Journal of Web Engineering, Vol. 5, No. 1 (2006) 003–024
© Rinton Press

REACTIVITY ON THE WEB: PARADIGMS AND APPLICATIONS OF THE LANGUAGE XCHANGE

FRANÇOIS BRY, MICHAEL ECKERT, PAULA-LAVINIA PĂTRÂNJAN
Institute for Informatics, University of Munich, Oettingenstr. 67
D-80538 Munich, Germany
{bry,eckert,patranjan}@pms.ifi.lmu.de

Received April 17, 2005
Revised January 13, 2006

Reactivity on the Web is an emerging research issue covering: *updating data* on the Web, exchanging information about *events* (such as executed updates) between Web sites, and *reacting* to combinations of such events. Reactivity plays an important role for upcoming Web systems such as online marketplaces, adaptive Web and Semantic Web systems, as well as Web services and Grids. This article introduces the paradigms upon which the high-level language *XChange* for programming reactive behaviour and distributed applications on the Web relies. Then, it briefly presents the main syntactical constructs of XChange and their declarative and operational semantics.

Keywords: Event-condition-action rules, composite events, reactive languages, Web

Communicated by: S Comai

1 Introduction

Many resources on the Web and the Semantic Web are dynamic in the sense that they can change their content over time. The need for changing (updating) data on the Web has several reasons: new information comes in, calling for insertions of new data; information is out-of-date, calling for deletions and replacements of data. Such changes need to be mirrored by other Web resources whose data depends on the initial changes. In other words, updates need to be propagated over related Web resources.

Reactivity on the Web is the ability of Web sites to detect happenings, or events, of interest that have occurred on the Web and to automatically react to them through reactive programs. Events may have various levels of abstraction ranging from low-level ones, such as insertions into XML or RDF documents, to high-level application-dependent ones. For example, in a tourism application, events of interest include delays or cancellations of flights, and new discounts for flights offered by an airline. Reactions to such events include notifying colleagues about delays, looking for and booking another flight, or booking flights from a particular airline.

Following a declarative approach to reactivity on the Web, a novel reactive, rule-based language called *XChange* [12, 6, 13, 5, 14, 20, 30] has been developed. XChange provides the following benefits over the conventional approach of using general-purpose programming languages like Java to implement reactive behaviour on the Web:

(i) XChange reactive rules have a highly declarative nature. They allow programming on a high level of abstraction, and are easy to analyse for both humans and machines (e.g. for optimisation, verification, or termination).
(ii) The various parts of a rule all follow the same paradigm of specifying patterns for XML data, thus making XChange an elegant, easy to learn language.

(iii) Both atomic and composite events can be detected and relevant data extracted from events. Composite events, temporal combinations of events, are an important requirement in composing an application from different services.
(iv) XChange embeds an XML query language, Xcerpt, allowing to access Web resources and reason with their data in a natural way.
(v) A typical reaction to some event is to update some Web resource; XChange provides an integrated XML update language for doing this.
(vi) XChange reactive rules enforce a clear separation of persistent data (Web resources) and volatile data (events). The distinction is important for programmers: the former relates to *state*, while the latter reflects *changes in state*.
(vii) XChange's high abstraction level and its powerful constructs allow for short and compact code. The programmer is released from programming tasks like access to or modification of data (events, Web resources) "manually".

This article is an extended and enhanced version of [13]. (In particular, Section 5 has no counterpart in [13].) Section 2 introduces the paradigms upon which the high-level, reactive language XChange relies. Section 3 gives a brief introduction into the Web query language Xcerpt, which is embedded in XChange. Section 4 subsequently describes the core language constructs of XChange, accompanied by examples. Section 5 provides declarative and operational semantics. Section 6 discusses related work on the topic of reactivity on the Web. Finally, Section 7 concludes with a summary and perspectives for future research.

2 XChange: Paradigms

Clear paradigms that a programming language follows provide a better language understanding and ease programming. Moreover, explicitly stated paradigms are essential for Web languages, since these languages should be easy to understand and use also by novice practitioners. This section introduces the paradigms upon which the language XChange relies.

2.1 Event vs. Event Query

An *event* is a happening to which each Web site may decide to react in a particular way or not to react to at all. For example, an insertion of new discounts for flights or just "8 o'clock every morning" are events. One might argue that defining an event in such a way is too vague. The intention here is to emphasise that one can conceive every kind of changes on the Web as events. However, each Web-based reactive system can be interested in different types of events or in different combinations of (like a given temporal order between) such events. Thus, the large spectra of possible events are always filtered relatively to one's interests (e.g. the owner of a personal travel organiser). In order to notify Web sites about events and to process event data, events need to have a data representation. In XChange, incoming events are represented as XML documents (see Section 4).

Event queries are queries against event data. Event query specifications differ considerably from event representations (e.g. event queries may contain variables for selecting parts of the events' representation). Most proposals dealing with reactivity do not significantly differentiate between event and event query. Overloading the notion of event precludes a clear language semantics and thus, makes the implementation of the language and its usage much more difficult. Event queries in XChange serve a double purpose: detecting events of interest and temporal combinations of them, and selecting data items from events' representation. This double purpose is novel in the field of reactivity and reactive rules.

2.2 Volatile vs. Persistent Data

The development of the XChange language – its design and its implementation – reflects the novel view over the Web data that differentiates between *volatile data* (event data communicated on the Web between XChange programs) and *persistent data* (data of Web resources, such as XML or HTML documents). One can imagine volatile data as *speech* and persistent data as *(computer-)written text*. Speech cannot be modified. If one has communicated some information in this way he/she can correct, complete, or invalidate what he/she has told – through further speech. In contrast, written text can be updated in the usual sense. Likewise, volatile data (i.e. events) is *not* updatable but persistent data (i.e. Web content) is updatable. To inform about, correct, complete, or invalidate former volatile data, new *event messages* (i.e. data containing information about events that have occurred) are communicated between Web sites. The clear distinction between volatile and persistent data aims at easing programming and avoiding the emergence of a parallel Web of events.

2.3 Rule-Based Language

Reactivity can be specified and realised by means of reactive rules [19, 29, 34]. XChange is a rule-based language that uses reactive rules for specifying the desired reactive behaviour and deductive rules for constructing views over Web resources' data.

An XChange program is located at one Web site and contains reactive rules, more precisely Event-Condition-Action rules (ECA rules) of the form *Event query – Web query – Action*. Every incoming event is queried using the *event query* (query against volatile data). If an answer is found and the *Web query* (query to persistent data) has also an answer, then the *Action* is executed. The fact that the event query and the Web query have answers (i.e. evaluate successfully) determines the rule to be fired; the answers influence the action to be executed, as information contained in the answers are generally used in the action part.

XChange embeds the Web query language Xcerpt for expressing the *Web query* part of reactive rules and for specifying deductive rules in XChange programs. Xcerpt (deductive) rules allow for constructing views over (possibly heterogeneous) Web resources that can be further queried in the *Web query* part of XChange reactive rules.

2.4 Pattern-Based Approach

XChange is a *pattern-based language*: event queries, Web queries, event raising specifications, and updates describe *patterns* for events requiring a reaction, Web data, raising event messages, and updating Web data, respectively. Patterns are templates that closely resemble the structure of the data to be queried, constructed, or modified, thus being very intuitive and also straight-forward to visualise [8].

2.5 Communication Paradigms

Peer-to-Peer Model With XChange, the communication between Web sites is based on a *peer-to-peer* communication model, i.e. all parties have the same capabilities and every party can initiate a communication session. *Event messages*, notifications containing data of events that have occurred on the Web, are directly communicated between Web sites without a centralised processing of events or event messages.

Push Strategy For communicating events on the Web two strategies are possible: the *push* strategy, i.e. a Web site informs (possibly) interested Web sites about events, and the *pull* strategy, i.e. 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 Web queries (e.g. XQuery [3] or Xcerpt), i.e. queries to persistent data. XChange offers the *push* strategy for communicating events. A push strategy has several advantages over a strategy of periodical polling: it allows faster reaction, avoids unnecessary network traffic, and saves local resources.

2.6 Processing of Events

Local Processing and Incremental Evaluation Event queries are processed locally at each 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). For efficiency reasons, (composite) event queries should be evaluated in an *incremental* manner.

Bounded Event Lifespan 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. This is consistent with the clear distinction between events as volatile data and standard Web data as persistent data.

Event queries are such that no data on any event has to be kept forever in memory, i.e. the event lifespan should be bounded. Hence, design enforces that volatile data remains volatile.

If for some applications it is necessary to make part of volatile data persistent, then the applications should turn events into persistent Web data by explicitly saving events, following the programming metaphor of XChange by turning speech into text.

2.7 Relationship Between Reactive and Query Languages

A working hypothesis of the XChange project is that a reactive language for the Web should build upon, 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. 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 time-related constructs not needed for querying Web contents.

3 The Web Query Language Xcerpt

The Web query language Xcerpt [32, 31] is *embedded* in XChange. Xcerpt is a pattern and rule-based language for querying Web data (i.e. persistent data). Xcerpt uses (query) *patterns* for querying Web contents and (construction) *patterns* for constructing new data items.

The language Xcerpt offers programmers the freedom to choose between two syntaxes for writing query programs, an XML syntax and a compact syntax where the building blocks are *terms*. The latter is used in this introduction to Xcerpt for readability and space reasons. Terms are used for denoting query patterns (*query terms*), construction patterns (*construct terms*) and also for denoting data items of Web contents (*data terms*). Common to all terms is that they represent tree or graph-like structures. In such a tree or graph structure, 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, an *ordered term specification* is denoted by square brackets [ ], an *unordered term specification* by curly braces {}.

Data Terms Data terms represent data items (i.e. 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 their Web address(es). Figure 1 presents two Xcerpt data terms that represent part of the data of a flight timetable and of a hotel directory. Note that XML element names are term labels and child elements are represented as subterms surrounded by curly braces (in case of ordered child elements, square brackets are used).

Query Terms Query terms are (possibly incomplete) patterns for the Web data that is to be queried and from which parts (subterms of data terms) are to be retrieved.

Total (complete) or *partial* (incomplete) query patterns can be specified. Partial query specifications are useful when the structure of the queried documents is not completely known, but also for minimising the terms that need to be written for meeting users' query requests. 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 retrieving data items, i.e. for selecting subterms of queried data terms. Xcerpt variables are place holders for data, very much like logic programming variables are. In Xcerpt, variables are preceded by the keyword `var`. Variable restrictions can be also specified, by using the construct `->` (read *as*), which restrict the bindings of the variables to those terms that are matched by the restriction pattern (given on the right hand side of `->`).

Example. The Xcerpt query term of Figure 2 is used to query the data found at Web site `http://airline.com` (see Figure 1) for flights from Paris to Munich.

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

At `http://airline.com`:

```
flights {
  last-changes { "2005-08-15" },
  currency { "EUR" },
  flight {
    number { "AI2011" },
    from { "Paris" },
    to { "Munich" },
    date { "2005-08-21" },
    departure-time { "10:30" },
    arrival-time { "12:00" },
    class { "economy" },
    price { "75" }
  },
  flight {
    number { "AI2021" },
    from { "Paris" },
    to { "Munich" },
    date { "2005-08-21" },
    departure-time { "17:30" },
    arrival-time { "19:00" },
    class { "economy" },
    price { "80" }
  },
  ...
}
```

At `http://hotels.net`:

```
accommodation {
  currency { "EUR" },
  hotels {
    city { "Paris" },
    country { "France" },
    hotel {
      name { "Ambassade" },
      category { "2 stars" },
      price-per-room { "62" },
      phone { "+33 1 88 8219 213" },
      no-pets {}
    },
    hotel {
      name { "Winston" },
      category { "3 stars" },
      price-per-room { "60" },
      phone { "+33 1 82 8156 135" }
    },
    hotel {
      name { "Villa Royale" },
      category { "4 stars" },
      price-per-room { "120" },
      phone { "+33 1 77 8123 414" }
    },
    ...
  },
  ...
}
```

Fig. 1. Xcerpt Data Terms

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

Fig. 2. An Xcerpt Query Term

`desc`) [32]. In order to pose queries expressing that a certain subterm should not be found in the queried data term, Xcerpt supports *subterm negation*.

Query terms are "matched" with data or construct terms by a non-standard unification method called *simulation unification* dealing with partial and unordered query specifications. More detailed discussions on simulation unification can be found in [32, 31].

Construct Terms Construct terms are patterns that make use of variables (the bindings of which are specified in query terms) so as to construct new data terms. Being templates for new data, incomplete specifications do not make sense and thus are not allowed in construct terms. They are similar to data terms, but augmented by *variables* playing the role of place holders for data retrieved in a query. Also, construct terms may contain *grouping constructs* for collecting *some* or *all* instances that result from different variable bindings.

Construct-Query Rules Construct-query rules ("rules" for short) 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 constraints (e.g. arithmetic constraints) in a so-called condition box (introduced by the keyword `where`). The `where` clause has been introduced to source out all restrictions that are not pattern-based and thus to keep patterns for the queried data as "clean" as possible.

Example. The Xcerpt rule of Figure 3 gathers information about the hotels in Paris with a price limit. Note that the variable P that is to be bound to the price per room is constrained in the `where` clause and not inside the query pattern.

```
CONSTRUCT
  answer [
    all var H ordered by [ var P]  ascending
   ]
FROM
  in { resource { "http://hotels.net" },
    accomodation {{
      hotels {{ city { "Paris" },
                var H -> hotel {{
                          price-per-room { var P } }}
           }}
     }}
   } where var P < 90
END
```

Fig. 3. An Xcerpt Construct-Query Rule

An Xcerpt program consists of one or more rules. Complex querying problems can be solved very elegantly using Xcerpt: rules are a means for structuring complex programs (keeping a clear overall structure of programs) and the *chaining of rules* (i.e. rules can query the result of other program rules) is the mechanism through which complex programs can be realised. More on Xcerpt can be found in [32] and at `http://xcerpt.org`.

4 XChange: Language Constructs

This section introduces the core constructs of XChange by means of which distributed reactive applications can be implemented.

4.1 Events and Event Messages

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 (e.g. local updates of data or system clock events); where necessary they can be given a representation as an event message.

Event messages communicate events between (same or different) Web sites. An XChange *event message* is an XML document with a root element labelled `event` and the five parameters (represented as child elements as they may contain complex content): `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. a unique identifier given at the recipient Web site). Note that all times are according to the local clocks, since no globally synchronised clock can be assumed on the Web.

An event message is an envelope for arbitrary XML content. Thus, multiple event messages can (where necessary) be nested making it possible to create trace histories. Note that XChange messages are compatible with the messages and the “message exchange patterns” of SOAP [2].

Example. Assume that a flight has been cancelled. The control point that has observed this event raises it and sends the event message of Figure 4 to `http://airline.com`. Note the use of the `xchange` namespace prefix for the element `event` and for the parameters of an XChange event message.

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 many Web sites is not adequate for a non-centrally managed structure such as the Web.

4.2 XML Syntax vs. Term Syntax

The language XChange, like the query language Xcerpt integrated in XChange, has a *compact* syntax (which is a term-based syntax where a term represents an XML document, a query pattern, or an update pattern), and an *XML* syntax. The compact syntax has been developed

```
<xchange:event xmlns:xchange="http://pms.ifi.lmu.de/xchange">
   <xchange:sender>http://control.com/point-A20</xchange:sender>
   <xchange:recipient>http://airline.com</xchange:recipient>
   <xchange:raising-time>2005-08-21T12:00:25</xchange:raising-time>
   <cancellation>
      <flight>
        <number>AI2021</number>
        <date>2005-08-21</date>
      </flight>
   </cancellation>
</xchange:event>
```

Fig. 4. An XChange Event Message

for the programmers, while the XML syntax is for machine processing. However, programmers have the freedom to choose whichever syntax they prefer.

For readability and space reasons, the compact syntax of XChange is used throughout this paper. The event message given in Figure 4 using XML syntax is represented using compact term syntax in Figure 5.

```
xchange:event {
  xchange:sender {"http://control.com/point-A20"},
  xchange:recipient{"http://airline.com"},
  xchange:raising-time {"2005-08-21T12:00:25"},
  cancellation {
    flight { number { "AI2021" }, date { "2005-08-21" } }
  }
 }
```

Fig. 5. An XChange Event Message - Term Representation

4.3 Event Queries

For detecting situations that have occurred on the Web and require a reaction to be automatically executed, incoming event messages (i.e. representations of events that have occurred on the Web) need to be queried. Section 2.2 has pointed out differences that exist between data of incoming events and data of Web resources, recognising that Web query languages are not suitable for querying event data: Real life situations need for their detection not just one event to occur, but (more often) more than one event to occur. Moreover, the temporal order of these (component) events and the specified temporal restrictions on their occurrence time points need also to be taken into account in detecting situations.

Mirroring these practical requirements, XChange offers not only *atomic event queries* but also *composite event queries.*

4.3.1 Atomic Event Queries

An *atomic event query* refers to one single event and describes a pattern for its representation (i.e. an event message). It specifies one event query term, i.e. an Xcerpt query term with an (optional) *absolute time restriction* specification. *Absolute time restrictions* are used to restrict the event instances that are considered relevant for an event query to those that have occurred (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 can be implicit (i.e. the time point of event query definition), 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 in Figure 6.

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

Fig. 6. An XChange Atomic Event Query

4.3.2 Composite Event Queries

The capability to detect and react to *composite events*, e.g. sequences of events 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 ([11] considers detecting composite events, but XChange's notion of composite events goes beyond their notion, cf. Section 6). One of the novelties introduced by XChange is the detection of *composite events*. To this aim, XChange offers *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 *composite events* (detected using composite event queries) do not have time stamps, as atomic events do. Instead, a composite event inherits from its components a beginning time (i.e. the reception time of the first received constituent event that is part of the composite event) and an ending time (i.e. 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).

Temporal Restrictions Like for atomic event queries, temporal restrictions can be specified also for composite event queries, posing temporal restrictions on the answers' constituent events. Besides absolute temporal restrictions, also *relative temporal restrictions*, given by a duration, can be specified for composite event queries. This decision is rather straightforward considering that each composite event has a length of time and restricting it may be very useful in practice. Relative temporal restrictions can be given as positive numbers of years, days, hours, minutes, or seconds and their specification follows the keyword `within` (an example is given in Figure 7 and explained later in this section).

XChange requires every (legal) composite event query to be accompanied by a temporal restriction specification. This makes it possible to release each (atomic or semi-composed composite) event at each Web site after a finite time. Thus, language design enforces the requirement of a bounded event lifespan and the clear distinction persistent vs. volatile data.

Event Compositions XChange core constructs for event compositions of event queries are shortly introduced next.

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. Figure 7 gives an XChange event query that is used to detect the notification of a flight cancellation and, within two hours from its reception, the detection of a notification informing that the accommodation is not granted by the airline.

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

```
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!"}
   }}
 ] within 2 h
```

Fig. 7. An XChange Composite Event Query Specifying Temporally Ordered Conjunction

company SAP on the 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 signalling an increase for the shares of the companies, all reports registered between these two instances.

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 (indicated by curly braces). Keyword `and` introduces such a composite event query in XChange.

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 having visited 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 (Figure 8).

```
or {
  xchange:event {{
    xchange:sender {"http://service-nimes.fr"},
    service-notification {{
      train {{  date {"2005-08-10"},
                from {"Orange"}, to {"Nimes"}  }},
      hotel {{ }}
    }}
   }},
  xchange:event {{
    xchange:sender {"http://reservations-arles.fr"},
    reservation-notification {{
      train {{ date {"2005-08-10"},
               from {"Orange"}, to {"Arles"} }},
      accomodation {{ }}
     }}
   }}
 } before 2005-05-02T21:30:00
```

Fig. 8. An XChange Composite Event Query Specifying Disjunction

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. The XChange event query of Figure 9 detects if the notification of an online reservation made on 10th of July 2005 is not received within ten days.

Occurrences constructs for event queries refer to the number of times an event query

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

Fig. 9. An XChange Composite Event Query Specifying Exclusion

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. Figure 10 gives the travel organiser's event query 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]
```

Fig. 10. An XChange Composite Event Query Specifying Quantifications

Repetitions are used for detecting e.g. every second, fourth, sixth, and so on, instance 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 quit slowly smoking so she answers only to every second call from her colleague suggesting a smoking break. Such an event query can be specified in XChange and is given in Figure 11. 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 the CaTTS system [15].

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

Fig. 11. An XChange Composite Event Query Specifying Repetitions

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 e.g. 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.

4.3.3 Event Queries' Answers

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* – contains all atomic events that are used for answering the composite event query; it is a sequence of (constituent) atomic events. Recall the example of the previous section where the answer to a temporally ordered conjunction of event queries contains *all* stock market reports that have been registered between occurrences of increase of two companies' share values. Like atomic events, composite events are represented as XML documents having an artificial root with child elements the constituent atomic events (of the sequence). One of the advantages of representing composite events as XML documents is that it allows further processing.

4.4 Actions

An XChange action 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*.

Update Terms 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 such patterns containing update operations for the data to be modified. An update term may contain different types of update operations. An *insertion operation* specifies an Xcerpt construct term that is to be inserted, a *deletion operation* specifies an Xcerpt query term for deleting all data terms matching it, and a *replace operation* specifies an Xcerpt query term to determine data terms to be modified and an Xcerpt construct term as their new value.

Example. At `http://airline.com` the flight timetable needs to be updated as reaction to the event given in Figure 4. The update term that realises this is given in Figure 12.

```
in { resource { "http://airline.com" },
    flights {{
      last-changes { var L replaceby var RTime -> "2005-08-21" },
      flight {{ number { "AI2021" }, date { var RTime },
                delete departure-time {{ }},
                delete arrival-time {{ }},
                insert news { "Flight has been cancelled!!" }
      }}
    }}
   }
```

Fig. 12. An XChange Update Term for Updating Flight Timetable

Intensional updates, i.e. a description of updates in terms of Web queries, can be specified in XChange as the language inherits the querying capabilities of the language Xcerpt. This eases considerably the specification of updates, as the next example shows.

Example. Figure 13 gives an XChange update term that specifies the modification of the used currency from EUR to Dollar. The prices for *all* flights offered by a specific airline are modified accordingly to an exchange rate.

XChange supports *complex updates* (e.g. ordered conjunction of atomic or complex updates, meaning that all specified updates are to be executed and in the specified order). In XChange, the keywords `and` and `or` denote conjunction and disjunction of updates, respectively. Like Xcerpt, XChange uses square brackets and curly braces for expressing that the order of evaluation is of importance and of no importance, respectively.

Example. Figure 16 specifies an XChange rule. After the keyword `ACTION`, the desired action to be executed is specified, namely an ordered conjunction of updates that first inserts into the data at `http://hotels.net/reservations/` a new hotel reservation for Mrs. Smith and then inserts the phone number of the hotel into her secretary's diary. The other parts of the given XChange rule are explained in the next section.

```
in { resource { "http://airline.com" },
    flights {{
      last-changes { var L replaceby var Today },
      currency { "EUR" replaceby "Dollar"},
      flight {{
        price {{ var OldPrice replaceby var OldPrice * var ExchangeRate }}
       }}
     }}
   }
```

Fig. 13. An XChange Update Term for Modifying Prices

Event Terms An XChange *event specification* is a (complete) *pattern* for the event message(s) to be constructed and sent to one or more Web sites. The notion of *event terms* is used to denote such patterns for events to be raised. An event term represents a restricted Xcerpt construct term having root labelled `event` and at least one sub-term `recipient` that specifies a Web site's address. An example of an event term is given as head of the XChange rule in Figure 14.

4.5 Rules

An XChange program is located at one Web site and consists of one or more (re)active rules of the form *Event query – Web query – Action*. Every incoming event is queried using the *event query* (introduced by keyword `ON`). If an answer is found and the *Web query* (introduced by keyword `FROM`) has also an answer, then the specified action(introduced by keyword `DO`) is executed.

Rule parts communicate through variable substitutions. Substitutions obtained by evaluating the event query can be used in the Web query and the action part, those obtained by evaluating the Web query can be used in the action part.

Example. The site `http://airline.com` has been told to notify Mrs. Smith's travel organiser of delays or cancellations of flights she travels with. The XChange rule realising this is given in Figure 14.

```
DO
 xchange:event {
   xchange:recipient {"http://travelorganiser/Smith"},
   cancellation-notification { var F }
  }
ON
 xchange:event {{
  xchange:sender {"http://airline.com"},
  cancellation {{
    var F -> flight {{ number {"AI2021"},
                       date {"2004-08-21"} }}  }}
 }}
END
```

Fig. 14. An XChange Rule for Raising Events

Example. The travel organiser of Mrs. Smith uses the following rule (specified in XChange in Figure 15): if the return flight of Mrs. Smith is cancelled then look for and book another suitable flight.

Example. If no other suitable return flight is found and the airline does not provide an accommodation, then book for Mrs. Smith a cheap hotel and inform her secretary about the changes in her schedule. The travel organiser's rule is given in Figure 16.

5 XChange: Declarative and Operational Semantics

XChange combines an event language, a query language, and an update language into ECA-rules. Accordingly, we give declarative and operational semantics separately for each rule

```
DO
  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" },
                    date { "2005-08-21" }  }}
          }}
       }}
FROM
  in { resource { "http://airline.com" },
   flights {{
     var F -> flight {{
       from { "Paris" }, to { "Munich" },
       date { "2005-08-21" }, departure-time { var T }
       }}
    }}
   } where var T after 2005-08-21T14:00:00
END
```

Fig. 15. An XChange Rule for Booking a Flight

```
DO
  and [
    in { resource { "http://hotels.net/reservations/" },
         reservations {{
           insert reservation {
                    var H, name { "Christina Smith" },
                    from { "2005-08-21" }, until { "2005-08-22" } }
         }}   },
     in { resource { "http://diary/my-secretary" },
          diary {{
            news {{
              insert my-hotel {
                       remark { "I'm staying in Paris over night!" },
                       phone { var Tel },  reason { "Flight cancellation." } }
      }}  }}   }
   ]
ON
  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 [2005-08-21T10:00:00 .. 2005-08-21T19:00:00]
  ] within 2 hour
FROM
  in { resource { "http://hotels.net" },
       accommodation {{
         hotels {{
            city { "Paris" },
            desc var H -> hotel {{
                             price-per-room { var P }, phone { var Tel } }} }}
     }}
   } where var P < 90
END
```

Fig. 16. An XChange Rule for Booking a Hotel and Announcing Flight Cancellation

part. The semantics of an XChange ECA-rule follows immediately from the semantics of its parts; the "glue" between the parts is given by the substitutions sets for the variables.

Semantics of event queries are the most interesting aspect of XChange semantics; for space reasons, we concentrate on event queries in this article and only give the underlying ideas for the semantics of Web queries and updates.

5.1 Event Queries

5.1.1 Declarative Semantics

Comparisons of (composite) event query languages such as [35], show that interpretation of similar language constructs can vary considerably. To avoid misinterpretations, clear semantics are indispensable. Furthermore, they 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 queries 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 powerful 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 have allowed a successful evaluation of q and a corresponding set of substitutions Σ for the free variables of q. We write $s = \langle a_1, \dots a_n \rangle_b^e$ 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 $rcpt(a_i) := r_i$. We have $b \leq r_1 < \dots < r_n \leq e$; 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`, but also (instances of) 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 data terms to variables. 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.

In the upcoming definition of the answering-relation we need to make statements about an event sequence being a subsequence of the event stream (which is in turn another event sequence). We say that an event sequence s is a *subsequence* (or extract) of another event sequence s', written $s \subset s'$, if the atomic events of s occur also in s'. Formally, $\langle a_1, \dots a_n \rangle_b^e \subset \langle a'_1, \dots a'_m \rangle_{b'}^{e'}$ iff $b' \leq b$, $e \leq e'$, and $\{a_1, \dots a_n\} \subset \{a'_1, \dots a'_m\}$. Similarly we say that s is a *complete subsequence* (or continuous extract) of s', written $s \sqsubset s'$, if s contains *all* atomic events of s' between the beginning and ending time of s. Formally, $\langle a_1, \ldots a_n\rangle_b^e \sqsubset \langle a'_1, \ldots a'_m\rangle_{b'}^{e'}$ iff $b' \leq b$, $e \leq e'$, and $\{a_1, \ldots a_n\} = \{a'_i \mid b \leq rcpt(a'_i) \leq e,\ 1 \leq i \leq m\}$.

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 [31].

$q = \mathtt{and}[q_1, \ldots, q_n]$: $q \lhd_{\mathcal{E}} (s, \Sigma)$ iff there exist event sequences $s_1, \ldots s_n$ s.t. (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 $s = \bigcup_{1\leq i\leq n} s_i$).

$q = \mathtt{andthen}[[q_1, q_2]]$: $q \lhd_{\mathcal{E}} (s, \Sigma)$ 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.

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

$q = q'\ \mathtt{within}\ w$: $q \lhd_{\mathcal{E}} (s, \Sigma)$ 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.

5.1.2 Bounded Event Lifespan

Our declarative semantics provide a sound basis for formal proofs about language properties. We have used it for proving the bounded event lifespan property (see 2.6) for all legal event queries. Legal event queries are atomic event queries and composite event queries that are accompanied by temporal restrictions, such as q `within` d, q `in` $[t_1..t_2]$, q `before` t_2, or `without` q `during` $[t_1..t_2]$.

More exactly, 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, this can be stated as the following theorem:

Theorem (Evaluation with Bounded Life-Span) For all legal event queries q, there exists a time bound $\beta \in \mathbb{D}$ (a length of time), such that for all time points $t \in \mathbb{T}$, all event streams $\mathcal{E}$ ($end(\mathcal{E}) \geq t$), and all answers (s, Σ) with $end(s) = t$ we have:

$$q \lhd_{\mathcal{E}} (s, \Sigma) \quad \iff \quad q \lhd_{\mathcal{E}\mid_{t-\beta}^{t}} (s, \Sigma).$$

The theorem follows immediately from two lemmas:

Lemma 1 (Bound for Answers to Legal Event Queries) For all legal event queries q, there exists a time bound $\beta \in \mathbb{D}$ (a length of time), such that for all event streams $\mathcal{E}$ ($end(\mathcal{E}) \geq t$), and all answers (s, Σ) with $end(s) = t$ we have:

$$q \lhd_{\mathcal{E}} (s, \Sigma) \quad \Longrightarrow \quad end(s) - begin(s) \leq \beta.$$

Lemma 2 (Restriction of the Event Stream) For all event queries q, all event streams $\mathcal{E}$, and all answers (s, Σ) we have:

$$q \lhd_{\mathcal{E}} (s, \Sigma) \quad \iff \quad q \lhd_{\mathcal{E}\mid_{begin(s)}^{end(s)}} (s, \Sigma).$$

Lemma 1 gives a bound β for all legal event queries. The proof works by analysing the required temporal restriction applied to the query q. (Note that the proof is not inductive!)

Proof of Lemma 1 If q is an atomic event query (denoted $q \in \mathcal{T}^q$), then let $\beta = 0$. The definition of $\lhd_{\mathcal{E}}$ for q atomic event query gives us $s = \langle d^r \rangle_r^r$, and we have $end(s) - begin(s) = r - r = 0 \leq \beta$.

If $q = q'$ `within` w, then let $\beta = w$. Immediately from the definition we get $end(s) - begin(s) \leq w = \beta$.

The other cases are similar, so we do not elaborate them here.

Lemma 2 is a more general claim about event queries (whether legal or not). Its essence is that the beginning time $begin(s)$ and ending time $end(s)$ of an answer (s, Σ) to some event query q are such that only incoming atomic events happening between $begin(s)$ and $end(s)$ have been considered in the query evaluation of q. Events happening before or after have not and will not influence whether (s, Σ) is an answer or not. Proof of Lemma 2 is by structural induction on q; again we give only a few illustrative cases.

Proof of Lemma 2 In order to be able to apply the induction hypothesis, we need to generalise our statement from $\mathcal{E}$ to $\mathcal{E}\mid_b^e$ and will prove:

$$\forall b \leq begin(s) \forall e \geq end(s). \quad q \lhd_{\mathcal{E}|_b^e} (s, \Sigma) \quad \Longleftrightarrow \quad q \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s, \Sigma)$$

As an auxiliary proposition, observe that for the subsequence relations $\sqsubset$ and $\subset$ it holds that:

$$\forall b \leq begin(s) \forall e \geq end(s). \quad s \sqsubset \mathcal{E}\mid_{begin(s)}^{end(s)} \Longleftrightarrow s \sqsubset \mathcal{E}\mid_b^e,$$

$$\forall b \leq begin(s) \forall e \geq end(s). \quad s \subset \mathcal{E}\mid_{begin(s)}^{end(s)} \Longleftrightarrow s \subset \mathcal{E}\mid_b^e .$$

The base of the induction is $\underline{q \in \mathcal{T}^q}$, i.e., q is an atomic event query.

$(\Longrightarrow)$ From left to right we start with $q \lhd_{\mathcal{E}|_b^e} (s, \Sigma)$, which gives us (1) $begin(s) = end(s) = r$ and (2) $s = \langle d^r \rangle_r^r \sqsubset \mathcal{E}\mid_b^e$ (by definition of $\lhd_{\mathcal{E}}$ for q atomic). We must show (i) $s = \langle d^r \rangle_r^r \sqsubset \mathcal{E}\mid_r^r$; but this is simply the auxiliary proposition from above applied to (1) and (2).

$(\Longleftarrow)$ From right to left we start with $q \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s, \Sigma)$, which gives us (1) $begin(s) = end(s) = r$ and (2) $s = \langle d^r \rangle_r^r \sqsubset \mathcal{E}\mid_{begin(s)}^{end(s)} = \mathcal{E}\mid_r^r$. We must show $s = \langle d^r \rangle_r^r \sqsubset \mathcal{E}\mid_b^e$. Again, this is just the auxiliary proposition applied to (1) and (2).

As an exemplary case for the induction steps consider $\underline{q = \texttt{andthen}[[q_1, q_2]]}$.

$(\Longrightarrow)$ From left to right we have (1) $q_i \lhd_{\mathcal{E}|_b^e} (s_i, \Sigma_i)$ for $i = 1, 2$, (2) $s' \sqsubset \mathcal{E}\mid_b^e$, (3) $begin(s) \leq begin(s_i)$, $end(s_i) \leq end(s)$ for $i = 1, 2$. We must show (i) $q_i \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s_i, \Sigma_i)$ for $i = 1, 2$ and (ii) $s' \sqsubset \mathcal{E}\mid_{begin(s)}^{end(s)}$.

Statement (ii) follows directly from the auxiliary proposition. For (i) apply the induction hypothesis $\Longrightarrow$ to (1) to obtain $q_i \lhd_{\mathcal{E}|_{begin(s_i)}^{end(s_i)}} (s_i, \Sigma_i)$ for $i = 1, 2$. Use of (3) and the induction hypothesis $\Longleftarrow$ on this yields (i).

$(\Longleftarrow)$ From right to left for we have (1) $q_i \lhd_{\mathcal{E}|_{begin(s)}^{end(s)}} (s_i, \Sigma_i)$ for $i = 1, 2$, (2) $s' \sqsubset \mathcal{E}\mid_{begin(s)}^{end(s)}$, (3) $begin(s) \leq begin(s_i)$, $end(s_i) \leq end(s)$ for $i = 1, 2$. We must show (i) $q_i \lhd_{\mathcal{E}|_b^e} (s_i, \Sigma_i)$ for $i = 1, 2$ and (ii) $s' \sqsubset \mathcal{E}\mid_b^e$.

Again, (ii) is just the auxiliary proposition on (2). For (i), an application of induction hypothesis $\Longrightarrow$ to (1) using (3) gives $q_i \lhd_{\mathcal{E}|_{begin(s_i)}^{end(s_i)}} (s_i, \Sigma_i)$. Now apply induction hypothesis $\Longleftarrow$ to get (i).

More detailed proofs and a deeper discussion on legal event queries can be found in [20].

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

5.1.3 Operational Semantics: Event Query Evaluation

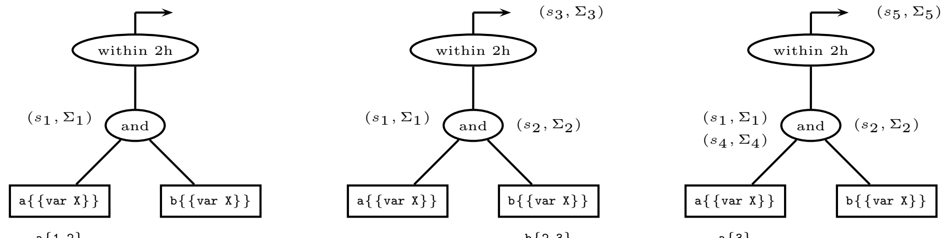
Evaluation of event queries should be performed in an incremental manner: work done in one evaluation step of an event query on some incoming atomic event should not be redone in future evaluation steps on further incoming events. To evaluate an XChange composite event query in an incremental manner, 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 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 17 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 18. For simplicity, 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 18(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 18(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.



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

At $t = 2$ we receive the event message `b{2,3}` (Figure 18(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 $\mathit{end}(s_3) - \mathit{begin}(s_3) = 1 \leq 2$ 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 18(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 $\mathit{begin}(s)$ and $\mathit{end}(s)$ for all answers (s, Σ) stored in their subtrees. In our example, all events (s, Σ) in the subtree of `within 2h` must satisfy $t - 2 \leq \mathit{begin}(s)$, 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.

5.2 Web Queries

Xcerpt, the Web query language embedded in XChange, has declarative semantics following the approach of Tarski-style semantics for first order logic. Operational semantics are based on backward-chaining of deductive rules and Simulation Unification [16], which is currently implemented by means of a constraint solver. An in-depth discussion of both declarative and operational semantics of Xcerpt can be found in [31].

5.3 Updates

Declarative semantics for updates are based on the following observation: an effect of an update can be described as transforming the data prior to the update into the resulting data. Hence, semantics of an XChange elementary update u can be expressed by an Xcerpt construct-query rule (cf. Section 3) constructing the resulting data from the prior data. For such Xcerpt rules, declarative semantics are available (see above).

Specifying update semantics now reduces to obtaining the corresponding Xcerpt construct-query rule for a given update term. This can be expressed by means of a set of rewriting rules. The underlying ideas of the rewriting process and a concrete example of obtaining a construct-query rule for an update term can be found in [30].

An advantage of this approach is that the intricacies of state-dependence usually associated with updates are lifted. Moreover, operational semantics can mirror the declarative semantics. Drawbacks are, however, that expressing an update by a transformation is not an efficient solution for operational, nor a very elegant one for declarative semantics. Better approaches are currently being investigated.

6 Related Work

The issue of automatic reaction in response to happenings of interest has its roots in the field of active databases [19, 29, 34]. In particular, the ability to react to *composite events*, i.e. possibly time-related combinations of events has received considerable attention (e.g. [17, 22, 23]). 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. Also, composite events reflecting a user- and application-centred (e.g. travel planning related situations as exemplified in this paper) and not a system-centred view are needed on the Web.

Composite Events A number of works in active databases, such as COMPOSE [23], SAMOS[22], and Snoop [17], have introduced composite events into reactive rules. The composition operators found in these systems are similar, though subtle differences exist [35]. XChange offers a richer event query language: In addition to simple operators like sequence, conjunction, or disjunction, XChange offers high-level constructs, e.g. multiple inclusions and exclusions, quantifications, and repetitions. Temporal restrictions can also be specified; they allow for detecting events that have occurred e.g. in a given time interval. XChange event queries not only detect (atomic and composite) events, data is extracted from events and used to correlate atomic events within one composite event. [4], a situation monitoring system for distributed event sources that has been proposed recently, shares with XChange the high-level nature of constructs. In contrast, however, event data plays a less important role than in XChange.

Some systems have introduced modes for event consumption and event instance selection [35], which filter the events that actually trigger rules. For example, Snoop [18] uses so-called parameter contexts for this purpose and defines the *recent* (takes only the most recent occurrences of event instances into account), *chronicle* (uses the chronological order of the notified events), *continuous* (each occurrence of an event is considered as possible component candidate), and the *cumulative* context (detected composite events include all occurrences of instances of component events). [4] shows how Snoop's operators in the recent, chronicle, and continuous contexts can be expressed in Amit. Composite event specifications of most systems are not easy to write and understand (partly because they lack a precise or good explanation of the constructs supported). Thus, when combined with event consumption or instance selection specifications, the behaviour of the specified rules is not so easy to grasp anymore. For this reason event consumption and instance selection are currently not considered in XChange. Moreover, XChange takes into account the data of events which can in many cases make event consumption and instance selection unnecessary. However, the design and implementation of the language do not preclude such extensions.

Popular approaches to composite event detection include finite state automata [23], Petri nets [22], and event trees or graphs (query trees/graphs with a bottom-up flow of events) [17, 25, 27]. Of these, the event tree approach seems to be the most widely adopted; it is used in the incremental evaluation of composite event queries in XChange, as described in Section 5.1.3. In contrast to earlier systems, XChange considers data in the form of variable bindings in the composite event detection. This aspect is very similar to the rete algorithm

[21], which describes an incremental forward-chaining algorithm for use in inference systems where over time new facts are told to the system (e.g., production rule systems).

Update Languages for the Web Most existing proposals for *update languages* for XML (such as XML-RL Update Language [24], XPathLog [26], and the update extensions to XQuery proposed in [33]) have a common trait: a path-expression is used to select nodes within the input XML document; the selected nodes are then considered as target of the update operations. This is not surprising: an update language represents an extension of a query language with update capabilities or at least needs a mechanism for selecting parts of XML documents that are to be modified. As XPath [1] (path-oriented language for addressing parts of XML documents) and XQuery [3] (query and transformation language for XML data based on XPath) are World Wide Web Consortium's recommendations, most update proposals are built upon these standards. However, these update languages support only the execution of simple updates (e.g. executing multiple updates in a desired order is not possible) and important features needed for propagation of updates on the Web are still missing.

Reactive Languages for the Web *Reactive languages* formerly developed for the Web support, like discussed already for the case of update languages, *simple* update operations on XML documents, i.e. there is no support for specifying and executing (two or more) updates in a desired order. For example, an Event-Condition-Action rule language for XML and RDF data is proposed in [28, 7], supporting as actions sequences of simple insertions or deletions to XML or RDF data. Moreover, these languages have the capability to react only to single events and do not provide constructs for querying for complex combinations of events (i.e. no composite event queries can be specified). XChange supports more general events than just modifications of data; an event can be an arbitrary, application-dependent happening, such as the flight cancellation mentioned earlier.

There is a single proposal for composite event detection *for XML documents* [11]. How this approach does (or even would) scale to the Web is unclear. This proposal does not represent a full reactive language for the Web, but it could be extended and integrated into a reactive language. Here, a *primitive event* occurs when a single node in the document tree is manipulated. *Composite events* are combinations (conjunctions, disjunctions, and sequences) of primitive and/or other composite events that have a given path type (restricted XPath expressions that determine on which path in the tree representation of an XML document node modifications have occurred). Thus, the possible situations (composite events) a user might be interested in have been restricted. Moreover, one cannot relate (primitive or composite) events that have occurred in XML documents distributed on the Web, as the communication of event data is not supported.

7 Conclusion

This article has presented the high-level language XChange for realising reactivity on the Web. 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 of event processing on the Web.

XChange is an ongoing research project. The design, the core language constructs, and the semantics of XChange are completed and have been presented here. The proof-of-concept implementation follows a modular approach that mirrors the operational semantics, the underlying ideas of which have been introduced in this paper. Issues of efficiency of the implementation, esp. for event detection and update execution, are subject to future work.

There are a couple of further research issues that deserve attention within the XChange project, such as the automatic generation of XChange rules (e.g. based on the dependencies between Web resources' data) or the development of a visual counterpart of the textual language (along this line, the visual rendering of Xcerpt programs – visXcerpt [10] – is to be extended).

Moreover, the integration with languages for specifying and reasoning with specific kinds of data is intended. Application scenarios such as travel organisation assess the practical need for reasoning with location data (e.g. to look for and book a hotel in a quiet area near to a metro station). This suggests to consider an integration of XChange with MPLL [9] (Multi Paradigm Location Language), a language for specifying and reasoning with different kinds of location data. Integrating XChange with CaTTS [15], a static typed calendar and time language that allows for declarative modelling of various calendars (e.g. Gregorian calendar, business calendars, holiday calendars, etc.), would provide the event language with richer temporal specifications.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful suggestions and comments.

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`).

References

1. *XML Path Language (XPath) 1.0*, W3C recommendation, World Wide Web Consortium, 1999.
2. *Simple Object Access Protocol (SOAP) 1.2*, W3C recommendation, World Wide Web Consortium, 2003.
3. *XQuery 1.0: An XML query language*, W3C working draft, World Wide Web Consortium, 2005.
4. Asaf Adi and Opher Etzion, *Amit – the situation manager*, Very Large Data Bases Journal **13** (2004), no. 2, 177–203.
5. James Bailey, François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan, *Flavours of XChange, a rule-based reactive language for the (Semantic) Web*, Proc. Int. Conf. on Rules and Rule Markup Languages for the Semantic Web, LNCS, no. 3791, Springer, 2005, pp. 187–192.
6. James Bailey, François Bry, and Paula-Lavinia Pătrânjan, *Composite event queries for reactivity on the Web*, Proc. Int. World Wide Web Conf. (Special interest tracks and posters), ACM, 2005, pp. 1082–1083.
7. James Bailey, Alexandra Poulovassilis, and Peter T. Wood, *An event-condition-action language for XML*, Proc. Int. World Wide Web Conf., ACM, 2002, pp. 486–495.
8. Sacha Berger, François Bry, Oliver Bolzer, Tim Furche, Sebastian Schaffert, and Christoph Wieser, *Xcerpt and visXcerpt: Twin query languages for the Semantic Web*, Proc. Int. Semantic Web Conf. (Demos track), 2004.
9. Sacha Berger, François Bry, Bernhard Lorenz, Hans Jürgen Ohlbach, Paula-Lavinia Pătrânjan, Sebastian Schaffert, Uta Schwertel, and Stephanie Spranger, *Reasoning on the Web: Language prototypes and perspectives*, Proc. Europ. Workshop on Integration of Knowledge, Semantics and Digital Media Technology, IEE, 2004, pp. 157–164.
10. Sacha Berger, François Bry, Sebastian Schaffert, and Christoph Wieser, *Xcerpt and visXcerpt: From pattern-based to visual querying of XML and semistructured data*, Proc. Int. Conf. on Very Large Databases, Morgan Kaufmann, 2003, pp. 1053–1056.
11. Martin Bernauer, Gerti Kappel, and Gerhard Kramler, *Composite events for XML*, Proc. Int. World Wide Web Conf., ACM, 2004, pp. 175–183.
12. François Bry, Tim Furche, Paula-Lavinia Patranjan, and Sebastian Schaffert, *Data retrieval and evolution on the (Semantic) Web: A deductive approach*, Int. Workshop on Principles and Practice of Semantic Web Reasoning, LNCS, no. 3208, Springer, 2004, pp. 34–49.
13. François Bry and Paula-Lavinia Patranjan, *Reactivity on the Web: Paradigms and applications of the language XChange*, Proc. ACM Symp. on Applied Computing, ACM, 2005, pp. 1645–1649.
14. François Bry, Paula-Lavinia Pătrânjan, and Michael Eckert, *Querying composite events for reactivity on the Web*, Proc. Intl. Workshop on XML Research and Applications, LNCS, no. 3842, Springer, 2006, pp. 38–47.

15. François Bry, Frank-André Rieß, and Stephanie Spranger, *CaTTS: Calendar types and constraints for Web applications*, Proc. Int. World Wide Web Conf., ACM, 2005, pp. 702–711.
16. François Bry, Sebastian Schaffert, and Andreas Schroeder, *A contribution to the semantics of Xcerpt, a Web query and transformation language*, Proc. Int. Conf. on Applications of Declarative Programming and Knowledge Management and Workshop on Logic Programming, LNCS, vol. 3392, Springer, 2004, pp. 258–268.
17. Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim, *Composite events for active databases: Semantics, contexts and detection*, Proc. Int. Conf. on Very Large Data Bases, Morgan Kaufmann, 1994, pp. 606–617.
18. Sharma Chakravarthy and D. Mishra, *Snoop: An expressive event specification language for active databases*, Data and Knowledge Engineering **14** (1994), no. 1, 1–26.
19. Klaus R. Dittrich and Stella Gatziu, *Aktive Datenbanksysteme. Konzepte und Mechanismen*, second ed., dpunkt.verlag, Heidelberg, Germany, 2000.
20. Michael Eckert, *Reactivity on the Web: Event queries and composite event detection in XChange*, Master's thesis, Institute for Informatics, University of Munich, Germany, 2005, `http://www.pms.ifi.lmu.de/publikationen#DA_Michael.Eckert`.
21. Charles L. Forgy, *A fast algorithm for the many pattern/many object pattern match problem*, Artificial Intelligence **19** (1982), no. 1, 17–37.
22. Stella Gatziu and Klaus R. Dittrich, *Samos: an active object-oriented database system*, IEEE Data Engineering Bulletin **15** (1992), no. 1-4, 23–26.
23. Narain H. Gehani, H. V. Jagadish, and Oded Shmueli, *Composite event specification in active databases: Model & implementation.*, Proc. Int. Conf. on Very Large Databases, Morgan Kaufmann, 1992, pp. 327–338.
24. Mengchi Liu, Li Lu, and Guoren Wang, *A declarative XML-RL update language*, Proc. Int. Conf. on Conceptual Modeling, LNCS, no. 2813, Springer-Verlag, 2003, pp. 506–519.
25. Masoud Mansouri-Samani and Morris Sloman, *GEM: A generalised event monitoring language for distributed systems*, Distributed Systems Engineering **4** (1997), no. 2, 96–108.
26. Wolfgang May, *XPath-Logic and XPathLog: A logic-programming-style XML data manipulation language.*, Theory and Practice of Logic Programming **4** (2004), no. 3, 239–287.
27. Douglas Moreto and Markus Endler, *Evaluating composite events using shared trees*, IEE Proceedings — Software **148** (2001), no. 1, 1–10.
28. George Papamarkos, Alexandra Poulovassilis, and Peter T. Wood, *Event-condition-action rule languages for the Semantic Web*, Proc. Int. Workshop on Semantic Web and Databases (co-located with VLDB), 2003, pp. 309–327.
29. Norman W. Paton (ed.), *Active rules in database systems*, Springer, 1999.
30. Paula-Lavinia Pătrânjan, *The language XChange: A declarative approach to reactivity on the Web*, Ph.D. thesis, Institute for Informatics, University of Munich, Germany, 2005, `http://www.pms.ifi.lmu.de/publikationen#PMS-DISS-2005-2`.
31. Sebastian Schaffert, *Xcerpt: A rule-based query and transformation language for the Web*, Ph.D. thesis, Institute for Informatics, University of Munich, Germany, 2004, `http://www.pms.ifi.lmu.de/publikationen#PMS-DISS-2004-1`.
32. Sebastian Schaffert and François Bry, *Querying the Web reconsidered: A practical introduction to Xcerpt*, Proc. of Extreme Markup Languages Conf., 2004.
33. Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld, *Updating XML*, Proc. ACM SIGMOD Int. Conf. on Management of Data, ACM, 2001, pp. 413–424.
34. Jennifer Widom and Stefano Ceri (eds.), *Active database systems: Triggers and rules for advanced database processing*, Morgan Kaufmann, San Francisco, CA, USA, 1996.
35. Detlef Zimmer and Rainer Unland, *On the semantics of complex events in active database management systems*, Proc. Int. Conf. on Data Engineering, IEEE Computer Society, 1999, pp. 392–399.