REACTIVITY ON THE WEB
PARADIGMS AND APPLICATIONS OF THE LANGUAGE XCHANGE

FRANÇOIS BRY
Institute for Informatics, University of Munich, Oettingenstr. 67
D-80538 Munich, Germany
francois.bry@ifi.lmu.de

PAULA-LAVINIA PĂTRÂNJAN
Institute for Informatics, University of Munich, Oettingenstr. 67
D-80538 Munich, Germany
paula.patranjan@ifi.lmu.de

Abstract

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, Semantic Web systems as well as Web services and Grids. After motivating the need for reactivity on the Web through an application scenario, 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. Finally, it sketches the implementation of a reactive Web-based application in XChange.

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data on the Web has several reasons: new information comes in, calling for insertions of new data; information is out-of-date, calling for deletions and replacements on data.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.

Evolution of data on the Web comprises updating Web resources' data and propagating updates on the Web. Thus, one can differentiate between

- *local evolution* that refers to updating data (inserting new data, deleting data, or replacing data) of a Web resource, and

- *global evolution* that refers to updating data of Web resources as consequence to remote updates (updates performed at other Web resources). Global evolution subsumes local evolution.

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. 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. From the authors' point of view reactivity *subsumes* evolution. Local evolution can be realised to some extent without reactive capabilities; global evolution, however, needs reactivity in order to propagate updates over related Web resources.

Following a declarative approach to reactivity on the Web, a novel reactive language called *XChange* [3, 9, 11] has been developed. The paradigms upon which XChange relies and its language constructs are introduced by this article. The novelty of the language is represented by the proposed *programming metaphor* intended to ease the language understanding and the supported *reactive features* tailored to the characteristics of the Web. Realising this has presupposed refining, extending, and adapting to a new medium some of the concepts on which active database systems are built upon.

This article is an extended and enhanced version of [11]. Its structure is as follows: Section 2 motivates the need for a language for programming reactive behaviour on the Web by briefly describing a motivating application scenario. Section 3 introduces the paradigms upon which the high-level, reactive language XChange relies. Section 4 gives a brief introduction into the Web query language Xcerpt, which is embedded in XChange. Section 5 subsequently describes the core language constructs of XChange accompanied by examples that partly implement the application scenario of Section 2. 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 A Motivating Application Scenario

The need for finding a solution for realising reactivity on the Web has been already recognised (e.g. in [2, 10, 24]) as a *must-have* ability of the actual Web. This section motivates the practicability of reactive abilities through a simple, everyday life application scenario – an *organising travels* scenario. Organising travels presupposes planning the desired travel, making the necessary reservations (e.g. booking flights and overnight stays), but also recognising changes that might affect the already made plan and reacting to them by adapting it properly.

Informally, it is clear what everyone understands under *booking* (a hotel, a flight, etc.). But, let's look behind the curtain to understand what booking on the Web involves: booking means changes of (i.e. updates to) one or more Web resources. E.g. booking a flight on the Web represents inserting into the airline's documents the information regarding the desired flight (number, date, departure city, arrival city) and the person who is going to travel (name, dietary requirements, credit card information). Moreover, the number of free seats for the respective flight needs to be changed. In case that the booking of flight has been done through a travel agency, changes need to be realised on some of the travel agency's documents, but they also need to be propagated to the airline offering the flight.

The task of organising travels implies two subtasks to be accomplished: *initial planning* and *adapting plans to changes*, which are detailed in the following by using the following concrete application scenario:

Mrs. Smith uses a travel organiser that plans her trips and reacts to happenings that could influence her schedule. One of the travel organiser's tasks is to plan Mrs. Smith's vacation in France. Mrs. Smith wants to visit Orange, Arles, Nîmes, Marseilles, and Paris. The trip should begin on 7th of August 2005 and end on 21st of August 2005.

Initial Planning The *initial planning* subtask of organising travels consists of two phases (what these phases might involve is exemplified with respect to the application scenario introduced above):

- Gathering the information about transportation, overnight stays etc. required for the travel, and developing a travel plan based on this information. This involves:
  - Searching for suitable flights from Munich to Paris and back. Schedules and price tables of several airlines have to be queried and compared. Only flights departing on 7th of August 2005 and arriving on 21st of August 2005 before 21:30h are of interest. Moreover, the search is constrained by a price limit of EUR 400.
  - Querying schedules and prices for train or flight connections for Paris – Orange – Arles – Nîmes – Marseilles – Paris. Like searching for flights between Munich and Paris, queries have time and price constraints.
  - Querying hotel reservation services' data to find suitable hotels in the cities Mrs. Smith wants to visit. Suitable refers to the quality of services (e.g. at least 2 stars), the prices (e.g. price per night for a room should be cheaper than EUR 90), the time period for which rooms are available for booking (e.g. Mrs. Smith wants to book a hotel in Paris from 17th of August 2005 until 21st of August 2005), and the location of the hotels (e.g. a quiet area near to a metro station).
  - Receiving notifications from different kind of information systems, like weather forecast services, or services announcing exhibitions. Moreover, reasoning with notifications' data and data of Web resources is needed for planning departures and arrivals but also for planning entertainment. For example, the weather forecast information can be used together with exhibition notifications in order to plan visiting an exhibition on a rainy day.
- Booking the travel according to the (initial) travel plan that resulted from (i). This involves:
  - Booking a flight from Munich to Paris and back and a suitable hotel in Paris. The hotel reservation needs to take into account the arrival and departure dates, which depend on the booked flight. Such reservations implying dependent changes to different Web resources need to be executed in an *all-or-nothing manner*, since e.g. a hotel reservation without a flight reservation is useless.
  - Making train reservations and corresponding hotel reservations in order to visit also Orange, Arles, Nîmes, and Marseilles.
  - Making arrangements for the desired entertainment programme, for example buying tickets for an operetta festival in Marseilles.

Adapting Plans to Changes The *adapting plans to changes* subtask of organising travels can be divided into two phases (again exemplified with respect to the scenario used throughout this section):

- Recognising (detecting) changes that might affect already made plans. Examples of such changes are:
  - The flight booked for Mrs. Smith from Paris to Munich on 21st of August 2005 has a delay. In such a case, Mrs. Smith's new arrival time in Munich and her appointments need to be taken into account (implying again reasoning with notifications' and Web resources' data), before deciding how to react to such a happening.
  - The flight booked for Mrs. Smith from Paris to Munich on 21st of August 2005 has been cancelled. In such a case, there are two possibilities: either the airline provides an accommodation for the night of 21st to 22nd of August 2005, or the airline does not provide such an accommodation. These happenings need not only to be detected but also combined in order to detect a situation of interest (in this case, an ordered conjunction of changes).

    - A train, for which Mrs. Smith has a reservation, has a delay.

- Reacting to changes of interest (like the ones presented above) by adapting already made travel plans. The following reactions to the changes noted above can be conceived:

    - In the case of flight cancellations, the travel organiser could wait for a notification regarding accommodation only a fixed amount of time (e.g. it waits 2 hours from the reception time of the notification regarding the flight cancellation). The reaction to be taken depends on this notification.

        * If an accommodation is granted, the travel organiser needs to announce Mrs. Smith's delay to the affected persons (e.g. her secretary, her boss, colleagues).
        * If the airline does not grant an accommodation, beside announcing the delay of Mrs. Smith, the travel organiser needs to book an extra overnight stay in Paris for Mrs. Smith.

    - Train or flight delays require reactions similar to the ones presented above. Cancelling train, flight, or hotel reservations can be also conceived as results to such happenings.

The above presented application scenario poses requirements that proposals for realising reactivity on the Web should fulfil. Concrete, a personalised travel organiser should have the capability to

- query and reason with data of Web resources,
- update data of Web resources and propagate these updates to possibly interested Web sites,
- detect not only single notifications but also (time related and time restricted) combinations of notifications,
- reason with notifications' data, and
- automatically react to situations of interest by sending notifications or executing updates (possibly in an all-or-nothing manner).

Existing Web-based applications support (to some extent, since one can not execute updates in an all-or-nothing manner) the subtask of initial planning but this still requires a lot of human interaction. Beside XChange, there are no other means for easily programming Web applications capable of fully accomplishing the subtask of adapting plans to changes.

The scenario given above is, of course, simplified. A real life application of this kind would probably require more details, more features, and therefore more complex reactions. Nonetheless, the scenario given above stresses salient aspects of reactivity on the Web.

The scenario given above is specific to a class of applications. Arguably, many other different application classes, such as Web service deployment, Web-based workflows, e-commerce, e-learning, share "reactivity traits" with the scenario given above.

3 XChange: Paradigms

In order to meet the requirements for realising reactivity on the Web, which have been revealed by developing use cases such as the one presented in the previous section, a novel language, *XChange*, for programming reactive behaviour on the Web has been developed. This section introduces the paradigms upon which the language XChange relies.

3.1 Volatile vs. Persistent Data

An *event* is a happening on which each Web site (through a reactive program) may decide to react in a particular way or not to react to at all. For example, an insertion of new discounts for flights, a query posed to Web resources, 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 XChange, incoming events are represented as XML documents (see Section 5).

For detecting situations of interest, incoming events (more precise, their representation) need to be queried. In order to determine what abilities such queries need to have and thus to determine if existing Web query languages (such as XQuery [37] or Xcerpt [31]) can also be used for querying events, the differences between Web resources and incoming events (if they exist) need to be revealed.

Data of incoming events and data of Web resources are indeed different. To better understand the differences one should imagine data of incoming events like *speech* and data of Web resources like *written text*.

- Speech is like a stream of words or propositions (phonology work exist that refer to the notion of *stream of speech* [23]). Considering two particular moments in time, for example during a researcher's presentation, the information "received" through speech differ in the sense that at the most recent moment one has more information as before. Still, one can not predict what information he/she will receive by the end of the presentation. Likewise, events are received in a stream-like manner. If one is interested in a particular sequence of events and such a sequence has not been entirely received until the present moment, one needs to wait to see whether other events of interest will also occur. The article the researcher is explaining in its presentation is written text, thus reading it for selecting (querying) information of interest would give the same information independent from the time point of reading.

- Speech cannot be modified. If one has communicated some information in this way one can correct, complete, or invalidate what one has told – through further speech. In contrast, written text can be updated in the usual sense. Likewise, data of incoming events is *not* updatable but data of Web resources is updatable. To inform about, correct, complete, or invalidate former events, new notifications are communicated between Web sites.

Thus, a distinction is made between *volatile data* (data of events) and *persistent data* (data of Web resources). For querying volatile data XChange offers so-called *event queries*, which are introduced in Section 5.3. This distinction not only reflects the kind of differences noted above, but, through the analogy with speech and written text, it is intended to ease the understanding of and thus programming with a reactive language for the Web.

3.2 Composite Events Defined as Answers to Event Queries

Composite event queries allow to recognise temporal patterns over incoming events – to recognise *composite events*. The notion of composite events has no precise definition in the literature. XChange's (occurrences of) *composite events* are defined through *composite event queries* (see Section 5.3) – they are *answers* to composite event queries. This is a novel way of defining composite events, but the authors consider it the only intuitive one. E.g. an XChange event query can ask for occurrences of an increase of share values by more than 5 percent for the company Siemens, followed by an increase of share values for the company SAP 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 signaling an increase for the shares of the companies, all reports registered between these two instances. The capability to query for all events having a particular pattern that have occurred between the instances of two specified event queries is one of the novelties of the language XChange.

3.3 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. XChange assumes no instance controlling (e.g. synchronising) communication on the Web, even though such instance can be realised using XChange.

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 meaningful. The pull strategy is supported by languages for Web queries (e.g. XQuery or Xcerpt), i.e. queries to persistent data. Therefore, so as to complement the framework, XChange offers the *push* strategy. The push strategy requires event queries to be incrementally evaluated (by so-called *event managers*). In the case of XChange, this is done at every (XChange-aware) Web site.

3.4 Transactional Reactivity

Complex Updates An *elementary update* is a change (i.e. insert, delete, or replace operation) within a persistent data item (e.g. XML or RDF data). *Complex updates* expressing ordered or unordered conjunctions or disjunctions of (elementary or complex) updates are offered by XChange. Such updates are required by real applications. E.g. when booking a trip on the Web one might wish to book an early flight *and* of course the corresponding hotel reservation, *or* else a late flight *and* a shorter hotel reservation. The application scenario of Section 2 has motivated the need for executing such complex updates in an *all-or-nothing manner*. Thus, XChange has a concept of transactions [32].

Transactions and ACID Properties XChange transactions obey the ACID properties (Atomicity, Consistency, Isolation, and Durability) [32]. Atomicity and isolation are considered in XChange, the issues of consistency and durability for transactions are currently not investigated in the project. XChange will build on standard solutions from database systems.

Transactional Events Transactional events (i.e. commit, abort, or request transaction) are offered by XChange. They are needed for supporting transactions.

3.5 Processing of Events

Local Processing No central processing of event queries is assumed as such an approach is not suitable on the Web. Instead, 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), and for releasing event query instances after a finite time.

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

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. Neither Web queries nor event queries posed at other Web sites can influence the size of the event history. This is consistent with the clear distinction between events as volatile data and standard Web data as persistent data. The time period for which an atomic event is kept in memory at a Web site is automatically detected from the event queries already posed at this Web site.

Event queries need to be in such a way that no data on any event can be kept for ever in memory, i.e. the event lifespan should be bounded. In the Web context it is not desirable to save the whole history of incoming events as perhaps the most of them are not even of interest, like a kind of spam for notifications. By design, XChange event queries are such 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 for turning speech into text.

3.6 Authentication, Authorisation, and Accounting

XChange in its present stage of development does not offer specific means for security (especially authentication and authorisation). However, such extensions are neither incompatible with the current version of XChange nor precluded in future versions of the language. The protocols of a Grid architecture (such as Globus [18]) would provide with convenient means for such an extension. Extending XChange with accounting functionalities is a promising perspective for future research. Vice versa, XChange could be seen as a (core of a) high-level reactive language for Grids.

3.7 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 constructs not needed for querying Web contents.

4 The Web Query Language Xcerpt

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

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. 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 the Web address(es) where the data is to be found. Figure 1 presents two Xcerpt data terms that represent part of the data of a flight timetable and of a hotel reservation offer. 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).

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" }
     },
    ...
  },
  ...
}
```

Figure 1: Xcerpt Data Terms

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.

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

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

Figure 2: Xcerpt Query Term

Xcerpt query terms may be augmented by additional constructs like *subterm negation* (keyword `without`), *optional subterm specification* (keyword `optional`), and *descendant* (keyword `desc`) [31]. In order to pose queries expressing that a certain subterm should not be found in the queried data term, Xcerpt supports *subterm negation.* A simple example is given next to illustrate the use of it.

Example 2. Assume that Mrs. Smith is interested in information about hotels in Paris where pets are allowed. Figure 3 gives an Xcerpt query term that gathers such information. Posing this query term against the data found at `http://hotels.net` (an excerpt of which is given in Figure 1) retrieves as bindings for the variable `H` the subterms containing information about the hotel Winston and Villa Royale, respectively.

```
accommodation {{
  city {"Paris"},
  var H -> hotel {{
             without no-pets {}
            }}
}}
```

Figure 3: Xcerpt Query Term Specifying Subterm Negation

For specifying optional patterns inside query terms, Xcerpt offers the `optional` construct. The query term of Figure 4 specifies, by means of `optional`, that if a `county` subterm is found in the queried data term then this subterm should be bound to the variable `C`. In the case that no `county` subterms exist in a data term labelled `accommodation`, the query term still matches the data term but the variable has no binding. The query term posed against the data term containing information on hotels of Figure 1 retrieves no binding for the variable `C`, as no `county` subterm is found.

```
accommodation {{
  optional var C -> county {{ }}
}}
```

Figure 4: Xcerpt Query Term Specifying Optional Subterms

As already seen, partial specifications in Xcerpt query terms express incompleteness in breadth. For expressing incompleteness in depth, Xcerpt offers a special construct – the *descendant* construct. The informal meaning of an expression `desc subterm` inside a query term is that the given subterm is to be found in the queried document(s) but its depth in the document tree is unknown.

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 [31, 30].

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 (short rules) relate a construct term (introduced by the keyword `CONSTRUCT`) to a query (introduced by the keyword `FROM`) consisting of AND and/or OR connected query terms. Queries or parts of a query may be further restricted by 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 3. The Xcerpt rule of Figure 5 gathers informations about the hotels in Paris with a price limit. Note that the variable 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
```

Figure 5: Xcerpt Construct-Query Rule

An Xcerpt program consists of one or more rules. Complex querying problems can be solved very elegant by using Xcerpt: rules are 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 [31] and at `http://xcerpt.org`.

5 XChange: Language Constructs

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

5.1 Events and Event Messages

Events XChange distinguishes between two kinds of atomic events: *explicit* events and *implicit* events. *Explicit events* are explicitly raised by a user or by a (predefined) XChange program. They are raised at a Web site and sent internally or to other Web sites through *event messages*. *Implicit events* are local events not expressed through event messages (e.g. local updates of data or system clock events). Events are transmitted from one Web site to another through event messages. Thus, an event sent from one Web site to another is necessarily explicit.

The kinds of *events* considered in XChange are presented in Table 1. An *update* executed or a *query* posed locally at a Web site are for XChange local events, i.e. raised at this Web site and processed at this Web site. *Transactional events* (transaction commit, transaction abort, transaction request) are local events needed as XChange supports the concept of transactions (cf. Section 3). *System events* (e.g. system clock events) are events that are coming from the encompassing "system" and might be useful to handle together with explicit and/or implicit events. A system event might be explicit or implicit, depending whether or not it is transmitted from one Web site to another.

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

Table 1: XChange Events

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

Event Messages *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. an identifier given at the recipient Web site). An event message is an envelope for an arbitrary XML content. Thus, multiple event messages can (but not necessarily) 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 [36].

Example 4. Assume that a flight has been cancelled. The control point that has observed this event raises it and sends to `http://airline.com` the event message of Figure 6.

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

Figure 6: XChange Event Message - Term Representation

Note the use of the `xchange` namespace for the keyword `event` and for the parameters of an XChange event message. The examples are intended to give flavours of the XChange constructs and thus abstract away from a particular communication protocol. In the previous example `control` denotes a communication protocol used by the airline and its control points.

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.

5.2 Compact Syntax vs. XML 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 for the programmers, while the XML syntax is for machine processing. However, programmers have the freedom to choose whichever syntax they prefer. Thus, the example given previously using the XChange's compact syntax is given next using the XML syntax. For readability and space reasons, the compact syntax of XChange is used throughout this paper.

Example 5. The event message given in Figure 6 is represented using XML syntax as given in Figure 7.

```
<xchange:event xmlns:xchange="http://pms.ifi.lmu.de/xchange">
   <xchange:sender>control://controlpoint-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>
```

Figure 7: XChange Event Message - XML Representation

5.3 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 3.1 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. For this reasons, XChange offers *event queries* – queries against event data.

Real life situations, like the ones exemplified by the application scenario of Section 2, 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.*

5.3.1 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 it follows the keyword `before`.

Example 6. 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 8.

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

Figure 8: XChange Atomic Event Query

5.3.2 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 ([7] 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 processing 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 9 and explained later in this section).

XChange requires every 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. The practical need for total *and* partial specifications for such event queries has been already motivated by the examples of Section 3.2.

Example 7. Figure 9 gives an XChange event query is used to detect the notification of a flight cancellation and afterwards, within two hours from its reception, the detection of a notification informing that the accomodation is not granted by the airline.

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

Figure 9: XChange Composite Event Query Specifying Temporally Ordered Conjunction

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

Example 8. Mrs. Smith wants to visit an exhibition of G. Barthouil on a rainy day. The XChange event query of Figure 10 is used to detect the conjunction of the exhibition notification and the desired weather forecast notification that are sent by appropriate Web services.

```
and {
  xchange:event {{
    xchange:sender {"http://artactif.com"},
    exhibition {{ painter {"G. Barthouil"},
                  location {"Marseilles"},
                  time-interval { var TI }
               }}
   }},
  xchange:event {{
    xchange:sender {"http://weather.com"},
    forecast { date { var D }, city {"Marseilles"},
               info {"It's going to rain."} }
   }}
 } before 2005-08-16T11:15:00
 where var D included-in var TI
```

Figure 10: XChange Composite Event Query Specifying Conjunction

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 9. After Orange, Mrs. Smith wants to visit Arles and Nîmes. The next city to visit is chosen depending on the notification of train tickets and hotel reservation made by appropriate services (Figure 11).

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 10. The XChange event query of Figure 12 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 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*.

```
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
```

Figure 11: XChange Composite Event Query Specifying Disjunction

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

Figure 12: XChange Composite Event Query Specifying Exclusion

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 11. Figure 13 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]
```

Figure 13: XChange Composite Event Query Specifying Quantifications

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

Example 12. 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 14. 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 [12].

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

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

Figure 14: XChange Composite Event Query Specifying Repetitions

of event query instances and the non-occurrence of instances of the other specified event queries. It expresses a generalised exclusive disjunction of event queries. *Overlapping* and *meet* constructs for composite event queries detect instances of the specified event query if their component (composite) events overlap or meet, respectively, on the time axis of the incoming events.

5.3.3 Event Queries' Answers

Before explaining what answers to XChange event queries are and how they are represented, let's take a look at how event queries are processed in order to obtain answers to them.

XChange atomic event queries are patterns for incoming event instances that are of interest for a Web site (cf. Section 5.3). The event manager of an XChange-aware Web site tries to *match* each incoming event received with the currently posed atomic event queries (which themselves may be part of composite event queries). The matching of an atomic event query with an incoming event is based on the *simulation unification* [30], a novel unification method developed for matching query terms (i.e. Web queries) with data or construct terms (i.e. persistent data) in Xcerpt. The volatile nature of events does not preclude the usage of the same method for *simulating* atomic event queries into incoming events.

Composite event queries are made up of atomic event queries that are assembled by means of event composition and temporal restrictions constructs. For each composite event query posed at a Web site, an operator tree is constructed having the event query constructs as nodes and the component atomic event queries as leaves. The results obtained by applying simulation unification to the content of the leaves and the incoming events are pushed into the tree in a bottom-up manner. Inner nodes have an event storage for semi-composed (composite) events allowing for an incremental evaluation. This approach is very similar to the evaluation of composite events in Snoop [15].

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 Section 3.2 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.

5.4 Transactions

An XChange transaction specification is a group of update specifications and/or explicit event specifications (expressing events that are constructed, raised, and sent as event messages) that are to be executed in an *all-or-nothing manner*.

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 13. At `http://airline.com` the flight timetable needs to be updated as reaction to the event given in Example 4. The update term that realises this is given in Figure 15.

```
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!!" }
        }}
      }}
   }
```

Figure 15: XChange Update Term for Updating Flight Timetable

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

Example 14. Figure 16 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.

```
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 }}
        }}
      }}
   }
```

Figure 16: XChange Update Term for Modifying Prices

As mentioned in Section 3.4, 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 15. Figure 19 specifies an XChange rule. After the keyword `TRANSACTION`, the desired transaction 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.

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 subterm `recipient` that specifies a Web site's address. An example of an event term is given as head of the XChange rule in Figure 17.

5.5 (Re)Active Rules

An XChange program is located at one Web site and consists of one or more (re)active rules of the form *Event query – Web query – Transaction/Raised events*. 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 is executed. There are two kinds of XChange rules that differ in the action to be executed:

- *event-raising rules*, i.e. the head of the rules (introduced by keyword `RAISE`) specifies explicit events to be constructed and sent to one or more Web sites, and
- *transaction rules*, i.e. the head of the rules (introduced by keyword `TRANSACTION`) specifies transactions to be executed.

Both kind of rules are exemplified in the following by implementing in XChange parts of the scenario given in Section 2. Note that rule components 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 16. 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 raising rule realising this is given in Figure 17.

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

Figure 17: XChange Raising Rule

Example 17. The travel organiser of Mrs. Smith uses the following rule (specified in XChange in Figure 18): if the return flight of Mrs. Smith is cancelled then look for and book another suitable flight.

Example 18. If no other suitable return flight is found and the airline does not provide an accomodation, then book for Mrs. Smith a cheap hotel and inform her secretary about the changes in her schedule. This XChange transaction rule is given in Figure 19.

Complex applications specifying reactivity on the Web require a number of features that cannot always be specified by simple programs. In XChange transactions can also be used as *means for structuring* complex XChange programs.

6 Related Work

The issue of automatic reaction in response to happenings of interest has its roots in the field of active databases [17, 28, 34]. In particular, the ability to react to *composite events*, i.e. possibly

```
TRANSACTION
  in { resource { "http://airline.com/reservations/" },
       reservations {{
         insert reservation { var F, name { "Christina Smith" } }
        }}
     }
ON
  xchange:event {{
     xchange:sender { "http://airline.com" },
        cancellation-notification {{
          flight {{ number { "AI2021" },
                    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
```

Figure 18: XChange Transaction Rule for Booking a Flight

```
TRANSACTION
  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 { "diary://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 {{ }} }}  }}
            } before 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
```

Figure 19: XChange Transaction Rule for Booking a Hotel and Announce Flight Cancellation

time-related combinations of events has received considerable attention (cf. e.g. [17, 28, 34]). 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, complex events reflecting a user-centered (e.g. travel planning related situations detailed in Section 2) and not a system-centered view are needed on the Web.

Active Databases A considerable number of active database system prototypes have been proposed (e.g. Chimera [14], COMPOSE [21], HiPAC [29], Ode [20], REACH [38], SAMOS [19], Snoop [15], Starburst [33], the SQL3 standard, and commercial systems supporting triggers such as Oracle, Informix, Ingres, InterBase) that bear evidence for the intensity of work done in the past in the active database field (an annotated bibliography on active databases can be found in [22]).

One common trait of existing active database systems is their ability to query events that have been received in the past. The set of all events that an active database system is notified about forms the so-called *event history* that, in most active database applications, is stored completely. This approach is suitable for centralised systems with no huge amount of event occurrences, but is not amenable to distributed systems with different kinds of component systems.

Event consumption modes have been introduced in active databases for offering a precise semantics for different kinds of applications. For example, Snoop [16] 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). [1] shows how Snoop's operators in the recent, chronicle, and continuous contexts can be expressed in Amit, a situation monitoring system for distributed event sources that has been proposed recently. 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 specifications, the behaviour of the specified rules is not so easy to grasp anymore. This is the reason why event consumption is not considered in XChange. However, the design and implementation of the language do not preclude their extension to allow consumption of events.

As the research on (centralised) active database systems is actually saturated and has already entailed a number of commercial systems, recent research on this topic has moved its attention to other frameworks (distributed ones), such as the (Semantic) Web. Thus, new languages have been proposed first just for updating data on the Web (realised by means of *update languages*) and then also for propagating and reacting to such updates (realised by means of *reactive languages*).

Update Languages for the Web Most existing proposals for *update languages* for XML (like XML-RL Update Language [25] and XPathLog [26]) 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 [35] (path-oriented language for addressing parts of XML documents) and XQuery [37] (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 and in an *all-or-nothing* manner. Moreover, these languages have the capability to react only to single event instances and do not provide constructs for querying for complex combinations of event instances (i.e. no composite event queries can be specified and detected). For example, an Event-Condition-Action rule language for XML and RDF data is proposed in [27, 4], supporting as actions sequences of simple insertions or deletions to XML or RDF data.

There is a single proposal for composite event detection *for XML documents* [7], but the kind of composite events considered are rather simple ones. 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). Based on the differences between these kinds of data, the language metaphor is that of *written text* vs. *speech.* XChange's language design enforces this clear separation and entails new characteristics of event processing on the Web.

XChange is an ongoing research project. At present, the design of an extended core language for XChange is completed. For implementing the language XChange and specifying its semantics, a modular approach that mirrors the three components of rules is followed:

- A prototype for the *event query* evaluation has been developed and a declarative semantics for the event language has been specified.
- A first version of Xcerpt, the language used in XChange for expressing *Web queries*, is fully designed and a reference implementation is available (cf. http://xcerpt.org). The declarative semantics of Xcerpt is formalised in [13, 30].
- The implementation and the semantics' specification of the *action part* is under development.

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 [6] – is to be extended). To what extent the language XChange could be employed for coordinating Grid services' jobs (thus, enhancing proposals such as [8]) is an issue that needs also to be investigated.

Moreover, the integration with languages for specifying and reasoning with specific kind of data is intended. The presented application scenario assessed the practical need of 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 [5] (Multi Paradigm Location Language), a language for specifying and reasoning with different kinds of location data. Integrating XChange with CaTTS [12], 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

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`).

The authors would like to thank Michael Eckert for fruitful discussions that helped improving this article and his collaboration on refining and implementing the XChange event query language.

References

References

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

[2] J.J. Alferes, W. May, and F. Bry, *Towards Generic Query, Update, and Event Languages for the Semantic Web*, Workshop on Principles and Practice of Semantic Web Reasoning, Springer, 2004.

[3] J. Bailey, F. Bry, and P.-L. Pătrânjan, *Composite Event Queries for Reactivity on the Web*, Proc. of 14th Int. World Wide Web Conference (Chiba, Japan), ACM, May 2005.

[4] J. Bailey, A. Poulovassilis, and P.T. Wood, *An Event-Condition-Action Language for XML*, Int. World Wide Web Conference (Honolulu, Hawaii, USA), May 2002.

[5] S. Berger, F. Bry, B. Lorenz, H.J. Ohlbach, P.-L. Pătrânjan, S. Schaffert, U. Schwertel, and S. Spranger, *Reasoning on the Web: Language Prototypes and Perspectives*, Proc. of European Workshop on the Integration of Knowledge, Semantics and Digital Media Technology, London, United Kingdom, The Institution of Electrical Engineers, IEE, 2004, pp. 157–164.

[6] S. Berger, F. Bry, S. Schaffert, and C. Wieser, *Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data*, Proc. of 29th Int. Conference on Very Large Data Bases, 2003.

[7] M. Bernauer, G. Kappel, and G. Kramler, *Composite Events for XML*, Proc. of 13th Int. World Wide Web Conference, ACM, 2004.

[8] L. Bocchi, P. Ciancarini, R. Moretti, V. Presutti, and D. Rossi, *An OWL-S Based Approach to Express Grid Services Coordination*, Proc. of 20th Annual ACM Symposium on Applied Computing (Santa Fe, New Mexico), vol. 2, ACM Press, March 2005, pp. 1661–1667.

[9] F. Bry, T. Furche, P.-L. Pătrânjan, and S. Schaffert, *Data Retrieval and Evolution on the (Semantic) Web: A Deductive Approach*, Workshop on Principles and Practice of Semantic Web Reasoning at 20th Int. Conference on Logic Programming (Saint Malo, France), vol. LNCS 3208, Springer, 2004, pp. 34–49.

[10] F. Bry and M. Marchiori, *Ten Theses on Logic Languages for the Semantic Web*, Proceedings of W3C Workshop on Rule Languages for Interoperability, Washington D.C., USA, W3C, 2005.

[11] F. Bry and P.-L. Pătrânjan, *Reactivity on the Web: Paradigms and Applications of the Language XChange*, Proc. of 20th Annual ACM Symposium on Applied Computing (Santa Fe, New Mexico), vol. 2, ACM Press, March 2005, pp. 1645–1649.

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

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

[14] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca, *Active Rule Management in Chimera*, Active Database Systems: Triggers and Rules For Advanced Database Processing, Morgan Kaufmann, 1996, pp. 151–176.

[15] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, *Composite Events for Active Databases: Semantics, Contexts and Detection*, Proc. of 20th Int. Conference on Very Large Data Bases (J.B. Bocca, M. Jarke, and C. Zaniolo, eds.), Morgan Kaufmann, 1994, pp. 606–617.

[16] S. Chakravarthy and D. Mishra, *Snoop: An Expressive Event Specification Language for Active Databases*, Data Knowledge Engineering **14** (1994), no. 1, 1–26.

[17] K.R. Dittrich and S. Gatziu, *Aktive Datenbanksysteme, Konzepte und Mechanismen*, Internat. Thompson Publ., 1996.

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

[19] S. Gatziu and K.R. Dittrich, *SAMOS: An Active Object-Oriented Database System*, IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Databases **15** (1992), no. 1-4, 23–26.

[20] N. H. Gehani and H. V. Jagadish, *Ode as an Active Database: Constraints and Triggers*, Proc. of 17th Conference on Very Large Data Bases, Morgan Kaufman pubs. (Los Altos CA), Barcelona, 1991.

[21] N. H. Gehani, H. V. Jagadish, and O. Shmueli, *Composite event specification in active databases: Model and implementation*, Proc. of 18th Int. Conference on Very Large Data Bases, 1992.

[22] U. Jaeger and J.C. Freytag, *An Annotated Bibliography on Active Database*, SIGMOD Record **24** (1995), no. 1, 58–69.

[23] M. Kasuya, *Teaching Features of the Stream of Speech in Japanese Classrooms*, Tech. report, Open Learning Program, University of Birmingham, 1999.

[24] M. Levene and A. Poulovassilis (eds.), *Web Dynamics - Adapting to Change in Content, Size, Topology and Use*, Springer, Germany, 2004.

[25] M. Liu, L. Lu, and G. Wang, *A Declarative XML-RL Update Language*, Proc. Int. Conf. on Conceptual Modeling (ER 2003), LNCS 2813, Springer-Verlag, 2003.

[26] W. May and E. Behrends, *On an XML Data Model for Data Integration*, Proc. Int. Workshop on Foundations of Models and Languages for Data and Objects (Viterbo, Italy), September 2001.

[27] G. Papamarkos, A. Poulovassilis, and P.T. Wood, *Event-Condition-Action Rule Languages for the Semantic Web*, Workshop on Semantic Web and Databases at 29th Int. Conference on Very Large Data Bases (Berlin, Germany), September 2003.

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

[29] A. Rosenthal, U. S. Chakravarthy, B. Blaustein, and J. Blakely, *Situation Monitoring for Active Databases*, Proceedings of 15th Int. Conference on Very Large Data Bases (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., 1989, pp. 455–464.

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

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

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

[33] J. Widom, *The Starburst Rule System: Language Design, Implementation, and Applications*, IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Databases **15** (1992), no. 1-4, 15–18.

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

[35] World Wide Web Consortium (W3C), http://www.w3.org/TR/xpath, *XML Path Language (XPath)*, 1999.

[36] World Wide Web Consortium (W3C), http://www.w3.org/TR/soap, *Simple Object Access Protocol (SOAP) 1.1*, 2000.

[37] World Wide Web Consortium (W3C), http://www.w3.org/TR/xquery/, *XQuery: A Query Language for XML*, February 2001.

[38] J. Zimmermann and A.P. Buchmann, *REACH*, Active Rules in Database Systems, 1999, pp. 263–277.