Composite Event Queries for Reactivity on the Web

James Bailey
Dept. of Computer Science
University of Melbourne
Victoria, 3010
Australia
jbailey@cs.mu.oz.au

François Bry
Institute for Informatics
University of Munich
Oettingenstr. 67
D-80538 Munich
Germany
francois.bry@ifi.lmu.de

Paula-Lavinia Pătrânjan
Institute for Informatics
University of Munich
Oettingenstr. 67
D-80538 Munich
Germany
patranja@pms.ifi.lmu.de

ABSTRACT

Reactivity on the Web is an emerging issue. The capability to automatically react to events (such as updates to Web resources) is essential for both Web services and Semantic Web systems. Such systems need to have the capability to detect and react to complex, real life situations. This paper introduces the high-level language *XChange*, for programming reactive behaviour on the Web, emphasising the capability of the language to express complex situations that occur on the Web by means of *composite event queries*.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*

Keywords

Web, reactive languages, event-condition-action rules, composite events

1. INTRODUCTION

Reactivity on the Web is gaining importance and is recognised as a solution for many everyday life problems. For example, a Web Service provided by an airline could report delays on departures or arrivals, and flight cancellations. A Web-based personalised organiser might be conceived so as to automatically react to (possibly combinations of) reports which affect its owner. Such reports can be sent from different Web Services (like a weather forecast service). A delayed arrival might cause either an email to be sent to some other person or the cancellation of a hotel reservation.

Reactive languages (such as [12]) formerly developed for the Web support *simple* update operations on XML documents, i.e. there is no support for specifying and executing (two or more) updates in a desired order and in an *all-or-nothing* manner. Moreover, these languages have the capability to react only to single event instances and do not provide constructs for querying for complex combinations of event instances.

[0]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 (cf. `http://rewerse.net`).

WWW2005, May 10–14, 2005, Chiba, Japan.

The issue of reacting to so-called *complex events*, i.e. (possibly time-related) combinations of event instances, has received considerable attention in the field of active databases (cf. e.g. [13, 19]). Thus, 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 central clock and a central management are missing, necessitate new approaches. In particular, complex events reflecting a user-centered (and not a system-centered) view are needed for the Web. One such approach is proposed by the language *XChange* [5] and is presented in this article. XChange builds upon the Web query language Xcerpt [15, 3] and provides constructs for detecting complex (or composite) events on the Web. The principles that led to the design of these constructs are: i) reflecting the temporal order of events through syntactical representation, and ii) having a collection of constructs convenient for a natural expression of common-sense reasoning with events on the Web.

Preliminary work on XChange is introduced in [5]. As the development stage of the language was not been mature enough regarding the issue of composite events on the Web, just some of the XChange constructs were briefly introduced. This paper explains the XChange composite event queries for detecting composite events in more detail (introduces also new constructs), discusses the processing of event queries and examins the notion of answers to an event query.

2. EVENTS AND EVENT QUERIES FOR REACTIVITY ON THE WEB

2.1 Atomic Events

Informally, an *atomic event* is a happening (e.g. an update of a possibly remote Web resource) to which each Web site (through a reactive program) may decide to react in a particular way or not to react to at all. XChange distinguishes between two kinds of atomic events: *explicit* events and *implicit* events. *Explicit events* are explicitly raised by a user or by a (predefined) XChange program. They are raised at a Web site and sent internally or to other Web sites through *event messages*. *Implicit events* are local events not expressed through event messages (e.g. local updates of data or system clock events). Events are transmitted from one Web site to another through event messages. Thus, an event sent from one Web site to another is necessarily explicit.

The kinds of *events* considered in XChange are presented

Table 1: XChange Events

local events	explicit events (event messages)	
	implicit	updates
		queries
		transactional events
		system events
remote events	explicit events (event messages)	

in Table 1. An *update* executed or a *query* posed locally at a Web site are for XChange local events, i.e. raised at this Web site and processed at this Web site. *Transactional events* (transaction commit, transaction abort, transaction request) are local events needed as XChange supports the concept of transactions (cf. Section 5). *System events* (e.g. system clock events) are events that are coming from the encompassing "system" and might be useful to handle together with explicit and/or implicit events. A system event might be explicit or implicit, depending whether or not it is transmitted from one Web site to another.

Remote events, i.e. events informing a Web site of queries, updates, transactional or system events or of any other (application specific) matter, are always explicit and are expressed through event messages.

2.2 Composite Events

XChange's (occurrences of) *composite events* are defined as answers to *composite event queries* (see Section 3). E.g. an XChange event query can ask for occurrences of an increase of share values by more than 5 percent for the company Siemens, followed by an increase of share values for the company SAP at a particular stock market. An answer to such an event query contains instances of the two specified component event queries (i.e. increase of share values). Another XChange event query can ask for *all* stock market reports that have been registered between the occurrences of an increase of share values for the two mentioned companies. An answer to such an event query contains, besides the instances of the events signaling an increase for the shares of the companies, all reports registered between these two instances. The capability to query for all events having a particular pattern that have occurred between the instances of two specified event queries is one of the novelties of the language XChange.

2.3 Event Query vs. Web Query

Volatile vs. Persistent Data. An important distinction is made between *persistent data* (data of Web resources) and *volatile data* (events). To query persistent data, *standard queries* are used (expressed using a query language like XQuery [17], or Xcerpt [15]). To query volatile data, *event queries* are used. Standard and event queries can be very similar. However, event queries are more likely to refer to time or event sequences.

Incremental Aspects. Event queries need to be evaluated in an *incremental* manner, as data (events) that are queried are received in a stream-like manner and are not persistent. For every incoming event that might be relevant to an XChange-aware Web site and could contribute as a component to an event query instance specified in the rules of the Web site's XChange program(s), a partial *instantiation* of the involved event queries is realised. An instance of a specified composite event query is detected when instances for all specified component event queries have been detected.

2.4 Metaphor: Speech vs. Written Text

The metaphor of XChange for reactivity on the Web is that of *speech* for volatile data and *written text* for persistent data. Speech cannot be modified. If one has communicated some information in this way one can correct, complete, or invalidate what one has told – through further speech. In contrast, written text can be updated in the usual sense. Likewise, volatile data (events) is *not* updatable but persistent data (Web content) is updatable. To inform about, correct, complete, or invalidate former volatile data, new *event messages* (data containing informations about events that have occurred) are communicated between Web sites. Since events are volatile, like speech, a Web site cannot pose event queries against events that have been received by another Web site. Otherwise, events would have to be made persistent – confusing the clear picture volatile vs. persistent data and thus potentially making programming more complicated.

2.5 Communication of Events

2.5.1 Event Messages

Event messages communicate events between the same or different Web sites. An XChange *event message* is an XML document with a root element labelled `event` and the four parameters (represented as child elements as they may contain complex content): `raising-time` (the time of the event manager of the Web site raising the event), `reception-time` (the time at which a site receives the event), `sender` (the URI of the site where the event has been raised), and `recipient` (the URI of the site where the event has been received). Note that XChange messages are compatible with the messages and the "message exchange patterns" of SOAP [18].

Example 1. Mrs. Smith uses a travel organiser that plans her trips and reacts to happenings that could influence her schedule. One of the travel organiser's tasks is to plan Mrs. Smith's vacation in Provence, France. Mrs. Smith wants to visit Orange, Arles, Nîmes, and Marseilles. Carrying out this task presupposes booking a flight from Munich to Lyon and back, making train reservations and corresponding hotel reservations, and to notify Mrs. Smith of events of interest (such as exhibitions). The following XChange event message is sent by `http://artactif.com` informing the travel organiser of Mrs. Smith about an exhibition of the painter G. Barthouil.

```
xchange:event {
  xchange:sender {"http://artactif.com"},
  xchange:recipient{"organiser://travelorganiser/Smith"},
  xchange:raising-time {"2005-05-05T10:15:00"},
  exhibition {
    painter {"G. Barthouil"}, location {"Marseilles"},
    time-interval{"[2005-05-08..2005-05-18]"},
    visit-hours { from {"10:00"}, until {"18:00"}}
  }
}
```

Note the use of the `xchange` namespace for the keyword `event` and for the parameters of an XChange event message.

The examples are intended to give flavours of the XChange constructs and thus abstract away from a particular com-

munication protocol. In the previous example `organiser` denotes a communication protocol (e.g. the protocol used by a mobile personalised organiser).

An event message is an envelope for an arbitrary XML content. Thus, multiple event messages can (but are not necessarily) be nested making it possible to create trace histories.

Example 2. Mrs. Smith notifies a friend of her about G. Barthouil's exhibition. The following XChange event message is sent by Mrs. Smith's travel organiser and contains the received event message from Example 1.:

```
xchange:event {
  xchange:sender {"organiser://travelorganiser/Smith"},
  xchange:recipient{"organiser://travelorganiser/myFriend"},
  xchange:raising-time {"2005-05-06T11:10:20"},
  content {
    xchange:event {
      xchange:sender {"http://artactif.com"},
      xchange:recipient{"organiser://travelorganiser/Smith"},
      xchange:raising-time {"2005-05-05T10:15:00"},
      xchange:reception-time {"2005-05-05T10:21:20"},
      exhibition {
        painter {"G. Barthouil"}, location {"Marseilles"},
        time-interval{"[2005-05-08..2005-05-18]"},
        visit-hours { from {"10:00"}, until {"18:00"} }
    }
   }
  }
 }
```

XChange excludes broadcasting of event messages on the Web (i.e. sending event messages to all sites of a portion of the Web), since indiscriminate sending of event messages to all Web sites introduces problems for a non-centrally managed structure such as the Web.

2.5.2 Compact Syntax vs. XML Syntax

The language XChange has i) a *compact* syntax (which is a term-based syntax where a term represents an XML document, a query pattern, or an update pattern) and ii) an *XML* syntax. The compact syntax has been developed for the programmers, while the XML syntax is for machine processing. However, programmers have the freedom to choose whichever syntax they prefer. Thus, the example given previously using the XChange's compact syntax is given next using the XML syntax.

Example 3. Using the XML syntax of XChange, the event message given in Example 1. looks like:

```
<xchange:event>
  <xchange:sender>
    http://artactif.com
  </xchange:sender>
  <xchange:recipient>
     organiser://travelorganiser/Smith
  </xchange:recipient>
  <xchange:raising-time>
     2005-05-05T10:15:00
  </xchange:raising-time>
  <exhibition>
    <painter>G. Barthouil</painter>
    <location>Marseilles</location>
    <time-interval>[2005-05-08..2005-05-18]</time-interval>
    <visit-hours>
      <from>10:00</from> <until>18:00</until>
    </visit-hours>
  </exhibition>
</xchange:event>
```

For readability and space reasons, the compact syntax of XChange is used throughout this paper.

2.5.3 Peer-to-Peer

In XChange, the *peer-to-peer* communication model is used for communicating data between Web sites. This means that all parties have the same capabilities and every party can initiate a communication session. Event messages are directly communicated between Web sites without a centralised processing of events. XChange assumes no instance controlling (e.g. synchronising) communication on the Web.

2.5.4 Self-Synchronising Reactive Programs

There is no central service on the Web that synchronises the actions that are to be executed by reactive programs found at different Web sites. Taking communication unreliability into account, rules can be defined locally at an XChange-aware Web site to synchronise actions with other XChange programs. For example, one Web site can wait for another Web site to execute some action (like sending an email to a person) until a time point; if the action has not been executed, the first Web site can decide to execute this (to send an email) on behalf of the second Web site.

2.5.5 Push Strategy

For communicating (propagating) events on the Web, two strategies are possible: the *push* strategy, where a Web site informs possibly interested Web sites about events, and the *pull* strategy, where interested Web sites query periodically (poll) persistent data found at other Web sites in order to determine changes. Both strategies are useful. The pull strategy is supported by languages for standard queries like XQuery or Xcerpt that query persistent data. Therefore, so as to complement the framework, XChange offers the *push* strategy. The push strategy requires event queries to be incrementally evaluated by so-called *event managers* (cf. Section 2.3). In the case of XChange, this is done at every XChange-aware Web site.

2.6 Local Control of Event Memorisation

An essential aspect of XChange is that each Web site controls its own event memory usage. In particular, the size of the event history kept in memory depends only on the event queries posed at this Web site. Neither standard queries nor event queries posed at other Web sites can influence the size of the event history. This is consistent with the clear distinction between events as volatile data and standard Web data as persistent data. The time period for which an atomic event is kept in memory at a Web site is automatically detected from the event queries already posed at this Web site. XChange composite event queries are such that no data on any event can be kept for ever in memory. If this is necessary for some applications, then the applications should turn events into standard Web data by explicitly saving events as persistent data, following the metaphor of Section 2.4 for turning speech into text.

3. COMPOSITE EVENT QUERIES

The capability to detect and react to *composite events*, e.g. 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

[1][4] considers "composite events". However, this notion refers in [4] to updates of several elements of a single XML document. The XChange notion of composite events goes beyond such updates of an XML document.

by XChange is the processing of *composite events*. To this aim, XChange offers *composite event queries*.

An XChange *event query* may be *atomic* or *composite*. An *atomic event query* refers to one single event, it represents a pattern for the single incoming event that is of interest (similar to Xcerpt query patterns). A special atomic event query is `any`, that "matches" an arbitrary atomic event query instance (i.e. a kind of wildcard for atomic event query instances). Two dimensions are distinguished for *composite event queries*: *temporal range* and *event composition*.

Note that *composite event query instances* (detected using composite event queries) do not have time stamps, like atomic event query instances do. Instead, a composite event query instance inherits from its components a beginning time (the reception time of the first received event query instance that is part of the composite event query instance) and an ending time (the reception time of the last received event query instance that is part of the composite event query instance).

The following notations are used throughout this section: `AtomicEventQuery` is an atomic event query specification, `EventQuery` is an (atomic or composite) event query specification, `FiniteTimeSpec` is a specification of a finite time interval (e.g. of the form `before TimePoint`), and `TimeSpec` is a specification of a time interval in which events are to be monitored.

3.1 Temporal Range

A time interval can be specified for (atomic or composite) event queries, meaning that event query instances are considered relevant only if they occur in the given time interval. Such a time interval always has a lower bound (the time point of event query definition, if not explicitly given) and might have an upper bound (i.e. the time interval is *finite* in the sense that it contains a finite number of reference granules [6]). Lower and upper bounds are important for efficiency reasons. They make it possible to release each event at each Web site after a finite time – thus keeping the distinction persistent vs. volatile data.

Along the temporal range dimension, XChange composite event queries can have one of the following forms: `EventQuery within TimeInterval` (note that, in this case, the time interval must not refer to event queries that refer to `EventQuery`), `EventQuery before TimePoint`, `EventQuery after TimePoint`, `AtomicEventQuery at TimePoint`, and `EventQuery during Duration` (the specified duration is considered beginning from the reception time of the first received event query instance that is part of the instance of `EventQuery`).

Example 4. An XChange composite event query that detects new discounts for flights from Munich to Lyon, but only until a given time point.

```
xchange:event {{
  flight {{
    from {"Munich"}, to {"Lyon"},
    new-discount { var D }
  }}
}} before "2005-04-01T10:00:00"
```

In principal, XChange can refer to dates in all possible formats (like the ISO 8601 standard) or to dates/calendars defined using the calendar and temporal type system CaTTS [6]. It is intended to combine XChange and CaTTS in the near future.

3.2 Event Composition

3.2.1 Non-Temporal Event Composition

Conjunctions of event queries are used for detecting instances for each specifed event query. The order in which these instances occur is not relevant. Thus, only unordered specifications of conjunctions of event queries (denoted as `and` $\{EQ_1, EQ_2, ..., EQ_n\}$ with EQ_i event query, for $i = 1..n$) are considered in XChange.

Example 5. 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"},
                  time-interval { var TI }
               }}
  }},
  xchange:event {{
    xchange:sender {"http://weather.com"},
    forecast { date { var D }, city {"Marseilles"},
               info {"It's going to rain."} }
  }}
} where var D included-in var TI
```

Inclusive disjunctions of event queries are used for detecting instances that are answers to one of the specified event query. An answer to an inclusive disjunction event query (denoted `or` $\{EQ_1, EQ_2, ..., EQ_n\}$ (EQ_i event query, for $i = 1..n$) is the first received instance of one of the EQ_i, $i = 1, ..., n$. Note that *exclusive disjunctions* of event queries can also be specified (in other way), as XChange offers a generalised exclusive disjunction by means of the multiple selection constructs (cf. Section 3.2.2.3).

Example 6. 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://service-nimes.fr"},
    service-notification {{
      train {{
        var D ⇝ date {"2005-05-03"},
        var F ⇝from {"Orange"}, to {"Nimes"}
      }},
      hotel {{ }}
    }}
  }},
  xchange:event {{
    xchange:sender {"http://reservations-arles.fr"},
    reservation-notification {{
      train {{ var D, var F, to {"Arles"} }},
      accomodation {{ }}
    }}
  }}
} before "2005-05-02T21:30:00"
```

3.2.2 Temporal Event Composition

Constructs for composite event queries that refer to a temporal order between event instances are introduced next. The *reception time* of incoming events is used to determine the temporal order of events. The possibility to specify that the temporal order of events should be considered in terms of raising time and/or reception time of incoming events is currently investigated in XChange.

3.2.2.1 Orderings.

1. **Temporally ordered conjunctions** of event queries are used for detecting successive, in terms of time, occurrences of events. The keyword `andthen` is used to denote such event queries. Only ordered event query specifications are allowed for `andthen`, as the idea is to query occurrences of events that are *ordered* on the time axis of the incoming events.

A total specification (using single square brackets) expresses that only instances of the specified event queries are of interest and are to be contained in the answer. Instances of other event queries that have possibly occurred between the instances of the specified event queries are not of interest and, thus, are not contained in the answer.

In contrast, a partial specification (using double square brackets) expresses interest in all incoming events that have been received between the instances of the specified event queries. Thus all these instances will be contained in the event query's answer. Moreover, one can ask only for instances of events having a given pattern that have occurred between the instances of two specified event queries.

Example 7. 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 accomodation 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"},
    important {"Accomodation is not granted!"}
   }}
 ] during 2 hour
```

2. **Overlappings** of *composite* event queries are used for detecting instances of composite event queries that overlap on the time axis of the incoming events. Two event query instances eq_1 and eq_2 *overlap* if the beginning time of eq_1 is before the beginning time of eq_2 and the ending time of eq_1 is before the ending time of eq_2 on the time axis of the incoming events, or viceversa.

Ordered and unordered specifications are possible for overlappings of composite event queries, i.e. one has the possibility to express that the temporal order is of importance or not. Only total specifications are allowed for overlappings of composite event queries.

Example 8. Mrs. Smith would love to meet a friend of her that lives in Marseilles. Thus, Mrs. Smith's staying in Marseilles needs to overlap with the free time of her friend. An XChange event query that detects such a situation:

```
overlap [
  andthen [[
   xchange:event {{ arrival {{ city {"Marseilles"} }} }},
   xchange:event {{
     departure {{ city {"Marseilles"}, time {var T} }} }}
  ]],
  andthen [[
   xchange:event {{
     xchange:sender {"organiser://softw/a-friend/"},
     info {"I leave off work now!"} }},
   optional xchange:event {{
        xchange:sender {"organiser://softw/a-friend/"},
        important {"Something interced, we'll meet tommorow."} }}
  ]] before var T
 ]
```

3. **Meets** for composite event queries are used for detecting event query instances whose components "meet" on the time axis of the incoming events. Two event query instances eq_1 and eq_2 *meet* if the ending time of eq_1 is the same as the beginning time of eq_2, or viceversa. Like for overlappings of composite event queries, ordered and unordered specifications are possible and only total specifications are allowed for meets of composite event queries.

Example 9. An XChange event query that detects situations where Mrs. Smith's delayed flight arrives in Lyon at the time of departure of her train connection to Orange:

```
meets [
  xchange:event {{ arrival {{ delayed-flight {{
                              from {"Munich"}, to {"Lyon"} }} }}
    }},
  xchange:event {{ train-departure {{
                  from {"Lyon"}, to {"Orange"} }}
    }}
 ]
```

4. **Overlappings or meets** for composite event queries are used for detecting event query instances whose components overlap or meet on the time axis of the incoming events. This XChange construct, introduced by keyword `overlap-or-meets`, is a kind of mixture between the two previously introduced language constructs, i.e. `overlap` and `meets`.

5. **Inclusions** for event queries are used for detecting instances of events that have occurred during the time interval determined by the beginning time and the ending time of an instance of a composite event query. The inclusion construct for *event queries* is to be understood as the Allen's *during* relation for *time intervals* [2].

Example 10. The following XChange event query is used to detect emergencies that occur during Mrs. Smith's vacation.

```
include [
  andthen [[
   xchange:event {{
    begin-vacation {{ arrival {{ country {"France"} }} }} }},
   xchange:event {{
     end-vacation {{ departure {{ time {var T} }} }} }}
  ]],
  or {
   xchange:event {{
     xchange:sender {"organiser://institute/secretary/"},
     info {"There is a problem in our project!"} }},
   xchange:event {{
     xchange:sender {"organiser://organiser/myBrother/"},
     important {"Emergency in our familiy!"} }}
   }
 ]
```

One issue that is currently investigated in the XChange project is the specification of different time granularities that are to be used instead of time points for deciding whether event query instances have been received in a particular temporal order or not. For example, using the time granularity week, one can ask whether or not instances of event queries of interest *meet*. In this case, two event query instances meet if there is no more than a week between the ending time of the first instance and the beginning time of the second one.

3.2.2.2 Event Exclusions.

Only *positive* event query specifications (i.e. expressing event queries whose instances are to be monitored) are allowed in the specifications of XChange composite event queries. Thus, using the above presented constructs for composite event queries, one cannot specify that one is interested

in occurrences of composite event query instances, but only if other event query instances have not occurred in a given time interval. Such kinds of composite event queries can be specified in XChange by means of the **event exclusion** constructs, i.e. `CompositeEventQuery without EventQuery FiniteTimeSpec` and `without EventQuery FiniteTimeSpec`.

Example 11. The following XChange event query detects if the notification of an online reservation made on 10th of April 2005 is not received within ten days.

```
without xchange:event {{
        online-reservation-notification {{ }}
    }} within [2005-04-10..2005-04-20]
```

3.2.2.3 Multiple Selections and Exclusions.

Multiple selections and exclusions for event queries, denoted m `of` $\{EQ_1, EQ_2, ..., EQ_n\}$ `FiniteTimeSpec`, with $1 \leq m \leq n$, are used for detecting instances of m of the specified event queries and the non-occurrence of instances of the other $n-m$ event queries, within the given finite time interval. The multiple selection and exclusions construct has other two variants, `atleast` m `of` $\{EQ_1, EQ_2, ..., EQ_n\}$ `FiniteTimeSpec` and `atmost` m `of` $\{EQ_1, EQ_2, ..., EQ_n\}$ `FiniteTimeSpec`. Multiple selection and exclusion specifications for event queries must always be accompanied by specifications of a finite time interval in which instances of the specified event queries are to be monitored. Note that if $m = 1$, the construct is equivalent to an exclusive disjunction for event queries.

Example 12. The following XChange event query detects notifications either of a flight cancellation or of an in-time departure for a flight:

```
1 of {
  xchange:event {{
    xchange:sender {"http://airline.com/"},
    cancellation-notification {{
      flight {{
        number {"AI2021"}, date {"2005-05-15"} }}
  }},
  xchange:event {{
   xchange:sender {"http://airline.com/"},
   notification {{
     flight {{
       number {"AI2021"}, date {"2005-05-15"},
       info {"There are no delays or other problems!"}
     }}  }}
  }}
 } before "17:00"
```

3.2.2.4 Branching.

1. An **if-then-else** composite event query is provided by XChange for querying different instances depending whether an instance of a specified event query (the one specified in the `if`-part) has occurred or not.

Example 13. The following XChange event query specifies that after receiving a notification from Mrs. Smith's secretary saying that a problem occurred, a request for a phone conference should be queried.

```
if xchange:event {{
    xchange:sender {"organiser://institute/secretary"},
    important {"There is a problem in our project!"}
   }}
then xchange:event {{
      xchange:sender {"organiser://institute/secretary"},
      request {{ phone-conference {{ at {var Time} }} }}
     }}
```

2. The **case** construct for event queries is a generalisation of the `if-then-else` construct for event queries. An `else`-part (default-like) can be specified for the `case` construct, where an event query is given that is to be posed if the event queries of the `case`-part could not be answered. In this situation, a finite time interval needs to be specified for the event queries in the `case`-part. This is due to the fact that the non-occurrence of instances of the specified event queries can be evaluated only over a finite time interval.

3.2.2.5 Occurrences.

1. **Quantifications** for event queries specify that event query instances should occur at least, at most, or exactly a given number of times in a specified time interval. The following quantification specifications for event queries can be used in XChange: n `times EventQuery FiniteTimeSpec`, `atmost` n `times EventQuery FiniteTimeSpec`, and `atleast` n `times EventQuery TimeSpec`.

Example 14. Mrs. Smith might want to react to situations like the reception of at least three messages from her secretary during one hour. Such a situation can be detected using the XChange event query:

```
atleast 3 times xchange:event {{
     secretary-message {{ important {{ }} }}
 }} during 1 hour
```

2. **Ranks** are used to detect the instance of a specified event query that has a given rank in the incoming flow of events. Such event queries are denoted in XChange as `EventQuery withrank` n and `last EventQuery FiniteTimeSpec` (the answer to such a composite event query is the last instance of `EventQuery` that occurs within the given finite time interval).

Example 15. As the airline might send several delay notifications for Mrs. Smith's flight, she is interested in the last notification received before nine o'clock in the evening:

```
last xchange:event {{
       xchange:sender {"http://airline.com"},
       delay-notification {{
         flight {{ number {"AI2021"},
           espected-arrival-time { var T }, date {"2005-05-15"} }}
        }}
 }} before "21:00"
```

3. **Repetitions** are used for detecting e.g. every second, forth, sixed, and so on, instances of a specified event query that occur in a given finite time interval. Such an event query is denoted as `every` n `EventQuery FiniteTimeSpec`.

Example 16. Mrs. Smith wants to quit slowly smoking so she answers only to every second call from her colleague suggesting a smoking break:

```
every 2 xchange:event {{
          xchange:sender {organiser://institute/myColleague/},
          break-for-a-smoke {{
            info {"Join me for a cigarette!"}  }}
    }} within workday
```

Note that `workday` denotes a temporal type defined using the CaTTS system [6].

3.3 Processing of Event Queries in XChange

XChange assumes no central processing of event queries as such an approach is not suitable on the Web. Instead, event queries are processed locally at each XChange-aware Web site. Each such Web site has its own local *event manager* for processing incoming events and evaluating event queries against the incoming event stream (volatile data), and for releasing event query instances after a finite time (cf. Section 2.6).

As explained in previous sections, XChange atomic event queries are patterns for incoming event instances that are of interest for a Web site. The event manager of an XChange-aware Web site tries 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* [14], a novel unification method developed for matching query terms (i.e. standard queries) with data or construct terms (i.e. persistent data) in Xcerpt. The volatile nature of events does not preclude the usage of the same method for *simulating* event queries into incomig events. For determining answers to composite event queries, matching component atomic event queries with incoming events is not sufficient, as the relationships between these events is not captured.

3.4 Event Queries' Answers

An answer to an XChange event query is an XML document. An answer to an atomic event query (i.e. an atomic event instance) is an XML document containing the information of the event message that matched with the specified event query. An answer to a composite event query (i.e. a composite event instance) is an XML document containing answers to the component event queries and the temporal relations between these answers. For example, an answer to a composite event query `andthen` $[EQ_1, EQ_2]$ is an XML document with root labelled `event-andthen` with two ordered children, eq_1 and eq_2 – instances of EQ_1 and EQ_2, respectively. A point which, to the knowledge of the authors, is novel is that (atomic and composite) events have a structure and are represented as XML documents (that might have a complex structure).

Example 17. An answer to the XChange event query of Example 7. is:

```
<xchange:event-andthen ordered="true">
  <xchange:event>
    <xchange:sender>http://airline.com</xchange:sender>
    <xchange:recipient>
       organiser://travelorganiser/Smith
    </xchange:recipient>
    <xchange:raising-time>
       2005-05-15T16:05:03
    </xchange:raising-time>
    <xchange:reception-time>
       2005-05-15T16:10:00
    </xchange:reception-time>
    <cancellation-notification>
      <flight>
        <number>AI2021</number><date>2005-05-15</date>
        <from>Lyon</from><to>Munich</to>
      </flight>
    </cancellation-notification>
  </xchange:event>
  <xchange:event>
    <xchange:sender>http://airline.com</xchange:sender>
    <xchange:recipient>
      organiser://travelorganiser/Smith
    </xchange:recipient>
    <xchange:raising-time>
      2005-05-15T17:01:12
    </xchange:raising-time>
    <xchange:reception-time>
      2005-05-15T17:07:20
    </xchange:reception-time>
    <important>Accomodation is not granted!</important>
  </xchange:event>
</xchange:event-andthen>
```

Answers to XChange composite event queries can be "put in an envelope" and sent as event messages to one or more Web sites. Just as it is easy to exchange and query information about atomic events, it is also easy to exchange and query information about composite events. Powerful pattern matching based on simulation unification can be applied to event messages.

4. QUERYING WEB RESOURCES

4.1 Relationship Between Reactive and Query Languages

A working hypothesis of the XChange project is that a reactive language for the Web should build upon, or more precisely, embed, a Web query language. There are two reasons for this. First, specifications of reactive behaviour often refer to actual Web contents - calling for querying Web contents (examples are given in Section 5). Second, reactive behaviour necessarily refers to (more or less recent) events - calling for querying events. For reasons of uniformity, it is highly desirable both for users and for system developers that the languages used for querying Web contents and querying events are as close as possible to each other. Note, however, that querying events calls for constructs not needed for querying Web contents.

4.2 The Web Query Language Xcerpt

The Web query language Xcerpt is *embedded* in XChange. Xcerpt is a pattern and rule-based language for querying Web contents (i.e. persistent data). Xcerpt uses query *patterns* for querying Web contents, and construction *patterns* for constructing new data items. *Terms* are used for denoting query patterns (i.e. *query terms*), construction patterns (i.e. *construct terms*) and also for denoting data items of Web contents (i.e. *data terms*). Common to all terms is that they represent tree or graph-like structures. The children of a node may either be *ordered*, i.e. the order of occurrence is relevant, or *unordered*, i.e. the order of occurrence is irrelevant. In the term syntax (used in this article as it is more readable as the Xcerpt's XML syntax), an *ordered term specification* is denoted by square brackets [ ], an *unordered term specification* by curly braces {}.

4.2.1 Data Terms

Data Terms represent data items (XML documents) that are found on the Web. In an Xcerpt program, the Web contents to be queried are specified using the keyword `resource` followed by the Web address(es) where the data is to be found.

Example 18. The following two Xcerpt data terms represent a flight timetable and a hotel reservation offer.

At `http://airline.com`:

```
flights {
  changed-on {"2004-05-01"},
  currency {"EUR"},
  flight {
   number {"AI2011"},
   from {"Munich"},
   to {"Lyon"},
   date {"2005-05-01"},
   departure-time {"10:00"},
   arrival-time {"11:25"},
   price {"125"}
  },
  flight {
   number {"AI2021"},
   from {"Lyon"},
   to {"Munich"},
   date {"2005-05-15"},
   departure-time {"17:20"},
   arrival-time {"18:45"},
   price {"125"}
  }...
}
```

At `http://hotels.net`:

```
accomodation {
  currency {"EUR"},
  hotels {
   city {"Orange"},
   country {"France"},
   hotel {
     name {"Ambassade"},
     category {"2 stars"},
     price-per-room {"62"},
     phone {"+3388219213"},
     no-pets {}
     },
   hotel {
     name {"Winston"},
     category {"3 stars"},
     price-per-room {"60"},
     phone {"+3388156135"}
     },
   hotel {
     name {"Royale"},
     category {"4 stars"},
     price-per-room {"120"},
     phone {"+3388123414"}
     },
   ...
  },
...
}
```

4.2.2 Query Terms

Query Terms are (possibly incomplete) patterns that are matched against Web contents represented by data terms. *Partial* (incomplete) or *total* (complete) query patterns can be specified. A query term t using a partial specification (denoted by *double* square brackets [[ ]] or curly braces {{}}) for its subterms matches with all such terms that (1) contain matching subterms for all subterms of t and that (2) might contain further subterms without corresponding subterms in t. In contrast, a query term t using a total specification (denoted by *single* square brackets [ ] or curly braces {}) does not match with terms that contain additional subterms without corresponding subterms in t. Query terms contain *variables* for selecting data items (i.e. subterms of data terms are to be bound to the variables). Variable restrictions can be specified using the $\rightsquigarrow$ construct (read *as*), which restricts the bindings of the variables to those terms that are matched by the restriction pattern.

Example 19. The following Xcerpt query term is used to query the data at `http://airline.com` about flights from Munich to Lyon.

```
in { resource { "http://airline.com" },
  flights {{ var F ⇝ flight {{
                       from {"Munich"}, to {"Lyon"}  }}
      }}
  }
```

Xcerpt query terms may be augmented by additional constructs like *subterm negation* (keyword `without`), *optional subterm specification* (keyword `optional`), and *descendant* (keyword `desc`) [15].

Query terms are "matched" with data or construct terms by a non-standard unification method called *simulation unification* [15] dealing with partial and unordered query specifications.

4.2.3 Construct Terms

Construct Terms serve to reassemble variables (the bindings of which are specified in query terms) so as to construct new data terms. They are similar to data terms, but augmented by *variables* (acting as place holders for data selected in a query) and the *grouping construct* `all` (which serves to collect all instances that result from different variable bindings). Occurrences of `all` may be accompanied by an optional sorting specification.

4.2.4 Construct-Query Rules

Construct-Query Rules (short rules) relate a construct term (introduced by the keyword `CONSTRUCT`) to a query (introduced by the keyword `FROM`) consisting of `AND` and/or `OR` connected query terms. Queries or parts of a query may be further restricted by arithmetic constraints in a so-called condition box (introduced by the keyword `where`).

Example 20. The following Xcerpt rule gathers informations about the hotels in Orange with a price limit.

```
CONSTRUCT
  answer [ all var H ordered by var P ascending ]
FROM
  in { resource { "http://hotels.net" },
    accomodation {{
      hotels {{ city {"Orange"},
                var H ⇝ hotel {{
                          price-per-room { var P } }}
             }}
    }}
   } where var P < 100
END
```

An Xcerpt program consists of one or more rules. Xcerpt rules may be *chained* to form complex query programs, i.e. rules may query the results of other rules. More on Xcerpt can be found in [15] and at `http://xcerpt.org`.

4.3 XChange and Xcerpt Syntaxes

The languages XChange and Xcerpt have similar syntaxes, but with the following semantical differences:

1. **Square brackets** in XChange composite event query specifications express temporal order between incoming instances of event queries, while square brackets in Xcerpt query term specifications express document order between specified subterms.
2. **Partial specifications** in XChange composite event query specifications express that the answer might include additional event query instances to those specified in the event query. The same event query posed at different points in time might have different answers, depending on the incoming events (that are additional to the ones specified). In Xcerpt query term specifications, partiality means that the answers are documents with possible additional subelements to those specified in the query term. These additional subterms exist in the queried documents.
3. **The `without` construct** in XChange composite event query specifications expresses that in a given time interval (possibly determined by occurrences of event query instances) instances of a specified event query should not occur. In Xcerpt query term specifications `without` expresses subterm negation for documents that are queried [15].
4. **Temporal constructs** are offered by XChange for posing composite event queries against incoming events (cf. Section 3). For Xcerpt queries (i.e. queries to Web resources) such temporal constructs are not required.

The authors believe that such syntax similarities between XChange and Xcerpt are a convenient means for an easy programming of reactive applications on the Web.

5. TRANSACTIONS AND REACTIVE RULES IN XCHANGE

5.1 Complex Updates

An *elementary update* is a change (i.e. insert, delete, replace) to a persistent data item (e.g. XML or RDF data). *Complex updates* expressing ordered or unordered conjunctions, or disjunctions of (elementary or complex) updates are also offered by XChange. Such updates are often required by real applications. E.g. when booking a trip on the Web, one might wish to book an early flight *and* the corresponding hotel reservation, *or* a late flight *and* a shorter hotel reservation. Since it is sometimes necessary to execute such complex updates in an *all-or-nothing manner* (e.g. when booking a trip, a hotel reservation without a flight reservation is useless), XChange has a concept of transactions.

Example 21. The following XChange complex update specifies that a flight reservation and a hotel reservation are to be executed in the specified order.

```
and [
    in { resource {"http://airline.com/reservations"},
    flights {{
      insert reservation { var Flight,
                           var Name⇝name {"Christina Smith"} }
    }}
  },
  in { resource {"http://hotels.net/reservations"},
    reservations {{
```

```
      insert reservation { var Hotel, var Name,
                   from {"2005-05-10"}, until {"2005-05-15"} }
    }}
  }
]
```

5.2 Transactions

An XChange transaction specification is a group of update specifications and/or explicit event specifications (expressing events that are constructed, raised, and sent as event messages) that are to be executed in an *all-or-nothing manner*. XChange transactions obey the ACID properties [16] (Atomicity, Consistency, Isolation, and Durability). Atomicity and isolation are considered in XChange, the issues of consistency and durability for transactions are currently not investigated in the project. XChange will build on standard solutions from database systems.

An XChange *update specification* is a (possibly incomplete) *pattern* for the data to be updated, augmented with the desired update operations. The notion of update terms is used to denote patterns which contain update operations for the data to be modified. An update term may contain different types of update operations.

Intensional updates are a description of updates in terms of (standard or event) queries. They can be specified in XChange, as the language inherits the querying capabilities of the language Xcerpt. This eases considerably the specification of updates, e.g. for specifying *modification* of the discounts for *all* flights offered by a specific airline.

5.3 (Re)active Rules

An XChange program is located at one Web site and consists of one or more (re)active rules of the form *Event query – Standard query – Transaction/Raised events*. Every occurrence of an event is queried using the *event query* (introduced by keyword `ON`). If an answer is found and the *standard query* (introduced by keyword `FROM`) also has an answer, then the action is executed (i.e. a transaction is executed – keyword `TRANSACTION`, or explicit events are raised and sent to one or more Web sites – keyword `RAISE`). There are two kinds of XChange rules: *event-raising rules* (specifying events to be raised) and *transaction rules* (specifying transactions to be executed). Note that the variable substitutions found in the `ON`-part can be used in the `FROM` and `RAISE`- or `TRANSACTION`-part of an XChange rule. Variable substitutions found in the `FROM`-part can be used in the `RAISE`- or `TRANSACTION`-part of an XChange rule.

Example 22. The site `http://airline.com` has been told to notify Mrs. Smith's travel organiser of delays or cancellations of flights she travels with.

```
RAISE
 xchange:event {
   xchange:recipient {"organiser://travelorganiser/Smith"},
   cancellation-notification { var F }
  }
ON
 xchange:event {{
  xchange:sender {"http://airline.com"},
  cancellation {{
    var F ⇝flight {{ number {"AI2021"},
                     date {"2005-05-17"} }}  }}
 }}
END
```

Example 23. The travel organiser of Mrs. Smith uses the following rule: if the return flight of Mrs. Smith is cancelled then look for and book another suitable flight. The rule is specified in XChange as:

```
TRANSACTION
 in { resource {"http://airline.com/reservations"},
   reservations {{
     insert reservation { var F, name {"Christina Smith"} }
     }}
 }
ON
 xchange:event {{
   xchange:sender {"http://airline.com"},
   cancellation-notification {{
     flight {{ number {"AI2021"}, var D ⇝date {"2005-05-17"} }}
    }}
 }}
FROM
 in { resource {"http://airline.com"},
   flights {{
     var F ⇝flight {{
             from {"Lyon"}, to {"Munich"},
             var D, departure-time { var T }
           }}
    }}
  } where var T after "17:30"
END
```

Example 24. If no other suitable return flight is found and the airline does not provide an accomodation, then book for Mrs. Smith a cheap hotel and inform her secretary about the changes in her schedule:

```
TRANSACTION
 and [
   in { resource {"http://hotels.net/reservations"},
     reservations {{
       insert reservation { var H, name {"Christina Smith"},
                from { var S }, until { var M ⇝"2005-05-16"} }
      }}
    },
   in { resource {"diary://diary/secretary"},
     diary {{ var M,
       news {{ insert my-hotel { phone { var Tel },
         remark {"My flight has been cancelled!},
         request{"Please cancel my appointments for" + var M}  }
        }}  }}
    }
  ]
ON
 and {
   xchange:event {{ xchange:sender {"http://airline.com"},
            cancellation-notification {{
               flight {{ number {"AI2021"},
                         date { var S ⇝"2005-05-15"} }}
             }}
    }},
    without xchange:event {{
              xchange:sender {"http://airline.com"},
              accomodation-granted {{ hotel {{ }} }}
       }} before "18:00"
  }
FROM
 in { resource {"http://hotels.net"},
   accomodation {{
     hotels {{ city {"Lyon"},
               var H ⇝hotel {{ price-per-room { var P },
                               phone { var Tel } }}
       }}  }}
  }
END
```

As the paper emphasises the XChange capabilities to detect composite events and the space is limited, the semantics of rules' execution is not discussed here.

6. RELATED WORK

Allen's Temporal Relations. As explained in previous sections, a composite event query instance has a beginning time and an ending time (time points that can be seen as the starting and the ending point of a time interval). Thus, determining the temporal order of two (or more) composite event query instances can be reduced to determining the relationship between the time intervals formed from the beginning and ending time of the these instances. The possible relationships between time intervals have been described and represented in a hierarchical manner by James F. Allen [2]. The thirteen possible relationships between time intervals

have provided a basis for the development of the XChange constructs for detecting composite events.

Active Databases. A number of active database prototypes have been built providing sophisticated event algebras (e.g. SNOOP [8], REACH [7], CHIMERA [10] and ODEGJS92). Work in [21] provides a meta-model for classifying a number of properties of complex event formalisms for active rules. These works are oriented towards a centralised system, as opposed to a distributed one like the Web and there is not much work that considers events in a distributed environment, with [20] being a notable exception. Two recent works which address complex events are [4], which does so in the context of updates for XML and [1], which outlines a situation monitoring system and expands upon much of the work in the active database literature.

Other related work in the XML context is found in [11]. [11] discusses 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 complex 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.

Two key features that distinguish our work from those above are i) events are represented as XML documents and consequently may have a nested structure to which pattern matching can be applied within event queries, ii) there is a clear separation between events as volatile data versus the persistent data which can be queried by the user. This has implications for the kinds of rules that can be defined.

7. CONCLUSION AND FUTURE WORK

This paper has introduced constructs for the composition of events on the Web. Such constructs are needed when realising reactivity on the Web, an essential issue for both Web services and Semantic Web systems. The work reported about here is part of the XChange project, which started one year ago. XChange builds upon the Web query language Xcerpt [15] – cf. Section 4. A first version of Xcerpt is fully designed and a reference implementation (cf. `http://xcerpt.org`) is available. Currently, the design of an extended core language for XChange is completed and a first (reference) implementation has begun.

A promising perspective for future work consists in extending the language XChange with security functionalities (especially authentication and authorisation). The protocols of a Grid architecture (such as Globus [9]) would provide with convenient means for such an extension. Vice versa, XChange could be seen as a (core of a) high-level reactive language for advanced services in a Grid architecture.

8. REFERENCES

[1] A. Adi and O. Etzion. Amit – the situation manager. In *Very Large Data Bases Journal*, volume 13, pages 177–203, 2004.

[2] J. F. Allen. Maintaining Knowledge about Temporal Intervals. In *Communications of the ACM*, volume 26, pages 832–843, 1983.

[3] S. Berger, F. Bry, S. Schaffert, and C. Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. In *Int. Conf. on Very Large Databases (VLDB)*, 2003.

[4] M. Bernauer, G. Kappel, and G. Kramler. Composite Events for XML. In *13th Int. Conf. on World Wide Web*. ACM, 2004.

[5] F. Bry and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *20th Annual ACM Symposium on Applied Computing (SAC'2005)*. ACM Press, 2005.

[6] F. Bry and S. Spranger. Towards a Multi-Calendar Temporal Type System for (Semantic) Web Query Languages. In *Workshop on Principles and Practice of Semantic Web Reasoning*. Springer, 2004.

[7] A. Buchmann, A. Deutsch, J. Zimmermann, and M. Higa. The REACH Active OODBMS. In M. Carey and D. Schneider, editors, *ACM SIGMOD Int. Conference on Management of Data*. ACM Press, 1995.

[8] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data and Knowledge Engineering*, 14(1), 1994.

[9] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid. Enabling Scalable Virtual Organizations. In *Int. Journal of Supercomputer Applications*, 2001.

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

[11] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML Data on the Web. In *ACM SIGMOD Int. Conference on Management of Data*. ACM Press, 2001.

[12] G. Papamarkos, A. Poulovassilis, and P. Wood. Event-Condition-Action Rule Languages for the Semantic Web. In *Workshop on Semantic Web and Databases*, 2003.

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

[14] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation, 2004.

[15] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Int. Conf. Extreme Markup Languages*, 2004.

[16] J. Ullman. *Principles of Database and Knowledge-base Systems*, volume 1. Computer Science Press, 1988.

[17] W3 Consortium. *XQuery: A Query Language for XML*, 2001.

[18] W3 Consortium. *SOAP Version 1.2 Part 1: Messaging Framework*, 2003.

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

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

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