ABSTRACT

XML repositories are now a widespread means for storing and exchanging information on the Web. As these repositories become increasingly used in dynamic applications such as e-commerce, there is a rapidly growing need for a mechanism to incorporate reactive functionality in an XML setting. Event-condition-action (ECA) rules are a technology from active databases and are a natural method for supporting such functionality. ECA rules can be used for activities such as automatically enforcing document constraints, maintaining repository statistics, and facilitating publish/subscribe applications. An important question associated with the use of an ECA rules is how to statically predict their run-time behaviour. In this paper, we define a language for ECA rules on XML repositories. We then investigate methods for analysing the behaviour of a set of ECA rules, a task which has added complexity in this XML setting compared with conventional active databases.

Keywords

Event-condition-action rules, XML, XML repositories, reactive functionality, rule analysis

1. INTRODUCTION

XML is becoming a dominant standard for storing and exchanging information. With its increasing use in areas such as data warehousing and e-commerce [13, 14, 18, 22, 28], there is a rapidly growing need for rule-based technology to support reactive functionality on XML repositories. Event-condition-action (ECA) rules are a natural candidate to support such functionality. In contrast to implementing reactive functionality directly within a programming language such as Java, ECA rules have a high level, declarative syntax and are thus more easily analysed. Furthermore, many commercial and research systems based on ECA rules have been successfully built and deployed, and thus their implementation within system architectures is well-understood.

ECA rules automatically perform actions in response to events provided stated conditions hold. They are used in conventional data warehouses for incremental maintenance of materialized views, for validation and cleansing of the input data streams, and for maintaining audit trails of the data. By analogy, ECA rules can also be used as an integrating technology for providing this kind of reactive functionality on XML repositories. They can also be used for checking key and other constraints on XML documents, and for performing automatic repairs when violations are detected. For a 'push' type environment, they are a mechanism for automatically broadcasting information to subscribers as the contents of relevant documents change. They can also be employed as a flexible means for maintaining statistics about document and web site usage and behaviour. In this paper, we present a language in which ECA rules on XML can be defined.

ECA rules have been used in many settings, including active databases [26, 29], workflow management, network management, personalisation and publish/subscribe technology [3, 13, 14, 17, 27], and specifying and implementing business processes [2, 16, 22]. However, one of the key recurring themes regarding the successful deployment of ECA rules is the need for techniques and tools for analysing their behaviour [15, 23]. When multiple ECA rules are defined within a system, their interactions can be difficult to analyse, 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. Thus, the second part of this paper explores techniques for analysing the behaviour of a set of ECA rules defined in our language.

Other ECA rule languages for XML have been proposed in [13, 14, 22] but none of these focus on analysing the behaviour of the ECA rules. It is proposed in [13] that analysis techniques developed for conventional active databases can be applied in an XML setting too, but details are not given.

A more recent paper [12] also defines an active rule language for XML, but is not concerned with rule analysis. The rule syntax it describes is similar to the one we define here, the rule format being based on the definition of triggers in SQL3. Its rule execution semantics is rather different from the model we adopt, however. Generally speaking, insertions and deletions of XML data (so-called bulk statements) may involve document fragments of unbounded size. [12] describes a semantics whereby each (top-level) update is decomposed into a sequence of smaller updates (which depend on the contents of the fragment being inserted/deleted) and then trigger execution is interleaved with the execution of these smaller updates. In contrast, we treat each top-level update as atomic and trigger execution is invoked 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 suitability in different applications.

In an earlier paper [7] we described some initial proposals for analysis, and also optimisation, of XML ECA rules. The language we discussed there was less expressive than the language we propose here in that it did not allow disjunction or negation in rule conditions. Moreover, here we examine more deeply the triggering and activation relationships between rules and have derive more precise tests for determining both of these relationships than the tests we described in [7].

2. THE ECA RULE LANGUAGE

An XML database consists of a set of XML documents. Event-condition-action (ECA) rules on XML databases take the following form:

```plaintext
on event
test condition
do actions
```

Rather than introducing yet another query language for XML, we use the XPath [32] and XQuery [33] languages to specify events, conditions and actions within our ECA rules. XPath is in a number of W3C recommendations, such as XPointer, XSLT and XQuery itself, for selecting and matching parts of XML documents and so is well-suited to the requirements of ECA rules. XQuery is used in our ECA rules only where there is a need to be able to construct new fragments of XML. We define each of the components of our ECA rule language below, give some example rules, and describe the rule execution semantics.

2.1 Rule Events

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

```plaintext
INSERT e
```

or

```plaintext
DELETE e
```

where 

is a *simple XPath expression* (defined in Section 2.4 below) which evaluates to a set of nodes. The rule is said to be triggered if this set of nodes includes any node in a new sub-document, in the case of an insertion, or in a deleted sub-document, in the case of a deletion.

The system-defined variable $\$delta$ is available for use within the condition and actions parts of the rule (see below), and its set of instantiations is the set of new or deleted nodes returned by $e$.

2.2 Rule Conditions

The *condition* part of an ECA rule is either the constant TRUE, or one or more simple XPath expressions connected by the boolean connectives and, or, not.

The condition part of an ECA rule is evaluated on each XML document in the database which has been changed by an event of the form specified in the rule’s event part. If the condition references the system-defined $\$delta$ variable, it is evaluated once for each instantiation of $\$delta$ for each document. Otherwise, the condition is evaluated just once for each document.

2.3 Rule Actions

The *actions* part of an ECA rule is a sequence of one or more actions:

```plaintext
action1; ...; actionn
```

These actions are executed on each XML document which has been changed by an event of the form specified in the rule’s event part and for which the rule’s condition query evaluates to True — we call this set of documents the rule’s set of *candidate documents*.

An ECA rule is said to *fire* if its set of candidate documents is non-empty.

Each action, above, is an expression of the form

```plaintext
INSERT r BELOW e [BEFORE/AFTER q]
```

or

```plaintext
DELETE e
```

where $r$ is a *simple XQuery expression*, $e$ is a *simple XPath expression* and $q$ is either the constant TRUE or an *XPath qualifier* — see Sections 2.4 and 2.5 below for definitions of the italicised terms.

In an INSERT action, the expression $e$ specifies the set of nodes, $N$, immediately below which new sub-document(s) will be inserted. These sub-documents are specified by the expression $r$. If $e$ or $r$ references the $\$delta$ variable then one sub-document is constructed for each instantiation of $\$delta$ for which the rule’s condition query evaluates to True. If neither $e$ nor $r$ references $\$delta$ then a single sub-document is constructed.

$q$ is an optional XPath qualifier which is evaluated on each child of each node $n \in N$. For insertions of the form *AFTER q*, the new sub-document(s) are inserted after the last sibling for which $q$ is true, while for insertions of the form *BEFORE q*, the new sub-document(s) are inserted before the first sibling for which $q$ is true. The order in which new sub-documents are inserted is non-deterministic.

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

Example 1. Consider an XML database consisting of two documents, $s$.xml and $p$.xml. The document $s$.xml contains information on stores, including which products are sold in each store:

```xml
<stores>
  <store id="s1">
    <location>...</location>
  </store>
</stores>
```

We observe that using the phrase `BEFORE e` to indicate where the update should happen is significant. Without it the placement of new sub-documents would be restricted to occurring only at the root node of documents.

Thus, both *document-level* and *instance-level* triggering are supported in our ECA rule language. If there is no occurrence of the $\$delta$ variable in a rule action, the action is executed at most once on each document each time the rule fires — this is document-level triggering. If there is an occurrence of $\$delta$ in an action part, the action is executed once for each possible instantiation of $\$delta$ on each document — this is instance-level triggering.
<manager>...
<product id="p1"/>
<product id="p2"/>
...
</store>
...</stores>

The document p.xml holds information on each product, including which stores sell each product:

<products>
  <product id="p1">
    <name>...<name>
    <category>...<category>
    <price>...<price>
    <store id="s1"/>
    <store id="s2"/>
  ...
  </product>
  ...
</products>

The following ECA rule updates the p.xml document whenever one or more products are added below an existing store in s.xml:

on INSERT document('s.xml')/stores/store/product
if not(document('p.xml')/products/product[@id=$id])
do INSERT 'store id={$id}'/product[@id=$id]
   B.E.L.O.W document('p.xml')/products/
   product[@id=$id] AFTER TRUE

Here, the system-defined $delta variable is bound to the newly inserted product nodes detected by the event part of the rule. The rule’s condition checks that the store which is the parent of the inserted products in s.xml is not already a child of those products in p.xml. The action then adds the store as a child of those products in p.xml.

In a symmetrical way, the following ECA rule updates the s.xml document whenever one or more stores are added below an existing product in p.xml:

on INSERT document('p.xml')/products/product/store
if not(document('s.xml')/stores/
    store[@id=$id]/product[@id=$id])
do INSERT 'store id={$id}'/store/@id=
   B.E.L.O.W document('s.xml')/stores/
   store[@id=$id] AFTER TRUE

The two rules ensure that the information in the two documents is kept mutually consistent.

Example 2. This example is taken from [1] which discusses view updates on semi-structured data. The XML database consists of two documents, g.xml and m.xml. g.xml contains a restaurant guide, with information about restaurants, including the entrees they serve, and the ingredients of each entree:

<guide>
  <restaurant>
    <name>Thai City</name>
    <rating>5</rating>
  </restaurant>
</guide>

restaurant>
  <name>Baghdad Cafe</name>
  <rating>9</rating>
  <entree>
    <name>Beef Stew</name>
    <ingredient>Mushroom</ingredient>
  </entree>
  <entree>
    <ingredient>Tomato</ingredient>
  </entree>
</restaurant>

restaurant>
  <name>Eats</name>
  <rating>Four stars</rating>
  <entree>
    <ingredient>Tomato</ingredient>
  </entree>
  <entree>
    <name>Cheeseburger Club</name>
    <ingredient>Cheese</ingredient>
    <ingredient>Beef</ingredient>
  </entree>
</restaurant>
...
</guide>

The document m.xml is a view derived from g.xml, and contains a list of those entrees at the Baghdad Cafe where one of the ingredients is Mushroom:

<entrees>
  <entree>
    <name>Beef Stew</name>
    <ingredient>Mushroom</ingredient>
  </entree>
</entrees>

Suppose now an ingredient element with value Mushroom is added to the second (unnamed) entree of the Baghdad Cafe in g.xml. The following ECA rule performs the view maintenance of m.xml:

on INSERT document('g.xml')/guide/restaurant/
  entree/ingredient
if $delta=-"Mushroom" and
  $delta/@name='Baghdad Cafe'
do INSERT $delta/.. B.E.L.O.W document('m.xml')/entrees
  AFTER TRUE

The resulting m.xml document is:

<entrees>
  <entree>
    <name>Beef Stew</name>
    <ingredient>Mushroom</ingredient>
  </entree>
  <entree>
    <ingredient>Tomato</ingredient>
  </entree>
  <entree>
    <ingredient>Mushroom</ingredient>
  </entree>
</entrees>

Note that inserting $delta/.. results in the complete entree being inserted, while AFTER TRUE causes it to be inserted after the last child of entrees.
2.4 Simple XPath Expressions

The XPath and XQuery expressions appearing in our ECA rules are restrictions of the full XPath [32] and XQuery [33] 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. 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 other functions other than \document{}(). Thus, the syntax of a \textit{simple} XPath expression \( e \) is given by the following grammar, where \( s \) denotes a string and \( n \) denotes an element or attribute name:

\[
e ::= \ \document{'}s'\} | (('/' | '::') p) |
\$delta' ('[' q ']')* ('[' q ']') p? \( . \)
\]

\[
p ::= p / p | p / p | p | p [ q ] | n | 's' |
\]

\[
q ::= q \text{ and } q | q =\text{ or } q | e | p | p | p [ q ] | n | 's' |
\]

\[
o ::= 's' | 'n' = | 'c' e |
\]

Expressions enclosed in [' and '] in an XPath expression are called \textit{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 \( e \).

If we delete all qualifiers (along with the enclosing brackets) from an XPath expression, we are left with a path of nodes. We call this path the \textit{distinguished path} of the expression and the node at the end of the distinguished path the \textit{distinguished leaf} of the expression.

The result of an XPath expression \( e \) is a set of nodes, namely, those matched by the distinguished leaf of the expression. The \textit{result type} of \( e \), denoted \textit{type}(\( e \)), is one of \textit{string}, element name or *, where * denotes any element name. The result type can be determined as follows.

Let \( e \) be the distinguished path of \( e \). If the leaf of \( p \) is \$n or \$*, \textit{type}(\( e \)) is \textit{string}. If the leaf of \( p \) is \$n or \$*, \textit{type}(\( e \)) is \$n or \$, respectively. If the leaf is \$ or \$, \textit{type}(\( e \)) is determined from the leaf of \textit{a modified distinguished path} which is defined below.

The \textit{modified distinguished path} is constructed from the distinguished path \( p \) of expression \( e \) by replacing each occurrence of \$ and \$ from left to right in \( p \) as follows.

If \( p \) starts with \$delta, then we substitute for \$delta the distinguished path of the XPath expression which occurs in the event part of the rule. If the step is \$ and it is preceded by \$ ('where \( a \) must be an element name or *'), then replace \$ with *'. If the separator preceding the occurrence of \$ or \$ is \$', then replace the step with *'. If the step is \$ and the separator which precedes it is \$, then delete the step and its preceding separator.

\textit{Example 3.} Consider the condition from the ECA rule of Example 2, namely,

\[
\text{if } \text{delta[=Mushroom]} \text{ and } \text{delta[../[name=Baghdad Cafe']}
\]

The result type of the first conjunct is \textit{ingredient} because the event part of the rule is

\[
\text{on INSERT document('g.xml')/guide/restaurant/entree/ingredient}
\]

The result type of the second conjunct is \textit{restaurant}, determined as follows. The distinguished path after substituting \$delta is

\[
document('g.xml')/guide/restaurant/entree/ingredient/.../
\]

So we replace 'ingredient/' with '.', we delete '/', we replace 'entree/' with '.', and finally we delete '/', leaving the modified distinguished path

\[
document('g.xml')/guide/restaurant
\]

Type inference is part of the XQuery formal semantics defined in [34]. This allows an implementation of XQuery to infer at query compile time the output type of a query on documents conforming to a given input type (DTD or schema). Since XPath expressions are part of XQuery, their result types can also be inferred. Thus, in the presence of a DTD or XML schema, it is possible to infer more accurate result types for XPath expressions using the techniques described in [34].

2.5 Simple XQuery Expressions

The XQuery fragment we use disallows the use of full so-called FLWRO expressions (involving keywords for, 'let,' 'where' and 'return'), essentially permitting only the 'return' part of an expression [33].

The syntax of a \textit{simple} XQuery expression \( e \) is given by the following grammar:

\[
r ::= e | c
\]

\[
c ::= ('/' | ('/' ts '<' n '>'))
\]

\[
\text{(n as } s | e') 's' a)?
\]

\[
t ::= s | c | e'
\]

\[
e' ::= ['e']
\]

Thus, an XQuery expression \( r \) is either a simple XPath expression \( e \) (as defined in Section 2.4) 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 \textit{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.

The \textit{result type} of an XQuery expression \( r \), denoted \textit{type}(\( r \)), is a tree, each of whose nodes is of type \$ (for element name \$n), \$n (for attribute name \$n), * or /, or string. The types with a suffix /// indicate that the corresponding node can be the root of an arbitrary subtree. This is necessary.

\[\text{Although the parent function in [34], which corresponds to a step of \$'/' always returns the type anyElement?}\]
to capture the fact that the results of XPath expressions embedded in r return sets of nodes which may be the roots of subdocuments. The tree for type(r) can be determined as follows. If r is an XPath expression e, then type(r) comprises a single node whose type is type(e)/*/# if type(e') is n or *, or string if type(e') is string. If r is an element constructor c, then we form a document tree T from c in the usual way, except that some nodes will be labelled with enclosed expressions. For each such enclosed expression e', we determine its result type type(e') in the same way as for the single XPath expression above. We then replace e' in T by type(e'). Now type(r) is given by the modified tree T.

The result type of an XQuery expression r denotes a set of trees S such that every tree returned by r in S (the converse does not necessarily hold because we do not type the results of enclosed expressions as tightly as possible). We call each tree in S an instance of type(r). Given an XPath expression e and a tree T, T satisfies e if e(T) ≠ ∅. We say that type(r) may satisfy e if some instance of type(r) satisfies e.

Given XPath expression e and XQuery expression r, it is straightforward to test whether or not type(r) may satisfy e. The test essentially involves checking whether the evaluation of e on the tree of type(r) is empty or not. However, since type(r) denotes a set of trees rather than a single tree, the evaluation needs to be modified as indicated in the following informal description: a node of type string in type(r) may satisfy any string in e; a node of type * in type(r) may satisfy any element name in e; a node of type n/* (respectively, */* in type(r) may satisfy any expression in e which tests attributes or descendants of an element name n (respectively, any element name).

Example 4. Let r be the XQuery expression in the action of the first rule in Example 1, namely

<store id='{$delta/..@@id}'/>,

The result type of $delta/..@@id is string, so type(r) is store($id(string)).

As another example, let r be the XQuery expression

<ax>...</delta/..</ax>

and assume that the result type of $delta/.. is *. Then type(r) is a(b/*+/*)/c.[c].

2.6 ECA Rule Execution

In this section we describe informally the ECA rule execution semantics, giving sufficient details for our purposes in this paper. We refer the interested reader to [7] for a fuller discussion.

The input to ECA rule execution is an XML database and a schedule. The schedule is a list of updates to be executed on the database. Each such update is a pair

(a_i,j docsAndDelta_s_i).

The component a_i,j is an action from the actions part of some rule r_i. The component docsAndDelta_s_i is a set of pairs (d, delta_s_i), where d is the identifier of a document upon which a_i,j is to be applied and delta_s_i is the set of instantiations for the $delta variable generated by the event and condition part of rule r_i with respect to document d.

For each rule r_i, we then determine its set of candidate documents generated by this update, together with the set delta_s_i for each candidate document d. For all rules r_i that have fired (i.e. whose set of candidate documents is non-empty) we place their list of actions a_i,1, a_i,2, ..., a_i,n, at the head of the schedule, placing the actions of higher-priority rules ahead of the actions of lower-priority rules. Each such action a_i,j is paired with the set docsAndDelta_s_i consisting of the set of candidate documents for rule r_i with the set of instantiations delta_s_i for each such candidate document d.

The execution proceeds in this fashion until the schedule becomes empty. Non-termination of rule execution is a possibility and thus rule analysis techniques are important for developing sets of ‘well-behaved’ rules.

3. ANALYSING ECA RULE BEHAVIOUR

Analysis of ECA rules in active databases is a well-studied topic and a number of analysis techniques have been proposed, e.g., [4, 5, 6, 8, 9, 10, 11, 16], mostly in the context of relational databases. Analysis is important, since within a set of ECA rules, unpredictable and unstructured behaviour may occur. Rules may mutually trigger one another, leading to unexpected (and possibly infinite) sequences of rule executions.

Two important analysis techniques are to derive triggering [4] and activation [10] relationships between pairs of rules. This information can then be used to analyse properties such as termination or confluence of a set of ECA rules, or reachability of individual 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.

Thus, two key analysis questions regarding ECA rules are:

1. Is it possible that a rule r_i may trigger a rule r_j?
2. Is it possible that a rule r_i may activate a rule r_j?

Once triggering and activation relationships have been derived, one can construct graphs which are useful in analysing rule behaviour:

A triggering graph [4] 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, by examination of the arcs in the graph.

An activation graph [10] again represents rules as vertices and 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.

In common with the SQL3 standard for database triggers [24] we assume that no two rules can have the same priority. This, together with our use of restricted sub-languages of XPath/XQuery, ensures that rule execution is deterministic in our language, up to the order in which new sub-documents are inserted below a common parent.
3.1 Triggering Relationships between XML ECA rules

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.

3.1.1 Insertions

For any insertion action a of the form

\[ \text{INSERT } r \text{ BELOW } e_1 \]\n
in some rule \( r_i \) and any insertion event \( e \) of the form

\[ \text{INSERT } e_2 \]

in some rule \( r_j \), we need to know whether \( e \) is independent of \( a \), that is, \( e_2 \) can never return any of the nodes inserted by \( a \).

The simple XQuery \( r \) defines which nodes are inserted by \( a \), while the simple XPath expression \( e_1 \) defines where these nodes are inserted. So, informally speaking, 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 \( e \) is not independent of \( a \). We formalise these notions below, based on tests for containment between XPath expressions [19, 20, 30].

A prefix of a simple XPath expression \( e \) is an expression \( e' \) such that \( e = e' | e'' \) or \( e = e' | /e'' \). We call \( e'' \) the suffix of \( e \) and \( e' \). Recall from Section 2.5 that, for XQuery \( r \), type(\( r \)) denotes the result type of \( r \), and we can test whether or not type(\( r \)) may satisfy an XPath expression \( e \).

Given XPath expressions \( e_1 \) and \( e_2 \), we say that \( e_1 \) and \( e_2 \) are independent if, for all possible XML documents \( d \), \( e_1(d) \cap e_2(d) = \emptyset \).

Now let us return to the action \( a \) and event \( e \) defined above. Event \( e \) is independent of action \( a \) if for all prefixes \( e' \) of \( e_2 \), either

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

Equivalently, we can say that rule \( r_i \) (containing action \( a \)) may trigger rule \( r_j \) (containing event \( e \)) if for some prefix \( e' \) of \( e_2 \), \( e_1 \) and \( e_2 \) are not independent and type(\( r \)) may satisfy \( e' \).

3.1.2 Deletions

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

\[ \text{DELETE } e_1 \]

belonging to a rule \( r_i \), and any deletion event \( e \) of the form

\[ \text{DELETE } e_2 \]

belonging to a rule \( r_j \), we have that \( r_i \) may trigger \( r_j \) if \( e \) 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 \( e_1 \). Event \( e \) is independent of action \( a \) if expressions \( e_1 \) and \( e_2 \) are independent (which can be determined as in Section 3.1.1).
3.2 Activation Relationships between XML
ECA rules

In order to determine activation relationships between 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; 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

\[ (l_{1,1} \land l_{1,2} \ldots \land l_{1,m_1}) \lor (l_{2,1} \land l_{2,2} \ldots \land l_{2,n_2}) \lor \ldots \lor (l_{m,1} \land l_{m,2} \ldots \land l_{m,n_m}) \]

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

3.2.1 Simple XPath expressions

The following table illustrates the transitions that the truth-value of a condition consisting of a 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 non-independent insertion (NI) and a non-independent deletion (ND):

<table>
<thead>
<tr>
<th>before</th>
<th>after NI</th>
<th>after ND</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True or False</td>
</tr>
<tr>
<td>False</td>
<td>True or False</td>
<td>False</td>
</tr>
</tbody>
</table>

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 which is non-independent of the condition of \( r_j \). Let the condition of \( r_j \) be the simple XPath expression \( c \).

For \( c \) to be True, we require that it returns a non-empty result. Thus, in a sense, the distinguished path of \( c \) plays the same role as an XPath qualifier in that we are interested in the existence of some path in the document matching the distinguished path. In addition, the insertion of an element whose name occurs only in a qualifier of \( c \) can turn \( c \) from False to True. For example, the insertion of a d element below /a/b can turn condition /a/b/d/a from False to True. Thus we need to consider all the qualifiers and the distinguished path in \( c \) in a similar way in any test for case (a). Moreover, the use of . . . in a condition is analogous to introducing a qualifier, so we need to rewrite conditions accordingly. For example, the condition a/b/.../d is equivalent to a[b]/d. This condition can be turned from False to True if either a d element is added below an a element which has a b element as a child, or a b element is added below an a element which has a d element as a child.

The procedure for determining non-independence 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 i.e. a distinguished path. 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 \( C \).

A single step of the decomposition is as follows:

1. For any \( u \) and \( v \), and \( u \) an element name \( n \) or *,

   - if condition \( c_i \) is of the form \( u/v/.../w \), then replace \( c_i \) in \( C \) by \( u/.[c_i]w \);
   - if \( c_i \) is of the form \( u/v/.../w \), then replace \( c_i \) in \( C \) by \( u/.[v]u \) and \( u/.[v]w \).

2. We can delete from \( c_i \) steps of the form / . . . , as well as replacing occurrences of / / . . . by //, thereby ensuring that . can occur only at the end of \( c_i \) preceded by // .

3. If \( c_i \in C \) is of the form \( u[v]w \), where \( u, v \), and \( w \) are all non-empty, then replace \( c_i \) in \( C \) by \( u[w] \) and \( u[w]w \).

4. If \( c_i \in C \) is of the form \( u[v] \), where \( u \) and \( v \) are non-empty, then delete \( c_i \) from \( C \) and add to \( C \) the conditions specified by one of the cases below, depending on the structure of the qualifier \( v \):

   - if \( v \) matches nonterminal \( p \) from the grammar for simple XPath expressions, then add \( u/v \) to \( C \);
   - if \( v \) matches nonterminal \( e \), then add \( e \) to \( C \);
   - if \( v \) matches \( x \) or \( y \) (where \( x \) and \( y \) must be qualifiers), then add \( u[x] \) and \( u[y] \) to \( C \);
   - if \( v \) matches \( x \) and \( y \) (where \( x \) and \( y \) must be qualifiers), then add \( u[x] \) and \( u[y] \) to \( C \);
   - if \( v \) matches \( x \) or \( y \), then if \( x \) or \( y \) matches nonterminal \( p \), add \( u/x \) or \( u/y \), respectively, to \( C \), while if \( x \) or \( y \) match nonterminal \( e \), add \( x \) or \( y \), respectively, to \( C \).

The decomposition process continues until all conditions in \( C \) are qualifier-free.

Now let one of the actions a from rule \( r_i \) be

**INS**ert \( r \) belo**w** \( e_1 \) [BE**F**O**R**]a[AF**T**ER] q

As in Section 3.1.1, we determine type(r) and consider prefixes and suffixes of each condition \( c_i \in C \), where \( c_i = c'_i . c''_i \).

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 dependent and type(r) may satisfy \( c''_i \).

Example 6. The conjunction of conditions from the ECA rule in Example 2 can be rewritten as the single condition \( c \):

\[ \delta \text{Mushroom'}/.../\{'name='Baghdad Cafe'\} \]

Set \( C \) initially comprises just condition \( c \), which, after substituting for \( \delta \) (and dropping document('g.xml') for simplicity), is decomposed into the conditions

$$\delta['Mushroom']/.../\{'name='Baghdad Cafe'\}$$
3.2.2 Negations of Simple XPath expressions

The following table illustrates 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 non-independent insertion (NI) and a non-independent deletion (ND):

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after NI</th>
<th>after ND</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True or False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True or False</td>
<td>True</td>
</tr>
</tbody>
</table>

For case (a), where rules \( r_1 \) and \( r_2 \) are distinct, it is clear from this table that \( r_1 \) can activate \( r_2 \) only if one of the actions of \( r_2 \) is a deletion which is non-independent of the condition of \( r_2 \).

Let the condition of \( r_j \) be not \( c \). We construct the set of conditions \( C \) from \( c \) as in Section 3.2.1. Now let the action from rule \( r_1 \) be

```
DELETE e1
```

and let \( e \) be the query \( e1/s \). We again use the intersection test from Section 3.1.1 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 \( c \) from False to True. Otherwise, \( e \) is deemed to be non-independent of \( c \), and \( r_1 \) may activate \( r_j \).

For case (b) above, a rule \( r_1 \) 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 \( r_1 \) is not \( c \), the rule will be self-disactivating if all its actions are deletions which subsume \( c \). For each deletion action

```
DELETE e1
```

we thus need to test if

```
e1/s \supseteq c
```

For the simple XPath expressions to which our ECA rules are constrained in this paper, and provided additionally that the only operator appearing in qualifiers is ‘\(^*\)’, it is known that containment is decidable [10]. Thus, it is possible to devise a test for determining whether rules are self-disactivating. The decidability of containment for larger fragments of the XPath language is an open problem [10]. However, even if a fragment of XPath is used for which this property is undecidable, it may still be possible to develop conservative approximations, and this is an area of further research.
may turn any of the $l_{ij}$ 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 $l_1 \land l_2 \land \ldots \land l_n$. There are three possible cases:

(i) All the $l_{ij}$ 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 $l_{ij}$.

(ii) All the $l_{ij}$ 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 $l_{ij}$.

(iii) The $l_{ij}$ are a mixture of simple XPath expressions and negations thereof. In this case, $r_i$ may or may not be self-disactivating.

3.2.4 Disjunctions

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

$$(l_{1,1} \land l_{1,2} \ldots \land l_{1,n_1}) \lor (l_{2,1} \land l_{2,2} \ldots \land l_{2,n_2})$$

... or

$$(l_{m,1} \land l_{m,2} \ldots \land l_{m,n_m})$$

we can use the test described in Section 3.2.3 above to determine if a rule $r_i$ may turn any of the disjuncts

$$l_{1,1} \land l_{1,2} \ldots \land l_{1,n_1}$$

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

$$(l_{1,1} \land l_{1,2} \ldots \land l_{1,n_1}) \lor (l_{2,1} \land l_{2,2} \ldots \land l_{2,n_2})$$

... or

$$(l_{m,1} \land l_{m,2} \ldots \land l_{m,n_m})$$

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

(i) all the $l_{ij}$ are simple XPath expressions and $r_i$ disactivates all the disjuncts of its condition as in case (i) of Section 3.2.3 above; or

(ii) all the $l_{ij}$ are negations of simple XPath expressions and $r_i$ disactivates all the disjuncts of its condition as in case (ii) of Section 3.2.3 above.

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

4. CONCLUSIONS

In this paper we have proposed a new language for defining ECA rules on XML, thus providing reactive functionality on XML repositories, and we have developed new techniques for analysing the triggering and activation dependencies between rules defined in this language. Our language is based on reasonably expressive fragments of the XPath and XQuery standards.

The analysis information that we can obtain is particularly useful in understanding the behaviour of applications where multiple ECA rules have been defined. Determining this information is non-trivial, since the possible associations between rule actions and rule events/conditions are not syntactically derivable and instead deeper semantic analysis is required.

One could imagine using XSLT to transform source documents and materialise the kinds of view documents we have used in the examples in this paper. However, XSLT would have to process an entire source document after any update to it in order to produce a new document whereas we envisage detecting updates of much finer granularity. Also, using ECA rules allows one to update a document directly, whereas XSLT requires a new result tree to be generated by applying transformations to the source document.

The simplicity of ECA rules is another important factor in their suitability for managing XML data. ECA rules have a simple syntax and are automatically invoked in response to events — the specification of such events is indeed a part of the Document Object Model (DOM) recommendation by the W3C. Also, as is argued in [13], the simple execution model of ECA rules makes them a promising means for rapid prototyping of a wide range of e-services.

The analysis techniques we have developed are useful in a context beyond ECA rules. Our methods for computing rule triggering and activation relationships essentially focus on determining the effects of updates upon queries — the ‘query independent of update’ problem [25]. We can therefore use these techniques for analysing the effects of other (i.e. not necessarily rule-initiated) updates made to an XML database, e.g. to determine whether integrity constraints have been violated or whether user-defined views need to be re-calculated. Query optimisation strategies are also possible: e.g. given a set of pre-defined queries, one may wish to retain in memory only documents which are relevant to computing these queries. As updates to the database are made, more documents may need to be brought into memory and these documents can be determined by analysing the effects of the updates made on the collection of pre-defined queries.

For future work there are two main directions to explore. Firstly, we wish to understand more fully the expressiveness and complexity of the ECA language that we have defined. For example, we wish to look at what types of XML Schema constraints can be enforced and repaired using rules in the language. Secondly, we wish to further develop and gauge the effectiveness of our analysis methods. Techniques such as incorporating additional information from document type definitions may help obtain more precise information on triggering and activation dependencies [31]. We also wish to investigate the use of these dependencies for carrying out optimisation of ECA rules.

5. REFERENCES


