An Event–Condition–Action Language for XML

James Bailey[1], George Papamarkos[2], Alexandra Poulovassilis[2], Peter T. Wood[2]

[1] Department of Computer Science, University of Melbourne, Melbourne, Australia
e-mail: jbailey@cs.mu.oz.au
[2] School of Computer Science and Information Systems, Birkbeck College, University of London, London WC1E 7HX
e-mail: gpapa05,ap,ptw@dcs.bbk.ac.uk

Summary. XML is now a widespread means of exchanging and storing information on the Web. Event–condition–action (ECA) rules are a natural candidate for the support of reactive functionality on XML repositories. This chapter discusses ECA rules in the context of XML data. We give a review of related work, and then define a language for specifying ECA rules on XML repositories. We specify the rule execution model of our language and describe a prototype implementation. We also discuss techniques for analysing the behaviour of sets of ECA rules, in particular for determining the triggering and activation relationships between pairs of rules. We conclude with a discussion of some directions for further research.

1 Introduction

XML is becoming a dominant standard for storing and exchanging information on the Web. With its increasing use in applications such as data warehousing, e-commerce and e-learning [1, 17, 18, 24, 28, 30, 35], there is a rapidly growing need for the support of reactive functionality on XML repositories. *Event–condition–action* (ECA) rules are a natural candidate for this.

ECA rules automatically perform actions in response to events provided that stated conditions hold. They are used in conventional data warehouses for incremental maintenance of materialised views, validation and cleansing of data, and maintaining audit trails. By analogy, ECA rules could also be used as an integrating technology for providing this kind of functionality on XML repositories. Further potential uses include checking key and other constraints on XML documents, and performing automatic repairs when violations of constraints are detected. In a 'push' type environment, they can be used for automatically broadcasting information to subscribers as the contents of relevant documents change. They can also be employed as a flexible means of maintaining statistics about document and Web site usage and behaviour.

There are two main advantages in using ECA rules to support such functionality as opposed to implementing it directly using a programming language such as Java. First, ECA rules allow an application's reactive functionality to be defined and managed within a single rule base rather than being encoded in diverse programs. This enhances the modularity and maintainability of such applications. Second, ECA rules have a high-level, declarative syntax and are thus amenable to powerful analysis and optimisation techniques, which cannot be applied if the same functionality is expressed directly in programming language code.

An alternative way to implement the functionality described above might be to use XSLT to transform source XML documents. However, XSLT would have to process an entire document after any update to it in order to produce a new document, whereas we are concerned with the detection and subsequent processing of updates of much finer granularity. Also, using ECA rules allows direct update of a document, wheareas XSLT requires a new result tree to be generated by applying transformations to the source document.

ECA rules have been used in many settings, including active databases [37, 33], workflow management, network management, personalisation and publish/subscribe technology [4, 17, 18, 20, 34], and specifying and implementing business processes [3, 19, 28]. In this chapter, we study ECA rules in the context of XML data. We begin with a review of related work. We then define a language for specifying ECA rules on XML repositories, illustrating the language by means of some examples. We specify the rule execution model of the language and describe a prototype implementation. We also discuss techniques for analysing the behaviour of sets of ECA rules defined in our language, in particular for determining the triggering and activation relationships between pairs of rules. We conclude with a discussion of directions for further research.

2 Event–Condition–Action Rules

An ECA rule has the general syntax

`on` *event* `if` *condition* `do` *actions*

The event part describes a situation of interest and dictates *when* the rule should be triggered. Events can be broadly classified into two categories: *primitive events* and *composite events*. Primitive events are atomic, detectable occurrences such as database updates (e.g. '`on` *insert into relation R*') or the reaching of a particular time (e.g. '`on` *13th March 2003 at 15:00*'). Composite events are combinations of primitive events, specified using an *event algebra* [22, 26]. Common operators in event algebras include (i) disjunction: event $e_1 \vee e_2$ occurs if either event e_1 occurs or event e_2 occurs, (ii) sequence: event

$e_1; e_2$ occurs if e_2 occurs, having been preceded by e_1, and (iii) conjunction: event $e_1 \wedge e_2$ occurs when both e_1 and e_2 have occurred in any order.

However, most implementations of ECA systems do not support an event algebra as rich as this. Rather, they settle for being able to detect just an appropriate set of primitive events, with no support of event operators. While this limits the range of situations that can be reacted to, rule execution can be more easily optimised and analysed.

Events can have associated with them parameters which provide extra information about the event occurrence. For example, in active databases, these parameters are known as *deltas* and may be referenced by the condition and action parts of the ECA rule. For each event E detectable by the database there are two deltas: *has_occurred_E* and *change_E*. The former is nonempty if event E occurred during the execution of the last action, and it contains information about the occurrences of event E, e.g. the time at which they occurred and which transaction caused them to occur. The delta *change_E* contains information about the changes that occurrences of event E made to the database during the execution of the last action.

For example, in a typical active relational database there may be for each user-defined relation R a set of six delta relations:

- *has_occurred_insertion_R*, which would be nonempty if one or more INSERT statements on relation R occurred during the execution of the last action;
- *change_insertion_R*, which would contain the set of new tuples inserted into relation R during the execution of the last action;
- *has_occurred_deletion_R*, would be nonempty if one or more DELETE statements on relation R occurred during the execution of the last action;
- *change_deletion_R*, would contain the set of tuples deleted from R during the execution of the last action;
- *has_occurred_update_R*, would be nonempty if one or more UPDATE statements on relation R occurred during the execution of the last action;
- *change_update_R*, would contain a set of pairs (*old_tuple*,*new_tuple*) for each tuple of R which was updated during the execution of the last action

The event part of an ECA rule is either *has_occurred_E* or *change_E* for some event E. The identifiers *has_occurred_E* and *change_E* may also occur within the rule's condition and action parts.

A rule is said to be *triggered* if its event part is nonempty. Allowing either *has_occurred_E* or *change_E* to appear as rule events means that both 'syntactic' and 'semantic' triggering of rules can be supported. Syntactic triggering happens if the rule's event part is *has_occurred_E* and instances of event E occur. Semantic triggering happens if the rule's event part is *change_E* and instances of event E occur and make changes to the database.

The condition part of an ECA rule determines if the database is in particular state. It is a query over the database and its environment, and its semantics are the same as that used for the database query language, e.g.

SQL. The condition may also refer to the state before the execution of the event and the state created after the execution, by making use of the deltas.

The action part of a rule describes the logic to be performed if the condition evaluates to True. It is usually a sequence of modifications applied to the database, expressed using the same syntax as that used by updates within a transaction.

More details on the foundations of ECA rules in active databases and descriptions of a range of implemented active database prototypes can be found in [33, 37].

2.1 Rule Execution Model

The rule execution model is a specification of the run-time behaviour of the system. In particular, it specifies when the various components of a rule are executed with respect to one another, and what happens when multiple rules are triggered simultaneously.

This first aspect is traditionally handled by the use of *coupling modes* [21]. A coupling mode specifies the timing activation of one part of an ECA rule with respect to another. Possible coupling modes for the condition part with respect to the event part are:

- Immediate: The condition is evaluated immediately the event is detected as having occurred within the current transaction.
- Deferred: The condition is evaluated within the same transaction, but after the last operation in the transaction and just before the transaction commits.
- Decoupled: The condition is evaluated within a separate, child transaction.

Possible coupling modes for the actions part with respect to the condition part are similar:

- Immediate: The action is executed immediately after the condition has been evaluated (if the condition is found to be True).
- Deferred: The action is performed within the same transaction, but after the last operation in the transaction and just before the transaction commits.
- Decoupled: The action is performed within a separate, child transaction.

Different types of coupling modes may be more or less suitable for certain categories of rules. For example, decoupled execution can help response time since the length of transactions does not grow too large because of rule execution and hence potentially more concurrency is available. Decoupled execution can also be useful in situations where the parent transaction aborts, yet rule execution is nevertheless desired in the child transaction, e.g. updating an access log regardless of whether or not authorisation is granted. Maintaining views is typically done immediately to ensure freshness, and either deferred or immediate coupling can be used for checking integrity constraints (immediate for constraints that should never be violated, and deferred for constraints that need only be satisfied when the database is in a stable state).

The second aspect of rule execution is the policy employed for determining which rule to execute next, given that several rules have been previously triggered and are awaiting execution. The time of triggering is an important factor here and thus maintaining rules in a data structure which reflects this timing information is natural, e.g. a first-in-first-out or a last-in-first-out list. For rules which were triggered at precisely the same time by the same event occurrence, further information such as priorities can be used for tie-breaking; each rule is assigned a unique priority and rules with higher priority are executed earlier.

2.2 The SQL3 Standard

The SQL3 standard specifies a syntax and execution model for ECA rules, or *triggers*, in relational databases [31]. Rule event parts may be triggered by update, insert or delete operations on the database. Triggers are of two kinds: BEFORE triggers and AFTER triggers. The former conceptually execute the condition and action before the triggering event is executed. The latter execute the condition and action after the triggering event is executed, using an immediate coupling mode between both event and condition, and between condition and action.

Conditions are evaluated on the database state that the action is executed on, and multiply triggered rules are handled using a last-in-first-out list. Each rule is assigned a unique priority. Only syntactic triggering is supported, i.e. the event parts of triggers have the semantics of the *has_occurred* deltas we described above.

Another important aspect is that of rule granularity, and two types of granularity are supported, *row-level* and *statement-level*. When a statement-level rule is triggered by some event E and is then scheduled for execution, one copy of its action part is placed on the list of pending rules. When a row-level rule is triggered by some event E, one copy of its actions part is placed on the pending list for each member of *change_E* for which the rule's condition evaluates to True. Hence, a single event can give rise to zero, one, or many copies of a triggered rule's actions for row-level rules.

2.3 ECA Rules for Object-Oriented Databases

Because of the richness of the object-oriented data model, ECA rules for object-oriented databases often contain additional features compared with ECA rules for relational databases. The principal difference is the availability of a richer set of primitive event types, for example, events which are triggered on the invocation of methods or on the creation of objects. Here again, in the same way as for relational databases, deltas can be used to define event contexts. So in a typical active object-oriented database there may be classes *has_occurred_M* and *change_M* for each method M whose invocation is detectable as an event by the database. The delta *has_occurred_M* would be nonempty if method M was invoked during the execution of the last action. The delta *change_M* would contain information about changes made to user-defined database objects by invocations of method M during the last action. Another important difference in active object-oriented databases stems from the ability to specify rules as objects. Relationships between rules can then be captured, using properties such as generalisation and specialisation between rule classes.

2.4 Analysing Rule Behaviour

When multiple ECA rules are defined within a system, their interactions can be difficult to predict, since the execution of one rule may cause an event which triggers another rule or set of rules. These rules may in turn trigger further rules, and there is indeed the potential for an infinite cascade of rule firings to occur.

Analysing the behaviour of ECA rules is a well-studied topic in active databases and a number of analysis techniques have been proposed, e.g. [5, 6, 8, 11, 12, 13, 14, 19, 25, 29]. Two key properties of a set of ECA rules are the *triggering* [5] and *activation* [13] relationships between pairs of rules, since this information can be used to analyse properties such as *termination* of the ECA rule set, or *reachability* of specific rules. The triggering and activation relationships between pairs of rules are defined as follows:

- A rule r_i *may trigger* a rule r_j if execution of the action of r_i may generate an event which triggers r_j.
- A rule r_i *may activate* another rule r_j if r_j's condition may be changed from False to True after the execution of r_i's action.
- A rule r_i *may activate* itself if its condition may be True after the execution of its action.

The *triggering graph* [5] represents each rule as a vertex, and there is a directed arc from a vertex r_i to a vertex r_j if r_i *may trigger* r_j. Acyclicity of the triggering graph implies definite termination of rule execution. Triggering graphs can also be used for deriving rule reachability information. The *activation graph* [13] also represents rules as vertices. In this case there is a directed arc from a vertex r_i to a vertex r_j if r_i *may activate* r_j. Acyclicity of this graph also implies definite termination of rule execution.

Triggering and activation graphs were combined in [11] in a method called *rule reduction*, which gives more precise results than either of the triggering or activation graphs alone. With this method, any vertex that does not have both an incoming triggering and activation arc can be removed from the graph, along with its outgoing arcs. This removal of vertices is repeated until there are no such vertices. If the procedure results in all the vertices being removed, then the rule set is definitely terminating.

More recently, we proposed using *abstract interpretation* to analyse ECA rules [6, 8]. With this approach, the ECA rules are 'executed' on an abstract database representing a number of real databases. Unlike the graph-based approaches, this technique tracks how the triggering and activation relationships between rules *evolve* during rule execution.

Determining triggering and activation relationships between ECA rules is more complex for semistructured data such as XML than for structured databases, because determining the effects of rule actions is not simply a matter of matching up the names of updated database objects with the event and condition parts of ECA rules. Instead, the associations between actions and events/conditions are more implicit, and more sophisticated semantic comparisons between sets of path expressions are required. In Sect. 4 we discuss techniques for determining the triggering and activation relationships for our XML ECA rules.

2.5 ECA Rules for XML

The semistructured nature of XML data gives rise to new issues affecting the use of ECA rules. These issues are principally linked to choice of appropriate language syntax and execution model:

- *Event granularity*: In the relational model, the granularity of data manipulation events is straightforward, since insert, delete or update events occur when a relation is inserted into, deleted from or updated, respectively. With XML, this kind of strong typing of events no longer exists. Specifying the granularity of where data has been inserted or deleted within an XML document becomes more complex and path expressions that identify locations within the document now become necessary.
- *Action granularity*: Again in the relational model, the effect of data manipulation actions is straightforward, since an insert, delete or update action can only affect tuples in a single relation. With XML, actions now manipulate entire subdocuments, and the insertion or deletion of subdocuments can trigger a set of different events. Thus, analysis of which events are triggered by an action can no longer be based on syntax alone. Also, the choice of an appropriate action language for XML is not obvious, since there is as yet no standard for an XML update language.

Compared to rules for relational databases, ECA rules for XML data are more difficult to analyse, due to the richer types of events and actions. However, rules for XML have arguably less analysis complexity than rules for object-oriented data. This stems from the fact that object-oriented databases may permit arbitrary method calls to trigger events, and determining triggering relationships between rules may therefore be as difficult as analysing a program written in a language such as C++ or Java. ECA rules for XML, in contrast, can be based on declarative languages such as XQuery and XPath, and so are more amenable to analysis, particularly with the use of natural syntactic restrictions, as in the language we describe in this chapter.

In recent work [9, 10], we specified a language for defining ECA rules on XML data, based on the XPath and XQuery standards. We also developed techniques for analysing the triggering and activation relationships between such rules. This language and the analysis techniques are the subject of the rest of this chapter. A number of other ECA rule languages for XML have also been proposed, although none of this work has focused on analysing the behaviour of the ECA rules:

Bonifati et al. [17] discuss extending XML repositories with ECA rules in order to support e-services. Active extensions to the XSLT [40] and Lorel [2] languages are proposed that handle insertion, deletion and update events on XML documents. Bonifati at al. [18] discuss a more specific application of the approach to push technology, where rule actions are methods that cannot update the repository, and hence cannot trigger other rules.

Bonifati et al. [16] also define an active rule language for XML, which is discussed in detail in the chapter by Bonifati and Paraboschi in this book. The rule syntax of this language is similar to ours, and is based on the syntax of triggers in SQL3. The rule execution model is rather different from ours though. Generally speaking, insertions and deletions of XML data may involve document fragments of unbounded size. Bonifati et al. [16] adopt an execution model whereby each top-level update is decomposed into a sequence of smaller updates (depending on the contents of the fragment being inserted/deleted) and then triggering of rules is interleaved with the execution of these smaller updates. In contrast, in our language we treat each top-level update as atomic and rules are triggered only after completion of the top-level update. In general, these semantics may produce different results for the same top-level update, and it is a question of future research to determine their respective performance trade-offs and suitability in different application situations.

ARML [23] provides an XML-based rule description for rule sharing among different heterogeneous ECA rule processing systems. In contrast to our language, conditions and actions are defined abstractly as XML-RPC methods, which are later matched with system-specific methods.

GRML [36] is a multipurpose rule markup language for defining integrity, derivation and ECA rules. GRML uses an abstract syntax based on RuleML, leaving the mapping to a real language for each underlying system implementation. GRML aims to provide semantics for defining access over distributed, heterogeneous data sources for rule evaluation and allows the user to declare most of the semantics necessary for processing a rule, and to evaluate events and conditions coming from heterogeneous data sources.

Other related work is [35], which proposes extensions to the XQuery language [41] to incorporate update operations. We refer the reader to that paper for a review of the provision of update facilities in other XML manipulation languages. The update operations proposed are more expressive than the actions supported by our ECA rule language since they also include renaming and replacement operations, and specification of updates at multiple levels of documents. Triggers are discussed in [35] as an implementation mechanism for deletion operations on the underlying relational store of the XML. However, provision of ECA rules at the 'logical' XML level is not considered.

3 Our ECA Rule Language for XML

An XML repository consists of a set of XML documents. In our language, ECA rules on XML repositories take the following form:

$$\texttt{on}\ \mathit{event}\ \texttt{if}\ \mathit{condition}\ \texttt{do}\ \mathit{actions}$$

We use the XPath [39] and XQuery [41] languages to specify the event, condition and actions parts of rules. XPath is used for selecting and matching fragments of XML documents within the event and condition parts. XQuery is used within insertion actions, where there is a need to be able to construct new XML fragments.

The *event* part of an ECA rule is an expression of the form

$$\texttt{INSERT}\ e$$

or

$$\texttt{DELETE}\ e$$

where e is a *simple XPath expression* (see Sect. 3.1) which evaluates to a set of nodes. The rule is *triggered* if this set of nodes includes any node in a new XML fragment, in the case of an insertion, or in a deleted fragment, in the case of a deletion. The system-defined variable `$delta` is available for use within the condition and actions parts of the rule, and its set of instantiations is the set of new or deleted nodes returned by e. The *condition* part of a rule is either the constant `TRUE`, or one or more simple XPath expressions connected by the boolean connectives `and`, `or`, `not`. The *actions* part of a rule is a sequence of one or more actions:

$$action_1;\ \ldots;\ action_n$$

where each $action_i$ is an expression of one of the following three forms:

$$\texttt{INSERT}\ r\ \texttt{BELOW}\ e\ \texttt{BEFORE}\ q$$
$$\texttt{INSERT}\ r\ \texttt{BELOW}\ e\ \texttt{AFTER}\ q$$
$$\texttt{DELETE}\ e$$

Here, r is a *simple XQuery expression*, e is a *simple XPath expression* and q is either the constant `TRUE` or an *XPath qualifier* (see Sect. 3.1 for definitions of the italicised terms).

In an `INSERT` action, the expression e specifies the set of nodes N immediately below which new XML fragment(s) will be inserted. These fragments are specified by the expression r. If e or r references the `$delta` variable, then one XML fragment is constructed for each instantiation of `$delta` for which the rule's condition evaluates to True. If neither e nor r references `$delta`, then a single fragment is constructed. The expression q is an XPath qualifier that is evaluated on each child of each node $n \in N$. For insertions of the form `AFTER` q, the new fragment(s) are inserted after the last sibling for which q

is True, while for insertions of the form `BEFORE` q, the new fragment(s) are inserted before the first sibling for which q is True. The order in which new fragments are inserted is nondeterministic.

In a `DELETE` action, the expression e specifies the set of nodes which will be deleted (together with their descendant nodes). Again, e may reference the `$delta` variable.

Example 1. Consider an XML repository containing an XML document `s.xml` that contains information about share prices on a stock exchange. We show below some of the information held for a particular share in the document, share `XYZ`. Under `day-info` the share price is recorded periodically for the specified date. The highest and lowest prices for each day and each month are also recorded, under `day-info` and `month-info`, respectively.

```
<shares>
  ...
  <share name="XYZ">
    ...
    <day-info day="03" month="03">
      <prices>
        <price time="09:00">123.25</price>
        <price time="09:05">123.50</price>
        <price time="09:10">123.00</price>
      </prices>
      <high>123.50</high>
      <low>123.00</low>
    </day-info>
    ...
    <month-info month="03">
      <high>133.75</high>
      <low>111.25</low>
    </month-info>
  </share>
  <share name="ABC">
  ...
  </share>
  ...
</shares>
```

Suppose that this document is updated in response to external events received from a share price information service. In particular, an insertion event will arrive periodically with the current price for share `XYZ`. For example, such an insertion event Ev might be the following update, which inserts the new share price of `123.75` after the last share price currently recorded:

```
INSERT <price time="09:15">123.75</price>
   BELOW document('s.xml')/shares/share[@name="XYZ"]/
         day-info[@day="03"][@month="03"]/prices
   AFTER TRUE
```

The following ECA rule r_1 checks whether the daily high needs to be updated in response to a new price insertion in some share:

```
on INSERT document('s.xml')/shares/share/day-info/
          prices/price
if $delta > $delta/../../high
do DELETE $delta/../../high;
   INSERT <high>$delta/text()</high>
          BELOW $delta/../.. AFTER prices
```

Here, the `$delta` variable is bound to the newly inserted `price` node detected by the event part of the rule. The rule's condition checks that the value of this `price` node is greater than the value of the `high` node under the same `day-info` node. The action then deletes the existing `high` node and inserts a `high` node whose value is that of the newly inserted `price`. The insertion event Ev above would trigger this rule r_1, which would then update the daily high of share `XYZ` to `123.75`.

The following ECA rule, r_2, similarly checks whether the monthly high price for a share needs to be updated in response to an insertion of a new daily high price:

```
on INSERT document('s.xml')/shares/share/day-info/
          high
if $delta > $delta/../../month-info[@month=$delta/../
            @month]/high
do DELETE $delta/../../month-info/high;
   INSERT $delta
          BELOW $delta/../../month-info
                [@month=$delta/../@month]
          BEFORE TRUE
```

In the `INSERT` action of this rule, a copy of the `high` node whose insertion triggered the rule is inserted as the first child of the corresponding `month-info` node.

The insertion event Ev above would trigger rule r_1, and the second action of r_1 would in turn trigger rule r_2. However, the condition of r_2 would then evaluate to False and so its action would not be executed. Similar ECA rules could be used to update the daily and monthly low prices, and for undertaking many other potentially useful tasks.

3.1 Simple XPath and XQuery Expressions

The XPath and XQuery expressions appearing in our ECA rules are restrictions of the full XPath and XQuery languages, to what we term *simple* XPath and XQuery expressions. These represent useful and reasonably expressive fragments which have the advantage of also being amenable to analysis, a topic we discuss in Sect. 4.

The XPath fragment we use disallows a number of features of the full XPath language, most notably the use of any axis other than the child, parent, self or descendant-or-self axes and the use of all functions other than `document()` and `text()`. Thus, the syntax of a *simple XPath expression* e is given by the following grammar, where s denotes a string and n denotes an element or attribute name:

$$
\begin{array}{rl}
e ::= & \text{`}\texttt{document(}\text{'}\ s\ \text{`}\texttt{)}\text{'}\ ((\ \text{`}\texttt{/}\text{'}\ \mid\ \text{`}\texttt{//}\text{'}\ )\ p)\ \mid \\
& \text{`}\texttt{\$delta}\text{'}\ (\text{`}\texttt{[}\text{'}\ q\ \text{`}\texttt{]}\text{'})^{*}\ ((\ \text{`}\texttt{/}\text{'}\ \mid\ \text{`}\texttt{//}\text{'}\ )\ p)? \\
p ::= & p\ \text{`}\texttt{/}\text{'}\ p\ \mid\ p\ \text{`}\texttt{//}\text{'}\ p\ \mid\ p\ \text{`}\texttt{[}\text{'}\ q\ \text{`}\texttt{]}\text{'}\ \mid\ n\ \mid\ \text{`}\texttt{*}\text{'}\ \mid \\
& \text{`}\texttt{@}\text{'}n\ \mid\ \text{`}\texttt{@*}\text{'}\ \mid\ \text{`}\texttt{.}\text{'}\ \mid\ \text{`}\texttt{..}\text{'}\ \mid\ \text{`}\texttt{text()}\text{'} \\
q ::= & q\ \text{`}\texttt{and}\text{'}\ q\ \mid\ q\ \text{`}\texttt{or}\text{'}\ q\ \mid\ e\ \mid\ p\ \mid\ (p \mid e \mid s)\ o\ (p \mid e \mid s) \\
o ::= & \text{`}\texttt{=}\text{'}\ \mid\ \text{`}\texttt{!=}\text{'}\ \mid\ \text{`}\texttt{<=}\text{'}\ \mid\ \text{`}\texttt{<}\text{'}\ \mid\ \text{`}\texttt{>=}\text{'}\ \mid\ \text{`}\texttt{>}\text{'}
\end{array}
$$

Expressions enclosed in '`[`' and '`]`' in an XPath expression are called *qualifiers*. So a simple XPath expression starts by establishing a context, either by a call to the `document` function followed by a path expression p, or by a reference to the variable `$delta` (the only variable allowed) followed by optional qualifiers q and an optional path expression p. Note that a qualifier q can comprise a simple XPath expression e.

The XQuery fragment we adopt disallows the use of full FLWR expressions (involving the keywords 'for', 'let', 'where' and 'return'), essentially permitting only the 'return' part of such an expression [41]. The syntax of a *simple XQuery expression* r is given by the following grammar:

$$
\begin{array}{rl}
r ::= & e\ \mid\ c \\
c ::= & \text{`}\texttt{<}\text{'}\ n\ a\ (\text{`}\texttt{/>}\text{'}\ \mid\ (\text{`}\texttt{>}\text{'}\ t*\ \text{`}\texttt{</}\text{'}\ n\ \text{`}\texttt{>}\text{'})) \\
a ::= & (n\ \text{`}\texttt{= "}\text{'}\ (s\ \mid\ e')\ \text{`}\texttt{"}\text{'}\ a)? \\
t ::= & s\ \mid\ c\ \mid\ e' \\
e' ::= & \text{`}\{\text{'}\ e\ \text{`}\}\text{'}
\end{array}
$$

Thus, an XQuery expression r is either a simple XPath expression e (as defined above) or an element constructor c. An element constructor is either an empty element or an element with a sequence of element contents t. In each case, the element can have a list of attributes a. An attribute list a can be empty or is a name equated to an attribute value followed by an attribute list. An attribute value is either a string s or an *enclosed expression* e'. Element contents t is one of a string, an element constructor or an enclosed expression. An enclosed expression e' is an XPath expression e enclosed in braces. The braces indicate that e should be evaluated and the result inserted at the position of e in the element constructor or attribute value.

3.2 Rule Execution Model

We now describe the rule execution model of our language. The input to the execution is a *schedule* `s` and an *XML repository* `db`. The schedule consists of a list of pairs $(action_{i,j}, delta_i)$, where $action_{i,j}$ is the jth action within the actions part of rule r_i and $delta_i$ is a set of instantiations for the `$delta` variable of rule r_i for which r_i's condition evaluated to True. Rules whose event parts reference the same XML document can potentially be triggered by the same update event on that document. To disambiguate the effect of such rules, we require that all rules whose event parts are insertions on the same document are totally ordered, as are all rules whose event parts are deletions on the same document. The relative priorities of such rules are specified by the user when defining a new rule.

The schedule which initiates rule execution consists of an action and a set of instantiations for the `$delta` variable upon which this action is to be applied, i.e. the initial schedule is a singleton of the form `[(action,delta)]`. The following pseudocode expresses how this update request is handled:

```
while s != [] do {
  (a,delta)    := head (s);
  s            := tail (s);
  (changes,db) := updateDB (db,a,delta);
  for each rule r_i in order of increasing priority do {
    if changes[i] != {} then {
      (value,delta) := evalCondition(i,changes[i],db);
      if value = True then
      for j := noOfActions[i] downto 1
        s := (action[i,j],delta):s
    }
  }
}
```

In the above pseudocode, the function `head` returns the first element of a list and the function `tail` returns a list minus its first element.

The function `updateDB` executes the action `a` that was at the head of the schedule. If `a` does not reference the `$delta` variable, this update is performed just once on the repository `db`. If `a` does reference the `$delta` variable, a *set* of updates is generated by substituting occurrences of `$delta` within `a` by each member of `delta`. Thus if n is the cardinality of `delta`, n updates will be generated.[3] These updates are then performed in an arbitrary order on the repository.

`updateDB` returns a pair `(changes,db)`, where `db` is the new repository resulting from the update and `changes` is an array such that `changes[i]` is

[3] n is guaranteed to be finite due to the syntax of our update language, which does not allow infinite new XML fragments to be created, and the fact that there are a finite number of ECA rules.

the set of newly inserted or newly deleted nodes corresponding to the event part of rule `r_i`. In particular, if `a` is an insertion then for each `r_i` which may be triggered by `a`, the event part of `r_i` is evaluated on the repository after `a` is executed, and `changes[i]` is the intersection of this result and the new nodes inserted by `a`. If `a` is a deletion then for each `r_i` which may be triggered by `a`, the event part of `r_i` is evaluated on the repository before `a` is executed, and `changes[i]` is the intersection of this result and the nodes that are subsequently deleted by `a`.[4]

The function `evalCondition` evaluates rule `r_i`'s condition, and there are two possible cases:

1. If the `$delta` variable occurs in the condition, then the condition is evaluated once for each member of `changes[i]`, and the subset of `changes[i]` for which it evaluates to True is determined. The variable `delta` is set to this subset. If `delta` is nonempty, then the variable `value` is set to `True`; otherwise it is set to `False`.
2. If the `$delta` variable does not occur in the condition, then the condition is evaluated just once and the variable `value` is set to the result. The variable `delta` is set to `changes[i]`.

`noOfActions[i]` is the number of actions in the actions part of rule `r_i` and `actions[i,j]` is the `j`th action of `r_i`. The loop `for j:=noOfActions[i] downto 1 do ...` ensures that the actions of a given rule are placed in the right order onto the schedule. Each such action `action[i,j]` is paired with that rule's `delta` and prefixed to the current schedule by the statement `s := (action[i,j],delta):s`.

Rules are considered in increasing order of their priority in the outer `for` loop. Thus the actions of higher-priority rules that have fired will be placed onto the schedule in front of the actions of lower-priority rules.

The execution proceeds in this manner until the schedule becomes empty. Nontermination of rule execution is a possibility and thus development of static rule analysis techniques is important to aid the design of 'well-behaved' rules. We discuss such techniques in Sect. 4.

There are a number of observations we can make regarding the above rule execution model:

- Triggering is semantic, not syntactic, since a rule `r_i` is triggered only if `changes[i]` is not empty.
- The event/condition coupling mode and the condition/action coupling modes are both `Immediate`, since conditions are evaluated immediately

[4] We will see in Sect. 4 how the set of rules that may be triggered by an action can be determined. It would also be correct to evaluate the event parts of all rules since for those that cannot be triggered by the action, `changes[i]` will necessarily be empty. Thus, limiting the evaluation to the set of rules that may be triggered is an optimisation.

after an event becomes true, and the actions of rules that have fired as a result of the current set of updates are placed at the head of the schedule.
- Rule conditions are evaluated against the repository state in which the rule was triggered (unlike in SQL3, where conditions are evaluated against the database state that the action will be executed on). It is possible to evaluate conditions against the database state that the action will be executed on by adding the conditions as additional qualifiers to the XPath expression e that is part of `INSERT` and `DELETE` actions.
- Both *document-level* and *instance-level* triggering are supported in our ECA rule language, depending on the occurrence of `$delta` in the condition and action parts of a rule:
  - If there is no occurrence of `$delta` in the condition or the action, the action is executed once if the condition is True—this is document-level triggering.
  - If `$delta` occurs in the action (and possibly in the condition), the action is executed once for each possible instantiation of `$delta` for which the condition is True—this is instance-level triggering.

3.3 A Prototype Implementation

As a proof of concept, we have developed a prototype system that implements our language and the execution model described above. For this first prototype we have used flat files and have exploited the functionality provided by the W3C DOM standard [43] for interacting with them. The architecture of our system is illustrated in Fig. 1.

The *Parser* parses and checks the syntactic validity of a new rule. For construction of the parser, we have used the JavaCC lexer-parser generator.[5] Valid rules are translated into an XML form and are added by the *Registration Unit* to the *Rule Base* (which is an XML file). Details about each rule are stored here, including its name, priority, and event, condition and action parts.

The *Execution Engine* encapsulates the rule processing functionality. In particular, the *Event Dispatcher*, *Condition Evaluator* and *Action Scheduler* implement these aspects of the rule processing, as we describe in more detail below. All of these components interface with the *Wrapper* in order to send and receive data to and from the underlying XML files.

The *Execution Schedule* contains a sequence of *updates*, which have the same syntax as rule actions except that they do not contain any $delta expressions within them. By '$delta expression' we mean an XPath expression (either stand-alone or possibly nested within another XPath expression) that starts with `$delta`. These portions of a rule's action part are replaced by the result of evaluating them on the current document—see below.

The *Wrapper* interfaces with the XML files on disk. All update and query requests from the upper levels of the system pass through this component,

[5] `http://javacc.dev.java.net/`

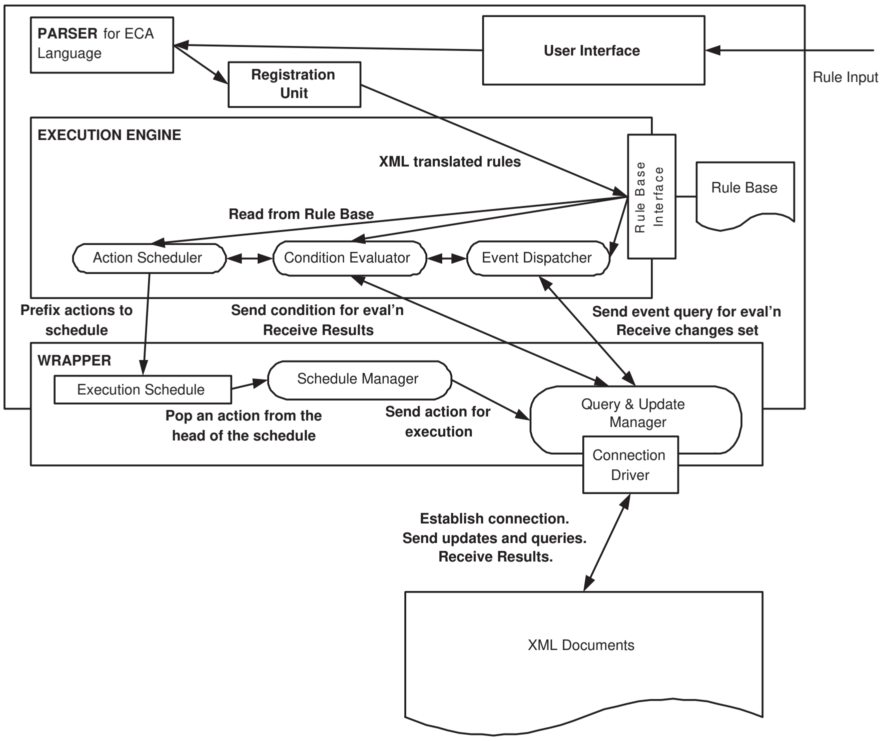

Fig. 1. System architecture

which coordinates them. It undertakes to open files, submit queries and updates, and receive back results from them. The Wrapper performs these services by using the functionality of the Apache Xalan API. All queries are performed directly by using XPath. For deletions, we identify the set of nodes that will be deleted by using the XPath expression within the `DELETE` part of the request, and we then remove all the subdocuments rooted at the nodes identified. For insertions, we identify the set of nodes that will be affected by using the XPath expression within the `BELOW` part of the request, and we then add the fragment specified within the `INSERT` part as a new child of each of the nodes identified, placing it relative to the existing children according to the `AFTER` or `BEFORE` qualifier.

Rule execution begins with a request from the *Schedule Manager* to the *Query & Update Manager* to execute the update currently at the head of the schedule. In case of an insertion, the Query & Update Manager executes the update and annotates the newly inserted nodes, while in the case of a deletion it annotates the nodes to be deleted without executing the deletion yet.[6]

Following the execution of the update, control then passes to the *Event Dispatcher*. This requests the Query & Update Manager to evaluate the XPath query of the event part of each rule that may be triggered by the update that was just executed. For each rule `r_i` whose query result set contains annotated nodes (either newly inserted or about to be deleted), the Event Dispatcher creates the `changes[i]` set containing these annotated nodes, and the rule is triggered.

The *Condition Evaluator* then requests the Query & Update Manager to evaluate the condition part of each triggered rule on the affected document, using as the evaluation context either the root node if there are no occurrences of `$delta` within a query, or each instance of the `changes` set otherwise. The rule's `delta` set is thus created (which will be a subset of its `changes` set). If this is nonempty, the rule fires and control is passed to the Action Scheduler to further process the rule. Otherwise, processing of the rule ends here.

The *Action Scheduler* reformulates a given rule's action(s) in order to eliminate any instances of $delta expressions within them. The reformulation algorithm performs the following steps for each node within the rule's `delta` set:

- replaces the `$delta` variable in each of the $delta expressions by the current node of the `delta` set;
- evaluates each of the modified $delta expressions with respect to the updated document;
- replaces each $delta expression within the rule's action(s) by the corresponding result of the previous step

The outcome of this reformulation is that one instance of the rule's action(s) is created for each node in the rule's `delta` set. These updates are now prefixed, in an arbitrary order, to the front of the schedule. Once this has been done for all the rules that have fired, control passes once more to the Schedule Manager and the cycle repeats.

If the last update executed by the Query & Update Manager was a `DELETE`, then before control passes back to the Schedule Manager, the actual deletion of the annotated nodes is first performed.

Example 2. Consider the XML file in Example 1 and the insertion event Ev. Initially the schedule consists of just this `INSERT` update. The Schedule Manager pops this update from the schedule and sends it to the Query & Update Manager. This executes the update and annotates the newly inserted nodes. The Event Dispatcher then evaluates the XPath expression in rule r_1's event part, which is the only one of the two rules in Example 1 that may be triggered

[6] The annotation of nodes is performed using non-DOM methods provided by Apache Xerces API that allow us to attach data to XML nodes without affecting the physical representation of the file.

by the update. It finds that the result set contains the newly inserted `price` node and it creates the `changes[1]` set, with this node as its only member. The Condition Evaluator then evaluates the condition part of rule r_1, finds it to be True for the single element of `changes[1]` and so creates the `delta` set, with the new `price` node as its only member, and passes control to the Action Scheduler.

The Action Scheduler extracts the $delta expression `$delta/../../month-info/high` from rule r_1's `DELETE` action and the $delta expressions `$delta/text()` and `$delta/../..` from its `INSERT` action and replaces `$delta` by the one member of the `delta` set (the new `price` node). It submits these three queries to the Wrapper, and then substitutes the resulting values for the three `$delta` expressions within r_1's actions. The resulting updates are placed at the front of the schedule and control is passed to the Schedule Manager.

The Schedule Manager pops the first `DELETE` update from the schedule and sends it to the Query & Update Manager. This 'executes' the deletion by annotating the nodes to be deleted. The Event Dispatcher determines that no rules can be triggered by this update, and the nodes annotated as 'to be deleted' are then actually deleted. The Schedule Manager then pops the next `INSERT` update from the schedule and sends it to the Query & Update Manager. This executes the update and annotates the newly inserted nodes. The Event Dispatcher evaluates the XPath expression for rule r_2 and finds that the result set contains the newly inserted `high` node. The Condition Evaluator then evaluates the condition part of the rule r_2, finds it to be False for the single element of `changes[2]` and so no further rules are scheduled. The schedule is now empty, so rule execution terminates.

4 Analysing and Optimising Rule Behaviour

Techniques for determining triggering and activation relationships between rules can be utilised in a variety of ways for analysing and optimising the behaviour of XML ECA rules defined in our language:

- They can then be 'plugged into' existing frameworks for ECA rule analysis (both static and dynamic) such as the approaches we reviewed in Sect. 2.4. For example, if we know the pairwise triggering and activation relationships between rules, we can use triggering graph analysis, activation graph analysis or the rule reduction method. It is also possible to use triggering and activation information within an abstract interpretation framework for a more precise analysis than these graph-based approaches, as discussed in [7].
- Information about which ECA rules may be triggered by the current rule action can be used during rule execution to limit the set of rule event parts that need to be evaluated after the execution of this action.
- The activation relationships between pairs of rules can be dynamically updated during rule execution, and this information can be used to avoid

evaluating rule conditions which can currently be inferred to be definitely True or False.

4.1 Determining Triggering Relationships

In order to determine triggering relationships between our XML ECA rules, we need to be able to determine whether an action of some rule may trigger the event part of some other rule. Clearly, `INSERT` actions can only trigger `INSERT` events, and `DELETE` actions can only trigger `DELETE` events.

For any insertion action a of the form

$$\texttt{INSERT}\ r\ \texttt{BELOW}\ e_1\ \texttt{BEFORE|AFTER}\ q$$

in some rule r_i and any insertion event ev of the form

$$\texttt{INSERT}\ e_2$$

in some rule r_j, we need to know whether event ev is *independent* of action a, that is, e_2 can never return any of the nodes inserted by a.

The XQuery r defines which nodes are inserted by a, while the XPath expression e_1 defines where these nodes are inserted. So if it is possible that some initial part of e_2 can specify the same path through some document as e_1 and the remainder of e_2 'matches' r, then ev is not independent of a. We now define these notions more formally:

We define a *prefix* of a simple XPath expression e to be an expression e' such that $e = e'/e''$ or $e = e'//e''$. We call e'' the *suffix* of e and e'. For an XQuery r, let *type*(r) be the result type of r—this can be determined using the type inference techniques described in [27] or [42]. Using the same techniques, we can test whether or not an XPath expression e can return a nonempty result when evaluated on documents of *type*(r) by first inferring the output type of e, given input *type*(r), and then checking whether the output type is the inconsistent type. If so, then e always returns an empty result on input of *type*(r), in which case we say that *type*(r) *cannot satisfy* e. If the output type is not the inconsistent type, we say that *type*(r) *may satisfy* e.

Given XPath expressions e_1 and e_2, we say that e_1 and e_2 are *independent* if, for all possible XML documents d, $e_1(d) \cap e_2(d) = \emptyset$. We discuss in [9] how testing for independence of two simple XPath expressions can be done by finding an XPath expression that corresponds to $e_1 \cap e_2$, and then checking that the containment $e_1 \cap e_2 \subseteq \emptyset$ holds (more general fragments of XPath which are also closed under intersection are presented in [15], while the complexity of the containment problem for various fragments of XPath is discussed in [32]).

Thus, event ev above is independent of action a if for all prefixes e_2' of e_2, either

1. e_1 and e_2' are independent, or
2. *type*(r) cannot satisfy e_2''.

Equivalently, a rule r_i (containing action a) may trigger rule r_j (containing event ev) if for some prefix e_2' of e_2, e_1 and e_2' are not independent and *type*(r) may satisfy e_2''.

Similarly to insertions, for any deletion action a of the form

DELETE e_1

belonging to a rule r_i, and any deletion event ev of the form

DELETE e_2

belonging to a rule r_j, we have that r_i may trigger r_j if ev is not independent of a. The test for independence of an action and an event in the case of deletions is simpler than for the insertion case above. Let e be the XPath expression e_1`//*`. Then event ev is independent of action a if expressions e and e_2 are independent.

Example 3. Consider the insertion event Ev and ECA rules r_1 and r_2 from Example 1. We can detect that Ev may trigger r_1 since (1) this prefix e'_2 of the XPath expression e_2 in the event part of r_1:

```
document('s.xml')/shares/share/day-info/prices
```

and the XPath expression from Ev:

```
document('s.xml')/shares/share[@name="XYZ"]/
  day-info[@day="03"][@month="03"]/prices
```

are not independent, and (2) the type of the XQuery fragment in Ev, namely `price`, satisfies the suffix e''_2 of e_2 (also `price`). We can also detect that Ev cannot trigger rule r_2 since every prefix of `document('s.xml')/shares/share/day-info/high` is independent of e_1.

4.2 Determining Activation Relationships

In order to determine activation relationships between our ECA rules, we need to be able to determine

a. whether an action of some rule r_i may change the value of the condition part of some other rule r_j from False to True, in which case r_i may activate r_j; and
b. whether all the actions of a rule r_i will definitely leave the condition part of r_i False, i.e. whether rule r_i is *self-disactivating*; if not, then r_i may activate itself

Without loss of generality, we can assume that rule conditions are in disjunctive normal form, i.e. they are of the form

$$(l_{1,1} \texttt{ and } l_{1,2} \ \ldots \texttt{ and } l_{1,n_1}) \texttt{ or } (l_{2,1} \texttt{ and } l_{2,2} \ \ldots \texttt{ and } l_{2,n_2}) \texttt{ or } \ldots \texttt{or } (l_{m,1} \texttt{ and } l_{m,2} \ \ldots \texttt{ and } l_{m,n_m})$$

where each $l_{i,j}$ is either a simple XPath expression c, or the negation of a simple XPath expression, `not` c.

Simple XPath expressions. The following table shows the transitions that the truth value of a condition consisting of a single simple XPath expression can undergo. The first column shows the condition's truth value before the update, and the subsequent columns its truth value after a nonindependent insertion (NI) and a nonindependent deletion (ND):

Before	After NI	After ND
$True$	$True$	$True$ or $False$
$False$	$True$ or $False$	$False$

For case (a) above, i.e. when r_i and r_j are distinct rules, it is clear from this table that r_i can activate r_j only if one of the actions of r_i is an insertion that is nonindependent of the condition of r_j.

Let the condition of r_j be the simple XPath expression c. The procedure for determining nonindependence of an insertion from a condition c involves constructing from c a set C of conditions, each of which is an XPath expression without any qualifiers. The objective is that condition c can change from False to True as a result of an insertion only if at least one of the conditions in C can change from False to True as a result of the insertion. We start with set $C = \{c\}$ and proceed to decompose c into a number of conditions without qualifiers, adding each one to C. See [9] for details of the decomposition algorithm.

Now let one of the actions a from rule r_i be

`INSERT` r `BELOW` e_1 `BEFORE|AFTER` q

We determine $type(r)$ and consider prefixes and suffixes of each condition $c_i \in C$, where $c_i = c_i' \cdot c_i''$. Set C of conditions is independent of a if for each $c_i \in C$ and for each prefix c' of c, either

1. e_1 and c_i' are independent, or
2. $type(r)$ cannot satisfy c_i''

If so, then action a cannot change the truth value of condition c in rule r_j from False to True. Equivalently, we can say that rule r_i may activate rule r_j if for some prefix c_i' of some $c_i \in C$, e_1 and c_i' are not independent and $type(r)$ may satisfy c_i''.

For case (b) above, if the condition part of r_i is a simple XPath expression c, the rule will be self-disactivating if all its actions are deletions which subsume c. For each deletion action

`DELETE` e_1

we thus need to test if

$$e_1//* \supseteq c$$

For simple XPath expressions and provided additionally that the only operator appearing in qualifiers is '`=`', it is known that containment is decidable [32]. The decidability of containment for various larger fragments of XPath is shown in [15, 32]. However, even if a fragment of XPath is used for which this property is undecidable, it is still possible to use conservative approximations. For example, if there are occurrences of comparison operators other than '`=`' in the condition part of a rule, then we can analyse each operand separately against each deletion action and if either operand is subsumed by the action, then we can infer that this action makes this condition False.

Example 4. Consider rule r_1 from Example 1. We can detect that its first (`DELETE`) action makes False its condition, since it deletes the existing `high` price. However, we cannot conclude that the rule is self-disactivating since the second action of r_1 is an `INSERT` and there is the possibility that this may insert nodes which cause r_1's condition to remain True. Only a deeper analysis of the rule set would detect that rule r_1 is in fact self-disactivating.

Negations of Simple XPath expressions. The following table shows the transitions that the truth value of a condition of the form `not` c, where c is a simple XPath expression, can undergo. The first column shows the truth value of the condition before the update, and the subsequent columns its truth value after a nonindependent insertion (NI) and a nonindependent deletion (ND):

Before	After NI	After ND
$True$	$True$ or $False$	$True$
$False$	$False$	$True$ or $False$

For case (a), where rules r_i and r_j are distinct, it is clear from this table that r_i can activate r_j only if one of the actions of r_i is a deletion which is nonindependent of the condition of r_j.

Let the condition of r_j be `not` c. We construct the set of conditions C from c as outlined in Sect. 4.1. Now let an action from rule r_i be

`DELETE` e_1

and let e be the query $e_1//*$. We again use the technique outlined in Sect. 4.1 in order to check whether e is independent of each of the conditions in C. If so, then e cannot change the truth value of `not` c from False to True. Otherwise, e is deemed to be nonindependent of `not` c, and r_i may activate r_j.

For case (b) above, a rule r_i activates itself if it may leave its own condition True. We again need the notion of a self-disactivating rule. If the condition part of r_i is `not` c, the rule will be self-disactivating if all its actions are insertions which guarantee that c will be True after the insertion.

Let an insertion action a from rule r_i be

`INSERT` r `BELOW` e_1 `BEFORE|AFTER` q

and let condition c comprise prefix c' and suffix c''. Action a guarantees that c will be True after the insertion if

$$c' \supseteq e_1$$

and *each* of the trees in the set of trees denoted by *type*(r) satisfies c''. Consequently, we need a stronger concept than the fact that *type*(r) *may satisfy* expression c''. As in Section 4.1, we can infer the output type of c'', given input type *type*(r); if this output type is equivalent to *type*(r), then every tree in *type*(r) satisfies c'', and we can conclude that r_i is self-disactivating.

Conjunctions. For case (a), if the condition of a rule r_j is of the form

$l_{j,1}$ `and` $l_{j,2}$ ... `and` l_{j,n_j}

we can use the tests described in the previous two subsections for conditions that are simple XPath expressions or negations of simple XPath expressions to determine if a rule r_i may turn any of the $l_{j,k}$ from False to True. If so, then r_i may turn r_j's condition from False to True, and may thus activate r_j.

For case (b), suppose the condition of rule r_i is of the form

$$l_{i,1} \;\texttt{and}\; l_{i,2} \;\ldots\; \texttt{and}\; l_{i,n_i}.$$

There are three possible cases:

1. All the $l_{i,j}$ are simple XPath expressions. In this case, r_i will be self-disactivating if each of its actions is a deletion which subsumes one or more of the $l_{i,j}$.
2. All the $l_{i,j}$ are negations of simple XPath expressions. In this case, r_i will be self-disactivating if each of its actions is an insertion which falsifies one or more of the $l_{i,j}$.
3. The $l_{i,j}$ are a mixture of simple XPath expressions and negations thereof. In this case, r_i may or may not be self-disactivating.

Disjunctions. For case (a), if the condition of a rule r_j is of the form

$$(l_{1,1} \;\texttt{and}\; l_{1,2} \;\ldots\; \texttt{and}\; l_{1,n_1}) \;\texttt{or}\; (l_{2,1} \;\texttt{and}\; l_{2,2} \;\ldots\; \texttt{and}\; l_{2,n_2}) \;\texttt{or}$$
$$\ldots \texttt{or}\; (l_{m,1} \;\texttt{and}\; l_{m,2} \;\ldots\; \texttt{and}\; l_{m,n_m})$$

we can use the test for conjunctions described above to determine if a rule r_i may turn any of the disjuncts

$$l_{k,1} \;\texttt{and}\; l_{k,2} \;\ldots\; \texttt{and}\; l_{k,n_k}$$

from False to True. If so, then r_i may turn r_j's condition from False to True and may thus activate r_j.

For case (b), suppose the condition of rule r_i is of the form

$$(l_{1,1} \;\texttt{and}\; l_{1,2} \;\ldots\; \texttt{and}\; l_{1,n_1}) \;\texttt{or}\; (l_{2,1} \;\texttt{and}\; l_{2,2} \;\ldots\; \texttt{and}\; l_{2,n_2}) \;\texttt{or}$$
$$\ldots \texttt{or}\; (l_{m,1} \;\texttt{and}\; l_{m,2} \;\ldots\; \texttt{and}\; l_{m,n_m})$$

Then r_i will be self-disactivating if it leaves False all the disjuncts of this condition. This will be so if

1. all the $l_{j,k}$ are simple XPath expressions and r_i disactivates all the disjuncts of its condition as in case 1 for conjunctions above; or
2. all the $l_{j,k}$ are negations of simple XPath expressions and r_i disactivates all the disjuncts of its condition as in case 2 for conjunctions above

In all other cases, r_i may or may not be self-disactivating.

5 Conclusions

In this chapter we discussed the provision of ECA rules for XML repositories. We reviewed ECA rules in conventional active databases and highlighted the main new issues that arise in the context of XML data. We described the design of a language for ECA rules on XML, described a prototype implementation, and presented techniques for analysing the behaviour of ECA rule sets defined in our language.

It would be straightforward to extend our language to also support `REPLACE` events and actions. A `REPLACE` event would have the syntax

$$\texttt{REPLACE}\;\; e$$

while a `REPLACE` action would have the syntax

REPLACE e BY r

where e is a simple XPath expression and r is a simple XQuery expression. The meaning of a REPLACE action is that the set of nodes identified by e (and their descendants) should be replaced by the XML fragments denoted by r. For example, the pair of actions in the rule r_1 in Example 1 could be replaced by the single action

```
REPLACE $delta/../../high;
BY      <high>$delta/text()</high>
```

Combinations of our analysis techniques for INSERT and DELETE actions could be applied to derive triggering and activation information for REPLACE events.

For future work there are several directions to explore:

a. There is as yet no accepted standard update language for XML. If ECA rules are to be supported on XML repositories, then whatever standard eventually emerges, there is also the parallel issue of designing the event language to match up with this update language. In this chapter we have done this in the context of our particular update language for XML. We have also shown how triggering and activation relationships can be detected for our particular ECA rules. In general, the ability to analyse ECA rule sets needs to be balanced against their complexity and expressiveness, and this issue also needs to be borne in mind in future developments in ECA rule languages for XML.
b. We would like to explore more deeply the expressiveness and complexity of the ECA language that we have defined. For example, what types of XML Schema constraints can be enforced and repaired using rules in this language?
c. In general, `updateDB` in Sect. 3.2 will undertake a set of updates on the repository. For INSERT actions, this may result in nondeterminism in the order in which a set of new fragments are inserted under a common parent, since the BEFORE and AFTER constructs only specify the ordering of new fragments with respect to the existing document content. It is an area of further work to extend our ECA language to capture ordering relationships between new fragments being inserted into a document.
d. At present our language supports only semantic triggering, though it would be easy to extend it to also support syntactic triggering. Similarly, although we currently assume immediate coupling mode for event/condition and condition/action, it would straightforward to also allow rules with the full range of other coupling modes. However, the practical applicability and performance implications of these extensions is an area that requires further detailed investigation.
e. We would like to further develop and gauge the effectiveness of our rule analysis and optimisation techniques. For example, incorporating knowledge that certain documents within the repository are valid with respect to a document type definition (DTD) or XML Schema specification may be

useful in at least two ways. First, this knowledge can allow us to simplify the XPath expressions used within ECA rules [38]. Second, it can help to obtain more precise type information when doing type inference, which in turn can allow more precise information on triggering and activation dependencies to be inferred.

f. A related issue is to develop techniques for determining whether a set of ECA rules is type-safe, in other words, whether execution of the rules ensures that each document remains valid with respect to its DTD or XML Schema.

g. We would like to improve our current prototype implementation, and in particular to integrate our ECA Rules Engine with existing XML repositories, once this technology has become mature enough.

h. An important issue is to evaluate the applicability and scalability of our language, its execution model and implementation. For this, we are planning to deploy it for providing reactive functionality on distributed RDF repositories of educational metadata, as part of the EU-funded SeLeNe project (see `http://www.dcs.bbk.ac.uk/selene`).

   The SeLeNe project is investigating the technical requirements, and possible technical solutions, for 'self e-learning networks'. A self e-learning network (SeLeNe) will have a peer-to-peer topology. Each peer will manage part of the overall distributed metadata, possibly with replication across peers. This metadata will be expressed in RDF which, if stored as XML, will be amenable to direct manipulation by our XML ECA rule language. Support of such networks will require:

   - techniques for reconciliation and integration of metadata describing heterogeneous distributed learning objects (LOs)
   - definition of personalised views over this distributed metadata resource
   - detection and notification of changes to the LO metadata descriptions
   - publish/subscribe functionality

   These requirements have a good fit with the functionality that could potentially provided by ECA rules and will provide a challenging testbed for application, evaluation and extension of our language and its implementation, including an opportunity to explore the applicability and performance of a variety of rule coupling modes, and to gauge the effectiveness of our analysis and optimisation methods.

   The SeLeNe project will also provide an opportunity to assess the impact of moving from a centralised to a distributed environment, with the additional challenges of network delay, network reliability, tolerance of delays and failures, synchronisation of event detection and action execution, maintaining consistency of the distributed information resource, etc. Some of the challenges of event-based systems in distributed heterogeneous environments are discussed in the chapters by Jacob et al. and Buchmann et al. in this book.

Many of the above open questions would make suitable PhD research topics, for example the questions raised in (a), (c), (e) and (h). Possible Masters-level projects would include (b), (d), (f) and (g).

References

1. S. Abiteboul, S. Cluet, G. Ferran, and M.-C. Rousset. The Xyleme project. *Computer Networks*, 39:225–238, 2002.
2. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The Lorel query language for semistructured data. *VLDB Journal*, 1(1):68–88, 1997.
3. S. Abiteboul, V. Vianu, B. S. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *JCSS*, 61(2):236–269, 2000.
4. A. Adi, D. Botzer, O. Etzion, and T. Yatzkar-Haham. Push technology personalization through event correlation. In *Proc 26th Int. Conf. on Very Large Databases*, pages 643–645, 2000.
5. A. Aiken, J. Widom, and J. M. Hellerstein. Static analysis techniques for predicting the behavior of active database rules. *ACM TODS*, 20(1):3–41, 1995.
6. J. Bailey and A. Poulovassilis. An abstract interpretation framework for termination analysis of active rules. In *Proc. 7th Int. Workshop on Database Programming Languages, LNCS 1949*, pages 249–266, Kinloch Rannoch, Scotland, 1999.
7. J. Bailey and A. Poulovassilis. Analysis of functional active databases. In P.M.D. Gray et al., editor, *The Functional Approach to Data Management: Modeling, Analyzing and Integrating Heterogeneous Data*. Springer, Berlin Heidelberg New York, 2003.
8. J. Bailey, A. Poulovassilis, and P. Newson. A dynamic approach to termination analysis for active database rules. In *Proc. 1st Int. Conf. on Computational Logic (DOOD stream), LNCS 1861*, pages 1106–1120, London, 2000. Springer, Berlin Heidelberg New York.
9. J. Bailey, A. Poulovassilis, and P.T. Wood. An Event–Condition–Action Language for XML. In *Proc. WWW'2002*, pages 486–495, Hawaii, 2002.
10. J. Bailey, A. Poulovassilis, and P.T. Wood. Analysis and optimisation for event–condition–action rules on XML. *Computer Networks*, 39:239–259, 2002.
11. E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In T. Sellis, editor, *Rules in Database Systems, LNCS 985*, pages 165–181. Springer, Berlin Heidelberg New York, 1995.
12. E. Baralis, S. Ceri, and S. Paraboschi. Compile-time and runtime analysis of active behaviors. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):353–370, 1998.
13. E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 475–486, Santiago, Chile, 1994.
14. E. Baralis and J. Widom. An algebraic approach to static analysis of active database rules. *ACM TODS*, 25(3):269–332, 2000.
15. Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper. Structural properties of XPath fragments. In *Proc. 9th Int. Conf. on Database Theory, LNCS 2572*, pages 79–95, Berlin, 2003.

16. A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *Proc. of the IEEE Conference on Data Engineering (ICDE)*, 2002.
17. A. Bonifati, S. Ceri, and S. Paraboschi. Active rules for XML: A new paradigm for e-services. *VLDB Journal*, 10(1):39–47, 2001.
18. A. Bonifati, S. Ceri, and S. Paraboschi. Pushing reactive services to XML repositories using active rules. In *Proc. 10th World-Wide-Web Conference*, 2001.
19. S. Ceri and P. Fraternali. *Designing Database Applications with Objects and Rules: The IDEA Methodology*. Addison-Wesley, 1997.
20. S. Ceri, P. Fraternali, and S. Paraboschi. Data-driven one-to-one web site generation for data-intensive applications. In *Proc. 25th Int. Conf. on Very Large Databases*, pages 615–626, 1999.
21. S. Chakravarthy. Architectures and monitoring techniques for active databases: An evaluation. *Data and Knowledge Engineering*, 16(1):1–26, 1995.
22. S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data and Knowledge Engineering*, 14(1):1–26, 1994.
23. E. Cho, I. Park, S. J. Hyum, and M. Kim. ARML: an active rule mark-up language for heterogeneous active information systems. In *Proc. RuleML 2002*, Sardinia, June 2002.
24. S. Cluet, P. Veltri, and D. Vodislav. Views in a large scale XML repository. In *Proc. 27th Int. Conf. on Very Large Databases*, pages 271–280, 2001.
25. A. Couchot. Improving the refined triggering graph method for active rules termination analysis. In *Proc. BNCOD 2002, LNCS 2405*, pages 114–133, Sheffield, 2002.
26. N. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *VLDB'92*, pages 327–338, 1992.
27. Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language (preliminary report). In *Proc. WebDB 2000: Int. Workshop on the Web and Databases*, pages 111–116, 2000.
28. H. Ishikawa and M. Ohta. An active web-based distributed database system for e-commerce. In *Proc. Web Dynamics Workshop, London*, 2001. `http://www.dcs.bbk.ac.uk/webDyn/`.
29. A. Karadimce and S. Urban. Refined triggering graphs: A logic based approach to termination analysis in an active object-oriented database. In *Proc. ICDE'96*, pages 384–391, New Orleans, 1996.
30. K. Keenoy et al. Self e-Learning Networks. See `http://www.dcs.bbk.ac.uk/selene/reports/Del22.pdf`, August 2003. SeLeNe Project Deliverable 2.2.
31. K. Kulkarni, N. Mattos, and R. Cochrane. Active database features in SQL3. In N. Paton, editor, *Active Rules in Database Systems*, pages 197–219. Springer, Berlin Heidelberg New York, 1999.
32. Frank Neven and Thomas Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Proc. 9th Int. Conf. on Database Theory, LNCS 2572*, pages 315–329, Berlin, 2003.
33. N. Paton, editor. *Active Rules in Database Systems*. Springer, Berlin Heidelberg New York, 1999.
34. J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *Proc 7th Int. Conf. on Cooperative Information Systems (CoopIS'2000)*, pages 162–173, 2000.
35. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 413–424, 2001.

36. G. Wagner. How to design a general rule markup language? In *Invited talk at the Workshop XML Technologien für das Semantic Web (XSW 2002)*, Berlin, June 2002.
37. J. Widom and S. Ceri. *Active Database Systems.* Morgan-Kaufmann, 1995.
38. Peter T. Wood. Containment for XPath fragments under DTD constraints. In *Proc. 9th Int. Conf. on Database Theory, LNCS 2572*, pages 300–314, Berlin, 2003.
39. World Wide Web Consortium. XML Path Language (XPath), Version 1.0. See `http://www.w3.org/TR/xpath`, November 1999. W3C Recommendation.
40. World Wide Web Consortium. XSL Transformations (XSLT), Version 1.0. See `http://www.w3.org/TR/xslt`, November 1999. W3C Recommendation.
41. World Wide Web Consortium. XQuery 1.0: An XML Query Language. See `http://www.w3.org/TR/xquery`, November 2002. W3C Working Draft.
42. World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Formal Semantics. See `http://www.w3.org/TR/query-semantics`, November 2002. W3C Working Draft.
43. World Wide Web Consortium. Document Object Model (DOM) Level 3 Core Specification. See `http://www.w3.org/TR/DOM-Level-3-Core/`, February 2003. W3C Working Draft.