig. 2. A flight database in data term and XML representation

2.5 Pattern-based Approach

Event queries, Web queries, event raising specifications, and updates describe *patterns* for events that require a reaction. Patterns are templates that closely resemble the structure of the data to be queried, constructed, or modified. XChange embeds the Web query language Xcerpt [16,15], for querying XML and other Web data formats. Data, queries, and updates can be represented either in a term-like syntax, or in XML. Figure 2 contrasts the (data) term and XML representations of a flight database. The term syntax is more compact and easier to read and hence we will use it throughout this paper.

An *ordered term specification* (denoted by square brackets [ ]) expresses that the order of subterms is relevant, an *unordered term specification* (denoted by

curly braces {}) expresses that the order of subterms is irrelevant and must not be kept. Ordered subterms are needed e.g. for describing the sequence of chapters in a book. Unordered subterms are convenient for database or set-like data items. Both *total* and *partial* (event and Web) query patterns can be specified. A query term q using a partial specification (denoted by *double* brackets [[ ]] or braces {{}}) for its subterms, matches with all such terms that (1) contain matching subterms for all subterms of q and that (2) might contain further subterms without corresponding subterms in q. In contrast, a query term t using a total specification (denoted by *single* brackets [ ] or braces {}) does not match with terms that contain additional subterms without corresponding subterms in q.

The patterns can contain variables for extracting pieces of information from data terms (representing event data or Web resources' data). Variables (preceded by the keyword `var`) are place holders for data, in the way that logic programming variables also are. Variable restrictions can also be specified, by writing `var X -> p` (read *as*), which restrict the bindings of the variables to those terms that are matched by the restriction pattern `p`.

Example. An Xcerpt query term that queries the data given in Figure 2 for flights departing from Paris on 2005-08-21.

```
flights {{
  var F -> flight {{
             from { "Paris" },
             date { "2005-08-21" }
  }}
```

Query terms are matched against event data or Web resource data, by means of a novel unification method called Simulation Unification [16,15], which can handle querying constructs such as partial specifications, optional subterms, or negation of subterms. Informally, a query term q simulation unifies (or simply matches) a data term d, if q's structure can be found in d. The outcome of simulation unifying q and d, is a set of substitutions for the variables in q. XChange event queries (event part) and Web queries (condition part) are based on query terms and find substitutions for the variables that are then subsequently used in the action part (event raising or transaction specification) of a rule.

Update patterns, which are used in the action part of XChange rules, extend query patterns: an update specification is a (possibly incomplete) pattern for the data to be updated, augmented with the desired update operations (insert, replace, delete). An update pattern may contain different types of update operations. An *insertion operation* specifies a construct term (i.e. a total pattern specification that makes use of variables for constructing new data) that is to be inserted. A *deletion operation* specifies a query term for deleting all data terms matching it. A *replace operation* specifies a query term to determine data terms to be modified and a construct term as their new value. A more detailed discussion on XChange update capabilities can be found in [5].

```
<xchange:event xmlns:xchange="http://pms.ifi.lmu.de/xchange">
   <xchange:sender>         http://airline.com          </xchange:sender>
   <xchange:recipient>      http://organiser.org/smith/ </xchange:recipient>
   <xchange:raising-time>   2005-08-21T12:00:25         </xchange:raising-time>
   <xchange:reception-time> 2005-08-21T12:00:55         </xchange:raising-time>
   <xchange:id>             4711                        </xchange:id>
   <cancellation>
      <flight-number> AI2021     </flight-number>
      <date>          2005-08-21 </date>
   </cancellation>
</xchange:event>
```

Fig. 3. XChange Event Message

3 Event Queries

XChange uses *Event Queries* to detect whether some situation requiring a reaction has occurred. Event queries serve a double purpose: they specify classes of events to indicate when a certain rule should fire, and they extract data from the events (as variable substitutions), that will be used in the subsequent condition- and action-part when the rule fires. This dual purpose of event queries sets XChange apart from related work on reactivity and in particular composite events.

An XChange *event query* may be *atomic* or *composite*. An *atomic event query* specifies a pattern for a *single* incoming event that is of interest. A *composite event query* specifies a situation of interest that is given not by a single atomic event query, but a temporal combination of several atomic events.

3.1 Events and their Representation

Events considered in XChange include updates to Web data, timer events, transactional events, and application-specific events such as the flight cancellation from above.

Event messages communicate events between (the same or different) Web sites. An event message is an envelope for arbitrary XML content; it is an XML document with root element `event`, arbitrary content, and in addition five fixed elements: `raising-time` (i.e. the time of the event manager of the Web site raising the event), `reception-time` (i.e. the time at which a site receives the event), `sender` (i.e. the URI of the site where the event has been raised), `recipient` (i.e. the URI of the site where the event has been received), and `id` (i.e. an identifier given at the recipient Web site). Figure 3 gives an event message for notifying a travel organiser of a flight cancellation. Specifications of time points follow ISO 8601 format.

3.2 Atomic Event Queries

An *atomic event query* is a pattern for the representation of a single event of interest (i.e. for an event message). Patterns can be accompanied by an optional *absolute temporal restriction*, to restrict the event instances that are considered relevant for the event query, to those that have occurred (or more precisely, their representations have been received) in the specified time interval. XChange absolute time restrictions can be specified by means of a fixed starting and ending point (i.e. a finite time interval), following the keyword `in`. The starting point of such a restricting interval may be implicit, e.g. the time point of event query registration, in which case the ending time point follows the keyword `before`.

Example. An XChange atomic event query that detects insertion of discounts for flights from Munich to Paris that are received as notifications before 7th of July 2005 is given next.

```
xchange:event {{
  flight {{
    from {"Munich"}, to {"Paris"},
    new-discount { var D }
   }}
 }} before 2005-07-07T10:00:00
```

3.3 Composite Event Queries

The capability to detect and react to *composite events*, i.e. sequences of event instances that have occurred possibly at different Web sites within a specified time interval, is needed for many Web-based reactive applications. However, to the best of our knowledge, existing languages for reactivity on the Web do *not* consider the issues of detecting and reacting to such composite events. [1] One of the novelties introduced by XChange is the detection of *composite events*. Composite events are defined exclusively as answers to *composite event queries*.

Composite event queries are specified by means of atomic event queries combined using XChange composite event query constructs. XChange offers a considerable number of such constructs along two dimensions: *temporal restrictions* and *event compositions*. This section introduces the constructs for temporal restrictions and the core constructs for event compositions.

Note that a *composite event* does not have a single occurrence time, as an atomic events does. Instead, a composite event inherits from its components a

[1] [3] considers “composite events”. However, this notion refers in [3] to updates of several elements of a single XML document. The XChange notion of composite events goes beyond such updates of an XML document.

beginning time (the reception time of the first received constituent event that is part of the composite event) and an ending time (the reception time of the last received constituent event that is part of the composite event). That is, in XChange composite events have a *duration* (a length of time).

3.3.1 Temporal Restriction

As for atomic event queries, temporal restrictions can be specified also for composite event queries, placing temporal restrictions on the answers' constituent events. Besides absolute temporal restrictions, *relative temporal restrictions*, given by a duration, can also be specified for composite event queries. Relative temporal restrictions can be given as positive numbers of years, days, hours, minutes, or seconds and their specification follows the keyword `within`.

In XChange, every (legal) event query must have a temporal restriction. This makes it possible to release each (atomic or semi-composed composite) event after a finite time. Thus, language design ensures that volatile data remains volatile and storage requirements for events are kept (in practice) constant.

3.3.2 Event Composition

Composition operators can be used to express a temporal pattern over atomic event occurrences that signifies a situation of interest.

Conjunctions specify that instances of each of the specified event queries need to be detected in order to detect the conjunction event query; the order in which event query instances occur is not of importance (denoted by curly braces). The keyword `and` introduces such a composite event query in XChange.

Example. Mrs. Smith wants to visit an exhibition of G. Barthouil on a rainy day. The following XChange event query is used to detect the conjunction of the exhibition notification and the desired weather forecast notification that are sent by appropriate Web services.

```
and {
  xchange:event {{
    xchange:sender {"http://artactif.com"},
    exhibition {{ painter {"G. Barthouil"},
                  location {"Marseilles"},
                  date { var D } }}
  }},
  xchange:event {{
    xchange:sender {"http://weather.com"},
    forecast {
      date { var D }, city {"Marseilles"},
      info {"It's going to rain."} }
  }}
} before "2005-11-11T20:15"
```

Temporally ordered conjunctions specify that the occurrences of component event queries' instances need to be successive in terms of time. The keyword `andthen` introduces such an event query whose component event queries are enclosed in square brackets. A total specification (i.e. single square brackets) expresses that the answer to such a composite event query contains

only the instances of the component event queries. In contrast, a partial specification (i.e. double square brackets) expresses that the answer contains also all events that have occurred in-between.

Example. The following XChange event query is used to detect the notification of a flight cancellation and afterwards, within two hours from its reception, the detection of a notification informing that the accommodation is not granted by the airline.

```
andthen [
  xchange:event {{
    xchange:sender {"http://airline.com"},
    cancellation-notification {{
     flight {{ number { var Number } }} }}
   }},
  xchange:event {{
    xchange:sender {"http://airline.com"},
    accommodation {"Not granted!"}
   }}
 ] within 2 hour
```

Inclusive disjunctions specify that the occurrence of an instance of any of the specified event queries suffices for detecting the disjunction event query. The keyword `or` denotes a disjunction in XChange and the event queries are enclosed in curly braces.

Example. After Orange, Mrs. Smith wants to visit Arles and Nîmes. The next city to visit is chosen depending on the notification of train tickets and hotel reservation made by appropriate services.

```
or {
  xchange:event {{
    xchange:sender {"http://nimes.fr"},
    service-notification {{
      train {{ date { var D },
               from {"Orange"}, to {"Nimes"}
        }}   }}
   }},
  xchange:event {{
    xchange:sender {"http://arles.fr"},
    reservation-notification {{
      train {{ date { var D },
               from {"Orange"}, to {"Arles"}
        }}   }}
   }}
 } before 2005-05-02T21:30:00
```

Exclusions specify that no instance of the given event query should have occurred in a time interval in order to detect the exclusion event query. Such a time interval is given by a finite time interval or by a composite event query (recall that their instances have a beginning and an ending time and thus determine a time interval). The keyword `without` introduces exclusion of event queries in XChange.

Example. An XChange event query that detects if the notification of an online reservation made on 10th of July 2005 is not received within ten days.

```
without {
  xchange:event {{
     reservation-notification {{ }}
    }}
 } during [2005-07-10..2005-07-20]
```

Occurrences constructs for event queries refer to the number of times an event query instance should occur or should be repeated to be of interest, or to the position that events of interest should have in the incoming event stream. The occurrences constructs supported by XChange (and explained in the following) are *quantifications*, *repetitions*, and *ranks*.

Quantifications in event queries are used to detect instances that occur

(at least, at most, or exactly) a number of times in a given time interval or between occurrences of other event query instances. The keyword `times` introduces such composite event queries in XChange.

Example. The travel organiser's event query given next is used to detect if Mrs. Smith receives at least three important messages from her secretary during a given time interval.

```
atleast 3 times {
   xchange:event {{
     secretary-message {{
        important {{ }} }}
    }}
 } during [2005-08-21..2005-08-22]
```

Repetitions are used for detecting e.g. every second, forth, sixth, and so on, instances of a specified event query in a given time interval or between occurrences of other event query instances. The keyword `every` introduces such event queries in XChange.

Example. Mrs. Smith wants to slowly quit smoking so she answers only to every second call from her colleague suggesting a smoking break (specified in XChange using the next event query).

```
every 2 {
  xchange:event {{
    xchange:sender {http://lmu.de/werner/},
      break-for-a-smoke {{
        info {"Join me for a cigarette!"}
      }}
   }}
  } within workday
```

Note that time intervals can be given as union of finite time intervals, thus periodical temporal specifications are also allowed in XChange. Here, `workday` denotes a temporal type defined using e.g. the Calendar and Time Type System CaTTS [6].

Ranks are used to detect instances of a specified event query having a given rank (or position) in the incoming stream of events. They are useful in specifying interest in the first or the last instance of an event query. The keywords `withrank` and `last` introduce such event queries in XChange.

Other composite event constructs are also supported by XChange. For example, the *multiple inclusions and exclusions* construct is used to detect occurrences of a given number of event query instances and the non-occurrence of instances of the other specified event queries. It expresses a generalised exclusive disjunction of event queries. *Overlapping* and *meet* constructs for composite event queries detect instances of the specified event query if their component (composite) events overlap or meet, respectively, on the time axis of the incoming events.

XChange constructs for composite event queries can be nested arbitrarily; thus, complex reactive applications can be easily and elegantly implemented in XChange. Figure 4 gives a composite event query for detecting occurrences of a flight cancellation, where the airline does *not* grant an accommodation. For this purpose, a temporal ordered conjunction construct, the exclusion

```
andthen [
    xchange:event {{
      xchange:sender { "http://airline.com" },
      cancellation-notification {{
          flight {{ number { "AI2021" }, date { "2005-08-21" }  }}
          }}
       }},
    without { xchange:event {{
              xchange:sender { "http://airline.com" },
              accomodation-granted {{ hotel {{ }} }}  }}
            } during [2005-08-21T17:00..2005-08-21T19:00]
] within 2 hour
```

Fig. 4. Nesting Composite Event Query Constructs

```
<xchange:event-seq>
    <xchange:event>
        <xchange:sender>    http://lmu.de/secretary </xchange:sender>
        <xchange:recipient> http://lmu.de/smith     </xchange:recipient>
        <xchange:raising-time>   2005-08-21T13:00 </xchange:reception-time>
        <xchange:reception-time> 2005-08-21T13:01 </xchange:reception-time>
        <xchange:reception-id>   42 </xchange:reception-id>
        <secretary-message>
            <important/>
            <text> Urgent call from Werner </text>
        </secretary-message>
    </xchange:event>
    <xchange:event>
        <xchange:sender>    http://lmu.de/secretary </xchange:sender>
        ...
    </xchange:event>
    <xchange:event>
        <xchange:sender>    http://lmu.de/secretary </xchange:sender>
        ...
    </xchange:event>
</xchange:event-seq>
```

Fig. 5. XML Representation of a Composite Event

construct and temporal restrictions are combined.

3.3.3 Answers to Event Queries

For determining answers to *atomic event queries*, the event manager of an XChange-aware Web site attempts to *match* each incoming event received, with the currently posed atomic event queries (which themselves may be part of composite event queries). The matching of an atomic event query with an incoming event is based on the *simulation unification* [15], a novel unification method developed for matching query terms (i.e. queries or patterns to XML and other data) with data terms (i.e. XML and other data). The same method is used in evaluating the Web queries in the condition part of rules, reducing learning effort for users. Section 4.2 discusses in more detail the algorithm used for the evaluation of composite event queries.

An answer to an atomic event query — an *atomic event* — is an event whose representation (as event message) matched the event query (and occurred in the given time interval, if a temporal restriction has been specified). Thus, the representation of an answer to an atomic event query is an event message – an XML document.

An answer to a composite event query — a *composite event* — should also be representable as an XML document, just like atomic events; this allows further

processing of composite events at local and remote Web sites. We choose a (flat) sequence (with an artificial root to make valid XML) of all atomic events that were used for answering the composite event query. Figure 5 shows an answer to the composite event query given as example for quantifications.

We have also investigated other approaches for representing answers, e.g., XML representations mirroring the nested structure of a composite event query. However, we found a flat sequence better. It is simpler and more intuitive for users, since no knowledge of the query structure is required. It leads to an easier definition of declarative semantics (presented in the next section), due to the similarity between sets and sequences. Finally, we believe it is desirable for a query language to have similar input and output — and the input of event queries is a collection of atomic events arriving sequentially. In principle, this allows using the answer to a composite event query as the input to another event query.

4 Semantics of Event Queries

Comparisons of (composite) event query languages such as [20], show that interpretation of similar language constructs can vary considerably. To avoid misinterpretations, clear semantics are indispensable. We next present a declarative semantics for XChange.

4.1 Declarative Semantics

Declarative semantics are not only beneficial to avoid misinterpretations of language constructs by both users and implementors; they also provide a basis for formal proofs of language properties, help us understand the language design and promote the construction of optimisations. We define a declarative semantics for XChange's event query language as a ternary relation between event queries, answers (i.e. composite events), and the stream of incoming event messages.

In the following, we draw particular attention to the following semantic features, that distinguish XChange from other related work on events.

- Event query specifications may contain features such as free variables and partial matches. The answers to composite event queries may contain complex bindings for such variables.
- The answer to an event query is a sequence and may include (atomic) events not referred to in the query. This is particularly useful since it allows pow-

erful composition of event queries (i.e. A second event query can be applied to the sequence(s) returned by the first event query).

Answers An answer to an event query q is a tuple (s, Σ). It consists of a (finite) sequence s of atomic events happening in a time interval $[b..e]$ that allowed a successful evaluation of q and a corresponding set of substitutions Σ for the free variables of q. We write $s = \langle a_1, \ldots a_n \rangle_b^e$ to indicate that s begins at time point $begin(s) := b$, ends at $end(s) := e$, and contains the atomic events $a_i = d_i^{r_i}$, which are data terms d_i received at time point r_i. We have $b \leq r_1 < \ldots < r_n \leq e$; note that $b < r_1$ and $r_n < e$ are possible.

Observe that the answer is an event sequence, and it is possible for instances of events not specified in the query to be returned. For example, a partial match `andthen[[a,b]]` returns not only event instances of `a` and `b` to be returned, but also instances all atomic events happening between them. This cannot be captured with substitutions alone.

Substitution Sets The substitution set Σ contains substitutions σ (partial functions) assigning variables to data terms. Assuming a standardisation of variable names, let V be the set of all free variables in a query having at least one defining occurrence. A variable's occurrence is *defining*, if it is part of a non-negated sub-query, i.e. does not occur inside a `without`-construct, and thus can be assigned a value in the query evaluation. Let $\Sigma \mid_V$ denote the restriction of all substitutions σ in Σ to V. For triggering rules in XChange, we are interested only in the maximal substitution sets.

Event Stream For a given event query q, all atomic events received after its registration form a *stream of incoming events* (or, *event stream*) $\mathcal{E}$. Events prior to a query's registration are not considered, as this might require an unbounded event life-span. Thus, since it fits better with the incremental event query evaluation (described in the next section), we prefer the term "stream" to the term "history" sometimes used in related work. Formally, $\mathcal{E}$ is an event sequence (as s above) beginning at the query's registration time.

Answering-Relation Semantics of event queries are defined as a ternary relation between event queries q, answers (s, Σ), and event stream $\mathcal{E}$. We write $q \lhd_{\mathcal{E}} (s, \Sigma)$ to indicate that q is answered by (s, Σ) under the event stream $\mathcal{E}$. Definition of $\lhd_{\mathcal{E}}$ is by induction on q, and we give only a few exemplary cases here.

q is an atomic event query: $q \lhd_{\mathcal{E}} (s, \Sigma)$ if and only if (1) $s = \langle d^r \rangle_r^r$, (2) d^r is an atomic event in the stream $\mathcal{E}$, (3) the data term d simulation unifies ("matches") with the query q under all substitutions in Σ. For a formal account of (3) see work on Xcerpt [15].

$q = \texttt{and}[q_1, \ldots, q_n]$: $q \lhd_{\mathcal{E}} (s, \Sigma)$ iff there exist event sequences $s_1, \ldots s_n$ such

that (1) $q_i \lhd_{\mathcal{E}} (s_i, \Sigma)$ for all $1 \leq i \leq n$, (2) s comprises all event sequences $s_1, \ldots s_n$ (denoted $s = \bigcup_{1 \leq i \leq n} s_i$).

$q =$ `andthen`$[[q_1, q_2]]$: $q \lhd_{\mathcal{E}} (s, \Sigma)$ iff there exist event sequences s_1, s', and s_2 such that (1) $q_i \lhd_{\mathcal{E}} (s_i, \Sigma)$ for $i = 1, 2$, (2) $s = s_1 \cup s' \cup s_2$, (3) $end(s_1) \leq begin(s_2)$, and (4) s' is a continuous extract of $\mathcal{E}$ (denoted $s' \sqsubset \mathcal{E}$) with (5) $begin(s') = end(s_1)$ and $end(s') = begin(s_2)$. The event sequence s' serves to collect all atomic events happening "between" the answers to q_1 and q_2 as required by the partial matching `[[ ]]`. The n-ary variant of this binary `andthen` is defined by rewriting the n-ary case associatively to nested binary operators.

$q =$ `without` $\{q_1\}$ `during` $\{q_2\}$: $q \lhd_{\mathcal{E}} (s, \Sigma)$ iff (1) $q_2 \lhd_{\mathcal{E}} (s, \Sigma)$, (2) there is no answer (s_1, Σ_1) to q_1 ($q_1 \lhd_{\mathcal{E}} (s_1, \Sigma_1)$) such that Σ contains substitutions for the variables V with defining occurrences that are also in Σ_1 ($\Sigma \mid_V \subseteq \Sigma_1 \mid_V$).

$q = q'$ `within` w: $q \lhd_{\mathcal{E}} (s, \Sigma)$ iff (1) $q' \lhd_{\mathcal{E}} (s, \Sigma)$ and (2) $end(s) - begin(s) \leq w$.

Discussion Our answering relation approach to semantics allows the use of advanced features in XChange's event query language, such as free variables in queries, event negation, and partial matches. Note that due to the latter two, approaches where answers are generated by a simple application of substitutions to the query would be difficult, if not impossible to define.

Our declarative semantics provide a sound basis for formal proofs about language properties. We have used it for proving that, in order to evaluate any legal event query q at some time t correctly, only events of bounded life-span are necessary; that is, it suffices to consider the restriction $\mathcal{E} \mid_{t-\beta}^{t}$ of the event stream $\mathcal{E}$ to a time interval $[(t - \beta) .. t]$. The time bound β (a length of time) is only determined from q and does not depend on the incoming events $\mathcal{E}$. Formally, we have proved the requirement that for all legal event queries q there exists a time bound β, such that for all time points t, all event streams $\mathcal{E}$, and all answers (s, Σ) with $end(s) = t$ we have: $q \lhd_{\mathcal{E}} (s, \Sigma) \iff q \lhd_{\mathcal{E}\mid_{t-\beta}^{t}} (s, \Sigma)$.

4.2 Operational Semantics: Event Query Evaluation

Evaluation of event queries should be performed in an incremental manner: work done in one evaluation step of an event query on some incoming atomic event should not be redone in future evaluation steps on further incoming events. To evaluate an XChange composite event query in an incremental manner, we store all partial evaluations in the query's operator tree. Leaf nodes in the operator tree implement atomic event queries, inner nodes implement composition operators and time restrictions. When an event message is received, it is injected at the leaf nodes; data in the form of event query

```
SetOfCompositeEvents evaluate( AndNode n, AtomicEvent a ) {
    // receive events from child nodes
    SetOfCompositeEvents newL := evaluate( n.leftChild, a );
    SetOfCompositeEvents newR := evaluate( n.rightChild, a );

    // compose composite events
    SetOfCompositeEvents answers := ∅;
    foreach ((s_L, Σ_L), (s_R, Σ_R)) ∈(newL × n.storageR) ∪
                                     (n.storageL × newR) ∪
                                     (newL × newR) {
            SubstitutionSet Σ := Σ_L ⋈ Σ_R;
            if (Σ ≠ ∅) answers := answers ∪ new CompositeEvent( s_L ∪ s_R, Σ );
    }

    // update event storage
    n.storageL := n.storageL ∪ newL;
    n.storageR := n.storageR ∪ newR;

    // forward composed events to parent node
    return answers;
}
```

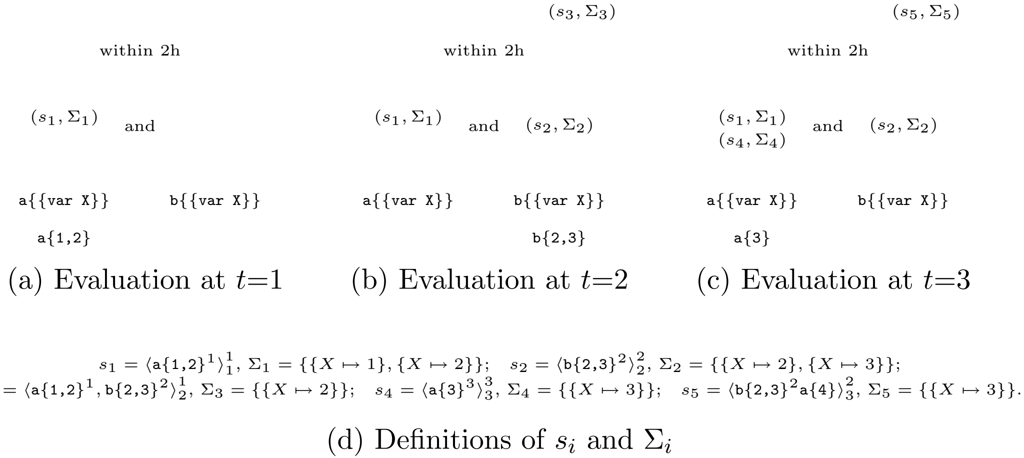
Fig. 6. Implementation of a (binary) `and` inner node in pseudo-code

answers (s, Σ) (cf. previous section) then flows bottom-up in the operator tree during this evaluation step. Inner nodes can store intermediate results to avoid recomputation when the next evaluation step is initiated by the next incoming event message.

Leaf nodes process an injected event message by trying to match it with their atomic event query (using Simulation Unification). If successful, this results in a substitution set $\Sigma \neq \emptyset$, and the answer (s, Σ), where s contains only the one event message, is forwarded to the parent node. Inner nodes process composite events they receive from their child nodes following the basic pattern:

(1) attempt to compose composite events (s, Σ) (according to the operator the inner node implements) from the stored and the newly received events,
(2) update the event storage by adding newly received events that might be needed in later evaluations,
(3) forward the events composed in (1) to the parent node.

Figure 6 sketches an implementation for the evaluation of a (binary) `and` inner node in java-like pseudo-code. Consider it in an example of evaluating the event query q =`and{ a{{var X}}, b{{var X}} } within 2h` (atomic event queries are abbreviated for notational convenience) in Figure 7. For simplicity,



(a) Evaluation at t=1 (b) Evaluation at t=2 (c) Evaluation at t=3

$s_1 = \langle \texttt{a\{1,2\}}^1 \rangle_1^1$, $\Sigma_1 = \{\{X \mapsto 1\}, \{X \mapsto 2\}\}$; $s_2 = \langle \texttt{b\{2,3\}}^2 \rangle_2^2$, $\Sigma_2 = \{\{X \mapsto 2\}, \{X \mapsto 3\}\}$;
$s_3 = \langle \texttt{a\{1,2\}}^1, \texttt{b\{2,3\}}^2 \rangle_2^1$, $\Sigma_3 = \{\{X \mapsto 2\}\}$; $s_4 = \langle \texttt{a\{3\}}^3 \rangle_3^3$, $\Sigma_4 = \{\{X \mapsto 3\}\}$; $s_5 = \langle \texttt{b\{2,3\}}^2 \texttt{a\{4\}} \rangle_3^2$, $\Sigma_5 = \{\{X \mapsto 3\}\}$.

(d) Definitions of s_i and Σ_i

Fig. 7. Incremental evaluation of an event query using bottom-up data flow in a storage-augmented operator tree

we let event messages arrive at time points $t = 1, 2, 3$ that are one hour apart; this is of course not the normal case in practice and not an assumption made by the algorithm.

Figure 7(a) depicts receiving the event message `a{1,2}` at time $t = 1$. The event message does not match with the atomic event query `b{{var X}}` (right leaf in the tree). But it does match with the atomic event query `a{{var X}}` (left leaf) with substitution set Σ_1 and is propagated upwards in the tree as answer (s_1, Σ_1) to the parent node `and` (Figure 7(d) defines s_i and Σ_i). The `and`-node cannot form a composite event from its input, yet, but it stores (s_1, Σ_1) for future evaluation steps.

At $t = 2$ we receive the event message `b{2,3}` (Figure 7(b)); it matches the right leaf node and (s_2, Σ_2) is propagated to the `and`-node. The `and`-node stores (s_2, Σ_2) and tries to form a composite event (s_3, Σ_3) from (s_1, Σ_1) and (s_2, Σ_2). Σ_3 is computed as a (variant of a) natural join ($\perp$ denotes undefined):
$\Sigma_3 = \Sigma_1 \bowtie \Sigma_2 =$
$\{\sigma_1 \cup \sigma_2 \mid \sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2, \forall X.\ \sigma_1(X) = \sigma_2(X) \vee \sigma_1(X) = \perp \vee \sigma_2(X) = \perp\}$.
Σ_3 now contains all substitutions that can be used simultaneously in all atomic event queries in `and`'s subtree. $\Sigma = \emptyset$ would signify that no such substitution exists and thus no composite event can be formed. In our case however there is exactly one substitution $\{X \mapsto 2\}$ and we propagate (s_3, Σ_3) to the `within 2h`-node. This node checks that $end(s_3) - begin(s_3) = 1 \leq 2$ and pushes (s_3, Σ_3) up (there is no need to store it). With this (s_3, Σ_3) reaches the top and we have our first answer to the event query q.

Figure 7(c) shows reception of another event message `a{3}` at $t = 3$, which results in another answer (s_5, Σ_5) to q.

After the query evaluation at $t = 3$, we can release (delete) the stored answer (s_1, Σ_1) from the operator tree: any composite event formed with use of (s_1, Σ_1) will not pass the `within 2h`-node. Event deletion is performed by top-down traversal of the operator tree. Temporal restriction operator nodes put restrictions on $begin(s)$ and $end(s)$ for all answers (s, Σ) stored in their subtrees. In our example, all events (s, Σ) in the subtree of `within 2h` must satisfy $t - 2 \leq begin(s)$, where t is the current time.

The idea to prove correctness of the incremental algorithm w.r.t. the declarative semantics is by dividing the problem into two: We first forget that the algorithm is incremental and stores events; to detect an event at a time point t we pretend that all incoming events are processed in one single evaluation. Then we prove that the operator tree will always have stored the right events, that is, at time point t it stores all events that can be constituting part of a composite event with occurrence time t or later. This requires checking that in the bottom-up data flow we store all needed events and that in the event deletion we do not delete needed ones.

5 Related Work

A number of active database prototypes have been built providing sophisticated event algebras (e.g. SNOOP [8], REACH [21], CHIMERA [11], Rock&Roll [9] and ODE [10]). The PFL system [14] uses the functional paradigm and includes active rules with event queries. Its philosophy of treating events as queries (rather than simple algebraic expressions) that can return complex bindings is shared by XChange. Work in [20] provides a meta-model for classifying a number of properties of composite event formalisms for active rules. Recent work in [1], outlines a situation monitoring system and expands upon much of the work in the active database literature. Unlike XChange, these works are oriented towards a centralised system, as opposed to a distributed one like the Web and do not focus on XML as the data format. One work that does consider events in a distributed environment is [19], which considers in detail the influence of (distributed) timestamps on event specification. This direction is complementary to our approach and we intend to evaluate in future work the ways in which this kind of detail may be specified in XChange event queries.

One related work that considers events for XML is [12]. This presents monitoring and subscription in Xyleme, an XML warehouse supporting subscription to web documents. A set of *alerters* monitor simple changes to web documents. A *monitoring query processor* then performs more composite event detection and sends notifications of events to a *trigger engine* which performs the necessary actions, including creating new versions of XML documents. The focus of

this reactive functionality is highly tuned to this specific application. A recent work which considers composite events for XML is [3], which does so in the context of updates to a single XML document. In contrast, XChange allows combinations of events that are not necessarily updates and which do not need to be associated with a particular document.

In summary, there are several features present in XChange that distinguish it from other related work.

- Events are represented as XML documents and consequently may have a nested structure, to which pattern matching can be applied within event queries.
- XChange is not only concerned with detecting events. It places special emphasis on the extraction of data from events, in the form of variable assignments. Event queries may include both partial matches and free variables. Answers to event queries are sequences, allowing powerful query composition.
- The evaluation algorithm for composite events must take into account the extraction of variable assignments. This makes it rather different from the evaluation methods used by, e.g. SNOOP [8].
- There is a clear separation between events as volatile data versus the persistent data which can be queried by the user. This in turn has implications for the kinds of rules that can be defined.

6 Conclusion

This article has presented the high-level language XChange for realising reactive Web applications. XChange is a language of Event-Condition-Action rules: The Event part queries incoming event messages (XML documents) to detect (composite) events an to extract data from them for use in the subsequent rule parts. The Condition part queries Web resources' data. The Action part specifies event messages to be raised and updates to be executed. This article improves upon work in [5] by focusing on querying events; we have presented language constructs for detecting composite events together with their declarative and operational semantics.

XChange introduces a novel view over the Web data by stressing a clear separation between *persistent data* (data of Web resources, such as XML or HTML documents) and *volatile data* (event data communicated on the Web between XChange programs). XChange's language design enforces this clear separation and entails new characteristics (such as deleting (atomic or semi-composed composite events) after a bounded time) of event processing on the Web.

XChange is an ongoing research project. At present, the design of an extended core language for XChange is completed. For implementing the language XChange and specifying its semantics, a modular approach that mirrors the three components of rules is followed. A prototype for the *event query* evaluation has been developed and a declarative semantics for the event language has been specified. A first version of Xcerpt, the language used in XChange for expressing *Web queries*, is fully designed and a reference implementation is available (cf. http://xcerpt.org). The declarative semantics of Xcerpt is formalised in [7,15]. The implementation and the semantics' specification of the *action part* is under development, and we have started working on use cases demonstrating XChange's practical usage.

References

[1] A. Adi and O. Etzion. Amit – The Situation Manager. *Very Large Data Bases Journal*, 13(2):177–203, 2004.

[2] J. Bailey, F. Bry, and P.-L. Pătrânjan. Composite Event Queries for Reactivity on the Web. In *Poster Proceedings:Proc. of 14th Int. World Wide Web Conference*, page To appear, Chiba, Japan, May 2005. ACM.

[3] M. Bernauer, G. Kappel, and G. Kramler. Composite Events for XML. In *Proc. of 13th Int. World Wide Web Conference*, pages 175–183, New York, USA, May 2004. ACM.

[4] F. Bry, T. Furche, P.-L. Pătrânjan, and S. Schaffert. Data Retrieval and Evolution on the (Semantic) Web: A Deductive Approach. In *Workshop on Principles and Practice of Semantic Web Reasoning at 20th Int. Conference on Logic Programming*, pages 34–49, Saint Malo, France, 2004. Springer.

[5] F. Bry and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *Proc. of 20th Annual ACM Symposium on Applied Computing*, Santa Fe, New Mexico, March 2005. ACM Press.

[6] F. Bry, F.-A. Rieß, and S. Spranger. CaTTS: Calendar Types and Constraints for Web Applications. In *Proc. of 14th Int. World Wide Web Conference, Chiba, Japan*, page To appear. ACM, 2005.

[7] F. Bry, S. Schaffert, and A. Schröder. A contribution to the Semantics of Xcerpt, a Web Query and Transformation Language. In *Proc. of 18th Workshop on (Constraint) Logic Programming*, pages 258–268, Potsdam, Germany, 2004. GLP, GI.

[8] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[9] Andrew Dinn, Norman W. Paton, M. Howard Williams, and Alvaro A. A. Fernandes. An active rule language for rock & roll. In *BNCOD*, pages 36–55, 1996.

[10] N. Gehani, H.V. Jagadish, and O. Shmueli. Event Specification in an Active Object-Oriented Database. In *ACM SIGMOD Int. Conference on Management of Data*, San Diego, 1992.

[11] R. Meo, G. Psaila, and S. Ceri. Composite Events in Chimera. In P.M.G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Proc. of 5th Int. Conference on Extending Database Technology*, pages 56–76. Springer, 1996.

[12] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML Data on the Web. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 437–448. ACM Press, 2001.

[13] N. W. Paton, editor. *Active Rules in Database Systems*. Springer, 1999.

[14] Swarup Reddi, Alexandra Poulovassilis, and Carol Small. Pfl: An active functional dbpl. In *Active Rules in Database Systems*, pages 297–308. 1999.

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

[16] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. of Int. Conference Extreme Markup Languages*, Montreal, Quebec, Canada, August 2004.

[17] R. Schulte. The growing role of events in enterprise applications. Technical report, Gartner Research Report AV-20-3900, 9 July 2003.

[18] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[19] S. Yang and S. Chakravarthy. Formal Semantics of Composite Events for Distributed Environments. In *Proc. of 15th Int. Conference on Data Engineering*, pages 400–407, Australia, 1999. IEEE Computer Society.

[20] D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proc. of 15th Int. Conference on Data Engineering*, pages 392–399, Australia, 1999. IEEE Computer Society.

[21] J. Zimmermann and A.P. Buchmann. REACH. In *Active Rules in Database Systems*, pages 263–277. Springer, 1999.