Active XQuery

Angela Bonifati[1] and Stefano Paraboschi[2]

[1] ICAR-CNR, Via P.Bucci, 41C - 87036 Rende CS, Italy bonifati@icar.cnr.it
[2] Universitá di Bergamo, Via Marconi, 5 - 24044 Dalmine BG, Italy
parabosc@unibg.it

Summary. We analyze some of the issues arising when an active rule mechanism is introduced within an XML management system. Apart from the presentation of a specific solution, the focus of the chapter is on two aspects. First, we analyze the coupling modes (immediate and deferred) that can be adopted to integrate the event application part and the rule evaluation part. Then, we illustrate how the component responsible of updating XML data and the rule processing engine can be integrated in a loose or tight way, producing two distinct rule execution semantics.

1 Introduction

The Web is currently migrating from a rather static asset to a dynamic configuration, in which automatized mechanisms have an increasing role. In particular, in several applications, such as negotiation portals, billing desks, reservation sites, monitoring services, index directories and many others, the promising XML technology needs to be augmented with special active features. By active we intend whatever piece of code is activated synchronously or asynchronously by an occurred event. The events can be general enough to include data modification events or simple invocations of a given function. We have identified many commonalities between this new active breed and traditional active database rules [18, 9, 8]. First, to locate the necessary information, they both need suitable querying capabilities. Next, like stored procedures and active rules, the data need to reside close to the active code, which exploits it to perform its computations. In previous work [3, 4, 5], we have showed that traditional triggers, once reshaped in any current XML language, may respond to these needs. Thus, in this chapter, we focus on the definition of an active language for XQuery [11], the standard XML query language.

The addition of active features to web languages creates many challenges and is reflected in many initiatives. Indeed, many efforts in the industry and in the research community aim at achieving a great level of automatization in a variety of mechanisms. Among them, we cite some commercial ones, such

as Macromedia MX [17], Apache Jelly [2], Sun JSP [16], PHP [19] and so on. They strictly integrate the functionality of a programming language or a script language with XML pages. These approaches are very similar to ours in the fact that an event-condition-action paradigm can be simulated. A specialized interpreter is needed each time an active feature is desired, and this requires some effort and customization. With Active XQuery, we propose a natural extension of the query processor to support triggers for XML data. As we will see while illustrating the architecture, the modifications of the query processor are not traumatic and once done, permit to exploit the full expressibility of the query language.

In a special application of XML triggers, such as that of monitoring web pages, one could argue that continuous queries can be fruitfully employed. The main advantage of the trigger solution resides in the possibility of programming a reaction to changes only when needed, leading to savings in the query instantiations and in the client-server connections. Moreover, triggers are located where the documents reside, and these do not need to be previously fetched in memory as for enacting a continuous query. Migration of the current XML query languages towards support of the ECA paradigm, is quickly feasible with little effort from a syntactic point of view. W.r.t. traditional technologies used to implement web services (i.e. agents), active rules can be preferred for their simplicity. Active rules will not be used as the final tool for each kind of e-service, because several brand-new e-services are too complex to be implemented by means of triggers and require more complex implementation mechanisms (such as using the agent technology in workflow management systems). However, triggers are inherently suitable for the rapid development of several conventional applications, such as view maintenaince, constraint handling, business rules etc. In addition, active rules can use as their action methods and procedures (compliant to any kind of an XML-compatible Web protocol, like SOAP [21]) that are executed remotely and implemented elsewhere. As a final observation, triggers are installed inside the interested web servers. Conversely, in an agent-based architecture, the agents are software modules instructed to go around in the network and to visit different hosts. Therefore, attacks to protected sites or execution of malicious code are more dangerous in an agent-based scenario.

A similar approach to ours has been developed within the Active XML project [1]. Active XML proposes a new framework for web services development, in which function calls are embedded into XML by using specialized `sc` tags. Calls are shipped on a remote server by means of a SOAP wrapper and a description of the web service in the *Web Services Description Language* (WSDL [27]) is available to distribute the service signature around the network. XQuery is used to express service calls as well as any programming language.

We organize this chapter as follows: in Section 2, we present the event model of Active XQuery and the choices of rule execution; in Section 3, we present some introductory examples of Active XQuery; in Section 3.1, we

describe the syntax and in Section 4, we describe the two alternative semantics, comparing their features. In Section 5, we show a classification of XML triggers and show, through examples, some other important application fields.

2 Active XQuery Event Base

XQuery is the standard query language for XML documents developed by W3C. It comes with a wide list of features for retrieving and re-constructing data within the XML documents, and with additional special functions for manipulating the text inside the data [24] or for treating special data types [28]. However, none of the manipulation operations, such as those issued by an update language, have yet been incorporated in the language. For this reason, in defining an active language for XQuery, we rely on the update model for XQuery recently developed in a research paper [23]. The update operations only affect the event model of the trigger language and the set of actions that are allowed in the action clause. Any other update language can be adopted instead of [23] for Active XQuery, by only modifying the syntax. Indeed, the semantics we have defined is general enough to explain the interpretation of any update operation.

Before illustrating the syntax and the execution model of Active XQuery, we briefly present the event model, the types of events and the event consumption modes. These are indeed the additional ingredients that need to be added to the query language in order to prepare the active extension.

2.1 Event Model

An event is something that happens at a point in time such that it is relevant to properly react to it. The event model permits to determine which is the event context, and how event occurrences are captured. Events are relatively easy to capture when they occur as a result of user's operations, either as update operations or plain queries: we name these events *explicit*. Indeed, in many settings the user may interact with an interface to submit his/her desired update operations. These operations cannot be always captured at run-time and thus the occurred modifications can only be grasped from a direct comparison of document versions. For instance, the operations over the document may be inferred by applying a diff algorithm [13] or a diff tool [15, 14], which compares two different snapshots of the same document. For conciseness, these events are called *derived* and they are stored into event sequences, which we call *event traces*.

Given an update language, its update commands may describe both types of events, *explicit* and *derived*. In some sense, the way events are collected or computed is orthogonal to the definition of the active language, which is the aim of this chapter. However, it may impact the behavior of the rules, as shown in our previous work [6].

As a final observation, the concept of event in the XML world is different from the one defined for tuples or for objects in the relational and object-oriented models. *Order* may be critical for documents. In the XML native data model [7], elements are ordered and attributes are unordered. However, if XML is used as an interchange format, order is less relevant than in other application fields. Relevance of order decreases as far as the focus is on data rather than on presentation.

This makes things more complicated and leads to two alternative interpretations of events, according to an *order-dependent* or *order-independent* semantics, along with the distinction between positional and non-positional operations. The *order-dependent* semantics defines a correct sequence of events that eventually produce a version of the document that depends on the order. Positional operations help to guarantee an order-dependent semantics. With an *order-independent* semantics, all the versions are equivalent to any that contains the same items in a different order. Non-positional operations are allowed in such a case. We do not distinguish between an order-dependent and an order-independent event semantics, because we consider the ordered semantics of the update language and directly transpose this semantics in the rule language. Thus, the sequence of operations in the update language enforces the order of events in the rule language evaluation.

2.2 Immediate versus Deferred event consumption

When a triggered rule is considered or actually executed, the events responsible for the triggering may undergo different treatments, that are called *event consumption modes*. Usually, the event consumption time is associated with the actual execution of rules rather than on condition evaluation. In our case, event consumption modes can be *immediate* or *deferred*[3]. When events are explicit, one can decide to process them (and to activate the corresponding rules) or to keep them for later evaluation. When events are derived from the cross comparison of two versions of the document, events processing (and rule execution) is naturally postponed to a subsequent time. Precisely, the explicit event detection supports the immediate and deferred event consumption modes, while the derived event detection only supports the *deferred* event consumption.

Therefore, in the *immediate* mode, events are consumed exactly when they occur (or equivalently rules are fired). The *immediate* mode of consumption enforces a sequence of state changes and rule sessions tightly interleaved (as shown in Figure 1). The state changes correspond to the sessions in which events are detected and rule sessions are the intervals in which events are consumed. In the *deferred* mode, events are consumed (rules are fired) promptly

[3] In active databases, the terms *immediate* and *deferred* are used to characterize the alternatives in the coupling mode between the event and the condition/action of rules. In this chapter we associate with the terms a different interpretation.

at the end of the transaction or at a given rule assertion point beyond the end of the transaction (as graphically represented in Figure 1). In such a case, the sequence of state changes and rule sessions are loosely interleaved. When an error occurs in rule evaluation, the last rule session is rolled back in the immediate mode and the entire transaction is rolled back in the deferred mode.

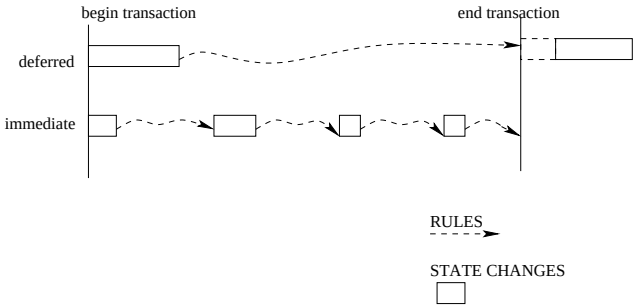

Fig. 1. Sequence of state changes and rule execution sessions in the immediate and deferred event consumption modes.

The event consumption mode also affects the rule execution mode and determines two ways to evaluate the triggers.

- The *immediate* execution mode processes the triggers exactly when events (due to update operations) occur.
- In the *deferred* execution mode, rules are not processed at the earliest opportunity after events, but in a separate session, and by analyzing the occurred event trace off-line.

In our discussion, we show the *immediate* execution mode. Nevertheless, a *deferred* execution mode has a similar treatment, as we will explain later.

Compared to relational updates, XQuery updates can be seen as *bulk* statements, since they may involve arbitrarily large fragments of documents, which are inserted or dropped by means of a single statement. These may trigger active rules which monitor events relative to internal portions of such document fragments. Thus, the main difficulty in extending the notion of triggers from the relational domain to the XML domain is indeed due to the different granularity between update events and rule events. To overcome this difficulty we have defined a first algorithm which expands bulk statements into a collection of equivalent statements, each one relative to a smaller fragment of the documents, so as to guarantee that any trigger defined for the document will be correctly considered. Each of these statements is in turn a self-standing XQuery update. This first algorithm defines a *loosely bundling immediate semantics.*

A second algorithm is defined that permits to expand the original update statement into a unique expanded statement. Rules and updates are more tightly interleaved than in the previous case, since only one expanded statement is generated. This second algorithm defines a *tightly bundling immediate semantics*.

3 Active XQuery by example

Let us assume a scenario based on the following `Library.xml` document, which belongs to the XML repository of a university library and describes the books stored on the shelves:

```
<Library>
  ...
  <Shelf nr="45">
    <Book id="AO97">
      <Author> J. Acute </Author>
      <Author> J. Obtuse </Author>
      <Title> Triangle Inequalities </Title>
    </Book>
    <Book id="So98">
      <Author> A. Sound </Author>
      <Title> Automated Reasoning </Title>
    </Book>
    ...
  </Shelf>
  ...
</Library>
```

An example of an update to the library is the bulk insertion of a whole shelf (nr. 45) into the document by means of the following XQuery update statement (`s0`). The new library content is extracted from a collection of new shelves, located in a separate document (within the repository). In order to insert fragment `$frag` the language requires to envelop the actual `INSERT` operation into an external `UPDATE` clause, targeted to a variable that is bound to the element that will *contain* the fragment (node `$target`). Recursively nested update statements (and therefore `UPDATE` clauses) are allowed within the curly brackets.

```
s0:
FOR $target IN document("Library.xml")/Library,
    $frag   IN document("New.xml")/Shelves/Shelf
WHERE $frag/@nr="45"
UPDATE $target  {  INSERT $frag  }
```

In our scenario, the library automatically maintains an index with a list of all authors, keeping pointers (IDREFs) to the library entries. The following XML excerpt demonstrates the author index:

```
<AuthorIndex>
 ...
 <AuthorEntry uni="PoliMi" pubs=".. A097 ..">
  <Name> J. Acute </Name>
  <PubsCount> ... </PubsCount>
 </AuthorEntry>
 ...
 <AuthorEntry uni="Princeton" pubs=".. So98 ..">
  <Name> A. Sound </Name>
  <PubsCount> ... </PubsCount>
 </AuthorEntry>
 ...
</AuthorIndex>
```

Triggers are responsible to guarantee referential integrity among the authors' publications (`pubs` attribute) and the books in the library. In particular, we want to guarantee the following properties:

1. **No dangling references**: *deletion of a* `Book` *element will cause all its authors (listed in the index part of the document) to lose their "dangling" references to that publication.*
2. **Automatic indexing**: *insertion of a* `Book` *element will cause a new reference to be inserted into all index items that represent new book's authors.* Note that this may require a new `Author` element to be inserted into the list.

Automatic deletion of dangling pointers is performed by trigger `NoDangle`, that updates `AuthorEntry` elements removing from their `pubs` attributes all references[4] to the deleted book (identified by keyword `OLD_NODE`):

```
CREATE TRIGGER NoDangle
AFTER DELETE OF document("Library.xml")//Book
FOR EACH NODE
DO ( FOR
  $AutIndex IN document("Library.xml")//AuthorIndex,
  $MatchAut IN $AutIndex/AuthorEntry
                    [Name = OLD_NODE/Author],
  $DangRef  IN $MatchAut/ref(pubs, OLD_NODE/@id)
UPDATE $AutIndex { DELETE $DangRef } )
```

Two other triggers perform the insertion of new references and new `AuthorEntry` elements. If one of the authors of the new book is not yet in the list, the higher-prioritized trigger `AddNewEntry` inserts a new "empty" `AuthorEntry` element. Thus, low-prioritized trigger `AddNewReference` can assume that the index already contains entries for all the incoming authors.

[4] Note that bindings to a single `IDREF` within an `IDREFS` attribute are declared according to the syntax extension proposed in [23].

```
CREATE TRIGGER AddNewEntry
AFTER INSERT OF document("Library.xml")//Book
FOR EACH NODE
LET $AuthorsNotInList := (
  FOR $n IN NEW_NODE/Author
  WHERE empty(//AuthorIndex/AuthorEntry[Name=$n])
  RETURN $n )
WHEN ( not( empty($AuthorsNotInList )) )
DO ( FOR $ai IN document("Library.xml")//AuthorIndex,
         $NewAuthor IN $AuthorsNotInList
     UPDATE $ai
     { INSERT <AuthorEntry>
               <Name> {$NewAuthor/text()} </Name>
                <PubsCount> 0 </PubsCount>
               </AuthorEntry> } )

CREATE TRIGGER AddNewReference
WITH PRIORITY -10
AFTER INSERT OF document("Library.xml")//Book
FOR EACH NODE
DO ( FOR $ai IN document("Library.xml")//AuthorIndex,
         $a  IN $ai/AuthorEntry[Name=$a]
     UPDATE $a
     { INSERT new_ref(pubs, NEW_NODE/@id)} )
```

Finally, triggers `IncrementCounter` and `DecrementCounter` maintain a counter of authors' publications (we only show `IncrementCounter` for brevity).

```
CREATE TRIGGER IncrementCounter
AFTER INSERT OF //new_ref(pubs)
FOR EACH NODE
LET $Counter := NEW_NODE/../PubsCount
DO ( FOR $AuthorEntry IN NEW_NODE/..
     UPDATE $AuthorEntry
     { REPLACE $Counter WITH $Counter + 1 } )
```

These triggers demonstrate that the execution of the action part of a trigger can cause the activation of other triggers (`AddNewReference` triggers `IncrementCounter`).

3.1 Quick Syntax of Active XQuery

The simplified syntax of an XQuery trigger is the following:

```
CREATE TRIGGER Trigger-Name
[WITH PRIORITY Signed-Integer-Number]
(BEFORE|AFTER)
  (INSERT|DELETE|REPLACE|RENAME)+
```

```
  OF XPathExpression (,XPathExpression)*
[FOR EACH (NODE|STATEMENT)]
[XQuery-Let-Clause]
[WHEN XQuery-Where-Clause]
DO (XQuery-UpdateOp|ExternalOp)
| TransactCom)
```

- The `CREATE TRIGGER` clause is used to define a new XQuery trigger, with the specified name.
- Rules can be prioritized in an absolute ordering, expressed with an optional `WITH PRIORITY` clause, which admits as argument any signed integer number. If this clause is omitted, the default priority is zero.
- The `BEFORE/AFTER` clause expresses the triggering time relative to the operation.
- Each trigger is associated with a set of update operations (insert, delete, rename, replace), adopted from the update extension of XQuery [23].
- The operation is relative to elements that match an XPath expression (specified after the `OF` keyword), i.e. a step-by-step path descending the hierarchy of documents (according to [12] and its update-related extensions[5]). One or more predicates (XPath *filters*) are allowed in the steps to eliminate nodes that fail to satisfy given conditions. Once evaluated on document instances, the XPath expressions result into sequences of nodes, possibly belonging to different documents.
- The optional clause `FOR EACH NODE/STATEMENT` expresses the trigger granularity. A *statement-level* trigger executes once for each set of nodes extracted by evaluating the XPath expressions mentioned above, while a *node-level* trigger executes once for each of those nodes. Based on the trigger granularity, it is possible to mention in the trigger the transition variables:
  - If the trigger is node-level, variables `OLD_NODE` and `NEW_NODE` denote the affected XML element in its before and after state.
  - If the trigger is statement-level, variables `OLD_NODES` and `NEW_NODES` denote the sequence of affected XML elements in their before and after state.
- An optional *XQuery-Let-Clause* is used to define XQuery variables whose scope covers both the condition and the action of the trigger. This clause extends the 'REFERENCING' clause of SQL:1999, because it can be used to redefine transition variables.
- The `WHEN` clause represents the trigger condition, and can be an arbitrarily complex XQuery `where` clause. If omitted, a trigger condition that specifies `WHEN TRUE` is implicit.

[5] The additional keyword `ref`, introduced in [23], can be used to denote a single IDREF within an attribute of type IDREFS.

- The action is expressed by means of the `DO` clause, and it can be accomplished through the invocation of an arbitrarily complex update operation. In addition, a generic *ExternalOp* syntax indicates the possibility of extending the XQuery trigger language with support to external operations, e.g. permitting to send mail or to invoke SOAP procedures. A *TransactCom* syntax indicates a generic transaction commands, such as a *fullRollback*, a *partialRollback* or a *commit* of the current transaction in which the trigger is executed.

For a complete syntax of XQuery refer to [10]. For the syntax of the update language, refer to [23].

4 The Immediate Execution Mode for Active XQuery

4.1 Intuitive semantics

The semantics of XQuery triggers should be as close as possible to the semantics of SQL:1999 triggers [20, 22], as already discussed. Accordingly, each XQuery operation should be computed in the context of a recursive procedure, such that:

- At the time of execution of an update, the set of affected nodes is computed (leading to the evaluation of transition variables).
- A given update statement is preceded by `BEFORE` triggers and followed by `AFTER` triggers[6]; statement and node level triggers may interleave, and are considered in priority order.
- If a given trigger executes an operation and this in turn causes some triggerings, the trigger execution context is suspended, and a new procedure is recursively invoked; the depth of recursion is limited by some given threshold, which is system specific.

However, such intuitive semantics cannot be immediately replicated in XQuery, given the hierarchical structure of XML and the "bulk" nature of update primitives. According to the update language of [23], the insertion of "content" may refer to an arbitrarily large XML fragment, and likewise the deletion of a node may cause the dropping of an arbitrarily large XML fragment. By "bulk" operations, we intend those operations of the update language composed by two or more constituents. The concept will be illustrated through an example.

Example 1. Consider the following update statement, which inserts a whole XML fragment containing sub-elements and attributes:

[6] In order to avoid nondeterministic and/or nonmonotonic behavior, `BEFORE` triggers may be subject to limitations in their actions.

```
s0:
FOR $target IN document("Library.xml")/Library,
    $frag   IN document("New.xml")/Shelves/Shelf
WHERE $frag/@nr="45"
UPDATE $target  {  INSERT $frag  }
```

The XML excerpt which is inserted is the following:

```
<Shelf nr="45">
   <Book id="A097">
      <Author> John Acute </Author>
      <Author> Jack Obtuse </Author>
      <Title> Applying Triangle Inequalities </Title>
   </Book>
   <Book id="So98">
      <Author> Anthony Sound </Author>
      <Title> Principles of Automated Reasoning </Title>
   </Book>
   ...
</Shelf>
```

Within `s0` there is one bulk statement `INSERT $frag` that needs to be expanded over the children of `Shelf`.

Expansion of bulk updates is the first step to semantics definition. We will show in the following section how expansion is accomplished.

Note that the SQL language instead supports update operations targeted to a given table and does not need to expand original statements. Therefore, a precise description of the semantics of XQuery triggers requires to be combined with a carefully defined management of bulk updates.

4.2 Update expansion

A first strategy (named *loosely binding semantics* (LBS)) with bulk updates consists of decomposing each original XQuery bulk update `s0` into a decomposition sequence S of smaller granularity self-standing updates, such that the change to each XML element involved in the update is addressed by a specific update operation of S. This strategy requires the definition of two separate mechanisms, one for expanding updates and one for executing them, where the latter includes the composition of updates with triggers. Note that update expansion requires access to the XML document; this is obvious in the case of bulk deletion (when the specific elements to be deleted need to be first accessed), but occurs with bulk insertions as well.

An alternative strategy is possible, consisting of interleaving update expansion and trigger evaluation. We call this semantics *tightly binding semantics* (TBS) as opposed to the *loosely binding semantics* aforementioned. In this strategy, the original XQuery update statement `s0` is decomposed into a single

nested update statement. In such a case, composition of updates with triggers is stricter, since triggers are invoked within the bulk update execution.

We have devised two algorithms addressing the two strategies. We will show both of them in this chapter, but we will focus on the first semantics for the implementation of the Active XQuery system.

In designing the expansion strategy (in both the semantics), we use a visit of the hierarchical structures which mimics the "natural" order of update propagation, in which inserts proceed top-down, and deletions proceed bottom-up. Such a visit strategy is described, in a simple case of bulk insert, in Figure 2 for the loosely binding semantics (the adopted notation displays attributes as black circles, elements as empty circles and PCDATA content as empty boxes); the considered fragment is that of the `Shelf` excerpt considered in the previous chapter.

Following this semantics, fragments are visited in a mixed breadth-depth order. In the first step of the algorithm, the first-level elements are visited and grouped into a common update statement. Root nodes need to be treated separately, since they lack a common ancestor. This is a precise choice with respect to the different semantics of sets introduced earlier in this chapter. Here, the sets depend directly on the expansion strategy. Figure 2 shows that four self-standing statements are obtained from the expansion of `s0`. The order of these statements is indicated by means of the progressive numbers.

When switching to the second semantics, represented in Figure 3, one can notice that the number of involved groups increases. The elements are visited in a depth-first order and separate groups correspond to separate update sub-statements. With this second strategy, `s0` remains a single statement, albeit nested. The smaller updates are grouped together taking apart the PCDATA content (that is considered in a separate update). The major difference with the first strategy relies in the number of the smaller updates that are treated separately in nested sub-statements. For example, as represented in Figure 2, the second level elements are inserted with a separate statement. Conversely, in Figure 3, the second-level elements are part of the same statement but inserted at a different time (the attribute and one element first and then, after the sub-statements from 4 to 10, the last element).

4.3 Expansion Algorithm (Loosely Binding Semantics)

In the following we detail the expansion process for `INSERT` statements. `DELETE` statement expansion is analogous (and omitted for brevity), while `REPLACE` statements can be rewritten in terms of `DELETE` statements immediately followed by suitable `INSERT` statements. Instead, `RENAME` operations perform mere name changes and do not require further expansion.

Essentially, expansion proceeds with an intermixed depth-first and breadth-first visit of the involved fragments. In particular, all the operations relative to the insertion of XML nodes with a common father are enveloped in the same update statement. Among these nodes (that can be attributes, elements

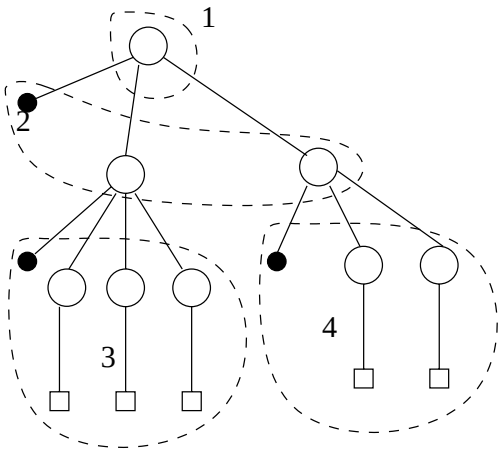

Fig. 2. Visit of an XML fragment (LBS)

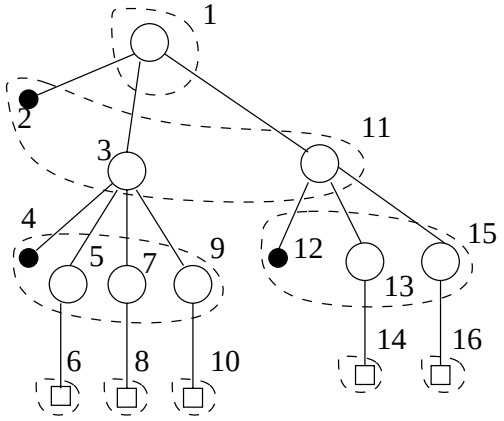

Fig. 3. Visit of an XML fragment (TBS)

or PCDATA content), attributes are inserted first, then elements and PCDATA content are inserted in their proper order. Elements are inserted as empty couples of tags if they contain a complex structure (and thus require further expansion), otherwise they solely have PCDATA content, and they are inserted together with such content.

Function *buildContentOfFirstUpdate* addresses the construction of the first expanded statement, targeted to the same variable as the user update operation (referenced in the external `UPDATE` clause). It takes the bulk statement ST as argument and outputs one update statement that inserts all the roots of the involved fragments. Thereafter, the algorithm starts from each complex root node and expands its complex sub-elements. This expansion is entrusted

to function `expandNode`, which is then recursively invoked on all complex sub-elements of these elements.

Algorithm 1 Bulk Statement Expansion. *Given an arbitrary bulk XQuery update statement ST, the algorithm returns UL, an ordered list of XQuery expanded update statements and directives.*

In a preprocessing phase, the algorithm computes the required data structures. Precisely, variable V represents the argument of the user update clause; variable S contains the bindings to the involved fragments; FClause and WClause are the `for` *and* `where` *clauses of ST; XFClause is an ad-hoc clause that is repeatedly used in the construction of the results; cur_path is a variable that is assigned to the current path, as soon as it is available by the visit of the fragments; variable name is used to name the generated statements and match them with their corresponding directives.*

This version of the algorithm takes care of the expansion of insert statements, and accepts for simplicity only flat user statements. Possible nested user updates can be treated via previous reduction to a list of flat statements.

```
begin
  V = getUpdateVariable(ST)
  S = bindInvolvedFragments(ST)
  FClause = getFORClause(ST)
  WClause = getWHEREClause(ST)
  XFClause = FClause + ", $curFrag IN " + V +
             "/*[empty(" + V + "/*[AFTER $curFrag])]"
  cur_path = "$curFrag"
  name = buildUniqueName(cur_path)
  UL = "EvalBefore(" + name + ") " + "Name:" + name + " "
  UL += FClause + WClause
  UL += "UPDATE " + V + "{ "
  UL += buildContentOfFirstUpdate(ST) + "} "
  for each fragment in S, consider again its root node N:
     if ( N is complex and requires further expansion )
       then UL += expandNode(N, cur_path, XFClause, WClause)
     UL += "EvalAfter(" + name + ") "
  return UL
end;

FUNCTION expandNode(Node N, String cur_path,
                    String XFClause, String WClause)
RETURNS OUT, an ordered list of update statements and directives
begin
  name = buildUniqueName(cur_path)
  OUT = "EvalBefore(" + name + ") " + "Name:" + name + " "
  if ( cur_path="$curFrag" )
    then OUT += XFClause + WClause +
                "UPDATE $curFrag { "
    else OUT += XFClause + ", $cur_node IN " + cur_path +
                WClause + "UPDATE $cur_node { "
  for each attribute A of N
```

```
   OUT += "INSERT new_attribute( "
   OUT += A.name + ", " + A.value + ") "
  for each subelement C of N
   if ( C is an XML-Element )
    then if ( C has only PCDATA content )
          then OUT += "INSERT " + buildTagWithPCDATA(C)
          else OUT += "INSERT " + buildEmptyTag(C)
    else OUT += "INSERT " + C.content
  OUT += "} "
  for each subelement C of N:
   if ( C is complex and requires further expansion )
    then cur_path += "/*[" + position of C + "]"
         OUT += expandNode(C, cur_path, XFClause, WClause)
  OUT += "EvalAfter(" + name + ") "
  return OUT
end;
```

Moreover, the algorithm interleaves the statements with special directives to the rule engine that enable the construction of conflict sets. In particular, the directive *EvalBefore* contains the name of the statement that it precedes, and the directive *EvalAfter* contains the name of the statement that it follows. The positions of these directives within the list of statements reflect the intrinsic semantics of the original statement. If we consider an `INSERT` operation, the *EvalBefore* directives precede each expanded statement, while *EvalAfter* directives solely follow the expanded statement of the leaf portions of the fragment. The *EvalBefore* directives that refer to the remaining (non-leaf) portions are postponed by recursion and follow the directive of the last leaf of the fragment, as shown below.

Example 2. We consider the expansion of the bulk statement `s0` (from section 4.1), that inserts an entire shelf in the document `Library.xml`. We need to expand `s0` into smaller self-standing update statements, in order to make the defined triggers sensitive to the insertion of the children of the `Shelf` element. The expansion algorithm outputs the following sequence:

```
EvalBefore(s1)

Name:s1
FOR $x IN document("Library.xml")/Library,
 $frag IN document("New.xml")/Shelves/Shelf[@nr="45"]
UPDATE $x
{  INSERT <Shelf/>  }

EvalBefore(s2)

Name:s2
FOR $x IN document("Library.xml")/Library,
 $frag IN document("New.xml")/Shelves/Shelf[@nr="45"],
```

```
 $curfragment IN $x/*[empty($x/*[AFTER $curfragment])]
UPDATE $curfragment
{  INSERT new_attribute(nr, "45")
   INSERT <Book/>
   INSERT <Book/>  }

EvalBefore(s3)

Name:s3
FOR $x IN document("Library.xml")/Library,
 $frag IN document("New.xml")/Shelves/Shelf[@nr="45"],
 $curfragment IN $x/*[empty($x/*[AFTER $curfragment])],
 $cur_node IN $curfragment/*[1]
UPDATE $cur_node
{  INSERT new_attribute(id, "AO97")
   INSERT <Author> J. Acute </Author>
   INSERT <Author> J. Obtuse </Author>
   INSERT <Title> Triangle Inequalities </Title>  }

EvalAfter(s3)

EvalBefore(s4)

Name:s4
FOR $x IN document("Library.xml")/Library,
 $frag IN document("New.xml")/Shelves/Shelf[@nr="45"],
 $curfragment IN $x/*[empty($x/*[AFTER $curfragment])],
 $cur_node IN $curfragment/*[2]
UPDATE $cur_node
{  INSERT new_attribute(id, "So98")
   INSERT <Author> A. Sound </Author>
   INSERT <Title> Automated Reasoning </Title>  }

EvalAfter(s4)

EvalAfter(s2)

EvalAfter(s1)
```

The order of evaluation of triggers corresponds to the intuitive order that could be expected by a user unaware of the decomposition process. This can be appreciated if one considers the hierarchy $s_1 < s_2 < \{s_3, s_4\}$ and the order of execution of before and after triggers. For instance, the "`AFTER INSERT`" triggers triggered by s_2 (referred to by the directive `EvalAfter(s2)`) see the side effects not only of statements s_3 and s_4, but also of all trigger executions caused by them.

4.4 Expansion Algorithm (Tightly Binding Semantics)

We synthetically present here the alternative expansion algorithm (that we have not adopted) describing a deeper fusion between the update statement and the triggers. This solution becomes eligible when the rule engine can be built on top of the query optimizer. The number of directives is greater than in the first expansion and is equal to the number of elements and of attributes of the fragment(see Figure 3). For these reasons, in the remainder of this chapter we will focus on the first kind of expansion strategy.

Algorithm 2 Bulk Statement Expansion Algorithm (2). *Given an arbitrary bulk XQuery update statement centered on node N, the algorithm returns EXP, a single XQuery expanded update statement, enriched with directives for rule evaluation.*

This version of the algorithm does not need a preprocessing phase, since the initial `FOR` *clause,* `WHERE` *clause and update variables are replicated only once in the final statement. The algorithm merely computes the variable X attached to the root node N, and calls expandNode, a recursive function on it. The function descends in the children of the root node.*

The underlying hypothesis assumes that there is one root for each update statement, that is exactly what happens in the adopted update language.

This version of the algorithm is also focused on the expansion of insert statements, and accepts for simplicity only flat statements. Possible nested updates can be treated via previous reduction to a list of flat statements.

```
begin
    X = getUpdateVariable(N);
    EXP = expandNode(N, 1, 0);
end;
FUNCTION expandNode(Node N, int counter, int position)
RETURNS ONE, a single expanded XQuery update statement
begin
    ONE = "EvalBefore(" + N.name + ") "
    ONE+= "INSERT" + buildEmptyTag(N)
    if isRoot(N)
        ONE+= "FOR $V1 IN " + X +
                "/*[empty(" + X + "/*[AFTER $V1]"+"UPDATE $V1{"
    else          ONE+= "FOR $V" + counter +" IN $V" + counter-1
                "+ /*[" + position + "] UPDATE $V"+counter+"{"
    for (int i=0; i < N.getAttributesNumber(); i++)

        ONE+= "EvalBefore(" + N.getAttribute(i).name + ") "
        ONE+= "INSERT new_attribute(" + N.getAttribute(i).name+
            ", " " + N.getAttribute(i).value + " " )";
        ONE+= "EvalAfter(" + N.getAttribute(i).name + ") "

    for (int i=0; i < N.getChildrenNumber(); i++)

        if (N.isElement())
```

```
            ONE+= expandNode(N.getChild(i), counter+1, i+1)
        else
            ONE+= " INSERT " " + N.getChild(i).content + " " ";

    ONE+= "} ";
    ONE+= "EvalAfter(" + N.name + ") "
    return ONE
end;
```

The output of the expansion is longer than the one provided by the first algorithm. However, the update statement is unbroken. For lack of space, we omit here the update expanded according to the second algorithm; it can be retrieved from the book Website.

4.5 Trade-off analysis between the LBS and TBS semantics

The loosely binding and the tightly binding semantics have been described through their respective algorithms for expansion of bulk updates into elementary updates. In this section, we compare them taking into account some aesthetic and functional criteria.

- In the loosely binding model, expanded updates are autonomous and separate. Triggering operations are computed for each update and involved triggers occur exactly when the update ends its execution. When operating with the tightly binding model, instead, the original update operation does not lose its integrity, and involved triggers need to be coalesced with the ongoing update execution.
- Separation of triggers and updates in the loosely binding semantics leads to a separation between the trigger engine and the query evaluator. Indeed, this permits to obtain a quick implementation of a trigger mechanism upon the existing querying technology. In the tightly binding semantics, this separation is not feasible since there is one monolithic update on which triggers are invoked. As shown in the previous point, in such a case triggers evaluation is not detached from the update evaluation; hence, the trigger engine and the query engine need to be unified.
- As with SQL:1999, the set of nodes affected by an update is computed before computing the update by the decomposer, i.e. according to the semantics of the original, user-level update primitive. This happens for both the semantics.
- In the loosely binding expansion, the obtained update statements are self-standing and contain heavily duplicated path expressions. This is due to the need to isolate the context of the operation. In the tighly binding semantics, replication is avoided since the update is unbroken and a single context specification is exploited. In practice, in the loosely binding semantics, this disadvantage is minor, because transparent caching mechanisms

should make such traversals very efficient; moreover, simple optimizations could be done, such as preserving pointers to already computed nodes from one statement execution to the next one.

- While in the LBS expansion yields a sequence of well-defined and compact statements, in the TBS semantics, expansion leads to a single statement that is rather lenghty and verbose. Moreover, while the LBS is more readable, the second expansion seems to be more natural and intuitive.
- The LBS decomposition of a bulk statement into a sequence of statements leads to exposing intermediate states - the ones left by a prefix of the decomposition sequence; therefore, the decomposition strategy affects the *semantics* of triggers, as this in turn is order-dependent. This is inevitable: order dependence characterizes even relational systems and is amplified by a hierarchical structure which can be processed in many ways. In the TBS decomposition, this problem is eliminated since the update evaluation is intertwined with the triggers invocation. In such a case, however, the technology of the rule engine has to be heavily modified in order to be combined with query execution.

	LBS semantics	TBS semantics
evaluation of triggers and updates	separate	integrated
separation between trigger and query engines	feasible	not feasible
computation of set of nodes	before decomposition	before decomposition
repetition of the same path traversals	high	low
decomposition features	intermediate states	no intermediate states

Table 1. Differences and similarities between LBS and TBS semantics

In Table 4.5, we illustrate the main differences between the two kinds of decompositions.

At a first glance, note that the TBS semantics is reasonably more intuitive than the LBS semantics. Indeed, the TBS semantics has few path expression replicas, and is even more compact. However, to adopt the TBS semantics, the update processing has to be intermixed with rule evaluation and technology of the rule engine is heavily affected.

Due to the results of this trade-off analysis, we chose to deploy the LBS approach since it can be easily supported "on top" of an existing XQuery optimizer, in the same way as a trigger engine can be easily supported on a relational storage system [25].

Therefore, in the remainder of this section, we will focus on the execution mode for LBS-expanded updates and XQuery triggers.

4.6 Description of trigger execution mode

Given the update decomposition algorithm, we can now define the trigger execution mode precisely, thus giving an operational semantics of the combined execution of updates and triggers. As observed, our query execution mode is inspired to the SQL:1999 execution mode, however adapted to the hierarchical nature of XML, and exploiting the expansion of statements.

Assume the XQuery engine is starting the execution of a generic bulk update statement S, which has been submitted by the user. The following algorithm defines this semantics operationally.

```
PROCEDURE EXECUTE_STATEMENT(Statement S)
1 Call EXPAND_STATEMENT(S) and store the
  returned structures (RF and SIL)
2 For each item Ii in SIL, if it is
2.1 an 'EvalBefore' instruction, call
    COMPUTE_BEFORE_CONFLICT_SET(Sn, RF),
    where Sn is the statement related to Ii
2.2 an 'EvalAfter' instruction, call
    COMPUTE_AFTER_CONFLICT_SET(Sn, RF),
    where Sn is the statement related to Ii
2.3 an update statement, execute Ii,
    updating the XML repository

PROCEDURE EXPAND_STATEMENT(Statement S)
RETURNS FragmentSequence RF,
        StatementInstructionList SIL
1 Retrieve RF, the set of fragments that are
  relevant for the execution of S
2 Expand S into SIL by visiting RF with the
  expansion algorithm
3 Return SIL and RF (NEW_RF and/or OLD_RF)

PROCEDURE COMPUTE_BEFORE_CONFLICT_SET
     (Statement Sn, FragmentSequence RF)
1 Compute BT, the set of eligible
  BEFORE triggers activated by Sn
2 Order all computed triggers according to
  their global ordering
3 For each trigger T in BT, PROCESS_TRIGGER(T)

PROCEDURE COMPUTE_AFTER_CONFLICT_SET
          (Statement Sn, FragmentSequence RF)
1 Compute AT, the set of eligible AFTER
  triggers activated by Sn
2 Order all computed triggers according
  to their global ordering
3 For each trigger T in AT, PROCESS_TRIGGER(T)
```

```
PROCEDURE PROCESS_TRIGGER(trigger T)
1 Calculate pointers corresponding to
  NEW_NODE(S) and OLD_NODE(S).
2 If any, evaluate the Let clause of T and bind
  the new variables.
3 Evaluate the condition C of T
4 If C evaluates to TRUE, EXECUTE_STATEMENT(A)
  (A being the action of T)
```

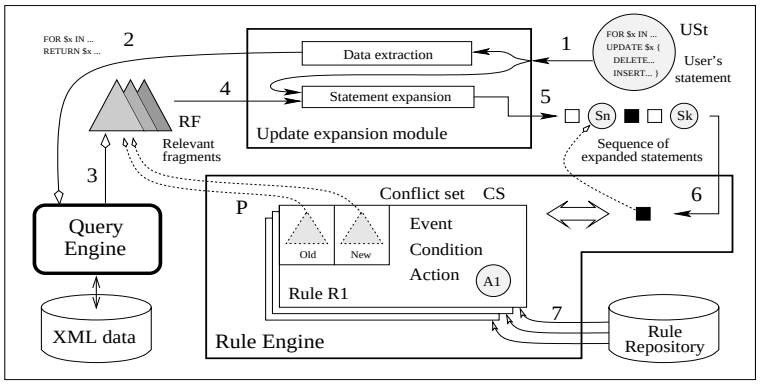

Fig. 4. System architecture

In the former algorithm `EXECUTE_STATEMENT` is invoked on `S`. The expansion algorithm 1 retrieves the set of relevant fragments (`RF`, involved by S), and produces a sequence of statements and directives (`SIL`). `SIL` is generated according to the mixed depth-breadth order visit of the fragments, and this is accomplished by calling procedure `EXPAND_STATEMENT`. The algorithm needs to communicate with the query engine in order to inspect XML data and to build RF, which constitutes a separate structure. More precisely, `REPLACE` operations will yield both `NEW_RF` and `OLD_RF` structures, deletions will produce only `OLD_RF` structures and insertions only `NEW_RF` structures. These structures are accessed by rules in order to bind transition variables, as explained below. The last task of `EXPAND_STATEMENT` is to return `SIL` and to rename `RF` as `NEW_RF` and/or `OLD_RF` (depending on the type of original statement `S`).

Once the `EXPAND_STATEMENT` has been completed, each item I_i in `SIL` is one of the following mutually exclusive cases. If the item is an *EvalBefore* directive, then the procedure `COMPUTE_BEFORE_CONFLICT_SET` is invoked; if it is an *EvalAfter* directive, then the procedure `COMPUTE_AFTER_CONFLICT_SET` is invoked; otherwise, it is an expanded statement, ready to be executed. Both `COMPUTE_BEFORE_CONFLICT_SET` and `COMPUTE_AFTER_CONFLICT_SET` receive as parameters the statement S_n referred by the directive and `RF`.

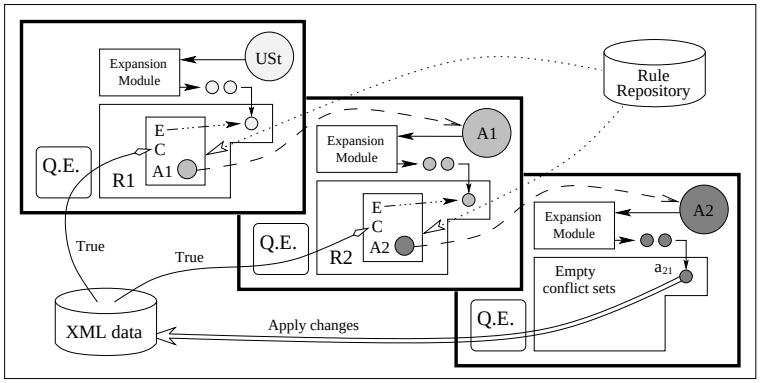

Fig. 5. Recursive rule execution

Procedures `COMPUTE_BEFORE_CONFLICT_SET` and `COMPUTE_AFTER_CONFLICT_SET` respectively compute `BT` and `AT`, the sets of triggered rules. This defines the conflict sets pertaining to statement S_n, which include both statement level and node level triggers, in priority order. For each trigger T in these conflict sets, `PROCESS_TRIGGER` is executed with T as parameter. Local pointers are calculated in order to bind `OLD_NODE(S)` and `NEW_NODE(S)` transition variables (they may have been used in a `LET` clause of T). When the condition evaluates to true, the action of T is executed by invoking `EXECUTE_STATEMENT` and recursively iterating the execution. When a new statement is executed, the current trigger execution context is suspended, and the entire process restarts.

During the processing of bulk statements, data is maintained in two places: the XML repository and the separate global fragments `NEW_RF` and/or `OLD_RF`, which are pointed to by local pointers (so we have a useful structure sharing that avoids redundance). The XML repository is persistent, while the trigger execution contexts and the global fragments are stored only until the execution of the bulk statement is completed.

4.7 System architecture

Figure 4 summarizes all the components of our proposed architecture. The update expansion module is responsible for the transformation of bulk statements (1). It consists of two layers that operate in a cascading sequence.

The data extraction layer first instantiates a query upon the query engine (2). This query is directly drawn from the user update; precisely, the *ForClause*, *LetClause* and *WhereClause* of the update are directly replicated in the query. Then, the query accesses the XML repositories to retrieve the relevant fragments `RF` (3) that are taken as inputs (4) by the statement expansion layer. This layer is responsible for the effective instantiation of the expansion algorithm over the user update. The result is the list of statements

and directives (5)(displayed as circles and boxes in the figure) that have been given as outputs by the algorithm.

Directives (6) are commands issued to the rule engine; they schedule the times at which the the conflict sets are calculated. The rule engine searches the rule repository (7) for rules addressing the nodes on which the referred statement S_n operates and whose triggering event matches the operation type. A rule might have more than one activation, since many similar nodes can be affected by S_n, and each rule instance is provided with the two pointers `OLD_NODE(S)` and `NEW_NODE(S)`, that represent the old/new fragment(s) (one of them might be set to null).

All triggered rules are collected in a conflict set (CS), where they are ordered with respect to their priority. Note that node level and statement level rules are mixed in CS, and that CS contains rules addressing nodes with different tagnames as well, since S_n affects all subelements and attributes of a single node. Only the actions of those rules with true condition are performed.

Statement A1 can be a bulk statement itself, and thus it must be processed by a recursive replication of the abstract machine described in figure 4. If we imagine a scenario in which the execution of a rule action causes another [cascading] rule to activate, we can represent the resulting stack of execution contexts like in figure 5. Here statement USt triggers rule R1, the action A1 of R1 is itself expanded, it triggers rule R2 and an update is eventually performed, since statement a_{21} causes no rule activation (the conflict sets related to a_{21} are empty).

5 Experiences of Applications of XML triggers

In this section, we focus on the application fields that Active XQuery is able to cover. In 5.1, we first provide an outline of these fields and their perspectives w.r.t. the current XML advanced technology. In appendix, we show a representative selection of triggers that are able to cover the various application fields.

5.1 Summarization of application fields

Table 5.1 summarize the application fields along two dimensions. The vertical dimension distinguishes between those special-purpose triggers that are implemented ad-hoc by the programmer and those that, conversely, are supposed to be produced automatically. The distinction is inspired to the classification provided in [8] for DBMS triggers. The horizontal dimension aims to characterize triggers as integrated in the XML repository, or provided as internal services exportable to the outside, or, alternatively, provided by third parties as external services.

Starting from the first column, "internal only XML repository services" are services not exportable to the outside. They basically are devoted to the

	Internal Only XML Repository Services	Internal XML Repository Services	External Services
Handcrafted	Metadata Management, Document Versioning, Internal Audit Trails	Alerters, Extenders, Audit Trails	Web Business Rules Supply Chain Manag. Transactional Services
Generated	XML Schema Identity and Domain Constraints, XML View Maintenance	Generic Integrity Constraints	B2B process integration (e.g. in WFMS)

Table 2. XML Trigger Applications

management of XML data internal to the repository. Handcrafted triggers of this species include those that contribute to build semantic knowledge over the data [7], document histories and traces of user behavior internal to the repository. Generated triggers are triggers that are supposed to be automatically generated by examining the document schemas or the document views. These are mainly triggers for maintaining XML Schema Identity and Domain Constraints, and triggers for keeping views of XML data consistent with the original documents.

The second column includes triggers that are programmed within the repository, and are instrumental in offering to the outside exportable services or in providing fruitful information. These includes handcrafted triggers such as prevalently notification mechanisms, extenders (separate structures, like flat files or specialized indexes that are kept aligned with the XML content), and audit trails. Generated triggers are instead generic integrity constraints that cannot be expressed in XML Schema, and need to be maintained separately. This kind of constraints are mainly semantic constraints that resemble the assertions of traditional databases. All these can be automatically generated once they have been encoded in a powerful constraint language.

The third column indicates triggers that come from external applications. Handcrafted triggers include business rules that include personalizers, classifiers, alerters centered on documents published on the Internet. Notification mechanisms fall again in this category as they are provided by external applications to deliver business information with rich content. Moreover, transactional services such as reservations or orders and supply chain management can be programmed by means of asynchronous business rules. Finally, generated triggers mostly conglobate B2B process integration, such as triggers to be used in workflow enactment. In such a case, triggers can be created automatically as long as there is a high level conceptual specification of the

[7] This is a very relevant application field. The Semantic Web Initiative is a W3C outstanding activity which is attracting the Web community. We feel that active rules may effectively serve as facilities to generate machine-understandable data and put them on the Web.

business process. External applications that follow these criteria (e.g. have an abstract model of rules) are eligible for the automatic generation of concrete triggers.

6 Summary and Future directions

This chapter has proposed Active XQuery and enriched the presentation with a few examples; additional examples can be retrieved by the Website of the book. We have also shown some relevant application fields, which happen to be different from those of traditional database language. In particular, we argue that active rules can be employed as rapid prototypes for web services and respond to the novel need of rendering the web as much dynamic as possible. We have presented two alternative semantics for XQuery triggers that enable the use of rules in a rich spectrum of applications.

We have highlighted the cases of *immediate* and *deferred* event detection. We have specialized the *immediate* semantics for Active XQuery, aware of the fact that the *deferred* semantics has an equivalent evaluation, once the event trace is known.

We have shown the *immediate* semantics for Active XQuery. We aimed at compatibility with SQL:1999, in spite of some difficulties due to "bulk" updates and to the hierarchical nature of XML; this goal has been achieved by defining a rule execution schema based on the preliminary expansion of user-provided update statements. We have provided two kinds of expansion, based on LBS and TBS semantics. We have shown that the main advantage produced by the former is the clean separation between the rule engine and the query optimizer, yielding a modular architecture in which the query optimizer can be merely plugged in.

Our study raises several optimization and research issues, briefly discussed below.

- **Schema-driven optimization.** The expansion of bulk statements can benefit of the availability of the documents schema, either expressed as a DTD or in terms of XML Schema. In particular, it is possible to avoid expanding the updates relative to those parts of a document that do not activate rules. Such an optimization is relevant within most XML repositories, as few of their element types correspond to distinctive real-world entities and are therefore targeted to triggers.
- **Internal optimizations.** Specialized indexing techniques can be devised in order to optimize the rule engine for repeated executions of the same query over different fragments, e.g. by adding data structures which point to "sibling" nodes. Such data structures can be set up by the expansion module, as it is aware of both the queries and the fragments. Other indexes may link the nodes of the XML repository to rules in the rule repository, in order to facilitate the extraction of the rules which are triggered by given operations.

- **Limitations of expressive power.** As with the SQL:1999 standard, the class of "legal" `BEFORE` triggers remains to be defined [20]. A simple solution can be the exclusion from their actions of any update operation or lookup into transition variables, but this in practice limits the expressive power of `BEFORE` triggers to error signaling. In many cases, however, such lookups and updates do not cause any inconsistency; therefore, the class of "legal" triggers - maybe relative to given XML schemas or documents - remains to be defined.
- **Definition of illegal executions.** Similarly, certain classes of executions are illegal, for instance when the update performed by a trigger affects the data which have caused the triggering, thus yielding to contradictory situations. Such illegal executions should be identified, and execution-time "traps" should be instrumented in order to detect them.
- **Compile-time trigger analysis.** Trigger analysis, applied to a given XML repository and rule set, could be used to detect anomalous behavior, such as the lack of termination or confluence [26]; conversely, trigger analysis could be used to "validate" triggers, thus excluding that a given rule set could lead to any illegal execution. These techniques are defined for relational triggers but need to be extended for XQuery triggers.

References

1. S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML Documents with Distribution and Replication. In *ACM SIGMOD*, San Diego, CA, June 2003.
2. Jelly: Executable XML. `http://jakarta.apache.org/commons/sandbox/jelly/`.
3. A. Bonifati, D. Braga, A. Campi, and S. Ceri. "Active XQuery". In *Proc. of ICDE 2002*, February 2002.
4. A. Bonifati, S. Ceri, and S. Paraboschi. "Active Rules for XML: A New Paradigm for E-Services". In *1st Workshop on E-Services (TES)*, Cairo, Egypt, September 2000.
5. A. Bonifati, S. Ceri, and S. Paraboschi. "Pushing Reactive Services to XML Repositories using Active Rules". In *Proc. of 10th World Wide Web Conf.*, pages 633–641, Hong Kong, China, May 2001.
6. A. Bonifati, S. Ceri, and S. Paraboschi. "Event Trace Independence of Active Behavior". In *Submitted for publication.*, July 2003.
7. T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds). "Extensible Markup Language (XML) 1.0 (2nd Edition)". W3C Recommendation, Oct. 2000. http://www.w3.org/TR/2000/REC-xml-20001006.
8. S. Ceri, R. J. Cochrane, and J. Widom. "Practical Applications of Triggers and Constraints: Success Stories and Lingering Issues". In *Proc. of 26th VLDB Conf.*, pages 254–262, Cairo, Egypt, September 2000.
9. S. Ceri and J. Widom. "Deriving Production Rules for Constraint Maintenance". In *Proc. of 16th VLDB Conf.*, pages 566–577, Brisbane, Australia, September 1990.

10. D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu (Eds). "XQuery 1.0: An XML Query Language". W3C Working Draft, Jun. 2001. http://www.w3.org/TR/2001/WD-xquery-20010607/.
11. D. Chamberlin et al.(Eds.). "XQuery: A Query Language for XML. W3C Working Draft, 15 February 2001.". http://www.w3.org/TR/2001/WD-xquery-20010215/.
12. J. Clark and S. J. DeRose (Eds). "XML Path Language (XPath) Version v1.0". W3C Recommendation, Apr. 1999. http://www.w3.org/TR/xpath.
13. G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *Proc. of ICDE*, San Jose, USA, February 2002.
14. "Delta XML Tool (Commercial)", 2001. http://www.deltaxml.com/.
15. "IBM Alphaworks XML Diff Tool", 1999. http://www.alphaworks.ibm.com/tech/xmldiffmerge.
16. Sun's JavaServer Pages. `http://java.sun.com/products/jsp/`.
17. Macromedia Dreamweaver. `http://www.macromedia.com/`.
18. N. Paton, editor. *"Active Rules in Database Systems"*. Springer-Verlag, 1999.
19. PHP. `http://www.php.net/`.
20. K. G. Kulkarni R. Cochrane and N. Mendonça Mattos. "Active Database Features in SQL3". In *"Active Rules in Database Systems"*, pages 197–219. Springer-Verlag, 1999.
21. "Simple Object Access Protocol (SOAP) 1.1 (W3C Note)", 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.
22. SQL:1999/Foundation. ISO-IEC 9075-2:1999, june 1999.
23. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S Weld. "Updating XML". In *ACM SIGMOD*, Santa Barbara, May 2001.
24. XQuery and XPath Full-text Use Cases. `http://www.w3.org/TR/xmlquery-full-text-use-cases/`.
25. J. Widom. "The Starburst Active Database Rule System". *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 8(4):583–595, 1996.
26. J. Widom and S. Ceri. *"Active database systems: Triggers and Rules for Advanced Database Processing"*. Morgan Kaufmann, 1996.
27. "Web Service Description Language (WSDL) 1.1 (W3C Note)", 2001. http://www.w3.org/TR/wsdl.
28. XML Query Use Cases. `http://www.w3.org/TR/xmlquery-use-cases/`.