Flavours of XChange, a Rule-Based Reactive Language for the (Semantic) Web

James Bailey[1], François Bry[2], Michael Eckert[2], and Paula-Lavinia Pătrânjan[2]

[1] The University of Melbourne, Australia, http://www.cs.mu.oz.au/~jbailey/
[2] University of Munich, Germany, http://pms.ifi.lmu.de/

Abstract. This article introduces XChange, a rule-based reactive language for the Web. Stressing application scenarios, it first argues that high-level reactive languages are needed for both Web and Semantic Web applications. Then, it discusses technologies and paradigms relevant to high-level reactive languages for the (Semantic) Web. Finally, it presents the Event-Condition-Action rules of XChange.

1 Introduction

A common perception of the Web is that of a distributed repository of hypermedia documents, with clients (in general browsers) that download documents, and servers that store and update documents. Although reflecting a widespread use of the Web, this perception is not completely accurate. In fact, many Web applications rely on the updating of server data in response to the requests of clients, or client data in response to the requests of servers. Indeed, there are many kinds of reactions for both servers and clients, to events or messages exchanged on the Web. The Web has an infrastructure for updates and reactivity: the protocol HTTP. This article first argues that complementing HTTP with high-level languages for updates and reactivity is needed for both standard Web and Semantic Web applications. It then introduces XChange, a novel high-level language for Event-Condition-Action rules for updates and reactivity on the (Semantic) Web.

2 Motivation

Updates on the Web. Many Web applications build upon servers that update data according to client requests or actions. This is the case for e-Commerce systems that receive, process and buy orders, e-Learning systems that select and deliver teaching materials depending on students' test performances, and of communication platforms such as wikis, where several users modify the same documents. Conversely, some Web applications also build upon clients that update data according to server requests. This is the case with so-called *cookies,* i.e. descriptions on a client of the states of a connection to a server, or when a client keeps, after a connection to a server, data collected during the connection, e.g. for a railways or airline electronic ticket.

Reactivity on the Web. Many Web applications not only build upon the updating of data, but also upon complex *reactions to messages* or *events*, exchanged not only between clients and servers but also (via servers) between clients. This is the case when contributors to a Web-based communication platform are informed of other contributors joining in or leaving a session. It is the case for Web-based business management systems, such as business travel applications and planning and reimbursement in large companies, that rely upon complex workflows of actions and messages, possibly realized using Web services. It is also the case for Web-based systems offering context-dependent services. e.g. a time and location dependent car park directory, that adapts the information it delivers and reacts to time changes, with clients changing places and car parks announcing their free parking capacities.

Updates and Reactivity on the Semantic Web. Updates and reactivity are as much "Semantic Web issues", as they are "standard Web issues". The application scenarios stressed above, might involve both standard Web and Semantic Web data and techniques, such as HTML, XML, RDF, Topic Maps and OWL data, as well as inference from RDF triples. For example, e-Commerce offers might be described by RDF meta-data and an e-Learning system might refer to inference rules expressed in terms of RDF triples, RDFS, and OWL.

HTTP/1.1: The Infrastructure for Updates and Reactivity on the Web. Updates and reactivity on the Web are realized using HTTP/1.1, the current version of the *Hypertext Transfer Protocol.* HTTP's communication paradigm is a *client-server model* of *request-response interactions* offering among others, the requests `GET` (by which a client can retrieve from a server, information identified by a URI) and `POST` (by which a client can submit information to an "entity" on a server identified by a URI). HTTP has roles for (software) intermediaries such as *proxies, gateways,* and *tunnels*, giving rise to the specification of various forms of communication. HTTP has two kinds of messages: *requests* and *responses.* Message *headers* give rise to the specification of network communication and system parameters (e.g. whether the message sender wants to close the connection), caching directives, encoding information (e.g. indicating that the message body is encoded using the gzip file format), communication protocols other than HTTP/1.1 a client offers the server to use, the length of the message body, a base for relative URIs referred to in a message body, etc.

High-Level Languages for Updates and Reactivity on the Web. Although HTTP/1.1 can help implement updates and reactivity on the Web, as needed by the afore-mentioned applications, more abstract and higher-level languages are needed that (1) abstract away network communication and system issues, (2) ease the specification of complex updates of Web resources (e.g. XML, RDF, and OWL data), (3) are convenient for specifying complex flows of actions and reactions on the Web. The need for high-level Web update and reactivity languages is similar to the need for high-level (Semantic) Web query languages (cf. [2] for a survey). High-level reactivity languages will *complement,* not replace HTTP. Indeed, the simplest and therefore most desirable way to implement a high-level reactivity language for the (Semantic) Web, is to use HTTP/1.1.

3 Technologies and Paradigms

Atomic Events, Event Messages, and Composite Events: Web and Semantic Web applications require a number of different kinds of *atomic events:* i) events *exchanged between Web nodes*, to make it possible for a Web node to trigger reactions at remote Web nodes, ii) events *local to a Web node*, to help express local reactivity, e.g. local updates, and iii) *system* events, making it possible for reaction to the functioning, or non-functioning, of the encompassing "system(s)", e.g. the operating system of a node or of the network. A simple and natural assumption is that events exchanged between Web nodes are expressed as *event messages*, expressed in a Web format such as XML. Reacting to *composite* (or *complex*) *events,* is essential in practice. Complex events have received considerable attention in the field of active databases, cf. e.g. [16, 14]. However, differences between (generally centralised) active databases and the Web, where central clock and management are missing and message deliveries between Web nodes can be delayed, necessitate new approaches. Furthermore, composite events reflecting a user-centered, and not a system-centered view are needed on the Web.
Event Messages vs. Web Resources. Event messages and standard Web resources should be kept in two separate data kinds, since otherwise the development of the reactive Web may be insufficiently distinguishable from the (Semantic) Web. In short: *No URIs for event messages!*
Temporal Dependencies often have to be expressed when composite events are specified. An application example might be "depend on an event E_1 occurring *before* an event E_2", or "depend on an event E_3 occurring within a time interval". Thus, a reactive language requires sophisticated temporal notions and *temporal event composition constructs.*
Event-Condition-Action Rules, (*ECA rules*) fit well with the widespread and intuitive view of the Web as a distributed repository of documents mentioned above. Indeed, ECA rules build on queries by the use of "conditions". Thus, ECA rules building on a Web or Semantic Web query language are a very natural paradigm for updates and reactivity on the Web. Updates and reactivity pertain to *imperative programming*, because they refer to *state changes* on the Web. Arguably, ECA rules building on a Web query language are more convenient for reactivity on the Web, than conventional imperative programming languages.
Distributed Processing and Communication. On the Web, reactive programs call for distributed processing. Arguably, reactive languages making each Web node capable of *controlling its own reactive behaviour*, would fit the de-centralised management of the (Semantic) Web.

4 XChange in a Nutshell

XChange is a language of *ECA rules.* Each rule consists of three parts: (1) an event query, also called *"event"*, accessing (local or remote) event messages and (local) system events, (2) a Web query refereed to as the *"condition"* accessing

standard Web data, and (3) an *"action"* expressing (3.a) single updates, (3.b) transactions i.e. a group of actions to be realized in an all-or-nothing manner, or (3.c) messages to be sent to Web nodes.

The *atomic events* of XChange are happenings (e.g. an update of a possibly remote Web resource) to which each Web node (through a reactive program) may or may not react. XChange distinguishes between two kinds of atomic events: *explicit events* and *implicit events. Explicit events* are explicitly raised by a user or by an XChange program at a Web node and sent to this and/or other Web nodes as *event messages.* XChange's event messages are (arbitrary) XML documents within an *event message envelope* expressed itself as a (specific) XML document. An envelope mentions at least the *sender,* a *recipient,* and, upon reception at a Web node, the *reception time.* An envelope, might in addition, mention other recipients and the event's raising time (at the sender's node). Figure 2 presents an envelope in the *term syntax* of Xcerpt [5, 2, 18] and XChange. This syntax is a slight variation of the XML syntax with unlabeled parentheses instead of labeled tags and element names occurring immediately before the parentheses, e.g. `article[ title{"Flavour of XChange"}, body[...] ]` is the term syntax for the XML document of Figure 1. Figure 2 presents an XChange event mes-

```
<article>
   <title>Flavour of XChange</title> <body>...</body>
</article>
```

Fig. 1.

sage (note the namespace prefix `xchange`). The respective meanings of the square brackets [ ] and curly braces { } are explained in Section 5. Nesting messages with their former envelopes make it possible to track the origin of messages, removing envelopes before forwarding messages hides their origin.

```
xchange:event {
   xchange:sender { "http://www.pms.lmu.de/" },
   xchange:recipient { "http://ruleml.org" },
   xchange:recipient { "http://www.cs.mu.oz.au/~jbailey/" },
   xchange:raising-time { "2005-06-29T18:15:00" },
   info { "Here is an article for RuleML'06!" ],
   article [ title {"Flavour of XChange"}, authors [...], body [...] ]
}
```

Fig. 2. An XChange Event Message

Implicit events are local events such as updates of local Web resources and system clock events. They are not expressed through event messages. Events are transmitted from one Web node to another via event messages. Thus, an event sent from one Web site to another is necessarily explicit. *Composite events* are defined in XChange as *answers* of composite *event queries,* cf. Section 6.

XChange makes a strict distinction between *persistent data,* i.e. Web resources,[3] and *volatile data,* i.e. *by definition* events. XChange relies on the query

[3] A Web resource might be computed on request from other resources, using e.g. views, and therefore be more "dynamic" than "persistent". XChange's distinction between Web resources and events is not affected by the existence of "dynamic" resources.

language Xcerpt [5, 2, 18] for accessing persistent, i.e. Web, data. XChange uses a novel query language especially tuned to events for accessing volatile data, i.e. events. This *event query language* builds upon Xcerpt and extends it with constructs for *temporal event composition.* Event messages can be turned into Web resources, and Web resources might be included in event messages. The metaphor for XChange's distinction between "volatile data" or events, versus "persistent data" or Web resources, is that of *speech vs. written text:* Speech cannot be stored and cannot be updated once produced, but only completed by further speech, whilst written text can be stored and updated. However, (non-storable, non-updatable) speech can be turned into (storable and updatable) written text.

XChange's event query evaluation is *not* based on *event consumption:* an event already used in answering an event query can be used for answering another event query. XChange supports a limited form of *event selection* [19] with the temporal ranges of its event queries. The authors believe that rejecting event consumption and instead using "pure" event selection, greatly contributes to making the event query language closer to database and Web query languages, especially Xcerpt, thus making programming in XChange easier. Note that event consumption and selection strategies can easily be implemented using XChange rules.

XChange's *communication model* is *peer-to-peer,* i.e. all Web nodes, whether they be clients or servers according to HTTP, have the same communication capabilities and every party can initiate a communication with every other Web node. Two basic *communication strategies* are possible on a network: a *push strategy* where senders inform recipients of messages they want to send to them, and a *pull strategy* where (potential) recipients keep querying all (potential) senders for messages. Arguably, the pull strategy is convenient for querying (persistent) Web resources, while the push strategy is convenient for querying (volatile) events. XChange relies on the push strategy for event queries and on the pull strategy for Web resource queries.[4] XChange's message communication is *asynchronous,* i.e. XChange's 'send operation' is *non-blocking:* the execution of an XChange program immediately continues after a 'send operation' without waiting for the message transmission, an acknowledgement of receipt, or a reply. Note that blocking sending can easily be implemented using XChange rules.

XChange programs are processed in a *distributed* manner, each (XChange-aware) Web node processes, possibly by delegation to another Web node, the XChange programs locally specified. XChange relies neither on "super-peers", nor on central services, such as a central synchronization point.

XChange ensures a *local control of events*, as well as of *event memorisation.* A Web node might reject an update request from a remote Web node (sent in

[4] Push communication can be simulated in a pull communication framework by "continuous queries", i.e. potential recipients periodically polling all potential senders. This simulation has severe drawbacks: Increased network traffic, delayed receptions, communication initiated by potential recipients and not actual senders, potential recipients must be aware of all potential senders, etc.

an event message), e.g. because of a lack of credentials. Furthermore, the events memorized at a Web node only depend on the XChange event queries posed *at that node.* The time during which an atomic event, e.g. an event message or a local implicit event, is kept in memory at a Web node, only depends on the event queries posed *at that node.* By design, XChange composite event queries can be evaluated without keeping any event forever in memory. If this is necessary for some applications, (volatile) events should be explicitly stored as (persistent) Web resources.

XChange event queries have a *declarative semantics* expressed in terms of a *Tarski-style model theory,* i.e. recursively on the structure of composite event queries, making them easy to understand and to implement. XChange event query evaluation is *data-driven,* incoming events are used for *incrementally evaluating* queries. In contrast, the evaluation of queries against Web resources, e.g. Xcerpt queries and XChange conditions, is in general query driven.

5 Conditions or Web Queries

The *condition parts* of XChange rules are expressed in Xcerpt [5, 2, 18], a Web and Semantic Web query language. Xcerpt has *query patterns,* called *query terms,* for querying Web resources, and *construction patterns,* called *construct terms,* for re-assembling data selected by queries into new data items. Common to query, construct, and data terms, is that they represent graphs, i.e. ID/IDREF and other XML references are dereferenced by Xcerpt. The children of the node of an XML document might be either ordered or unordered, i.e. during querying with Xcerpt, their order might be modified. In the *term syntax* of Xcerpt, an ordered term specification is denoted by square brackets [ ], an unordered term specification by curly braces { }.

```
bib {  massimo @ person{ first { "Massimo" }, last { "Benedetti" } },
       article { title { "Querying XML" }, authors { ^ massimo } },
       article { title { "Updating XML"}, authors { ^ massimo } } }
```

Fig. 3. A Xcerpt Data Term

Data Terms represent Web resources, i.e. XML documents. In an Xcerpt program, the Web resources to be queried are specified using the keyword `resource`, followed by the URIs of the resources. Figure 3 presents an Xcerpt data term describing a bibliography. Note the defining occurrence `massimo @ ...` and the referencing occurrence `^ massimo` of the reference, or pointer, `massimo`.
Query Terms are (possibly incomplete) patterns that are matched against Web resources. Both *partial* (i.e. incomplete) or *total* (i.e. 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 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 subterms of data terms that are bound to the variables. *Variable restrictions* can be expressed using the `->` construct (read *as*), which restrict the bindings of the variable at the left of `->` to those terms matching with the query term at the right of `->`. Figure 4 presents a query term retrieving the authors of articles listed in the data term of Figure 3 at `http://library.com`.

```
in { resource { "http://library.com" }
     bib {{ article {{ var Wrote -> authors }} }} }
```

Fig. 4. A Xcerpt Query Term

Xcerpt query terms may use additional constructs like *subterm negation* (keyword `without`), *optional subterms* (keyword `optional`), and *descendant* (keyword `desc`) [18]. Query terms are "matched" against data or construct terms by a non-standard unification method called "simulation unification" [6].

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

Construct-Query 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 constraints (e.g. arithmetic constraints), in a so-called *condition box* (introduced by the keyword `where`). A *goal* is like a rule, except for the keyword `CONSTRUCT` replaced by `GOAL`. Goals specify data to be computed and delivered, while rules specify possible intermediate computations. A *(query) program* consists of one or more rules and of at least one goal. 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 at [5, 18].

6 Event Queries

6.1 Syntax of Atomic Event Queries

Atomic Event Queries are patterns to be "matched" against incoming events (at the Web node where the atomic event query is evaluated). They are in XChange like they are in Xcerpt, *query terms,* the only difference being that event query terms never refer to a Web resource, unlike Xcerpt query terms (recall: *No URIs for event messages!*). Figure 5 presents an atomic event query (watching events announcing flight cancellations for Massimo Benedetti) within an XChange rule (the action and condition of which are not given).

```
RAISE
  < action >
ON
  xchange:event {{
     flight-cancellation {{
        flight-number { var Nb },
        passenger {{ name { first { "Massimo" }, last { "Benedetti" } } }}
     }}
   }}
FROM
   < condition >
END
```

Fig. 5. An Atomic Event Query Within an XChange Rule

6.2 Syntax of Composite Event Queries

A *Composite Event Query* consists of (1) a connection of (atomic or composite) event queries (possibly with `where` clauses limiting variable bindings) with *event composition operators* such as `and`, `andthen`,`or`, and `without` and (2) an optional *temporal range* limiting the time interval in which event occurrences are relevant to the composite event query. *Atomic events* are event messages. *Composite events* are only defined as answers to, or instances of composite event queries. Composite events do not have time stamps, in contrast to event messages, i.e. atomic events. Instead, a composite event inherits from its components a *beginning time* (the reception time of the first event message received that contributes to the composite event) and an *ending time* (the reception time of the last event message received that contributes to the composite event).

The following notations, possibly with indices, are used in the rest of the section: `AtomicEventQuery`, `CompositeEventQuery`, `EventQuery` (an atomic or composite event query), `TimePoint`, `AbsTimeRange`(an *absolute time range,* i.e. an *anchored* finite time interval like e.g. "from June 3, 2005 to July 4, 2005"), `RelTimeRange` (a *relative time range,* i.e. an *unanchored* finite time interval like e.g. "3 days"), and `TimeRange`(an absolute or relative time range).

6.2.1 Temporal Ranges

A *Temporal Range* is a time interval during which a composite event query is to be evaluated, i.e. only events received during this time interval might be relevant to the event query. Temporal ranges are necessary for ensuring that no events received by this node might have to be kept forever for answering some composite event queries posed at this Web node. A temporal time range can be *absolute* or *relative.*

a An *absolute temporal range* has either the form `in AbsTimeRange`, or the form `before TimePoint`. Figure 6 presents two composite event queries that detect new discounts for flights from Munich to Paris.

The current prototype implementation of XChange accepts time points expressed, like in the examples of Figure 6, using the "restricted profile" of ISO 8801 [17], possibly leaving out the "time zone designator" (then considering im-

plicitly the time zone of the Web node evaluating the event query). XChange has additional constructs by which one can define absolute temporal ranges referring to the raising or reception times of atomic events, or to the beginning or ending (raising or reception) times of composite events. If a temporal range can be derived from the temporal ranges of the components of a composite event query, then no explicit temporal range need be specified for this query.

```
xchange:event {{                        xchange:event {{
   flight {{                               flight {{
      from { "Munich" },                      from { "Munich" },
      to { "Paris" },                         to { "Paris" },
      new-discount { var D }                  new-discount { var D }
   }}                                      }}
}} before "2005-08-10T14:00:00"         }} in [ "2005-08-10", "2005-08-31" ]
```

Fig. 6. Composite Event Queries with Absolute Temporal Ranges

b A *relative temporal range* has the form `within Duration.` The current implementation of XChange accepts durations expressed as numbers of years, days, hours, minutes and seconds. Figure 7 presents a (composite) event query with a relative temporal range.

6.2.2 Event Composition Operators

a *Non-Temporal Event Composition Operators*

a.1 *Conjunctions* of event queries are used for detecting instances for each specified event query regardless of their order. They have the form:
`and` { $\texttt{EventQuery}_1$, ..., $\texttt{EventQuery}_n$ }

a.2 *Inclusive Disjunctions* of event queries are used for detecting instances for one of the specified event queries. They have the form:
`or` { $\texttt{EventQuery}_1$, ..., $\texttt{EventQuery}_n$ }
An answer to this event query is the first answer of one of the $\texttt{EventQuery}_i$ the evaluation of which can be completed using the events received so far.

b *Temporal Event Composition Operators*
The *reception time* of incoming atomic events determines the temporal order of events that the temporal event composition operators refer to (the raising time is another alternative currently being investigated). The temporal event composition operators of XChange include: *orderings, event exclusions, multiple selections and exclusions, branchings,* and *occurrences.*

b.1 *Orderings*

b.1.1 *Temporally ordered conjunctions* of event queries detect successively, in terms of the event temporal order, instances of events. They have the forms:
`andthen [` $\texttt{EventQuery}_1$, ..., $\texttt{EventQuery}_n$ `]`
`andthen [[` $\texttt{EventQuery}_1$, ..., $\texttt{EventQuery}_n$ `]]`
A total specification (using `[ ]`) expresses that only instances of the $\texttt{EventQuery}_i$ ($i = 1, \ldots, n$) are of interest and are included in the answer. Instances of other

events that possibly have occurred between the instances of the `EventQuery`$_i$ are not of interest and, thus, are not contained in the answer. In contrast, a partial specification (using `[[ ]]`) expresses interest in all incoming events that have been received between the instances of the `EventQuery`$_i$. Thus, all these instances are contained in the event query's answer. Figure 7 presents an event query that detects notifications of flight cancellations that are followed, within two hours of reception, by notifications that the airline is granting no accommodation.

```
andthen [
   xchange:event {{ xchange:sender { "http://airline.com" },
      cancellation-notification {{ flight {{ number { var Nb } }} }} }},
   xchange:event {{ xchange:sender { "http://airline.com" },
      important { "Accommodation not granted!" } }}
] during 2 hours
```

Fig. 7. A Temporally Ordered Conjunction Event Query

b.1.2 *Overlappings* of *composite* event queries detect instances of event queries that overlap on the time axis of the incoming events.
`EventQuery`$_1$ and `EventQuery`$_2$ overlap if the beginning time of `EventQuery`$_1$ is before the beginning time of `EventQuery`$_2$ and the ending time of `EventQuery`$_1$ is after the beginning time of `EventQuery`$_2$, or vice versa.

Ordered and unordered specifications are possible for overlapping of composite event queries, i.e one has the possibility to express that the temporal order is of importance or not. Thus, overlappings have the forms:
`overlap [ EventQuery`$_1$`, EventQuery`$_2$ `]`
`overlap { EventQuery`$_1$`, EventQuery`$_2$ `}`

b.1.3 *Meets* for *composite* event queries detect event query instances whose components "meet" on the time axis of the incoming event. `EventQuery`$_1$ meets `EventQuery`$_2$ if the ending time of `EventQuery`$_1$ is the same as the beginning time of `EventQuery`$_2$, or vice versa. As for overlappings of composite event queries, ordered and unordered specifications are possible. The keyword for this event composition operator is `meet`.

b.1.4 *Overlappings or meets* for *composite* event queries, detect event query instances whose components overlap or meet on the time axis of the incoming events. `overlap-or-meet` is the keyword for this event composition operator.

b.1.5 *Inclusions* for event queries detect instances of events that have occurred during the time interval determined by the beginning time and ending time of an instance of a composite event query. The keyword for this event composition operator is `include`. This operator is Allen's *during* relation [1].
b.2 *Event Exclusions*
The *event exclusion* operator enables the monitoring of the *non-occurrence* of (atomic or composite) event query instances. i.e. to express event queries excluding some event query instances. Event exclusions queries have one of the forms:
`CompositeEventQuery without EventQuery`
`without EventQuery AbsTimeRange`

Figure 8 presents an event exclusion query detecting if the notification of an online reservation made on 10th of August 2005 is not received within ten days.

```
without xchange:event {{
           online-reservation-notification {{ }}
        }} in [ "2005-08-10", "2005-08-20" ]
```

Fig. 8. An Event Exclusion Query

b.3 *Multiple Selections And Exclusions* for event queries have the forms:
m `of EventQuery`$_1$`, ..., EventQuery`$_n$ `AbsTimeRange`
`atleast` m `of EventQuery`$_1$ `, ..., EventQuery`$_n$ `AbsTimeRange`
`atmost` m `of EventQuery`$_1$ `, ..., EventQuery`$_n$ `AbsTimeRange`
The first form requires $1 \leq m \leq n$. It detects instances of m of the specified event queries and the non-occurrence of instances of the $n - m$ event queries, within the given time interval. The other forms are self-explanatory. Multiple selection and exclusions event queries must always be accompanied by the specification of an absolute time range in which instances of the specified event queries are to be monitored. Note that if $m = 1$, the first form is equivalent to an exclusive disjunction. Figure 9 presents a composite event detecting notifications of either cancellation or of an in-time departure for a flight.

```
1 of { xchange:event {{ xchange:sender { "http://airline.com" },
          cancellation-notification {{ varFlight ->
             flight {{ number { "AI2021" }, date { "2005-08-10" } }} }}
       }},
       xchange:event {{ xchange:sender { "http://airline.com" },
          in-time-departure-notification {{ var Flight }} }}
} before "17:00"
```

Fig. 9. A Multiple Selections And Exclusions Event Query

b.4 *Branchings*
b.4.1 An `if-then-else` event composition operator gives rise to query different event instances, depending whether an instance of an event query (specified in the `if` part) has occurred or not.
b.4.2 The `case` event composition operator generalizes the `if-then-else` operator. An optional `else` part can be specified for detecting event query instances if none of the event queries of the `case` part could be answered.
b.5 *Occurrences*
b.5.1 *Quantifications* of event queries serve to detect event query instances occurring at least, at most, or exactly a given number of times in a time range. Quantifications have the forms:
n `times EventQuery TimeRange`
`atmost` n `times EventQuery TimeRange`
`atleast` n `times EventQuery TimeRange`
Figure 10 presents a composite event query making it possible to react only after three identical notifications from a secretaries pool within 2 hours.

```
at least 3 times xchange:event {{
    xchange:sender { "http://xchange.com/secretaries/" },
   var Notification -> important {{ }}
}} within 2 hours
```

Fig. 10. An Event Query Specifying 3 Occurrences of A Same Notification

b.5.2 *Ranks* serve to detect instances of event queries with a given position, or rank, in the flow of events. Composite event queries with ranks have the forms:
`EventQuery withrank` n `TimeRange`
`last EventQuery TimeRange`
Negative ranks denote event query instances counted from the end of the event query sequence, with -1 denoting the last occurrence. Event queries with last or negative ranks must have a time range so as to determine the last incoming event. Figure 11 presents an event query detecting the last notification before 10:00 of a delayed arrival.

```
last xchange:event {{
  xchange:sender {"http://airline.com"}, delay-notification {{var Notif}}
}} before "10:00"
```

Fig. 11. An Event Query With Rank

b.5.3 *Repetitions* are used to detect e.g. every second, third, etc. instances of a specified event query in a given time range. They have the form:
`every` n `EventQuery TimeRange`
Figure 12 presents a composite event query making it possible to react only to every fourth message labeled `important` from the secretaries pool within a workday, a temporal type defined using the calendar system CaTTS [4], the integration of which with XChange is currently underway.

```
every 4 xchange:event {{
   xchange:sender { "http://xchange.com/secretaries/" }, important {{ }}
}} within workday
```

Fig. 12. An Event Query Specifying a Repetition

6.3 Semantics of Event Queries

XChange event queries have a *declarative semantics* [9] defined in terms of a *Tarski-style model theory,* i.e. the valuation of a composite event query with respect to a stream of incoming events is defined recursively on a query's structure. An answer to an event query, i.e. an *event query instance,* can take *two forms:* an *event stream* form and a *"resource" form.*

The *event stream form* of an event query instance, is the stream of events selected from the incoming stream by the event query. This form ensures the *answer-closedness* of the event query language, i.e. answers are of the same kind as the data queried. Answer-closedness is a necessary feature for modular programs specifying computations in a stepwise manner.

The *"resource" form* of an event query instance is an XML document (storable as a Web resource, hence the name) structured after the event query and expressed either in the term syntax or in the XML syntax of XChange event queries.

The "resource" form of an event query instance is obtained by binding a variable using the Xcerpt `->` ('as') construct to the event query, thus *"resource-ifying"* event query instances. This variable serves as a "handle" for "resource-ifying" event instances and thus making them usable in the "condition" and/or "action" of an XChange rule. "Resource-ification" of event query instances is needed because of XChange's position: *"No URIs for event messages!"*

XChange event queries have a *procedural semantics* which is *data driven,* i.e. the evaluation of a composite event query is performed *incrementally*, using the atomic events as they are received. The procedural semantics is *sound* and *complete* with respect to the declarative semantics.

7 Actions

The *action* of an XChange rule can be (1) the specification of (one or more) events to *raise,* i.e. the specification of event messages to send to (one or several) Web nodes, (2) (one or more) *update* requests to (one or several) Web nodes, or (3) (one or more) *transactions,* i.e. groups of events to raise and/or update requests to perform in an *all-or-nothing* manner: The notification by a remote Web node of the rejection of a part of a transaction, is interpreted by the (XChange-aware) Web node that had emitted the transaction as the rejection of the whole transaction. Specific *transactional event messages* are provided for notifying the rejection of update requests or transactions.

XChange transactions obey the *ACID properties* [15] (Atomicity, Consistency, Isolation, and Durability). They can be called *weak* because, in contrast to database systems, XChange does not automatically *roll-back* rejected transactions, but leaves this task to the Web node that emitted the transaction, where it can be implemented with specific XChange rules. This is consistent with the XChange *communication model* discussed in section 4, especially with XChange's *non-blocking* 'send operation', and with XChange's *local control of events* mentioned in section 4. The database systems style of automatic rollback would contradict these two paradigms of XChange and the de-centralised management of the Web.

XChange update requests are expressed by *update terms* built up from Xcerpt's query and construct terms: subterm *deletions* are specified similarly to Xcerpt *query terms*, so as to make possible *intensional* specifications of deletions, subterm *insertions* similarly to Xcerpt *construct terms.* Figure 13 presents a composite action (1) canceling all appointments of Benedetti on the 10th of August 2005, (2) notifying via email all the persons concerned by the cancellations, and (3) inserting an novel appointment for Benedetti on that same day. Note the intensional specification of the deletions using the variables `DelApp` and `EmailAdd`, the grouping `all`, and the temporally ordered conjunction `andthen [...]` specifying in which order the three action components should be performed. Enclosing an composite action in a term labeled `transaction` makes it a transaction.

```
andthen [
   in { resource { "organizer://xchange.com/~benedetti/" },
      organizer {{ delete var DelApp appointment {
         receiving { last { "Benedetti" } },
         visiting {{ email { var EmailAdd } }},
         when {{ day { "2005-08-10" } }}
       } }}
    },
   in { resource { "organizer://xchange.com/benedetti/" },
      organizer {{ insert appointment {
         receiving { first { "Massimo" }, last { "Benedetti" } },
         when { day { "2005-08-10" }, from { "9:00" }, to { "18:00" } }
      } }}
   },
   all raise {
      in { resource { "mailto:" var EmailAdd },
         appointment-cancellation { var DelApp }
      }
 }
]
```

Fig. 13. Composite Ordered Action Specifying Updates and Raising Messages

8 Related Work And Conclusion

Allen's Temporal Relations [1] have been an important inspiration in defining the temporal event composition operators.
Active Databases prototype systems have been developed that provide sophisticated event algebras, e.g. [8, 7, 11]. Their composite events have been an inspiration for XChange event query language.
High-Level Reactive Languages for the Web formerly developed, e.g. [13], support *simple* update operations on XML documents. They offer no means to specify several updates to be executed in a given order or in an *all-or-nothing* manner. Other related work can be found in [12], where Xyleme is described. Xyleme is a system for monitoring and subscription on the Web. *Alerters* monitor simple updates of Web resources, a *monitoring query processor* performs more complex event detection and send notifications of events to a *trigger engine* which performs actions. The reactive functionality of Xyleme is highly tuned to its specific application field.

XChange significantly differs from and/or extends over the above-mentioned approaches with (1) its structured event messages, (2) its distinction between Web resources and event messages, (3) its logical variables possibly shared by its event queries, conditions, and actions, (4) its weak transactions, (5) its declarative semantics, and (6) its communication and distributed processing models.

This article has introduced XChange, stressing its principles and outlining its syntax, semantics, and processing. XChange is an on-going research prototype, which was presented in an earlier state in [3]. The present paper extends this previous publication by reporting on the newly developed temporal composite

event operator. On-going work on XChange is devoted to the transaction specification language, the transactional messages, event query optimization, and on integrating the CaTTS calendric type system [4]. A promising perspective for future work is to extend XChange with security functionalities, especially *authentication* and *authorization*. The protocols of a Grid architecture such as Globus [10], would provide means for such an extension. Conversely, XChange could be used as the core of a high-level reactive language for advanced services on the Grid.

References

1. J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Comm. ACM*, 26:832–843, 1983.
2. J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and Semantic Web Query Languages: A Survey. In *Reasoning Web*, LNCS 3564. Springer-Verlag, 2005.
3. F. Bry and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *Proc. 20th Annual ACM Symp. Applied Computing*, 2005.
4. F. Bry, F.-A. Rieß, and S. Spranger. CaTTS: Calendar Types and Constraints for Web Applications. In *Proc. 14th Intl. World Wide Web Conference*, 2005.
5. F. Bry and S. Schaffert. A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proc. Int. Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, 2002.
6. F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proc. Int. Conf. Logic Programming*, LNCS 2401. Springer-Verlag, 2002.
7. A. Buchmann, A. Deutsch, and J. Zimmermann. The REACH Active OODBMS. In *Proc. ACM SIGNMOD Int. Conf. on the Management of Data*, 1995.
8. S. Chakravarthy and D. Mishra. SNOOP: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(1), 1994.
9. M. Eckert. Reactivity on the Web: Event Queries and Composite Event Detection in XChange. Master's thesis, Inst. for Informatics, Univ. Munich, Germany, 2005.
10. I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid. Enabling Scalable Virtual Organizations. *Int. Jour. of Supercomputer Applications*, 2001.
11. R. Meo, G. Psaila, and S. Ceri. Composite Events in Chimera. In *Proc. 5th Int. Conf. on Extending Database Technology*, 1996.
12. B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML Data on the Web. In *Proc. ACM SIGMOD Conf. on the Management of Data*, 2001.
13. G. Papamarkos, A. Poulovassilis, and P. Wood. Event-Condition-Action Rules Languages for the Semantic Web. In *Proc. Workshop on Semantic Web and Databases*, 2003.
14. N. W. Paton. *Active Rules in Database Systems*. Springer-Verlag, 1999.
15. J. D. Ullman. *Principles od Database and Knowledge-base Systems*, volume 1. Computer Science Press, 1988.
16. J. Widom and S. Ceri. *Active Database Systems*. Morgan Kaufmann, 1996.
17. M. Wolf and C. Wicksteed. Date and Time formats. W3C Note, 1997.
18. Xcerpt. http://xcerpt.org.
19. D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proc. 15th Int. Conf. Data Engineering*, 1999.