Querying Composite Events for Reactivity on the Web

François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan

University of Munich, Institute for Informatics
Oettingenstr. 67, D-80538 München
{bry, eckert, patranjan}@pms.ifi.lmu.de
http://www.pms.ifi.lmu.de

Abstract. Reactivity, the ability to detect events and respond to them automatically through reactive programs, is a key requirement in many present-day information systems. Work on Web Services reflects the need for support of reactivity on a higher abstraction level than just message exchange by HTTP. This article presents the composite event query facilities of the reactive rule-based programming language XChange. Composite events are important in the dynamic world of the Web where applications, or Web Services, that have not been engineered together are composed and have to cooperate by exchanging event messages.

1   Introduction

Reactivity, the ability to detect events or situations of interest and respond to them automatically through reactive programs, is a key requirement in many present-day information systems. The World Wide Web, undoubtedly by far the largest information system, has become a basis for many applications requiring reactivity, e.g., in commerce, business-to-business, logistics, e-Learning, and information systems for biological data.

It is natural to represent events that are exchanged between Web sites as XML messages. The Web's communication protocol, HTTP, provides an infrastructure for exchanging events or messages. Still, until recently the Web has been commonly perceived as a passive collection of HTML and XML documents; reactivity had to be implemented largely "by hand" (e.g., CGI scripts) and was limited to single Web sites (e.g., filling out forms on a shopping Web site). There is little support for reactivity on a higher abstraction level, e.g., in the form of reactive programming languages. Research and standardization in the Web Services area reflect the need to overcome what one might call the Web's passiveness.

XChange [1–3] is a rule-based reactive language for programming reactive behavior and distributed applications on the Web. Amongst other reactive behavior it aims at easing implementation and composition of Web Services. XChange is based on *Event-Condition-Action* rules (ECA rules). These specify that some *action* should be performed in response to some (class of) *events* (or situations of interest), provided that the *condition* holds.

To specify situations that require a reaction, XChange provides *event queries*, which describe classes of events. But event queries do more: they also extract and make available data from the events' XML representation that is relevant for the particular reaction. Considering a tourism application, it is not only important to detect when a flight has been canceled, but also to know its flight number and similar information in the reaction to the event.

Often, the situations are not given by a single atomic event, but a temporal combination of events, leading to the notion of *composite events* and *composite event queries*. Support for composite events is very important for the Web: In a carefully developed application, designers have the freedom to choose events according to their goal. They can thus often do with only atomic events by representing events which might by conceptually composite with a single atomic event. In the Web's open world however many different applications which have not been engineered together are integrated and have to cooperate. Situations that require a reaction might not have been considered in the original design of the applications and thus have to be inferred from many atomic events.

Consider again a tourism scenario: an application might want to detect situations where a traveler has already checked out of his hotel (first atomic event) but his flight has been canceled (second atomic event). On the Web the constituent events are emitted from independent Web sites (the airline and the hotel), which have not designed together. Hence an application has to infer the composite event from the given atomic events.

Similar motivating scenarios for composite events are filtering of stock trade reports, e.g., "recognize situations where the NASDAQ rises 10 points and the Dow Jones falls 5 points", or work-flow-management, e.g., "a student has fulfilled her degree-requirements if she has handed in her thesis and taken the final exams (which in turn can be a composite event)."

The remainder of this paper is structured as follows. We first give a short overview of XChange (Sect. 2), and then explain the event query language in detail. We introduce the syntax and intuitive meaning of its language constructs (Sect. 3), before turning to the formal semantics (Sect. 4) and the event query evaluation (Sect. 5). Conclusions (Sect. 6) complete this article.

2   XChange: ECA Rule-based Reactivity

XChange programs consist of ECA rules running locally at some Web site. In reaction to events, they can query and modify local and remote XML data and raise new events, which are sent as XML messages to other remote Web sites (in a push-manner).

XChange ECA rules have the general form *Action – Event Query – Web Query*. They specify to automatically execute the action (an update to a Web resource or rasing of a new event) as response to a situation specified by the event query (a query over the stream of incoming event messages), provided the Web query (a query to persistent Web data) evaluates successfully.

```
<flight-cancellation>                flight-cancellation [
    <number> UA917  </number>            number [ "UA917" ]        flight-cancellation {{
    <date> 2005-02-20 </date>            date [ "2005-02-20" ]         number [ var N ]
</flight-cancellation>               ]                            }}
```

Fig. 1. An event message in XML and term representation, and an atomic event query

Atomic event queries (queries to a single incoming event message) and Web queries rely on the query language *Xcerpt* [4]. Also, update specifications are an extension to Xcerpt. Xcerpt queries describe *patterns* of the queried XML data and are represented in a term-like syntax. Fig. 1 depicts a small XML document, its term representation, and a query term against this data. In the term syntax, square brackets indicate that the order of child elements is significant, while curly braces indicate it isn't. Double braces or brackets indicate a partial match, i.e., other children may exists, while single braces or brackets indicate a total match, i.e., no other children may exist. Queries can contain free variables (indicated by the keyword `var`) which are bound during the evaluation, which is based on a novel method called *Simulation Unification* [5].

XChange can provide the following benefits over the conventional approach of using imperative programming languages to implement Web Services:

- ECA rules have a highly declarative nature, allow programming on a high level of abstraction, and are easy to analyze for both humans and machines (see [6], for example).
- Event queries, Web queries, and updates follow the same paradigm of specifying patterns for XML data, thus making XChange an elegant, easy to learn language.
- Both atomic and composite events can be detected, the latter being an important requirement in composing an application from different Web Services (cf. Sect. 1) and relevant data extracted.
- Having an XML query language embedded in the condition part allows to access Web resources in a natural way. Also Xcerpt's *deductive rules* allowing to reason with data and to query not only pure XML but also RDF [7].
- A typical reaction to some event is to update some Web resource; XChange provides an integrated XML update language for doing this.
- ECA rules of XChange enforce a clear separation of persistent data (Web resources with URIs) and volatile data (event messages, no URIs). The distinction is important for a programmer: the former relates to *state*, while the latter reflects *changes in state*.

3   Event Queries and Composite Events

Event queries detect atomic events (receptions of single event messages) and composite events (temporal patterns in the reception of event messages) in the stream of incoming events and extract data in the form of variables bindings from them.

3.1 Atomic Events and Atomic Event Queries

Atomic events are received by XChange-aware Web sites as XML messages. Typically these messages will follow some standardized envelope format (e.g., SOAP format) providing information like the sender or the reception time of the message; in this paper we skip such details for the sake of brevity.

Atomic event queries are query terms (as introduced in the previous section). On reception of an incoming event message, XChange tries to simulation unify the query term and the message. If successful, this results in a set of substitutions for the free variables in the query.

3.2 Composite Event Queries

Composite event queries are built from atomic (and smaller composite) event queries (`EQ`) by means of composition operators and temporal restrictions. They describe a pattern of events that have to happen in a some time frame. Composite events are sequences of atomic events that answer a given composite event query; they happen over a period of time and thus have a starting and ending time.

Temporal Restrictions limit the time frame in which events are considered relevant. XChange supports absolute and relative temporal restrictions. Absolute restrictions are introduced by the keyword `in` and a time interval specification, e.g., `[1978-02-20 .. 2005-02-20]` following an (atomic or composite) event query. Answers to the event query are only considered if they happen in the specified time interval. Relative restrictions are introduced by the keyword `within` and a specification of a duration, e.g., `365 days`. They limit the duration (difference between starting and ending time) of answers to the event query.

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

Composition Operators express a temporal pattern of atomic event occurrences. XChange provides a rich set of such composition operators, a selection of which is presented here.

Conjunctions of event queries detect instances for each specified event query regardless of their order. They have the form: `and { EQ`$_1$`, ..., EQ`$_n$` }`.

Inclusive Disjunctions of event queries detect instances of any of the specified event queries. They have the form: `or { EQ`$_1$`, ..., EQ`$_n$` }`.

Temporally Ordered Conjunctions of event queries detect successive instances of events: `andthen [ EQ`$_1$`, ... , EQ`$_n$` ]` and `andthen [[ EQ`$_1$`, ... , EQ`$_n$` ]]`.

A total specification (using `[ ]`) expresses that only instances of the `EQ`$_i$ ($i = 1, \ldots, n$) are of interest and are included in the answer. Instances of other events that possibly have occurred between the instances of the `EQ`$_i$ 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 EQ_i. Thus, all these instances are contained in the event query's answer.

Example. The composite event query on the right side detects notifications of flight cancellations that are followed, within two hours of reception, by notifications that the airline is not granting accommodation. Note the use of the variable P to make sure that the notifications apply to the *same* passenger.

```
andthen [
    flight-cancellation {{
        number { var N },
        passenger { var P }
    }},
    no-accommodation {{
        passenger { var P },
    }}
] within 2 hours
```

Event Exclusions enable the monitoring of the *non-occurrence* of (atomic or composite) event query instances during an absolute time interval or the answer to another composite event query: `without EQ during CompositeEQ` or `without EQ during [s .. t]`.

Other operators include `n times EQ` to detect n occurrences of the same event, and `m of {EQ1, ... EQn}` to detect occurrences of m instances in a given set of event queries.

4 Semantics of Event Queries

Comparisons of (composite) event query languages such as [8] show that interpretation of similar language constructs can vary considerably. To avoid misinterpretations, clear semantics are indispensable.

The notion of answers to event queries is twofold. An answer to some query consists of (1) a sequence s of atomic events that allowed a successful evaluation of the query on the one hand, and (2) a set of variable substitutions Σ on the other hand. Variable substitutions can influence the reaction to some event specified in the remaining part of an XChange ECA rule. The sequence of events allows for events not being specified in the query to become a part of the answer (e.g., a partial `andthen[[`EQ_1`, `EQ_2`]]` returns not only answers to EQ_1 and EQ_2 but also any atomic events in-between) and gives answer closedness, i.e., the result of a query can be in turn queried by further queries.

We define a declarative semantics for XChange's event query language similar to a model-theoretic entailment relation. Unlike the traditional binary entailment relation $\models$, which relates models to queries (under some environment giving bindings for the free variables), however, our answering relation has to be ternary: it relates the stream of incoming event messages (which corresponds to a model), queries, *and answers* (as discussed above these include the "environment" Σ). The reason for the need of answers is that in our event query language answers cannot be simply obtained from queries by applying the variables substitutions to them, since they may contain events not having a corresponding constituent query. The answering relation is defined by induction on the structure of a query.

This allows easy recursive evaluation of composite (event) queries, where each constituent query can be evaluated independently of the others.

    We now give a formal account of the declarative semantics.

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

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

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

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

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

    q is an atomic event query: $q \lhd_{\mathcal{E}} (s, \Sigma)$ if and only if (1) $s = \langle d^r \rangle_r^r$, (2) d^r is an atomic event in the stream $\mathcal{E}$, (3) the data term d simulation unifies ("matches") with the query q under all substitutions in Σ. For a formal account of (3) see work on Xcerpt [9].

    $q = \text{and}[q_1, \ldots, q_n]$: $q \lhd_{\mathcal{E}} (s, \Sigma)$ iff there exist event sequences $s_1, \ldots s_n$ such that (1) $q_i \lhd_{\mathcal{E}} (s_i, \Sigma)$ for all $1 \leq i \leq n$, (2) s comprises all event sequences $s_1, \ldots s_n$ (denoted $s = \bigcup_{1 \leq i \leq n} s_i$).

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

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

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

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

The declarative semantics provide a sound basis for formal proofs about language properties. In particular, we have used it for proving the *bounded event lifespan* property for all legal event queries. Legal event queries are atomic event queries and composite event queries that are accompanied by temporal restrictions, such as $q\ \mathtt{within}\ d$, $q\ \mathtt{in}\ [t_1 .. t_2]$, $q\ \mathtt{before}\ t_2$, or $\mathtt{without}\ q\ \mathtt{during}\ [t_1 .. t_2]$. All legal event queries are such that no data on any event has to be kept forever in memory, i.e., the lifespan of every event is bounded.

More exactly, to evaluate any legal event query q at some time t correctly, only events of bounded life-span are necessary; that is, it suffices to consider the restriction $\mathcal{E} \mid_{t-\beta}^{t}$ of the event stream $\mathcal{E}$ to a time interval $[(t - \beta)\ ..\ t]$. The time bound β (a length of time) is only determined from q and does not depend on the incoming events $\mathcal{E}$. A more formal account of this and detailed proofs can be found in [10].

5   Evaluation of Composite Event Queries

Evaluation of composite event queries against the stream of incoming event messages should be performed in an incremental manner: work done in one evaluation step of an event query on some incoming atomic event should not be redone in future evaluation steps on further incoming events. Following the ideas of the rete algorithm [11] and previous work on composite event detection like [12], we evaluate a composite event query incrementally by storing all partial evaluations in the query's operator tree. Leaf nodes in the operator tree implement atomic event queries, inner nodes implement composition operators and time restrictions. When an event message is received, it is injected at the leaf nodes; data in the form of event query answers (s, Σ) (cf. Sect. 4) then flows bottom-up in the operator tree during this evaluation step. Inner nodes can store intermediate

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

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

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

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

Fig. 2. Implementation of a (binary) **and** inner node in pseudo-code

results to avoid recomputation when the next evaluation step is initiated by the next incoming event message.

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

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

Fig. 2 sketches an implementation for the evaluation of a (binary) **and** inner node in java-like pseudo-code. Consider it in an example of evaluating the event query $q =$**and**{ **a**{{var X}}, **b**{{var X}} } **within 2h** in Fig. 3. (Keep in mind, Fig. 2 covers only the **and**-node; **within** is a separate node with a separate implementation)

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

Fig. 3(a) depicts receiving **a**{1,2} at time $t = 1$. It does not match with the atomic event query **b**{{var X}} (right leaf in the tree). But it does match with the atomic event query **a**{{var X}} (left leaf) with substitution set Σ_1 and is propagated upwards in the tree as answer (s_1, Σ_1) to the parent node **and** (Fig. 3(d) defines s_i and Σ_i). The **and**-node cannot form a composite event from its input, yet, but it stores (s_1, Σ_1) for future evaluation steps.

At $t = 2$ we receive **b**{2,3} (Fig. 3(b)); it matches the right leaf node and (s_2, Σ_2) is propagated upwards. The **and**-node stores (s_2, Σ_2) and tries to form a

(a) $t=1$ (b) $t=2$ (c) $t=3$

$s_1 = \langle \mathtt{a\{1,2\}} \rangle$, $\Sigma_1 = \{\{X \mapsto 1\}, \{X \mapsto 2\}\}$; $s_2 = \langle \mathtt{b\{2,3\}} \rangle$, $\Sigma_2 = \{\{X \mapsto 2\}, \{X \mapsto 3\}\}$;
$s_3 = \langle \mathtt{a\{1,2\}}, \mathtt{b\{2,3\}} \rangle$, $\Sigma_3 = \{\{X \mapsto 2\}\}$; $s_4 = \langle \mathtt{a\{3\}} \rangle$, $\Sigma_4 = \{\{X \mapsto 3\}\}$; $s_5 = \langle \mathtt{b\{2,3\}a\{4\}} \rangle$, $\Sigma_5 = \{\{X \mapsto 3\}\}$.

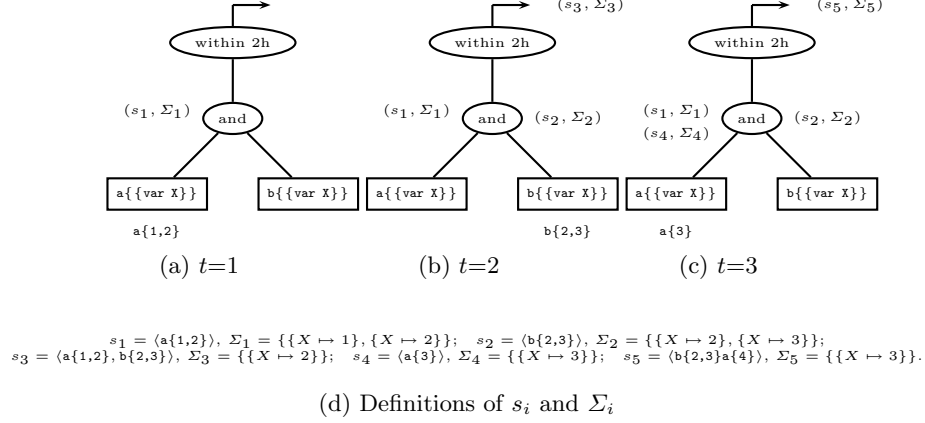(d) Definitions of s_i and Σ_i

Fig. 3. Incremental evaluation of an event query using bottom-up data flow in a storage-augmented operator tree

composite event (s_3, Σ_3) from (s_1, Σ_1) and (s_2, Σ_2). To be able to compose the events they have to agree on the variables substitutions with a common Σ_3. This can be computed as a (variant of a) natural join ($\bot$ denotes undefined): $\Sigma_3 = \Sigma_1 \bowtie \Sigma_2 = \{\sigma_1 \cup \sigma_2 \mid \sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2, \forall X.\ \sigma_1(X) = \sigma_2(X) \vee \sigma_1(X) = \bot \vee \sigma_2(X) = \bot\}$. Σ_3 now contains all substitutions that can be used simultaneously in all atomic event queries in **and**'s subtree. $\Sigma = \emptyset$ would signify that no such substitution exists and thus no composite event can be formed. In our case however there is exactly one substitution $\{X \mapsto 2\}$ and we propagate (s_3, Σ_3) to the `within 2h`-node. This node checks that $end(s_3) - begin(s_3) = 1 \leq 2$ and pushes (s_3, Σ_3) up (there is no need to store it). With this (s_3, Σ_3) reaches the top and we have our first answer to the event query q.

Fig. 3(c) shows reception of another event message $\mathtt{a\{3\}}$ at $t = 3$, which results in another answer (s_5, Σ_5) to q. After the query evaluation at $t = 3$, we can release (delete) the stored answer (s_1, Σ_1) from the operator tree: any composite event formed with use of (s_1, Σ_1) will not pass the `within 2h`-node. Event deletion is performed by top-down traversal of the operator tree. Temporal restriction operator nodes put restrictions on $begin(s)$ and $end(s)$ for all answers (s, Σ) stored in their subtrees. In our example, all events (s, Σ) in the subtree of `within 2h` must satisfy $t - 2 \leq begin(s)$, where t is the current time.

6 Conclusions

This article has presented the event query facilities of the reactive rule-based language XChange. Event queries detect (composite) events in the stream of incoming event messages and extract data from them for use in the subsequent reaction. The event query language is tailored for the Web: Events are represented as XML messages, so it is necessary to extract data from them with an

XML query language. When composing Web Services or other reactive applications in an ad-hoc manner, situations that require a reaction oftentimes are not given through a single atomic event, requiring support for composite events.

While composite event detection has been explored in the active database community, this work doesn't consider or extract data contained in events. An important novelty in the XChange event query language are the free variables, which allow to "correlate" data from different events during the composite event detection and to extract data in the form of variable bindings for use in the rest of a reactive rule. This work has defined declarative semantics for composite event queries in the presence of free variables. Existing approaches to composite event detection have been extended to incrementally evaluate such queries.

Acknowledgments

References

1. Bry, F., Pătrânjan, P.L.: Reactivity on the Web: Paradigms and applications of the language XChange. In: Proc. 20th ACM Symp. on Applied Computing. (2005)
2. Bailey, J., Bry, F., Eckert, M., Pătrânjan, P.L.: Flavours of XChange, a rule-based reactive language for the (Semantic) Web. In: Proc. Intl. Conf. on Rules and Rule Markup Languages for the Semantic Web. (2005)
3. Pătrânjan, P.L.: The Language XChange: A Declarative Approach to Reactivity on the Web. PhD thesis, Institute for Informatics, University of Munich (2005)
4. Schaffert, S., Bry, F.: Querying the Web reconsidered: A practical introduction to Xcerpt. In: Proc. Extreme Markup Languages. (2004)
5. Bry, F., Schaffert, S.: Towards a declarative query and transformation language for XML and semistructured data: Simulation Unification. In: Proc. Int. Conf. on Logic Programming. (2002)
6. Bailey, J., Poulovassilis, A., Wood, P.T.: Analysis and optimisation of event-condition-action rules on XML. Computer Networks **39** (2002)
7. Berger, S., Bry, F., Bolzer, O., Furche, T., Schaffert, S., Wieser, C.: Querying the standard and Semantic Web using Xcerpt and visXcerpt. In: Proc. European Semantic Web Conf. (2005)
8. Zimmer, D., Unland, R.: On the semantics of complex events in active database management systems. In: Proc. 15th Int. Conf. on Data Engineering. (1999)
9. Schaffert, S.: Xcerpt: A Rule-Based Query and Transformation Language for the Web. PhD thesis, Institute for Informatics, University of Munich (2004)
10. Eckert, M.: Reactivity on the Web: Event queries and composite event detection in XChange. Master's thesis, Institute for Informatics, University of Munich (2005)
11. Forgy, C.L.: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence **19** (1982)
12. Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.K.: Composite events for active databases: Semantics, contexts and detection. In: Proc. 20th Int. Conf. on Very Large Data Bases. (1994)