Analysis and Optimisation of Event-Condition-Action Rules on XML

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

[1]Department of Computer Science, University of Melbourne
jbailey@cs.mu.oz.au
[2]School of Computer Science and Information Systems, Birkbeck College, University of London
{ap,ptw}@dcs.bbk.ac.uk

Abstract

XML is a now a dominant standard for storing and exchanging information. With its increasing use in areas such as data warehousing and e-commerce, there is a rapidly growing need for rule-based technology to support reactive functionality on XML repositories. Event-condition-action (ECA) rules automatically perform actions in response to events and are a natural facility to support such functionality. In this paper, we study ECA rules in the context of XML data. We define a simple language for specifying ECA rules on XML repositories. The language is illustrated by means of some examples, and its syntax and semantics are then specified more formally. We then investigate methods for analysing and optimising these ECA rules, a task which has added complexity in this XML setting compared with conventional active databases.

1 Introduction

Event-condition-action rules are used to provide reactive functionality in many settings, including active databases [35, 31], workflow management, network management, personalisation and publish/subscribe technology [32, 20, 16, 17, 4, 29], and specifying and implementing business processes [19, 2, 26].

In this paper we propose a simple ECA rule language for providing reactive functionality over corpora of XML documents. The motivation for this work is the increasing use of XML as a mechanism for data warehousing — see for example recent extensions of commercial database systems to support storage, viewing and exporting of XML data, and discussions in [1, 16, 23, 21, 33, 29]. In this context, there is a need for incremental maintenance of the materialised views in the XML data warehouse, for validating and cleansing the input data streams, and for maintaining audit trails of the data. By analogy to their use in conventional databases, ECA rules can be used as an integrating technology for providing this kind of reactive functionality on corpora of XML documents. For a 'push' type environment, ECA rules 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.

One of the key recurring themes regarding the successful deployment of ECA rules is the need for techniques and tools for *analysing* and *optimising* their behaviour [19, 27, 18]. Thus, the second part of this paper explores techniques for analysing and optimising our proposed ECA rule language for XML.

The closest work to ours is [26, 16, 17, 15]. [26] describes a language for interacting with XML sources on the web which incorporates ECA rules. However, details of the syntax and semantics of these rules are not given, and their analysis and optimisation are not addressed. [16] discusses extending XML repositories with ECA rules in order to support e-services. Active extensions to the XSLT [39] and Lorel [1] languages are proposed which handle insertion, deletion, and update events on XML documents — in contrast, we consider only insertion and deletion events here (see Section 2 below). It is proposed in [16] that analysis techniques developed for conventional active databases can be applied to these extended languages too but details are not given. [17] discusses 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.

A more recent paper [15] 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. [15] 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.

Other related work is [29, 33]. [29] discusses monitoring and subscription in Xyleme, an XML warehouse supporting subscription to web documents. A set of *alerters* monitor simple changes to web documents. A *monitoring query processor* then performs more complex event detection and sends notifications of events to a *trigger engine* which performs the necessary actions, including creating new versions of XML documents. The focus of this reactive functionality is highly tuned to this specific application.

[33] proposes extensions to the XQuery language [40] 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 [33] 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.

This paper is structured as follows. Section 2 defines a simple language for specifying ECA rules on XML documents. The language is illustrated by means of some examples, and its syntax and semantics are then specified more formally. Section 3 discusses techniques for analysing rule behaviour, including issues such as triggering and activation relationships between rules, and a more sophisticated analysis technique based on abstract interpretation. Section 4 discusses techniques for optimising the execution of our ECA rules. Section 5 gives our concluding remarks and outlines directions for further work.

2 The ECA Rule Language

An XML database consists of a set of XML documents. ECA rules on XML databases take the following form

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

We first illustrate our ECA rules by means of three examples. We then define their syntax and semantics more formally.

Example 1 The following XML database consists of one document which lists the books and magazines held by a bookstore, where the `books-in-stock` element summarises the ISBN numbers of books that are currently in stock i.e. whose `quantity-in-stock` value is greater than 0:

```
<bookstore>
  <books-in-stock>
    <isbn>0-436-20012-0</isbn>
    ...
  </books-in-stock>
  <book>
    <author>
      <first-name>John</first-name>
      <last-name>Coetzee</last-name>
    </author>
    <title>Age of Iron</title>
    <date><year>1990</year></date>
    <isbn>0-436-20012-0</isbn>
    <quantity-in-stock>3</quantity-in-stock>
  </book>
  <magazine>
    <title>The Economist</title>
    <date><day>24</day><month>June</month><year>2000</year></date>
  </magazine>
  <book>
    <author>
      <first-name>Nadine</first-name>
      <last-name>Gordimer</last-name>
    </author>
    <title>Something Out There</title>
    <date><year>1984</year></date>
    <isbn>0-670-65660-7</isbn>
    <quantity-in-stock>0</quantity-in-stock>
  </book>
  ...
</bookstore>
```

The following ECA rule updates the books-in-stock information in the above document after an insertion of one or more new books into the document:

```
on INSERT /bookstore/book
if $delta[quantity-in-stock > 0]
do INSERT $delta/isbn BELOW /bookstore/books-in-stock AFTER TRUE
```

Here, the system-defined $delta variable is successively bound to each of the newly inserted book nodes detected by the event part of the rule. The condition part of the rule checks for each new book whether its quantity-in-stock is greater than zero. If so, the action part of the rule inserts a copy of the new isbn node as a child of books-in-stock. The phrase AFTER TRUE indicates that new isbn node will be added after any existing children of books-in-stock.

The following ECA rule similarly updates the books-in-stock information after a deletion of one or more books from the document:

```
on DELETE /bookstore/book
if TRUE
do DELETE /bookstore/books-in-stock/isbn[.=$delta/isbn]
```

Here each isbn node child of books-in-stock whose value matches the isbn of a deleted book is itself deleted. ■

Example 2 Consider an XML database consisting of two documents, stores.xml and products.xml. The document stores.xml stores the amount of sales of each product in each store on each day:

```
<sales>
  <day date="d1">
    <store id="s1">
      <product id="p1">
        <amount>...</amount>
      </product>
      ...
    </store>
    ...
  </day>
...
</sales>
```

The document products.xml holds information on each product, including which stores have sold some amount of each product:

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

The following ECA rule updates the `products.xml` document whenever one or more new products (and their amounts sold) are added to `stores.xml`:

```
on INSERT document('stores.xml')/sales/day/store/product
if TRUE
do DELETE document('products.xml')/products/
          product[@id=$delta/@id]/store[@id=$delta/../@id];
   INSERT <store id='{$delta/../@id}'/>
          BELOW document('products.xml')/products/product[@id=$delta/@id]
          AFTER TRUE
```

The first action of the rule deletes from `products.xml` the store which is the parent of the inserted products in `stores.xml`, if this store already exists as a child of those products in `products.xml`. The second action then adds the store as a child of those products in `products.xml`. The net effect is that only one copy of the store remains below each product. ■

We note that, although Example 2 above uses an attribute with the name `id`, the fragment of XPath which we use in this paper does not include the `id()` function used to dereference attribute values of type `IDREF`. Therefore in our XPath fragment, we cannot traverse the graph structure of a document which uses attributes of type `ID` and `IDREF`.

Example 3 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>
  <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/.. BELOW 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>
    <ingredient>Mushroom</ingredient>
  </entree>
</entrees>
```

■

Having introduced our ECA rule language informally via the above examples, we now specify its syntax and semantics more precisely.

2.1 ECA Rule Syntax

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

$$\texttt{INSERT}\ e$$

or

$$\texttt{DELETE}\ e$$

where e is an XPath expression [38] 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 ECA rule (see below) and its set of instantiations is the set of new or deleted nodes returned by e.

The *condition* part of an ECA rule is either the constant `TRUE` or a conjunction of XPath expressions. It 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 `$delta` variable then it is evaluated once for each instantiation of `$delta` for each document. Otherwise, it is evaluated just once for each document.

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

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

These are executed on each XML document which has been 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 ECA rule's set of *candidate documents*. An ECA rule is said to *fire* if its set of candidate documents is non-empty.

Each $action_i$ is an expression of the form

$$\texttt{INSERT}\ r\ \texttt{BELOW}\ e\ [\texttt{BEFORE}|\texttt{AFTER}\ q]$$

or

$$\texttt{DELETE}\ e$$

where r is an XQuery expression [40], e is an XPath expression, and q is either the constant TRUE or an XPath qualifier (see below).

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[1].

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

In order to be able to apply our analysis and optimisation techniques (which are discussed in Sections 3 and 4), we assume that the XPath and XQuery expressions used in ECA rules are restricted to be *simple* ones, defined below. Extending our techniques to larger fragments of the XPath and XQuery languages is an area of further work.

The XPath fragment we consider here 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 (including `not`) other than `document()`. Thus, the syntax of a <u>*simple XPath expression*</u> e is given by the following grammar, where s denotes a string and n denotes an element or attribute name[2]:

[1] We observe that using the phrase `BELOW` 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.

[2] We underline here the terms that will be referred to again later in the paper.

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

So an XPath expression starts by establishing a context, either by a call to the `document` function (the only function allowed) followed by a path expression p or by a reference to the variable `$delta` (the only variable allowed).

Expressions enclosed in ‘`[`’ and ‘`]`’ in an XPath expression are called *qualifiers*. 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 *distinguished path* of the expression and the node at the end of the distinguished path the *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 (simple) *result type* of e, denoted $type(e)$, is one of *string*, an element name n, or `*`, where `*` denotes any element name. The result type can be determined as follows.

Let p be the distinguished path of e. If the leaf of p is `@`n or `@*`, $type(e)$ is *string*. If the leaf of p is n or `*`, $type(e)$ is n or `*`, respectively. If the leaf is ‘`.`’ or ‘`..`’, $type(e)$ can be inferred 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 ‘a`/`’ (where a must be either an element name or `*`’), then replace ‘a`/..`’ 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.

We term the path that results the *modified distinguished path*, and $type(e)$ is determined from the leaf of this modified distinguished path[3].

Example 4 Consider the condition from the ECA rule of Example 3, namely,

```
if $delta[.='Mushroom'] and $delta/../..[name='Baghdad Cafe']
```

The result type of the first conjunct is `ingredient` because the event part of the rule is

```
on INSERT document('g.xml')/guide/restaurant/entree/ingredient
```

The result type of the second conjunct is `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
```

■

The XQuery fragment we consider here disallows the use of full so-called FLWR expressions (involving keywords ‘for,’ ‘let,’ ‘where’ and ‘return’), essentially permitting only the ‘return’ part of an expression. The syntax of a *simple* XQuery expression r is given by the following grammar:

[3]The modified distinguished path is simply used to determine the result type; it may not be equivalent to the original path p.

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

Thus an XQuery expression r is either a XPath expression e 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.

The <u>*result type*</u> of an XQuery expression r, denoted <u>$type(r)$</u>, is a tree, each of whose nodes is of type n (for element name n), `*`, n`//*` or `*//*`. The types with a suffix `//*` indicate that the corresponding node can be the root of an arbitrary subtree. This is necessary to capture the fact that the results of XPath expressions embedded in r return sets of nodes which may be the roots of sub-documents. 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 r is an element constructor c, then we form a tree T from the element names and enclosed expressions in c (so we ignore strings and attributes). For each enclosed expression e', we determine its result type $type(e')$. If $type(e')$ is string, we delete e'; otherwise we replace e' in T by $type(e')$`//*`. Now $type(r)$ is given by the modified tree T. This tree can be represented as an XPath expression.

Example 5 Let r be the XQuery expression in the `INSERT` action of the rule in Example 2. Then $type(r)$ is clearly `store` (although a more precise type would be given by the XPath expression `store[@id]`).

As another example, let r be the XQuery expression `<a><b>{$delta/..}</b><c/></a>`, and assume that the result type of `{$delta/..}` is `*`. Then $type(r)$ can be represented by the XPath expression `a[b/*//*][c]`. ■

2.2 ECA Rule Execution Semantics

We specify the rule execution semantics using procedural pseudocode below. The input to the execution is an *XML database*, *db*, and a *schedule*, *s*. The database consists of a set of XML documents. The schedule consists of a list of *updates* to be executed on this database. Each such update is a pair $(a_{i,j}, docsAndDeltas_i)$. $a_{i,j}$ is an action from the actions part of some rule r_i. $docsAndDeltas_i$ is a set of pairs $(d, deltas_{d,i})$, where d is the identifier of a document upon which $a_{i,j}$ is to be applied and $deltas_{d,i}$ 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 (computed at the time that the rule's event and condition part were evaluated) — we call this set of instantiations the set of *deltas*.

Rules are identified by numbers `1..noOfRules` in order of decreasing priority (thus, in common with the SQL3 standard for database triggers [28], we assume here that no two rules have the same priority). Without loss of generality, we assume that the schedule which initiates the ECA rule execution consists of an action from one of the rules, and an initial set of documents and deltas upon which this action is to be applied, i.e. the initial schedule is a singleton of the form `[(a,docsAndDeltas)]`:

```
while s != [] do {
  (a,docsAndDeltas):= head (s);
  s                := tail (s);
  (deltas,db)      := updateDB (db,(a,docsAndDeltas));
  for i := noOfRules downto 1 do {
    docsAndDeltas := candidateDocs (i,deltas,db);
    if docsAndDeltas != {} then
      for j := noOfActions[i] downto 1 do
        s := (action[i,j],docsAndDeltas):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` at the head of the schedule `s` on the set of documents specified in `docsAndDeltas`. If `a` does not reference the `$delta` variable, the update is performed just once on each candidate document. If `a` does reference `$delta`, the update is performed once for each instantiation on each candidate document. `updateDB` returns a pair `(deltas,db)`. `db` is the new database resulting from this update. `deltas` is a two-dimensional array such that `deltas[i,d]` is the set of deltas corresponding to the event part of rule `i` with respect to document `d`.

The function `candidateDocs` determines the set of candidate documents for rule `i` with respect to the update just executed on the database by `updateDB`, together with the set of instantiations of the `$delta` variable for each candidate document. It does this by evaluating rule `i`'s condition on all documents `d` for which `deltas[i,d]` is non-empty. There are two cases to consider for each such document:

(i) The `$delta` variable occurs in the condition of rule `i`:

In this case the condition is evaluated once for each element of `deltas[i,d]`, and the subset of `deltas[i,d]`, D say, for which the condition evaluates to True (i.e. to a non-empty set of nodes) is determined. If D is non-empty, then `d` is returned as a candidate document and D is paired with it as the set of instantiations for `$delta` w.r.t. `i` and `d`. Otherwise, `d` is not returned as a candidate document.

(ii) There is no occurrence of `$delta` in the condition of rule `i`:

In this case the condition need only be evaluated once for document `d`. If it evaluates to True, `d` is returned, paired with the set `deltas[i,d]` as the set of instantiations for `$delta` w.r.t. `i` and `d`. Otherwise, `d` is not returned as a candidate document.

`noOfActions[i]` is the number of actions in the actions part of rule `i`, and `actions[i,j]` is the `j`'th action of rule `i`. The statement `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 `a[i,j]` is paired with the set `docsAndDeltas` consisting of the set of candidate documents for rule `i` and the set of deltas for each such document. Since rules are considered in increasing order of priority (in the statement `for i := noOfRules downto 1 do ...`), 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 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 — we discuss such techniques in Section 3 below.

Observations. There are a number of observations we can make regarding the above execution semantics:

- *Set-based updates*: In general `updateDB` will be undertaking a simultaneous set of updates on the database. For `INSERT` actions, this may result in non-determinism in the order in which a set of new sub-documents are inserted under a common parent since the `BEFORE/AFTER` construct only determines the ordering of new sub-document(s) with respect to existing sub-documents. It is an area of further work to extend our ECA language to capture ordering relationships between new sub-documents.

- *Coupling mode*: The rule coupling mode we have assumed here is `Immediate`, meaning that the actions of rules that have fired as a result of the current update are placed at the head of the current schedule. Other coupling modes are also possible in ECA systems e.g. `Deferred` and `Detached` (see [31]), and examining their applicability and impact in our context here is an area of further work.

- *Binding mode*: We have assumed that the instantiations of the `$delta` variable are determined at the time that the event part of a rule is detected. Again, there are other binding possibilities (see [31] for a discussion) and examining their implication in our context is an area of further work. Incidentally, the coupling and binding choices that we have made here correspond to those in the SQL3 standard for database triggers [28].

- *Document-level and instance-level triggering*: Our ECA language encompasses both document-level and instance-level triggering, by analogy to statement-level and row-level triggers in relational active databases [28]:

  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.

3 Analysing ECA Rule Behaviour

Analysis of ECA rules in active databases is a well-studied topic, with a number of approaches appearing in the literature e.g. [5, 6, 8, 10, 11, 12, 13, 14, 19], mostly in the context of relational databases. A key analysis question is that of *termination* of the rule execution, and a set of ECA rules is said to be *terminating* if for any initial event and any initial database state, the rule execution terminates.

Triggering and activation relations between rules have been used to determine whether a set of ECA rules is terminating:

A rule r_i *may trigger* a rule r_j if the action of r_i may generate an event which triggers r_j. The *triggering graph* [5, 6] 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.

An *activation graph* [13] also represents rules as vertices. In this case an arc between two distinct vertices r_i and r_j indicates that r_j's condition may be changed from False to True after the execution of r_i's action, while an arc from a vertex r_i to itself indicates that r_i's condition may be True after the execution of r_i's action. 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 previous methods. With this method, any vertex which 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 have been investigating the use of *abstract interpretation* [3, 30] for analysing termination of ECA rules [8, 10]. With this approach, the ECA rules are "executed" on an abstract database and abstract schedule, which represent a number of real databases and real schedules. Our abstract interpretation approach is a more costly but more precise technique than the graph-based approaches since these focus on triggering and activation relationships between *pairs* of rules whereas our approach considers the global relationships between rules, and how these relationships evolve during rule execution.

A natural question to ask is whether it is possible to re-use analysis techniques developed for ECA rules in relational databases in an XML setting by translating the set of XML documents and associated ECA rules into a relational form and then applying previous analysis techniques to these rules. The problem with this approach is that it may result in a significant loss of precision of the analysis, since the triggering graph is likely to become much less precise after transformation. For example, we can translate an XML document into a number of unary and binary relations such as relation $element(A)$, which states that A is an element node, relation $child(A, B)$, whose semantics are 'the element identified by A has the element identified by B as a child,' and relation $value(C, D)$, whose semantics are 'the data for the node identified by C is D' [34]. Relations *element*, *child* and *value* (along with others) are then used to store the XML document contents. For the insertion or deletion of element nodes, the ECA rules would now have the form 'on insertion/deletion into *child* if condition do action'. Consequently any triggering graph containing two or more such rules will always contain a cycle, since there is just one relation whose modification causes triggering.

For this reason, it is not sufficient to translate XML data and ECA rules into relational form for analysis, and new techniques need to be developed for conducting the analysis directly. We discuss some possible techniques below.

3.1 Triggering graphs for 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. For any insertion action a of the form

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

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 *ev is independent* of a.

Recall from Section 2 that $type(r)$ denotes the result type of r, which is a tree T in which each node is an element name n or `*`, with the leaf nodes possibly suffixed with `//*` indicating that the node may be the root of an arbitrary subtree. Let $path(T)$ be the set of paths in T and Q be the set of XPath queries

$$\{e_1/p \mid p \in path(T)\}$$

Then, the event ev is independent of the action a if the query e_2 is independent of the query set Q, that is, if for all possible XML documents d and queries $e \in Q$

$$e(d) \ \cap \ e_2(d) \ = \ \emptyset$$

(notice that the optional part of the insertion action, **BEFORE** | **AFTER** q, is not useful in the triggering analysis).

We can develop a safe test for independence, based on the distinguished paths of e and e_2, as follows.

Recall from Section 2.1 the construction of the *modified distinguished path*, say p, of an XPath expression e. The path p has no occurrence of `.` or `..`; it comprises only element names, attribute names, occurrences of `*` or `@*`, and the operators `/` or `//`. In addition, an attribute name or `@*` can occur only at the end of p since attributes cannot have children.

Let ev be an event and a be an action. Let Σ_e be the set of element names (interpreted as single symbols) and Σ_a be the set of attribute names (also interpreted as single symbols) occurring in the XML database, db, under consideration. A modified distinguished path p can be interpreted as a regular expression R_p over alphabet $\Sigma_e \cup \Sigma_a$ as follows. By overloading notation, let the *regular expression* Σ_e denote the alternation of all symbols in alphabet Σ_e and *regular expression* Σ_a denote the alternation of all symbols in alphabet Σ_a. Then R_p is formed from p by replacing each occurrence of `/` by concatenation, each occurrence of `*` by Σ_e, each occurrence of `@*` by Σ_a, and each occurrence of `//` by Σ_e^*. The distinguished path p matches a path p' of element names (possibly terminated by an attribute name) in a document in the XML database db if and only if p' (interpreted as a string over $\Sigma_e \cup \Sigma_a$) is in $\mathcal{L}(R_p)$ (the language denoted by R_p).

Let e be one of the XPath expressions in set Q for action a of rule r_i and e_2 be the XPath expression in event ev of rule r_j, as above. Let p and p_2 be the modified distinguished paths of e and e_2, respectively. Then we have that

$$\mathcal{L}(R_p) \ \cap \ \mathcal{L}(R_{p_2}) \ = \ \emptyset$$

implies that for all possible XML documents $d \in db$

$$e(d) \ \cap \ e_2(d) \ = \ \emptyset$$

Testing for the empty intersection of regular languages can be done using standard techniques [25]. If for all $e \in Q$ the intersection of $\mathcal{L}(R_p)$ and $\mathcal{L}(R_{p_2})$ is empty, we can conclude that e_2 is independent of Q. If not, then rule r_i may trigger rule r_j and an arc should be inserted into the triggering graph.

Since this test considers only the distinguished paths of e and e_2, it is a conservative test and it may be the case that r_i will not actually trigger r_j in any database state. For example, consider event ev:

```
INSERT /b/c[d]
```

and action a:

```
INSERT <c/> BELOW /b
```

The set of queries Q derived from action a comprises the single query `/b/c`. This is identical to the distinguished path of the XPath expression in event ev, and so the method described above would conclude that a rule containing action a may trigger a rule containing event ev. However, we know the exact structure of what has been inserted below `/b` by action a, namely, that it is an element `c` with no children. Therefore this action cannot trigger ev which requires that `c` has a `d` child. A more precise test could detect this, and developing such tests is an area of further research.

For testing whether DELETE actions may trigger DELETE events, for any deletion action a of the form

$$\texttt{DELETE } e_1$$

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

$$\texttt{DELETE } e_2$$

belonging to a rule r_j, we view e_1 as an XQuery expression and derive the query set Q from e_1 as for the insertion case above, except that in this case, because e_1 is simply an XPath expression, Q will be a singleton set containing just $e_1//*$. We then have that r_i may trigger r_j if e_2 is not independent of Q.

Once the triggering graph between a set of ECA rules has been constructed, acyclicity of this graph implies definite termination of the rules.

3.2 Activation Graphs for 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 there is an activation arc from r_i to 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 there is an activation arc from r_i to itself.

The following table illustrates the transitions that the truth-value of a condition can undergo, in the absence of negation in rule conditions. The first column shows its truth value of before the update, and the subsequent columns its truth value after a non-independent insertion (NI) and a non-independent deletion (ND):

before	after NI	after ND
$True$	$True$	$True$ or $False$
$False$	$True$ or $False$	$False$

For case (a), it is clear from this table that, in the absence of negation in rule conditions, 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 $c = c_1$ **and** $\ldots$ **and** c_n, where each c_i is an XPath expression. For c to be True, we require that each c_i, $1 \leq i \leq n$, returns a non-empty result. Thus, in a sense, the distinguished path of each c_i plays the same role as an XPath qualifier in that we are only 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_i can turn c_i from False to True. For example, the insertion of a `c` element below `/a/b` can turn condition `/a/b[c]/d` from False to True. Thus we need to consider all the qualifiers and the distinguished path in c_i in a similar way in any test for (a) above.

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 $C = \{c_1, \ldots, c_n\}$, the set of conjuncts appearing in c. We now proceed to decompose each conjunct in 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 w, and v an element name n or `*`,
   - if condition c_i is of the form uv`/..`w, then replace c_i in C by u`./[`v`]`w;
   - if c_i is of the form uv`//..`w, then replace c_i in C by u`./[`v`]`w and uv`//*[*]`.
2. We can delete from c_i steps of the form `/.` and `./`, 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 $u[v]$ and uw.
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 $x\ o\ y$, then if x or y match 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 from rule r_i be

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

As described in Section 3.1, we again construct a query set Q from e_1 and r. We now use the test of regular expression intersection in order to check whether every $e \in Q$ is independent of each of the conditions in C. If so, then Q cannot change the truth value of c from False to True. Otherwise, Q is deemed to be non-independent of c and we insert an activation arc from r_i to r_j. This again is a conservative test, and it is an area of further research to develop more precise tests.

Example 6 Let c be the condition `$delta[quantity-in-stock > 0]` in the first ECA rule in Example 1. The set of conditions C constructed by the decomposition procedure described above comprises only a single condition c', namely, `$delta/quantity-in-stock`. The modified distinguished path of c' after substituting for `$delta` is

`/bookstore/book/quantity-in-stock`

Thus, the only insertion actions that can activate c' are ones which

- insert below `/bookstore/book` a `quantity-in-stock`,
- insert below `/bookstore` a `book` with a `quantity-in-stock` child, or
- insert below `/` a `bookstore` with a `book` child which has a `quantity-in-stock` child.

In each of these three cases, the set Q of queries generated from the insertion action will contain a query which, when interpreted as a regular expression, will be equal to c' interpreted as a regular expression. This means that the condition and insertion are possibly not independent and an activation arc should be added from the rule containing the insertion action to the rule containing condition c. ■

Example 7 Let c be the condition

`$delta[.='Mushroom'] and $delta/../..[name='Baghdad Cafe']`

from the ECA rule in Example 3. After substituting for `$delta` (and dropping `document('g.xml')` for simplicity), set C initially comprises conditions

$$c_1 = \texttt{/guide/restaurant/entree/ingredient}$$

and

$$c_2 = \texttt{/guide/restaurant/entree/ingredient/../../name}$$

Condition c_2 is further decomposed into

$$\begin{aligned} c_3 &= c_1 = \texttt{/guide/restaurant/entree/ingredient} \\ c_4 &= \texttt{/guide/restaurant/entree} \\ c_5 &= \texttt{/guide/restaurant/name} \end{aligned}$$

Conditions c_3, c_4 and c_5 can be interpreted as stating that the only way an insertion can change condition c from False to True is if an `ingredient` is inserted as a child of an `entree`, an `entree` is inserted as a child of a `restaurant`, or a `name` is inserted as a child of a `restaurant`. Once again, the test described above will detect these possibilities and will correctly insert the necessary activation arcs. ■

For case (b) above, a rule r_i activates itself if it may leave its own condition True. From the above table, we see that with the analysis that we have used so far this will be the case for all rules. To obtain more precision, we need to develop the notion of *self-disactivating* rules, by analogy to this property of ECA rules in a relational database setting [12]. A self-disactivating rule is one where the execution of its action makes its condition False.

In the absence of negation in rule conditions, the only possibility for a rule to be self disactivating is if all its actions are deletions which subsume one or more of the conjuncts $c_1, \ldots, c_n$ in its condition. For each deletion action

$$\texttt{DELETE}\ e_1$$

let e be the query $e_1//*$. Then we need to test if

$$e \supseteq c_i$$

for some $1 \leq i \leq n$.

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 [22, 24, 36]. 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 [22]. 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.

Once the activation graph between a set of ECA rules has been constructed, acyclicity of this graph implies definite termination of the rules. The triggering and activation graphs can also be combined and the rule reduction method employed.

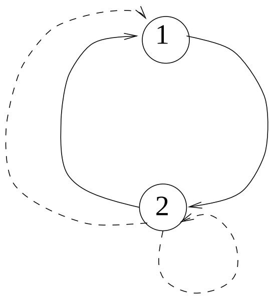
3.3 Abstract Interpretation of Rule Execution

The abstract interpretation approach to ECA rule analysis we discussed in [8, 10] can also be applied in the present XML setting. This approach generalises graph-based techniques since these focus on triggering and activation relationships between pairs of rules whereas our approach is more "global" in nature. To illustrate, consider the following pair of rules:

```
rule 1: on INSERT /a/*
        if /a/b and /a/c
        do DELETE /a/b

rule 2: on DELETE /a/*
        if TRUE
        do INSERT <c/> BELOW /a
```

It is easy to see that the triggering and activation graph for these rules is as follows, where the solid arcs are triggering ones and the broken arcs are activation ones:



Triggering analysis, activation analysis, and the rule reduction method are thus all inconclusive with this rule set. With our abstract interpretation approach (see [8] for details), the rule termination test consists of running the abstract rule execution once for each possible initial singleton abstract schedule, with an initial abstract database in which all non-trivial conditions have value *Unknown* (i.e. may be *True* or *False*). If all invocations of the abstract execution terminate, then definite termination of the set of ECA rules can be concluded. Otherwise, the set of rules is deemed to be possibly non-terminating.

With the above two rules, given an initial abstract schedule consisting of the action of rule 1, a trace of the abstract execution of these rules is as follows, where c_i denotes the condition query of rule i and a_i the action of rule i:

Iteration	c_1	c_2	Schedule
1	$Unknown$	$True$	$[a_1]$
2	$False$	$True$	$[a_2]$
3	$False$	$True$	$[]$

"Execution" of rule 1's action, a_1, at iteration 1 causes its condition, c_1, to become False, because the rule is self-disactivating. After the execution of rule 2's action, a_2, at iteration 2, c_1 remains False because a_2 can only make True the second conjunct of c_1 and the prior execution of a_1 has made False the first conjunct. At iteration 3, the falsity of c_1 means that rule 1 cannot fire at this point. We can therefore conclude that if rule 1 is the first rule triggered then rule execution will definitely terminate within 3 iterations.

Similarly, given an initial abstract schedule consisting of a_2, a trace of the abstract execution of the rules is as follows:

Iteration	c_1	c_2	Schedule
1	$Unknown$	$True$	$[a_2]$
2	$Unknown$	$True$	$[a_1]$
3	$False$	$True$	$[a_2]$
4	$False$	$True$	$[]$

This time, execution of a_1 at iteration 2 causes c_1 to become False, and at iteration 4 rule 1 cannot fire. We can therefore conclude that if rule 2 is the first rule triggered then rule execution will definitely terminate within 4 iterations. Overall, we can therefore conclude that this is a terminating rule set.

4 Optimising ECA Rule Execution

A number of techniques can be used for optimising the execution of our XML ECA rules:

(i) The triggering relationships between rules derived using the techniques discussed in Section 3.1 can be used at run-time in order to optimise event detection in the `updateDB` function described in Section 2.2.

In particular, if it is known that the action `a` just applied to the database by `updateDB` cannot trigger a rule `i`, then there is no need to evaluate rule `i`'s event query on the new database and the set `delta[i,d]` can be set to empty for all documents `d`.

(ii) By dynamically maintaining the activation relationships between rules as rules execute, as described in Section 3.3 above, it is also possible to avoid evaluating condition queries that are currently known to be definitely True or False. This optimisation would modify the `candidateDocs` function described in Section 2.2.

(iii) The individual expressions in the condition and action parts of rules can be optimised by using recent results on equivalence and minimisation of XPath/XQuery expressions [7, 37]. Further optimisations are possible if documents are known to satisfy various constraints by conforming to a schema or document type definition [7, 22, 37].

(iv) More "global" optimisation of rule conditions and actions is possible by using abstract interpretation to predict possible sequences of rule actions that may execute from the current database state, and then producing a specialised version of the rule execution semantics for each possible such sequence. This technique is described in [9] in a conventional active database setting, but it can be applied equally well in our XML setting here.

The technique gives the opportunity to optimise rule execution for specific sequences of rule actions, for example by abstracting common sub-queries from the set of queries that it is known will need to be evaluated. The technique can be used both statically, i.e. at rule compilation time, and dynamically, during rule execution.

5 Conclusions

We have proposed a simple ECA rule language for providing reactive functionality on XML databases, and have developed new techniques for analysing the triggering and activation dependencies between XML ECA rules. By combining these new techniques with recent theoretical results regarding containment and equivalence of XPath/XQuery expressions, and with analysis and optimisation approaches for ECA rules that were developed for conventional databases, we have thus proposed a number of techniques for analysing and optimising the behaviour of our XML ECA rules.

To our knowledge, this is the first time that such techniques have been proposed for ECA rules on XML data. The determination of triggering and activation relationships between ECA rules is more complex in an XML setting than for relational databases, because determining the effects of rule actions is not simply a matter of matching up the names of updated relations with potential events or with the bodies of rule conditions. Instead, the associations are more implicit and semantic comparisons between sets of path expressions are 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 invisage detecting updates of much finer granularity. Also, using active rules allows one to update a document directly, wheareas XSLT requires a new result tree to be generated by applying transformations to the source document.

The work that we have described here has opened up many directions for further work:

- exploring the practical effectiveness of our tests for independence of queries in triggering analysis and activation analysis, and investigating the possibility of developing more precise tests;
- exploring the decidability of independence and containment for larger fragments of XPath, and developing conservative approximations in cases where these properties are undecidable;
- extending the expressiveness of the various parts of our ECA rules, for example to support negation in rule conditions;

- incorporating information about document types in our treatment, in order to get more precise analysis information and more possibilities for rule optimisation;
- investigating other coupling modes and other binding modes for our ECA rules;
- developing analysis techniques for other properties of our ECA rules e.g. *confluence* of the rule execution if rules of the same priority are permitted.

References

[1] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J.L. Wiener. Incremental maintenance for materialized views over semistructured data. In *Proc. 24th Int. Conf. on Very Large Databases*, pages 38–49, 1998.

[2] S. Abiteboul, V. Vianu, B.S. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *JCSS*, 61(2):236–269, 2000.

[3] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[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. Behaviour of database production rules: Termination, confluence and observable determinism. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 59–68. ACM Press, 1992.

[6] 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.

[7] S. Amer-Yahia, S. Cho, L.V.S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 497–508, 2001.

[8] 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.

[9] J. Bailey, A. Poulovassilis, and C. Courtenage. Optimising active database rules by partial evaluation and abstract interpretation. In *Proc. 8th Int. Workshop on Database Programming Languages*, Rome, 2001.

[10] 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*, London, 2000.

[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-Verlag, 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] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *Proc. ICDE 2002. To appear*, 2002.

[16] A. Bonifati, S. Ceri, and S. Paraboschi. Active rules for XML: A new paradigm for e-services. *VLDB Journal*, 10(1):39–47, 2001.

[17] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing reactive services to XML repositories using active rules. In *Proc. 10th World-Wide-Web Conference*, 2001.

[18] S. Ceri, R. Cochrane, and J. Widom. Practical applications of triggers and constraints: Success and lingering issues. In *Proc. 26th Int. Conf. on Very Large Databases*, pages 254–262, 2000.

[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. 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.

[22] A. Deutsch and V. Tannen. Containment and integrity constraints for XPath fragments. In *Proc. 8th Int. Workshop on Knowledge Representation Meets Databases*, 2001.

[23] M.F. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middle-ware queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 103–114, 2001.

[24] D. Florescu, A.Y. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Proc. 17th ACM Symp. on Principles of Databases Systems*, pages 139–148. ACM Press, 1998.

[25] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[26] H. Ishikawa and M. Ohta. An active web-based distributed database system for e-commerce. In *Proc. Web Dynamics Workshop, London*, 2001.

[27] A. Kotz-Dittrich and E. Simon. Active database systems: Expectations, Commercial Experience and Beyond. In N. Paton, editor, *Active Rules in Database Systems*, pages 367–404. Springer–Verlag, 1999.

[28] 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–Verlag, 1999.

[29] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 437–448, 2001.

[30] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[31] N. Paton, editor. *Active Rules in Database Systems*. Springer-Verlag, 1999.

[32] 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.

[33] 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.

[34] P. Wadler. A formal semantics of patterns in XSLT. In *Markup Technologies 99*, 1999.

[35] J. Widom and S. Ceri. *Active Database Systems*. Morgan-Kaufmann, San Mateo, California, 1995.

[36] P.T. Wood. On the equivalence of XML patterns. In *Proc. 1st Int. Conf. on Computational Logic (DOOD stream), LNCS 1861*, pages 1152–1166, 2000.

[37] P.T. Wood. Minimising simple XPath expressions. In *Proc. WebDB 2001: Fourth Int. Workshop on the Web and Databases (Santa Barbara, Calif., May 24–25)*, pages 13–18, 2001.

[38] World Wide Web Consortium. XML Path Language (XPath), Version 1.0. See `http://www.w3.org/TR/xpath`, November 1999. W3C Recommendation.

[39] World Wide Web Consortium. XSL Transformations (XSLT), Version 1.0. See `http://www.w3.org/TR/xslt`, November 1999. W3C Recommendation.

[40] World Wide Web Consortium. XQuery 1.0: An XML Query Language. See `http://www.w3.org/TR/xquery`, June 2001. W3C Working Draft.