Composite Events for XML

Martin Bernauer
Institute for Software Technology and Interactive Systems, Vienna University of Technology, Austria
bernauer@big.tuwien.ac.at

Gerti Kappel
Institute for Software Technology and Interactive Systems, Vienna University of Technology, Austria
gerti@big.tuwien.ac.at

Gerhard Kramler
Institute for Software Technology and Interactive Systems, Vienna University of Technology, Austria
kramler@big.tuwien.ac.at

ABSTRACT

Recently, active behavior has received attention in the XML field to automatically react to occurred events. Aside from proprietary approaches for enriching XML with active behavior, the W3C standardized the Document Object Model (DOM) Event Module for the detection of events in XML documents. When using any of these approaches, however, it is often impossible to decide which event to react upon because not a single event but a combination of multiple events, i.e., a composite event determines a situation to react upon. The paper presents the first approach for detecting composite events in XML documents by addressing the peculiarities of XML events which are caused by their hierarchical order in addition to their temporal order. It also provides for the detection of satisfied multiplicity constraints defined by XML schemas. Thereby the approach enables applications operating on XML documents to react to composite events which have richer semantics.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures; H.2.3 [**Database Management**]: Languages; H.2.4 [**Database Management**]: Systems—*Rule-based databases*

General Terms

Languages

Keywords

XML, Composite Event, Event Algebra, Active Behavior, Event-Condition-Action Rule

1. INTRODUCTION

Recently, active behavior has received attention in the XML field after being widely used in other fields such as database systems [15] and workflow management [14]. Various proprietary approaches for active XML [2, 3, 4, 5, 6, 16] show how event-condition-action (ECA) rules can be used to automatically react to an occurred event by executing an action if a condition applies. Events that can be reacted on are modifications of XML data, such as the insertion of an element with a certain name, and invocations of operations against XML data in [16].

In parallel to these efforts, the W3C standardized the Document Object Model (DOM) Event Module [20]. It provides for detection of events in DOM documents so that application programs can react accordingly. Defined events comprise among others mutation events, which are events that represent modifications of XML data. The Event Module may thus be used to provide the proprietary approaches mentioned before and custom applications with mutation events.

When using any of the approaches [2, 3, 4, 5, 6, 20] it is sometimes impossible to decide upon which event to react. The reason is that often not a single event but a combination of multiple events determines a situation where some action has to be executed. A potential work-around in such a situation is to use the event that always occurs at last of multiple events or to use another event that usually occurs after multiple events. Using such a work-around, however, makes rules dependent on applications which define the order of event occurrences.

Obviously, a technique is needed to detect occurrences of combinations of multiple events, i.e., to detect so called *composite events*. This has long been studied in the active database literature where *event algebras* have been proposed for the description of composite events (e.g., cf. [8, 11, 13, 22, 23]) and several techniques for realizing detection of composite events have been proposed, namely event graphs [7], state automata [13], and petri nets [12].

Events in XML, however, differ from the concept of events in literature as follows:

(1) XML events are not only ordered by time but also by hierarchical structure. It is mostly undesired to use hierarchically unrelated events to form composite events as previous approaches do.

(2) An XML schema may constrain the number of element and attribute occurrences in documents. Existing approaches do not support the detection of when such constraints are satisfied.

(3) Event types, which are descriptions of events at the schema level, are hierarchically related as their events are. This allows for more expressive and more reusable event type definitions than in previous approaches.

Due to the above peculiarities of XML events, existing approaches for detecting composite events, such as [8, 11, 13, 22, 23], cannot be reasonably employed for XML events because one encounters the following problems: (i) depending on the order of multiple modifications that all result in the same XML data different composite events are detected, (ii) most of the detected composite events are meaningless since they are not hierarchically related, they have to be filtered out by application code, (iii) it cannot be detected when multiplicity constraints defined by an XML schema are satisfied, and (iv) event types are unrelated and their extents are disjunct, limiting expressiveness and reusability of composite event type definitions. For a motivating example that shows problems i and ii when using a refined existing approach see Section 2.3.

The contribution of the paper is to present an approach to detect composite events in XML that takes the above peculiarities of XML events into account. It *refines* an event algebra known from literature by defining the employed abstract model for XML data, XML events, and XML event types. Thereby it provides for more expressive and reusable event type definitions (addressing peculiarity 3). Moreover, it *extends* the semantics of the refined event algebra by introducing the hierarchical context to combine events according to hierarchy (addressing peculiarity 1), by introducing the multiplicity operator to detect when multiplicity constraints are satisfied (addressing peculiarity 2), and by introducing operator modifiers to provide for more expressive event type definitions.

In particular, the presented approach refines and extends the event algebra Snoop [7, 8], because it is both extensible and well suited for XML. Snoop is extensible because it uses contexts to define the semantics of an event expression, thus by defining a new context semantics can be extended. Snoop is well suited for XML because *event trees* are used to realize event expressions and demonstrate event detection. Event trees fit well for processing XML events because they are hierarchically ordered as well. Moreover, Snoop is used in the Sentinel active DBMS, is prominent among [11, 13, 22, 23] according to CiteSeer[1], and is still subject to active research [1].

The paper is structured as follows. The refinement of the event algebra and a motivating example is shown in Section 2, the algebra's extension in Section 3. Section 4 briefly discusses the implementation of a proof-of-concept prototype, and finally Section 5 concludes the paper.

2. REFINED EVENT ALGEBRA

This section shows how the event algebra Snoop is refined so that it can be used with XML events. First it presents the employed abstract model for XML data, a syntax for referrers to portions of XML data at the schema and instance level, and operators on referrers in Section 2.1. Second, an abstract model for events and event types is introduced in Section 2.2. Finally, Section 2.3 briefly introduces Snoop and shows an example that uses the refined event algebra with contexts from Snoop.

2.1 Path Types and Path Instances

An XML document is represented by a tree of nodes where an XML document's elements, attributes and text is represented by the tree using element, attribute, and text nodes respectively. As such it is a subset of the XML Infoset [18]. Each node has an identifier.

A *path type* identifies a node of a tree by using type information, i.e., independently of concrete documents. A path type accords to a restricted XPath expression [17] that refers to either element, attribute, or text nodes in each of its steps via respective axis and node tests. A path type is absolute or relative with respect to the root of the tree, e.g., $/\mathsf{order}/\mathsf{item}/\mathsf{price}$ denotes an absolute path type while $\mathsf{item}/\mathsf{price}$ denotes a relative path type to element price. Path type pt is a tuple comprising a $kind \in \{\mathsf{absolute}, \mathsf{relative}\}$ and an ordered set of steps, thus $pt = \langle kind, steps\rangle$ or $pt = null$. Two single steps are equal if they equal in their respective axis (child or $\mathsf{attribute}$) and node test (test for an XML-QName or $\mathsf{text}()$).

A *path instance* identifies a node of a tree representing a concrete document. For node price_1 its path instance comprises an ordered set of identifiers that starts with the identifier of the tree's root node and ends with price_1's identifier and is thus always absolute. A path instance is denoted similar to a path type by using "/" to separate nodes, e.g., $/\mathsf{order}_1/\mathsf{item}_1/\mathsf{price}_1$ denotes a path instance to node price_1. Path instance pi is a tuple comprising its absolute path type and an ordered set of node identifiers, thus $pi = \langle pt, ids\rangle$ or $pi = null$.

To compare and operate on path types, operators for testing for equality ($=$), containment ($\subset$), ending ($\subset_{\mathsf{e}}$), and intersection ($\cap_{\mathsf{lb}}$, $\cap_{\mathsf{ab}}$) are defined. The operators complement the ones defined by XPath which operate on path instances only, such as $=$ [17] and $\mathsf{intersection}$ [21]. The result of applying operators on path types are defined as follows (where $m = |pt_1.steps|$ and $n = |pt_2.steps|$):

- $pt_1 = pt_2$
  Path type pt_1 equals pt_2 iff $pt_1.kind = pt_2.kind \wedge m = n \wedge \forall 1 \le i \le m : pt_1.steps[i] = pt_2.steps[i]$.

  Example 1. $\mathsf{item}/\mathsf{price} = \mathsf{item}/\mathsf{price}$,
  $/\mathsf{order} = /\mathsf{order}$,
  $\mathsf{order} \neq /\mathsf{order}$.

- $pt_1 \subset pt_2$
  Path type pt_2 uniquely contains path type pt_1 iff $\forall 1 \le i \le m : pt_1.steps[i] = pt_2.steps[c+i]$ where c is a constant offset and $m + c \le n$. No $c' \neq c$ may exist for which the expression above applies as well. Additionally, if pt_1 and pt_2 are both absolute $c = 0 \wedge m < n$ must apply, if both are relative only $m < n$ must apply. A relative path type cannot contain an absolute one.

  Example 2. $\mathsf{item}/\mathsf{price} \subset /\mathsf{order}/\mathsf{item}/\mathsf{price}$,
  $\mathsf{order}/\mathsf{item} \subset /\mathsf{order}/\mathsf{item}/\mathsf{price}$,
  $/\mathsf{order} \not\subset \mathsf{order}/\mathsf{item}$.

- $pt_1 \subset_{\mathsf{e}} pt_2$
  Relative path type pt_1 ends path type pt_2 if the end of $pt_2.steps$ contains $pt_1.steps$ and pt_2 is more special than pt_1, i.e., $pt_1 \subset_{\mathsf{e}} pt_2$ iff $pt_1.kind = \mathsf{relative} \wedge (m < n \vee (pt_2.kind = \mathsf{absolute} \wedge m = n)) \wedge \forall 1 \le i \le m : pt_1.steps[i] = pt_2.steps[n - m + i]$.

  Example 3. $\mathsf{item}/\mathsf{price} \subset_{\mathsf{e}} /\mathsf{order}/\mathsf{item}/\mathsf{price}$,
  $\mathsf{price} \subset_{\mathsf{e}} \mathsf{item}/\mathsf{price}$,
  $\mathsf{order}/\mathsf{item} \not\subset_{\mathsf{e}} /\mathsf{order}/\mathsf{item}/\mathsf{price}$.

- $r := pt_1 \cap_{\mathsf{lb}} pt_2$
  The left-bound intersection operator is commutative and determines for path types pt_1 and pt_2 equal steps at the beginning of $pt_1.steps$ and $pt_2.steps$. If $pt_1.kind = pt_2.kind$, $r.kind := pt_1.kind$, otherwise it is $\mathsf{absolute}$. Resulting $r.steps := \{pt_1.steps[i] | pt_1.steps[i] = pt_2.steps[i]\}$ where $1 \le i \le j$ where j is either the largest index for which $pt_1.steps[j] = pt_2.steps[j]$ applies or the minimum out of m and n. The result is $null$ if $pt_1 = null \vee pt_2 = null \vee pt_1.steps[1] \neq pt_2.steps[1]$.

  Example 4. $\mathsf{order}/\mathsf{item} \cap_{\mathsf{lb}} /\mathsf{order}/\mathsf{billTo} = /\mathsf{order}$,
  $\mathsf{item}/@\mathsf{partnum} \cap_{\mathsf{lb}} \mathsf{item} = \mathsf{item}$,
  $\mathsf{order} \cap_{\mathsf{lb}} \mathsf{item} = null$.

- $r := pt_1 \cap_{\mathsf{ab}} pt_2$
  The absolute-path intersection is not commutative and makes path type pt_1 absolute according to absolute path type pt_2. If $pt_1 \subset pt_2 \vee pt_1 = pt_2$, r is defined by $r.kind := \mathsf{absolute}$ and $r.steps := \{pt_2.steps[i] | 1 \le i \le j\}$ where j is the last index where pt_2 contains pt_1. The result is $null$ if $pt_1 = null \vee pt_2 = null \vee (pt_1 \not\subset pt_2 \wedge pt_1 \neq pt_2)$.

[1]`http://citeseer.nj.nec.com`

Example 5. $\mathsf{item} \cap_{\mathsf{ab}} \mathsf{/order/item/price} = \mathsf{/order/item}$,
$\mathsf{item} \cap_{\mathsf{ab}} \mathsf{/order} = null$.

To compare and operate on path instances, operators for testing for equality ($=$) and projection (π) are defined. The result of applying operators on path instances are defined as follows:

- $pi_1 = pi_2$
  Two path instances pi_1 and pi_2 equal iff $pi_1.pt = pi_2.pt \wedge \forall 1 \leq i \leq |pi_1.ids| : pi_1.ids[i] = pi_2.ids[i]$.

  Example 6. $\mathsf{/order_1/item_1} = \mathsf{/order_1/item_1}$.

- $r := \pi_{pt}(pi)$
  A projection of path instance pi on path type pt is a path instance if $pt \subset pi.pt \vee pt = pi.pt$. Then $r.pt := pt$ and $r.ids := \{pi.ids[i] | j \leq i \leq k\}$ where j and k are the indexes between which $pi.pt.steps$ contains $pt.steps$. The result is $null$ if $pi = null \vee pt = null \vee (pt \not\subset pi.pt \wedge pt \neq pi.pt)$.

  Example 7. $\pi_{\mathsf{item}}(\mathsf{/order_1/item_1/price_1}) = \mathsf{item_1}$,
  $\pi_{\mathsf{/order}}(\mathsf{/order_1/item_1}) = \mathsf{/order_1}$,
  $\pi_{\mathsf{item}}(\mathsf{/order_1}) = null$.

2.2 Event Types and Events

The DOM Event Module defines among others event types for mutation events, which reflect modifications of DOM documents' data. They basically comprise one event type for the insertion of nodes, one for the deletion of nodes, one for manipulation of attributes, and one for manipulations of text nodes.

While the DOM event types are sufficient for a procedural handling of occurred events, they are too coarse grained for a declarative handling by an event algebra. Hence, for *every* path type pt the presented approach distinguishes three *primitive event types*, denoted as $\mathsf{ins}(pt)$, $\mathsf{upd}(pt)$, and $\mathsf{del}(pt)$. Like in the DOM Event Module, ins and del events reflect insertions and deletions of element, attribute, and text nodes, while upd events reflect modifications of text nodes and attribute nodes. Moreover, instead of an operation wildcard "$*$" can be used. The path type defines where events of that type occur. It can be relative or absolute. Primitive event type et is a tuple comprising an operation, which is one of $\{\mathsf{ins}, \mathsf{upd}, \mathsf{del}, *\}$, and a path type, thus $et = \langle op, pt \rangle$.

A *primitive event* occurs whenever a node is manipulated. Primitive event e is represented by a tuple comprising its identifier id, timestamp ts, event type et, and path instance pi which identifies the manipulated node, thus $e = \langle id, ts, et, pi \rangle$. The event type's operation does not equal wildcard "$*$", its path type is absolute, and the path instance's path type $pi.pt$ equals $et.pt$.

A *composite event type*, i.e., the event type of a composite event, is a tuple that comprises a unique name and a path type, thus $et = \langle name, pt \rangle$. Wildcard "$*$" can be used instead of a name, referring to any composite event type having the path type. A composite event type is denoted as $\mathsf{name}(pt)$ analogously to a primitive one. The path type of a composite event type defines, like the path type of a primitive event type, where events of that type occur. It can be relative or absolute.

A *composite event* is formed by combining primitive and other composite events, which are referred to as constituent events. Composite event e^c is represented by a tuple comprising its identifier id, composite event type et, path instance pi, which identifies where the event occurred, and a set of constituent events $cevts$, thus $e^c = \langle id, et, pi, cevts \rangle$. The composite event type's name does not equal wildcard "$*$", its path type is absolute, and the path instance's path type $e^c.pi.pt$ equals $e^c.et.pt$.

The clock for measuring the timestamps of primitive events is a logical one, i.e., a counter. Primitive events occur at distinct points in time and for simplicity it is assumed that the detection of composite events takes no time. Therefore one primitive and multiple composite events may be detected at a single point in time.

Primitive as well as composite event types are hierarchically related via their path type and lead to *more expressive* composite event type definitions than in Snoop. Most important this allows to constrain the combination of event types by an operator to related event types. For the constraints on operator nodes see Section 3.1.

An event can be an instance of more than one event type, providing for *more reusable* composite event type definitions than in Snoop. Event e is a direct instance of its type $e.et$ and an indirect instance of event types $et_i \neq e.et$ to which the event's type is compatible to, denoted by $e.et \succeq et_i$ (not commutative). Primitive event type et_1 is *compatible to* primitive event type et_2, i.e., $et_1 \succeq et_2$ iff $((et_1.op = et_2.op) \vee (et_2.op = \text{"}*\text{"})) \wedge ((et_2.pt \subset_{\mathsf{e}} et_1.pt) \vee (et_2.pt = et_1.pt))$. Analogously, for two composite event types $et^c_1 \succeq et^c_2$ iff $((et^c_1.name = et^c_2.name) \vee (et^c_2.name = \text{"}*\text{"})) \wedge ((et^c_2.pt \subset_{\mathsf{e}} et^c_1.pt) \vee (et^c_2.pt = et^c_1.pt))$. For more details on reuse see Section 3.1, especially Examples 13 and 14.

Example 8. Primitive event e_{p_1} reflecting an insertion in path type $e_{p_1}.et.pt = \mathsf{/order/item/price}$ is an instance of event types such as $E_1 = \mathsf{ins}(\mathsf{/order/item/price})$ and $E_2 = *(\mathsf{price})$ since $e_{p_1}.et \succeq E_1$ and $e_{p_1}.et \succeq E_2$. Analogously, composite event $e^c_{h_1}$ with path type $e^c_{h_1}.et.pt = \mathsf{/order/item}$ and name Nm is an instance of composite event types such as $E^c_1 = \mathsf{Nm}(\mathsf{/order/item})$ and $E^c_2 = *(\mathsf{item})$ since $e^c_{h_1}.et \succeq E^c_1$ and $e^c_{h_1}.et \succeq E^c_2$.

Like in Snoop, composite event types are defined by event expressions according to an event algebra. An expression combines events by the algebra's operators. The presented approach uses operators $\triangle$, $\triangledown$, and $;$ from Snoop to form conjunction, disjunction, and sequence of events. An expression is realized by an event tree, e.g., Figure 1 depicts the event tree that realizes the expression described in Section 2.3.

An event tree comprises event type nodes and operator nodes. An *event type node* is a tuple $\langle et, evts \rangle$ which stores a set of events $evts$. All events in $evts$ are a direct or indirect instance of event type et. An *operator node* combines events from child nodes nds to events of composite event type et according to operator opr and stores them in a set of composite events $evts$, and is thus a tuple $\langle opr, nds, et, evts \rangle$. Leaf nodes in an event tree are event type nodes while inner nodes are operator nodes.

In the presented approach the event type of an event tree node can be compared to the *static type* of a variable in strongly typed object-oriented programming languages (e.g., Java), while the event type of an event can be compared to the *dynamic type* of an expression, e.g., an object. An occurred event is stored in all leaf nodes that have a compatible event type. By using the ending operator in the definition of type compatibility a leaf node only stores events that occur directly in the node's path type and not in descendants.

Example 9. Continuing Example 8, when primitive event e_{p_1} with (dynamic) event type $e_{p_1}.et$ occurs, it is stored in event type nodes that have compatible (static) event types such as E_1 and E_2. Analogously, when composite event $e^c_{h_1}$ is raised, it is stored in event type nodes that have event types such as E^c_1 and E^c_2.

2.3 Example

This section exemplifies the need for composite events in XML and demonstrates the application of the refined event algebra. Consider an XML document that represents a purchase order (document element order) comprising items to be ordered (element

Table 1: Raised composite events when using contexts from Snoop

-	t_1	t_2	t_3	t_4	t_5	t_6
Cumulative Context						
S_1	e_{i_1}	e_{p_1}	$e_{q_1}, \{e_{i_1} e_{p_1} e_{q_1}\}^c$	e_{i_2}	e_{p_2}	$e_{q_2}, \{e_{i_2} e_{p_2} e_{q_2}\}^c$
S_2	e_{i_1}	e_{i_2}	e_{p_1}	e_{p_2}	$e_{q_1}, \{e_{i_1} e_{i_2} e_{p_1} e_{p_2} e_{q_1}\}^c$	e_{q_2}
S_3	e_{i_1}	e_{i_2}	e_{p_1}	$e_{q_1}, \{e_{i_1} e_{i_2} e_{p_1} e_{q_1}\}^c$	e_{p_2}	$e_{q_2}, (\{e_{p_2} e_{q_2}\}^c)$
S_4	e_{i_1}	e_{i_2}	e_{p_1}	$e_{q_2}, \{e_{i_1} e_{i_2} e_{p_1} e_{q_2}\}^c$	e_{p_2}	$e_{q_1}, (\{e_{p_2} e_{q_1}\}^c)$
Chronicle Context						
S_1	e_{i_1}	e_{p_1}	$e_{q_1}, \{e_{i_1} e_{p_1} e_{q_1}\}^c$	e_{i_2}	e_{p_2}	$e_{q_2}, \{e_{i_2} e_{p_2} e_{q_2}\}^c$
S_2	e_{i_1}	e_{i_2}	e_{p_1}	e_{p_2}	$e_{q_1}, \{e_{i_1} e_{p_1} e_{q_1}\}^c$	$e_{q_2}, \{e_{i_2} e_{p_2} e_{q_2}\}^c$
S_3	e_{i_1}	e_{i_2}	e_{p_1}	$e_{q_1}, \{e_{i_1} e_{p_1} e_{q_1}\}^c$	e_{p_2}	$e_{q_2}, \{e_{i_2} e_{p_2} e_{q_2}\}^c$
S_4	e_{i_1}	e_{i_2}	e_{p_1}	$e_{q_2}, \{e_{i_1} e_{p_1} e_{q_2}\}^c$	e_{p_2}	$e_{q_1}, \{e_{i_2} e_{p_2} e_{q_1}\}^c$
Recent Context						
S_1	e_{i_1}	e_{p_1}	$e_{q_1}, \{e_{i_1} e_{p_1} e_{q_1}\}^c$	e_{i_2}	$e_{p_2}, \{e_{i_2} e_{p_2} e_{q_1}\}^c$	$e_{q_2}, \{e_{i_2} e_{p_2} e_{q_2}\}^c$
S_2	e_{i_1}	e_{i_2}	e_{p_1}	e_{p_2}	$e_{q_1}, \{e_{i_2} e_{p_2} e_{q_1}\}^c$	$e_{q_2}, \{e_{i_2} e_{p_2} e_{q_2}\}^c$
S_3	e_{i_1}	e_{i_2}	e_{p_1}	$e_{q_1}, \{e_{i_2} e_{p_1} e_{q_1}\}^c$	$e_{p_2}, \{e_{i_2} e_{p_2} e_{q_1}\}^c$	$e_{q_2}, \{e_{i_2} e_{p_2} e_{q_2}\}^c$
S_4	e_{i_1}	e_{i_2}	e_{p_1}	$e_{q_2}, \{e_{i_2} e_{p_1} e_{q_2}\}^c$	$e_{p_2}, \{e_{i_2} e_{p_2} e_{q_2}\}^c$	$e_{q_1}, \{e_{i_2} e_{p_2} e_{q_1}\}^c$
Continuous Context						
S_1	e_{i_1}	e_{p_1}	$e_{q_1}, \{e_{i_1} e_{p_1} e_{q_1}\}^c$	e_{i_2}	$e_{p_2}, \{e_{i_2} e_{p_2} e_{q_1}\}^c$	$e_{q_2}, (\{e_{p_2} e_{q_2}\}^c)$
S_2	e_{i_1}	e_{i_2}	e_{p_1}	e_{p_2}	$e_{q_1}, \dagger_1$	e_{q_2}
S_3	e_{i_1}	e_{i_2}	e_{p_1}	$e_{q_1}, \dagger_2$	$e_{p_2}, (\{e_{p_2} e_{q_1}\}^c)$	$e_{q_2}, (\{e_{p_2} e_{q_2}\}^c)$
S_4	e_{i_1}	e_{i_2}	e_{p_1}	$e_{q_2}, \dagger_3$	$e_{p_2}, (\{e_{p_2} e_{q_2}\}^c)$	$e_{q_1}, (\{e_{p_2} e_{q_1}\}^c)$
$\dagger_1$: $\{e_{i_1} e_{p_1} e_{q_1}\}^c, \{e_{i_1} e_{p_2} e_{q_1}\}^c, \{e_{i_2} e_{p_1} e_{q_1}\}^c, \{e_{i_2} e_{p_2} e_{q_1}\}^c$ $\dagger_2$: $\{e_{i_1} e_{p_1} e_{q_1}\}^c, \{e_{i_2} e_{p_1} e_{q_1}\}^c$ $\dagger_3$: $\{e_{i_1} e_{p_1} e_{q_2}\}^c, \{e_{i_2} e_{p_1} e_{q_2}\}^c$						

item), each in turn described by a price (element price) and a quantity (element quantity). When defining a rule that reacts on the insertion of an item and re-calculates the overall order value by multiplying price by quantity of each item and summing it up, one encounters the problem to decide which event to react on. Upon the insertion of element item it does not comprise any of the necessary child elements, and upon the insertion of element price the quantity element may not be available and vice versa.

The problem can be overcome by using composite events. A composite event is raised according to its definition after certain events have occurred. In the example, a composite event should occur after the occurrence of events reflecting insertions of an item, a price, and a quantity element (where the latter two are children of the first) so that a rule can be defined on it. This can be achieved by event expression “$\mathsf{E_i}$; $(\mathsf{E_p} \triangle \mathsf{E_q})$”, where $\mathsf{E_i} := \mathsf{ins(item)}$, $\mathsf{E_p} := \mathsf{ins(item/price)}$, and $\mathsf{E_q} := \mathsf{ins(item/quantity)}$.

The event tree realizing the example’s expression is depicted in Figure 1. The event tree’s behavior when using contexts from Snoop, i.e., its processing of four sequences of primitive events $S_1..S_4$ is shown in Table 1. The sequences reflect the insertions of item (abbreviated as i_n), price (p_n), and quantity (q_n) elements. Numerical index n of an element represents its hierarchical position, meaning that elements with the same numerical index are hierarchically related, e.g., p_1 is a child of i_1. In case two numerical indices are separated by a dot, the first number represents the element’s hierarchical position and the second one the time of its insertion, e.g., the insertion of $q_{1.1}$ occurs before the insertion of $q_{1.2}$.

Briefly and informally introducing the contexts from Snoop, a composite event is raised by a conjunction operator node as soon as a child node’s set of stored events is modified, i.e., an event is added (the so called “terminator”) and every child node contains at least one event. Note that only the conjunction operator is described here, however, sequence and disjoint operator are defined analogously. A raised composite event’s constituent events are defined as follows:

- in *cumulative context* they comprise all events from every child node wherefrom they are removed.
- in *chronicle context* they comprise the oldest event from every child node wherefrom they are removed, i.e., all constituent events are consumed in chronological order of occurrence.
- in *recent context* they comprise the most recent event from each child node. All events that cannot be the earliest constituent event of subsequently raised composite events (i.e., that cannot be an “initiator”) are removed from child nodes.
- in *continuous context* its constituent events comprise the terminator and the most recent event from every child node except the one of the terminator. Subsequently all constituent events are removed from child nodes except the terminator. The procedure is repeated until there are no events left to be combined with the terminator. Finally the terminator is removed if it cannot be an initiator of subsequently raised composite events.

The rationale of Table 1 is to introduce the unfamiliar reader to Snoop’s contexts and to exemplify that incorrect composite events are raised when any of Snoop’s contexts is used (for which one falsifying event sequence would suffice). Incorrect events are raised because events are selected only by their occurrence time and not their hierarchical position. Where under a correct composite event it is referred to a composite event whose constituent events are hierarchically related. Composite events raised by the root of the tree are shown by their constituent events, e.g., $\{e_{i_1} e_{p_1} e_{q_1}\}^c$. Unconsumed composite events that remain in the tree after t_6 are shown at the time they are raised, but in brackets, e.g., $(\{e_{p_2} e_{q_2}\}^c)$. The table does not show the unrestricted context which basically forms the cartesian product of all events. Naturally, it raises even more incorrect composite events.

Summarized, the refined event algebra as presented in this section is still not applicable to detect composite events in XML when used with contexts from Snoop. The reason is that the requirements on applications using such an event algebra are inadequate, which would have to use the unrestricted context and filter out huge amounts of incorrect events or manipulate XML data in a defined temporal order so that some context only detects correct events. And still, the satisfaction of multiplicity constraints cannot be detected using the refined algebra.

3. EXTENDED EVENT ALGEBRA

This section presents an extension to the refined event algebra. The extension comprises the hierarchical context presented in Section 3.1, the multiplicity operator in Section 3.2, and operator modifiers in Section 3.3. Event trees which are generated from event expressions are used for presentational purposes throughout the section.

3.1 Hierarchical Context

The hierarchical context is introduced since it is necessary to combine events according to their hierarchical position, which is not supported by existing contexts. It raises only correct composite events, i.e., composite events whose constituent events are hierarchically related.

To combine events according to their hierarchical position an event tree maintains data concerning hierarchy. Therefore, as mentioned earlier, every node n in an event tree has an assigned path type $n.et.pt$. Naturally, an event type node specifies a path type, however, the path type of an operator node, if not specified by the event expression, has to be derived from its child nodes $c_1, c_2, ..., c_n$ by evaluating $c_1.et.pt \cap_{\mathsf{lb}} c_2.et.pt \cap_{\mathsf{lb}} ... \cap_{\mathsf{lb}} c_n.et.pt$ (left-bound intersection is used for simplicity). The derivation of path types for all operator nodes is done bottom up. For an event tree to be valid, every node n must have a non-$null$ path type whose steps are not empty and do not violate the constraints on child nodes (cf. later in this section).

Example 10. Figure 1 shows an exemplary event tree on the left defining composite event type $\mathsf{E}^c_i := \mathsf{E}_i \; ; \; (\mathsf{E}_p \triangle \mathsf{E}_q)$ for the insertion of item elements and the derivation of path types for operator nodes on the right. The derivation is done bottom up, step 1 determines the path type of operator node $\triangle$ by evaluating $\mathsf{item/price} \cap_{\mathsf{lb}} \mathsf{item/quantity} = \mathsf{item}$. Subsequently, step 2 determines the path type of operator node ; by evaluating $\mathsf{item} \cap_{\mathsf{lb}} \mathsf{item} = \mathsf{item}$.

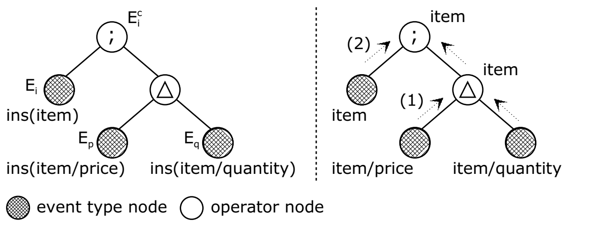
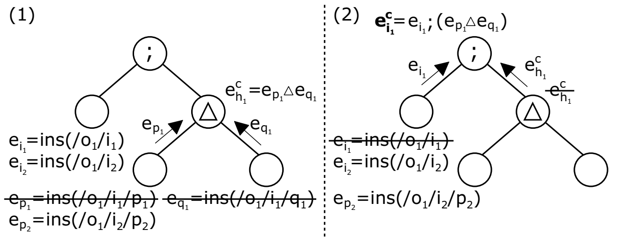


Figure 1: Derivation of operator nodes' path types

An operator node raises a composite event by selecting events from its child nodes that satisfy certain conditions. Basically, (a) at least one child node of a disjunction operator node must hold an event, (b) all child nodes of a conjunction operator node must hold an event, and (c) all child nodes of a sequence operator node must hold an event and they must have occurred in the specified order.

If operator node o combines multiple events from child nodes they must all have the same ancestor node, i.e., $\forall 1 \leq i \leq m-1$: $\pi_{o.et.pt}(e_i.pi) = \pi_{o.et.pt}(e_{i+1}.pi)$ where m is the number of events in o's child nodes that are to be combined. Events for which the projection evaluates to $null$ are not combined. Raised composite event e^c's path instance, which must be absolute, is derived from constituent events by $\pi_{o.et.pt \cap_{\mathsf{ab}} e_1.pi.pt}(e_1.pi)$. Event e_1 is the first constituent event, however, any other constituent event e_i could be used instead because if all constituent events e_i equal in $\pi_{o.et.pt}(e_i.pi)$ they equal in $\pi_{o.et.pt \cap_{\mathsf{ab}} e_i.pi.pt}(e_i.pi)$ as well.

Example 11. How the event tree in Figure 1 forms composite events in hierarchical context is shown in Figure 2. Its status after the occurrence of the event sequence $e_{i_1}, e_{i_2}, e_{p_1}, e_{p_2}, e_{q_1}$ is shown on the left. In path instances and event indices, o abbreviates order, i abbreviates item, p abbreviates price, q abbreviates quantity, and the numerical index indicates the hierarchical position, e.g., p_1 is a child of i_1. Events e_{p_1} and e_{q_1} are combined by operator $\triangle$ to form composite event $e^c_{h_1}$, since $\pi_{\mathsf{item}}(e_{p_1}.pi) = \pi_{\mathsf{item}}(e_{q_1}.pi)$ $(= \mathsf{i}_1)$. With the creation of $e^c_{h_1}$ the two primitive events are consumed (step 1). Subsequently, e_{i_1} and $e^c_{h_1}$ are combined by operator ; since $\pi_{\mathsf{item}}(e_{i_1}.pi) = \pi_{\mathsf{item}}(e^c_{h_1}.pi)$ $(= \mathsf{i}_1)$ (step 2). Upon the occurrence of e_{q_2} later on the same combination process starts over again.



(1)
;
△
e^c_{h_1}=e_{p_1}△e_{q_1}
e_{p_1}
e_{q_1}
e_{i_1}=ins(/o_1/i_1)
e_{i_2}=ins(/o_1/i_2)
~~e_{p_1}=ins(/o_1/i_1/p_1)~~
~~e_{q_1}=ins(/o_1/i_1/q_1)~~
e_{p_2}=ins(/o_1/i_2/p_2)
(2)
e^c_{i_1}=e_{i_1};(e_{p_1}△e_{q_1})
;
e_{i_1}
e^c_{h_1}
~~e^c_{h_1}~~
△
~~e_{i_1}=ins(/o_1/i_1)~~
e_{i_2}=ins(/o_1/i_2)
e_{p_2}=ins(/o_1/i_2/p_2)

Figure 2: Event selection and event consumption in hierarchical context

Constraints on operator node o's path type are enforced so that the projection of an event's path instance on o's path type is likely to return a non-$null$ value, i.e., so that for child event e_j expression $\pi_{o.et.pt}(e_j.pi) \neq null$ applies. The constraints seek a compromise between the operator's expressiveness for the reuse of event trees (cf. later in this section) and restrictions on child nodes and do thus not guarantee a non-$null$ value. They are as follows (where E_j refers to a child node's event type): (a) if both $o.et.pt$ and $E_j.pt$ are absolute, $o.et.pt \cap_{\mathsf{lb}} E_j.pt \neq null$ must apply, (b) if only $E_j.pt$ is absolute, $o.et.pt \subset E_j.pt$ must apply. If $E_j.pt$ is relative no constraints must apply so that it may contain only a single step that is not contained in $o.et.pt$. In such cases one must rely that an event's (dynamic) event type is compatible to the operator's (static) event type so that the event's path instance can be projected on the operator's path type. This can be compared to a type-cast in strongly typed object-oriented programming languages where an object's dynamic type must be compatible to the static casted type which can only be determined at runtime. For examples see Example 13 and 14.

Operators combine events with *interval-based semantics* not detection-based semantics, since composite events detected with the latter are not always exactly as, presumably, intended (cf. [1, 10]). Basically, this means that operators do not combine events according to their detection time but take the intervals during which the events occurred into account. A primitive event detected at t_d occurs in interval $[t_d, t_d]$, and a composite event starts at the beginning of the smallest interval and ends at the end of the latest one. Comparing intervals, $[t'_a, t''_a] < [t'_b, t''_b]$ iff $t''_a < t'_b$. Interval-based semantics only affect the sequence operator, so that expression $E_i\,;E_j$ combines two hierarchically matching events e_i and e_j only if the occurrence interval of the former is smaller than the other.

The hierarchical context is *orthogonal* to existing contexts from Snoop, because selection by XML hierarchy is orthogonal to selection by time. Thus it can be combined with any existing context, acting like a filter. For operator node o, first the hierarchical context groups all events from child nodes according to o's path type, and second o selects and consumes events within each group according to its context from Snoop.

Example 12. To detect when both price and quantity information of an order item are modified, event expression $*(\text{item/quantity}) \triangle \text{upd}(\text{item/price})$ can be used. Table 2 shows raised composite events, represented by their constituent events. for the above expression in the hierarchical variants of Snoop's contexts. Events that remain in the event graph after t_5 are shown in the rightmost column for completeness.

A composite event type definition can *reuse* other, existing event type definitions, i.e., and event tree can reuse other event trees. All event trees together form the *event graph*. A tree that defines composite event type E_i^c is reused in another tree through an event type node with event type E_j^c iff $E_i^c \succeq E_j^c$. Then it may be the case that the node's parent operator node o's path type $o.et.pt$ cannot be derived from its child nodes, because the left-bound intersection of $E_j^c.pt$ with the path types of o's other child nodes is $null$, e.g., if their path types are of different kind or $E_j^c.pt$ contains only a single step. In either case o's path type must be specified explicitly.

Example 13. The order element has aside from item elements a billTo and a shipTo child element. When two composite event types $\mathsf{E}_\mathsf{b}^\mathsf{c}$ and $\mathsf{E}_\mathsf{s}^\mathsf{c}$ for complete insertions of the latter two elements are defined in addition to $\mathsf{E}_\mathsf{i}^\mathsf{c}$ (from Example 10), a fourth composite event type can use the three to define a composite event that occurs after both addresses and at least one item have been inserted by $\mathsf{E}_\mathsf{i}^\mathsf{c} \triangle \mathsf{E}_\mathsf{s}^\mathsf{c} \triangle \mathsf{E}_\mathsf{b}^\mathsf{c}$. Since, e.g., $\mathsf{E}_\mathsf{i}^\mathsf{c}.pt \cap_{\mathsf{lb}} \mathsf{E}_\mathsf{s}^\mathsf{c}.pt \cap_{\mathsf{lb}} \mathsf{E}_\mathsf{b}^\mathsf{c}.pt = \text{item} \cap_{\mathsf{lb}} \text{shipTo} \cap_{\mathsf{lb}} \text{billTo} = null$, the conjunction operator's path type must be specified explicitly, e.g., as order. Because the (dynamic) event types of occurred events have absolute path types, the projection of their path instances on order is not $null$.

Reuse of event type definitions is facilitated by using type compatibility instead of type equality for storing occurred events in event type nodes, as mentioned before and in Section 2.2. The reason is that the more leaf nodes with "general" event types an event tree has (i.e., nodes with event types that have relative path types and/or wildcard "$*$" as operator or name), the more events will be stored in it because an event's "special" event type (i.e., one with absolute path type and given operation) may be compatible to a general one but not to another special one. The more events are stored, the more composite events are raised, and the more the tree, i.e., the event definition is reusable.

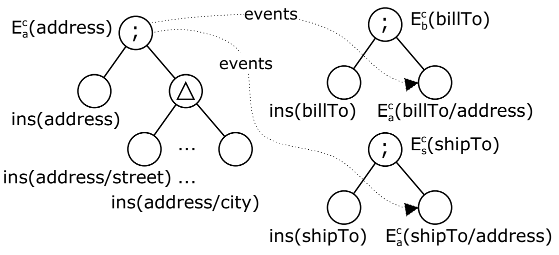


Figure 3: Reuse of event trees

Example 14. The order's shipTo and billTo elements both have an address child element that comprises other elements such as street and city. By defining composite event type $\mathsf{E}_\mathsf{a}^\mathsf{c}$ for the insertions of address elements and using only relative path types that start with address, the according event tree in Figure 3 raises composite events for complete insertions of address elements as childs of both shipTo and billTo. Thus $\mathsf{E}_\mathsf{a}^\mathsf{c}$ can be reused by both $\mathsf{E}_\mathsf{b}^\mathsf{c}$ and $\mathsf{E}_\mathsf{s}^\mathsf{c}$ from Example 13.

3.2 Multiplicity Operator

Because an XML schema can define multiplicity constraints on XML elements and attributes it is desirable to be able to detect when multiplicity constraints are satisfied by observing occurred events. A multiplicity constraint is defined by lower bound l and upper bound u ($l \leq u$), meaning that between l and u child elements or attributes with the same name may occur as child of a parent element. After l events reflecting insertions the multiplicity constraint is satisfied (if the parent element did not contain any such child elements before). The operator from Snoop that closest resembles the required functionality is the ANY operator. It detects a fixed number > 0 of events of distinct event types, however, in XML the required number is ≥ 0, since 0 events reflect optional elements, and the events are of the same event type.

To detect when multiplicity constraints are satisfied unary multiplicity operator "$\times$" is introduced, denoted as $\times[l,u]E_i$. It raises a composite event as soon as l events of E_i occurred, thereby indicating the constraint's satisfaction. The consumption of the composite event, however, may take place after other events of E_i occurred. To provide composite events with most extensive sets of constituent events, the multiplicity operator has *integrative semantics*.

Event integration starts after composite event e_1^c is raised upon the occurrence of the l^{th} event of E_i. A subsequently occurring event gives rise to the new composite event e_2^c which integrates e_1^c's constituent events. Event e_1^c is waived because it has been integrated and is thus assumed not to be of interest any longer. If u is reached or the integrating composite event is consumed, integration is suspended and starts over after the l^{th} occurrence of E_i.

When employed in a hierarchical context, the multiplicity operator shows the above behavior for every distinct path instance of its path type. If not specified explicitly, a multiplicity operator's path type is set to the child node's path type omitting the last step. The same constraints on a multiplicity operator's path type must apply as on other operator's path types (cf. Section 3.1). Multiplicity operator o raises a composite event e_1^c as soon as there exist l child events that equal in $\pi_{o.et.pt}(e_{i_j}.pi)$ for $1 \leq j \leq l$. Thereafter it raises new composite event e_2^c upon the occurrence of child event e_{i_k} where $\pi_{o.et.pt}(e_{i_k}.pi) = \pi_{o.et.pt}(e_1^c.pi)$ if e_1^c has not been consumed yet, where $e_2^c.cevts$ is defined by the union of $e_1^c.cevts$ and e_{i_k}.

Example 15. The event tree depicted in Figure 4 allows multiple insertions of quantity elements as childs of element item. For two insertions of quantity elements $e_{q_{1.1}}$ and $e_{q_{1.2}}$ the multiplicity operator first raises composite event $e_{q_1}^c$ upon the occurrence of $e_{q_{1.1}}$. If $e_{q_1}^c$ has not been consumed upon the occurrence of $e_{q_{1.2}}$ new composite event $e_{q_2}^c$ is raised, where $e_{q_2}^c.cevts = \{e_{q_{1.1}}, e_{q_{1.2}}\}$, and $e_{q_1}^c$ is waived. Otherwise, i.e., if $e_{q_1}^c$ has been consumed new composite $e_{q_2}^c$ is raised as well, however, with $e_{q_2}^c.cevts = \{e_{q_{1.2}}\}$.

A multiplicity operator in hierarchical context with a lower bound of zero raises a composite event without the occurrence of a child event. Instead, composite event e_1^c is raised with the event representing the creation of (absolute) path instance pi that satisfies the constraint of multiplicity operator o by having $o.et.pt \subset_{\mathsf{e}}$

Table 2: Combination of the hierarchical context with contexts from Snoop

-	t_1	t_2	t_3	t_4	t_5	unconsumed events
Hierarchical Cumulative Context						
S_5	$e_{q1.1}$	e_{q2}	$e_{q1.2}$	$e_{p2}, \{e_{p2}e_{q2}\}^c$	$e_{p1}, \{e_{p1}e_{q1.1}e_{q1.2}\}^c$	–
Hierarchical Chronicle Context						
S_5	$e_{q1.1}$	e_{q2}	$e_{q1.2}$	$e_{p2}, \{e_{p2}e_{q2}\}^c$	$e_{p1}, \{e_{p1}e_{q1.1}\}^c$	$e_{q1.2}$
Hierarchical Recent Context						
S_5	$e_{q1.1}$	e_{q2}	$e_{q1.2}$	$e_{p2}, \{e_{p2}e_{q2}\}^c$	$e_{p1}, \{e_{p1}e_{q1.2}\}^c$	$e_{p1}, e_{q1.2}, e_{p2}, e_{q2}$
Hierarchical Continuous Context						
S_5	$e_{q1.1}$	e_{q2}	$e_{q1.2}$	$e_{p2}, \{e_{p2}e_{q2}\}^c$	$e_{p1}, \{e_{p1}e_{q1.1}\}^c, \{e_{p1}e_{q1.2}\}^c$	e_{p1}, e_{p2}

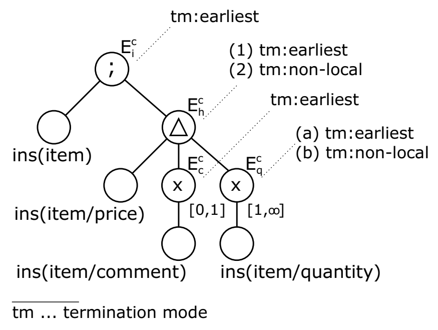


Figure 4: Event tree using a multiplicity operator

$pi.pt \vee o.et.pt = pi.pt$. It does not comprise any constituent event, its path type is set to $pi.pt$, and its path instance to pi. When event e_i with $\pi_{o.et.pt}(e_i.pi) = \pi_{o.et.pt}(e^c_1.pi)$ subsequently occurs in o's child node before e^c_1 is consumed, new composite event e^c_2 which integrates e^c_1 is raised. If composite event e^c_2 is consumed later on and $e^c_2.pi$ still exists, new composite event e^c_3 is raised since o's constraints are still satisfied.

Example 16. The event tree depicted in Figure 4 allows optional element comment as child of element item. With the insertion of item element i_1 composite event $e^c_{c_1}$ occurs with $e^c_{c_1}.et.pt =$ /order/item and $e^c_{c_1}.pi =$ /o_1/i_1. Thereafter, the conjunction operator raises composite event $e^c_{h_1}$ as soon as matching events are stored in its other child nodes, i.e., as soon as p_1 and q_1 are inserted as childs of i_1. If matching comment element c_1 is inserted before both elements p_1 and q_1 are inserted, event $e^c_{c_2}$ integrating $e^c_{c_1}$ is raised which becomes a part of $e^c_{h_1}$ later on.

3.3 Operator Modifiers

To enrich the expressiveness of event type definitions, operator nodes are parameterized by two modifiers to exactly define the points in time when composite events are raised. When an operator node detects events in child nodes that satisfy the operator's semantics, a "potential composite event", which is not stored in the tree, is detected. A composite event, which is stored in the tree, is raised with the detection of the potential event or is deferred to a later date. This is of importance, e.g., when a multiplicity operator or the hierarchical cumulative context is used, because the later a composite event is raised the more constituent events it will possibly have. Thus applications can react to deferred composite events comprising a rich set of constituent events and can determine the net-effect [15], i.e., overall effect of multiple events more easily.

First, the *termination mode* determines when a composite event is raised relative to the detection of a potential event. If the termination mode is earliest, composite event e^c_i is raised with the detection of potential composite event e^c_p. If it is non-local, e^c_i is raised after the detection of e^c_p and the first occurrence of event e_j, where $\pi_{o.et.pt}(e^c_p.pi) \neq \pi_{o.et.pt}(e_j.pi)$. This means that operator node o waits until an event occurs that reflects a manipulation in another subtree of the document, i.e., it assumes that the manipulation of a document is done hierarchically. If the termination mode is custom, composite events are raised upon flushing or closing the event tree (cf. later in this section).

Example 17. For the event tree depicted in Figure 4 and four exemplary event sequences $S_6..S_9$ Table 3 shows when composite events are raised depending on the termination mode of (i) the conjunction operator node and (ii) the multiplicity operator node of the quantity element. The other two operator nodes have termination mode earliest. To clearly point out the consequences of termination modes the table shows composite events raised by any operator node. A composite event is denoted by e^c with an alphabetical index indicating its event type and a numerical index indicating its hierarchical position. Note that due to the order of occurred events in $S_6..S_9$ the same composite events are raised irrespective of the context from Snoop that is combined with the hierarchical context.

Second, the *termination condition* must be fulfilled for a composite event to be raised. It is a condition on the subtree of the XML document with root node $e^c_p.pi$ in the form of an arbitrary XPath expression that evaluates to boolean. The termination condition differs from operators and termination modes in that it is used to test the document and not events, e.g., to test if text nodes contain any or certain text.

Example 18. Instead of adding event type nodes and operator nodes that test for the insertion of text nodes to the event tree depicted in Figure 4, termination condition "item/price $\geq 0 \wedge$ item/quantity > 0" can be defined on the root node so that a composite event is only raised when both price and quantity element contain a value that is a number. This has also the advantage that text nodes can be arbitrarily inserted, updated, and deleted because its their value that matters, not the operations that lead to it.

An event tree can be opened, flushed, and closed by an application. After opening it stores occurring primitive and composite events. If it is flushed, remaining composite events are raised by operator nodes with non-local or custom termination mode. Closing an event tree first flushes it and afterwards takes it out of order, e.g., when an XML document is closed after manipulation.

4. IMPLEMENTATION

The implemented proof-of-concept prototype extends the DOM event module of Apache's Xerces [9] and thus provides Java applications with composite events. This section briefly describes the prototype's execution model.

Table 3: Raised composite events in hierarchical context when different termination modes are used

-	t_1	t_2	t_3	t_4	t_5	t_6
(1a) $\triangle$: earliest, $\times[1,\infty]$: earliest						
S_6	$e_{i_1}, e^c_{c_1}$	e_{p_1}	$e_{q_1}, e^c_{q_1}, e^c_{h_1}, \mathbf{e^c_{i_1}}$	$e_{i_2}, e^c_{c_2}$	–	–
S_7	$e_{i_1}, e^c_{c_{1.1}}$	e_{p_1}	$e_{c_1}, e^c_{c_{1.2}}$	$e_{q_1}, e^c_{q_1}, e^c_{h_1}, \mathbf{e^c_{i_1}}$	$e_{i_2}, e^c_{c_2}$	–
S_8	$e_{i_1}, e^c_{c_{1.1}}$	e_{p_1}	$e_{q_{1.1}}, e^c_{q_{1.1}}, e^c_{h_1}, \mathbf{e^c_{i_1}}$	$e_{c_1}, e^c_{c_{1.2}}$	$e_{q_{1.2}}, e^c_{q_{1.2}}$	$e_{i_2}, e^c_{c_2}$
S_9	$e_{i_1}, e^c_{c_1}$	e_{p_1}	$e_{i_2}, e^c_{c_2}$	e_{p_2}	$e_{q_1}, e^c_{q_1}, e^c_{h_1}, \mathbf{e^c_{i_1}}$	$e_{q_2}, e^c_{q_2}, e^c_{h_2}, \mathbf{e^c_{i_2}}$
(1b) $\triangle$: earliest, $\times[1,\infty]$: non-local						
S_6	$e_{i_1}, e^c_{c_1}$	e_{p_1}	e_{q_1}	$e_{i_2}, e^c_{c_2}, e^c_{q_1}, e^c_{h_1}, \mathbf{e^c_{i_1}}$	–	–
S_7	$e_{i_1}, e^c_{c_{1.1}}$	e_{p_1}	$e_{c_1}, e^c_{c_{1.2}}$	e_{q_1}	$e_{i_2}, e^c_{c_2}, e^c_{q_1}, e^c_{h_1}, \mathbf{e^c_{i_1}}$	–
S_8	$e_{i_1}, e^c_{c_{1.1}}$	e_{p_1}	$e_{q_{1.1}}$	$e_{c_1}, e^c_{c_{1.2}}$	$e_{q_{1.2}}$	$e_{i_2}, e^c_{c_2}, e^c_{q_1}, e^c_{h_1}, \mathbf{e^c_{i_1}}$
S_9	$e_{i_1}, e^c_{c_1}$	e_{p_1}	$e_{i_2}, e^c_{c_2}$	e_{p_2}	e_{q_1}	$e_{q_2}, e^c_{q_1}, e^c_{h_1}, \mathbf{e^c_{i_1}}$
(2a) $\triangle$: non-local, $\times[1,\infty]$: earliest						
S_6	$e_{i_1}, e^c_{c_1}$	e_{p_1}	$e_{q_1}, e^c_{q_1}$	$e_{i_2}, e^c_{c_2}, e^c_{h_1}, \mathbf{e^c_{i_1}}$	–	–
S_7	$e_{i_1}, e^c_{c_{1.1}}$	e_{p_1}	$e_{c_1}, e^c_{c_{1.2}}$	$e_{q_1}, e^c_{q_1}$	$e_{i_2}, e^c_{c_2}, e^c_{h_1}, \mathbf{e^c_{i_1}}$	–
S_8	$e_{i_1}, e^c_{c_{1.1}}$	e_{p_1}	$e_{q_{1.1}}, e^c_{q_{1.1}}$	$e_{c_1}, e^c_{c_{1.2}}$	$e_{q_{1.2}}, e^c_{q_{1.2}}$	$e_{i_2}, e^c_{c_2}, e^c_{h_1}, \mathbf{e^c_{i_1}}$
S_9	$e_{i_1}, e^c_{c_1}$	e_{p_1}	$e_{i_2}, e^c_{c_2}$	e_{p_2}	$e_{q_1}, e^c_{q_1}$	$e_{q_2}, e^c_{h_1}, \mathbf{e^c_{i_1}}$
(2b) $\triangle$: non-local, $\times[1,\infty]$: non-local						
S_6	$e_{i_1}, e^c_{c_1}$	e_{p_1}	e_{q_1}	$e_{i_2}, e^c_{c_2}, e^c_{q_1}, e^c_{h_1}, \mathbf{e^c_{i_1}}$	–	–
S_7	$e_{i_1}, e^c_{c_{1.1}}$	e_{p_1}	$e_{c_1}, e^c_{c_{1.2}}$	e_{q_1}	$e_{i_2}, e^c_{c_2}, e^c_{q_1}, e^c_{h_1}, \mathbf{e^c_{i_1}}$	–
S_8	$e_{i_1}, e^c_{c_{1.1}}$	e_{p_1}	$e_{q_{1.1}}$	$e_{c_1}, e^c_{c_{1.2}}$	$e_{q_{1.2}}$	$e_{i_2}, e^c_{c_2}, e^c_{q_1}, e^c_{h_1}, \mathbf{e^c_{i_1}}$
S_9	$e_{i_1}, e^c_{c_1}$	e_{p_1}	$e_{i_2}, e^c_{c_2}$	e_{p_2}	e_{q_1}	$e_{q_2}, e^c_{q_1}, e^c_{h_1}, \mathbf{e^c_{i_1}}$

Every occurred primitive event is inserted into the event graph, which is processed to detect composite events. To determine the order in which the event trees of the graph are processed, which remains the same as long as neither the graph nor the trees are modified, the notion of *event tree dependency* is introduced. Event tree t_b directly depends on event tree t_a, denoted as $t_a \rightarrow t_b$, iff t_b uses composite event type et_b to which event type et_a defined by t_a is compatible to, i.e., if $et_a \succeq et_b$. An event tree may not depend directly or indirectly on itself.

Algorithm `processGraph` for processing event graph G upon the occurrence of primitive event e is shown below. It processes every event tree of the graph exactly once. Line *#1* determines the ordered set of event trees that do not directly depend on any other event tree and assigns it to T. Subsequently, the set of occurred and raised events E is initialized to e (line *#2*). While there are event trees left that need to be processed (line *#3*), every event tree $t_i \in T$ is processed (line *#4*–*#5*, see later in this section), where t_i denotes the i^{th} element of T. Subsequently, all event trees that directly depend on any $t \in T$ and thus need to be processed are determined and assigned to T (line *#6*). Finally, T is purged (line *#7*, see later in this section).

```
processGraph($e$)
#1  $T := \{t \in G | \nexists u \in G : u \rightarrow t\}$
#2  $E := \{e\}$
#3  while $T \neq \emptyset$ do
#4      for $i = 1$ to $|T|$ do
#5          $E :=$ processTree($t_i$, $E$)
#6      $T := \{t' \in G | \exists t \in T : t \rightarrow t'\}$
#7      purge($T$)
```

Algorithm `processTree(`t, E`)` processes event tree t with the set of events E as follows. The tree is traversed in postorder (a form of depth-first traversal) during which (a) every visited event type node n is tested for type compatibility with the event type of every event $e \in E$ and if they are compatible, i.e., $e.et \succeq n.et$, e is stored in $n.evts$ (but not taken out of E), and (b) every visited operator node o is evaluated if (i) an event was stored in a (direct) child node, (ii) if o specifies a termination condition and for the primitive event $e \in E$ it applies that $o.et.pt \subset e.pt \vee o.et.pt = e.pt$, (iii) if o has non-local termination mode, or (iv) if o is a multiplicity operator node with $l = 0$ and for the primitive event $e \in E$ it applies that $o.et.pt \subset_{\mathsf{e}} e.pt \vee o.et.pt = e.pt$. Finally, composite events that are raised in the root node of event tree t are added to E to be processed by dependent event trees later on.

Algorithm `purge(`T`)` removes every event tree $t_k \in T$ from T if it is contained in the closure of another event tree $t_j \in T$, where the closure of a tree refers to the set of trees that directly and indirectly depend on it. This guarantees that an event tree is processed exactly once at the latest time possible.

5. CONCLUSIONS

We have presented the first approach to detect composite events in XML, which refines and extends the event algebra Snoop to take the peculiarities of events in XML into account and provide for the detection of satisfied multiplicity constraints. Moreover, the introduced concept of compatibility of event types makes event expressions more expressive and reusable. The prototype including additional examples is available on the Web at `http://www.big.tuwien.ac.at/research/prototypes/composite-events`. Naturally our approach is independent of an implementation and can be used by any application in need of composite events in XML, presumably, such as [2] and [6].

The presented approach is fully compatible with Snoop and thus provides for combined expressiveness. First, the hierarchical context presented herein is orthogonal to Snoop's contexts and can thus be arbitrarily combined with the latter, providing for *simultaneous* event combination by hierarchical position and time. Second, composite XML event types can be combined by operators from Snoop, providing for *subsequent* event detection based on time (e.g., `NOT` detects non-occurrences of events in intervals).

Our ongoing research concentrates on the automatic derivation of event expressions from XML schemas (i.e., schemas expressed in XML Schema [19]) to provide a set of composite event type definitions an application engineer can start working with. We assume that if schemas use XML schema concepts such as type definitions, type hierarchies and model groups in a meaningful way, derived composite event type definitions will be meaningful as well.

6. ACKNOWLEDGMENTS

We want to thank Jan Wenger for implementing the prototype and giving valuable feedback and Elke Michlmayr for carefully reading the paper and giving helpful comments.

7. REFERENCES

[1] R. Adaikkalavan and S. Chakravarthy. Event Operators: Formalization, Algorithms, and Implementation. Technical Report CSE-2002-3, Department of Computer Science and Engineering, University of Texas at Arlington, 2002.

[2] J. Bailey, A. Poulovassilis, and P. T. Wood. An Event-Condition-Action Language for XML. In *Proceedings of the 11th International Conference on World Wide Web (WWW11), Honolulu, USA*, pages 486–495. ACM Press, 2002.

[3] J. Bailey, A. Poulovassilis, and P. T. Wood. Analysis and Optimisation of Event-Condition-Action Rules on XML. *Computer Networks*, 39(3):239–259, 2002.

[4] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *Proceedings of the 18th International Conference on Data Engineering (ICDE), San Jose, USA*, 2002.

[5] A. Bonifati, S. Ceri, and S. Paraboschi. Active Rules for XML: A New Paradigm for E-Services. *The VLDB Journal*, 10(1):39–47, 2001.

[6] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing Reactive Services to XML Repositories using Active Rules. In *Proceedings of the 10th International World Wide Web Conference (WWW10), Hong Kong, China*, pages 633–641. ACM Press, 2001.

[7] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), Santiago de Chile, Chile*, pages 606–617. Morgan Kaufmann, 1994.

[8] S. Chakravarthya and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data & Knowledge Engineering*, 14(1):1–26, Nov. 1994.

[9] A. S. Foundation. Xerces-Java. `http://xml.apache.org/xerces2-j/`, 2003.

[10] A. Galton and J. C. Augusto. Two Approaches to Event Definition. In *Proceedings of the 13th International Conference on Database and Expert Systems Applications (DEXA), Aix-en-Provence, France*, number 2453 in LNCS, pages 547–556. Springer, 2002.

[11] S. Gatziu and K. R. Dittrich. Events in an Active Object-Oriented Database System. In *Proceedings of the 1st Intl. Workshop on Rules in Database Systems (RIDS), Edinburgh, Scotland*, pages 23 – 29. Springer, 1993.

[12] S. Gatziu and K. R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In *Proceedings of the 4th Intl. Workshop on Research Issues in Data Engineering (RIDE): Active Database Systems, Houston, Texas*, 1994.

[13] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. In *Proceedings of the 18th International Conference on Very Large Databases (VLDB)*, 1992.

[14] G. Kappel, S. Rausch-Schott, and W. Retschitzegger. A Framework for Workflow Management Systems Based on Objects, Rules and Roles. *ACM Computing Surveys Electronic Symposium on Object-Oriented Application Frameworks*, 32(1), Mar. 2000.

[15] N. Paton and O. Diaz. Active Database Systems. *ACM Computing Surveys*, 31(1):63–103, Mar. 1999.

[16] M. Schrefl and M. Bernauer. Active XML Schemas. In *Proceedings of the Workshop on Conceptual Modeling Approaches for e-Business (eCOMO) at the International Conference on Conceptual Modeling (ER), Yokohama, Japan*, volume 2465 of *LNCS*. Springer, 2001.

[17] W3C. XML Path Language (XPath), W3C Recommendation. `http://www.w3.org/TR/xpath`, Nov. 1999.

[18] W3C. XML Information Set, W3C Recommendation. `http://www.w3.org/TR/xml-infoset`, Oct. 2001.

[19] W3C. XML Schema Part 1: Structures, W3C Recommendation. `http://www.w3.org/TR/xmlschema-1`, May 2001.

[20] W3C. Document Object Model (DOM) Level 3 Events Specification. `http://www.w3.org/TR/DOM-Level-3-Events/`, Mar. 2003.

[21] W3C. XQuery 1.0 and XPath 2.0 Functions and Operators, W3C Working Draft. `http://www.w3.org/TR/xpath-functions`, May 2003.

[22] R. J. Zhang and E. A. Unger. Event Specification and Detection. Technical Report CS-96-8, Department of Computing and Information Sciences, Kansas State University, 1996.

[23] D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proceedings of the 15th International Conference on Data Engineering (ICDE), Sydney, Australia*, pages 392–399. IEEE Computer Society Press, 1999.