ActiveXQBE: A Visual Paradigm for Triggers over XML Data

Daniele Braga[1], Alessandro Campi[1],
Davide Martinenghi[2], and Alessandro Raffio[1]

[1] Politecnico di Milano
Dip. di Elettronica e Informazione
p.zza L. da Vinci 32, 20133 Milano, Italy
{braga,campi,raffio}@elet.polimi.it

[2] Free University of Bozen/Bolzano
Faculty of Computer Science
p.zza Domenicani, 3, 39100 Bolzano, Italy
martinenghi@inf.unibz.it

Abstract. While XQuery is becoming a standard, the W3C is currently discussing the features of an update language for XML, and its requirements. Therefore, time is ripe for designing and defining the language features and extensions that are usually needed when updates are supported: reaction policies to constraint violations, business rules, and more. In the past years, several languages have been proposed for updates as well as for triggers in XML, such as *XUpdate* and *Active XQuery*.
In this paper, we propose a visual approach to the formulation of active rules building on XQBE, a graphical query language for XML data. Our approach is motivated by the need to provide unskilled users with the ability to express business rules in an intuitive fashion. Visual triggers are then translated into statements that can be interpreted by query engines.

1 Introduction

According to a well-known classification [6], data semantics can be represented declaratively under the form of *normative rules*, *constructive* (or passive) *rules*, and *reactive* (or active) *rules*.

Normative rules, also known as *integrity constraints* in the terminology of databases, are properties that the data must always satisfy. In the context of XML data, some forms of integrity constraints can be expressed through schema definition languages, such as DTDs and XML Schema. Although some attempts exist (e.g., [3]), a universally accepted paradigm for specifying general constraints (in the sense of SQL assertions) still seems to be missing for XML.

Constructive rules (*views* in databases) allow one to specify how to derive new data from data already available. Integrity constraints can also be expressed in this form and, thus, evaluated as constructive rules. Again, there is no standard for the specification of constructive rules in XML, although some support

to views is intrinsically available in XQuery. The issues concerning the materialization and maintenance of views in XML are discussed, for instance, in [1].

Finally, reactive rules specify how data should change depending on the current state of the store and, possibly, on events.

The above rules eventually serve the purpose of formally specifying process flows or business level requirements of the system to be described. According to good design principles, these so-called *business rules* should be expressed as part of the schema, so that the knowledge they carry is decoupled from the rest of the application. Business rules are used to describe the operations and constraints that apply to organizations. As such, they should be business owned and oriented and should be specified in the easiest and most intuitive way, so as to appeal to the broadest audience. Ideally, it should be possible to allow users to maintain the rules without the intervention of an IT professional. In this regard, an intuitive visual paradigm can simplify the specification, via user-friendly interfaces, of essential data management features and policies, such as queries and updates, of requirements of compliance with data constraints, and of consistency repairing actions that ought to take place upon violations of constraints. Complex rules are best specified by domain experts, who, however, may lack knowledge in data definition and manipulation languages. A number of applications can be envisaged, where XML is already established as the *de facto* data representation model, e.g., in the medical domain, where health care professionals sharing information stored in clinical records may need to impose constraints, say, on treatments and compatibility between medicines and patients' profiles.

The ActiveXQBE paradigm presented in this paper stands out from previous attempts for incorporating a visual approach with an emphasis on usability and intuitiveness, yet without heavily sacrificing generality and expressiveness. In particular, to achieve these goals, we designed and propose the visual tool ActiveXQBE for the specification of active rules, building on XQBE [5], a graphical query language for XML. XQBE as well as ActiveXQBE are based on annotated trees, so as to adhere to the hierarchical nature of the XML data model. Both XQBE and ActiveXQBE have a quite steep learning curve; although no formal tests were performed, our experience with under-graduated students has shown that a couple of lessons are enough to get acquainted with the visual paradigm.

Visual ActiveXQBE triggers can be translated into textual representations to be executed by external rule engines. To this end, we provide an algorithm for translating visual triggers into *Active XQuery* rules [4].

2 XQBE: a visual XML query language

XQBE (XQuery By Example) [5] is a graphical query language for XML designed to be intuitive and capable of running on top of XQuery engines. XQBE includes most of the expressive power of XPath, allows for arbitrarily deep nesting of XQuery FLWOR expressions, supports the construction of new XML elements, and permits to restructure existing documents. Figure 1 shows a query reading "*List books published by Addison-Wesley after 1991, including their year*

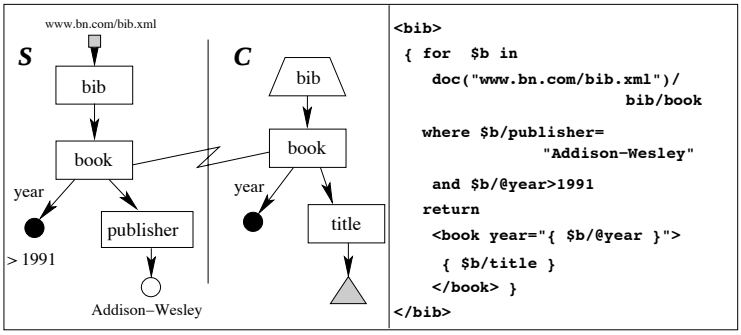


Fig. 1. XQBE and XQuery versions of the same query

and title". All XQBE queries have a vertical line in the middle, separating the *source* part (on the left) from the *construct* part (on the right); both parts contain labeled graphs that represent XML fragments and express properties of such fragments: the source part specifies the properties of the XML data to be included in the result, while the Construct Part describes the structure of the result in terms of newly generated XML items, together with the projection of the elements extracted from the source documents. The correspondence between the components of the two parts is expressed by means of explicit *binding edges* across the vertical line; these edges connect the nodes of the Source Part to the nodes that will take their place in the output document. The paths that branch out of a bound node in the Construct Part indicate which of its sub-items are to be retained, thus "projecting" the node. All the XML *elements* in the Source Part of the target documents are represented as labeled rectangles; *attributes* are represented as black circles, with the *attribute name* on the arc between the rectangle and the circle; *PCDATA content* is represented as an empty circle. Attribute and PCDATA nodes in the Source Part may be labeled, so as to express conditions on the values they represent.

In Figure 1 the Source Part matches all the `book` elements with a `year` attribute whose value is greater than 1991 *and* a `publisher` sub-element whose PCDATA content equals "Addison-Wesley". By means of a binding edge, the selected `book`s are sent to the Construct Part, where only the `title` and publication `year` are retained. Note the use of the triangular node to project the entire fragment rooted in the `title` element. The binding edge between the `book` nodes states that the query result shall contain *as many* `book` elements *as* those matched in the Source Part. Rectangular nodes in the Construct Part represent XML elements in direct correspondence with some XML data extracted from the source documents (either by a direct binding or by projection and renaming). Elements with no counterpart in the source documents are rendered by means of trapezoidal nodes, such as the `bib` node in the Construct Part of the example.

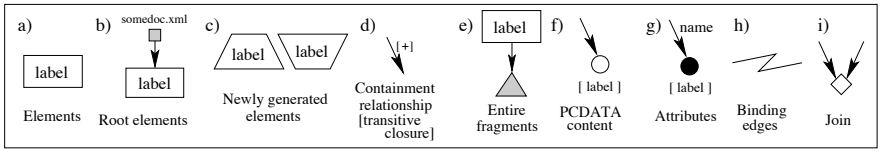


Fig. 2. Summary of the core XQBE constructs

Other nodes allow one to express more complex queries with joins, aggregates, sorting, negation and more. Figure 2 shows all the defined constructs. For a full and formal description of the language, refer to [5].

3 ActiveXQBE

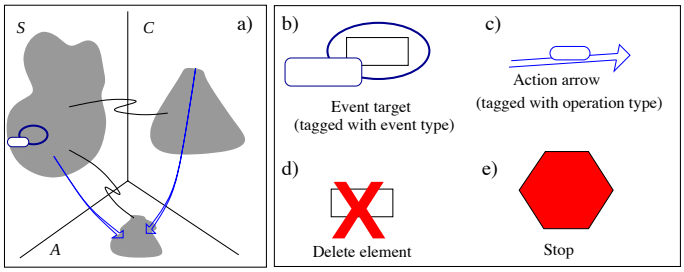


Fig. 3. Summary of the ActiveXQBE constructs

ActiveXQBE extends XQBE: while XQBE uses two regions to extract data and construct results, ActiveXQBE uses three regions (Figure 3a):

$\mathcal{S}$: the region on the left is the *Source Part* ($\mathcal{S}$); the graphs in this region locate the XML nodes on which the triggering events are defined, as well as the conditions that apply to such nodes for the rules to be triggered. Any valid XQBE source graph is allowed in $\mathcal{S}$; besides, one (and only one) node in this region *must* be tagged as the node on which the event occurs (the *event node*). One *action arrow* can go out of a node and point to an element in the action part $\mathcal{A}$, described below.

$\mathcal{C}$: the region on the right is the *Construct Part* ($\mathcal{C}$); the trees contained in this region, when present, define the constructed data structures to be inserted into suitable documents (as defined by the third region $\mathcal{A}$); such insertions may implement the actions of the active rules. If $\mathcal{C}$ is not empty, an *action arrow must* connect the root of the tree in $\mathcal{C}$ and a node in $\mathcal{A}$. Any regular XQBE construction tree is allowed in $\mathcal{C}$.

$\mathcal{A}$: a new region, the *Action Part* ($\mathcal{A}$) is placed below $\mathcal{S}$ and $\mathcal{C}$, where the action of the trigger is represented. ActiveXQBE supports different kinds of

```
<dept>                                  <!ELEMENT dept
 <budget>1000000</budget>                      (budget, manager, emp*)>
 <manager>                              <!ELEMENT manager
  <name>Smith</name>                          (name, salary, numOfEmps)>
  <salary>10000</salary>                <!ELEMENT emp (name, salary)>
  <numOfEmps>9</numOfEmps>              <!ELEMENT name (#PCDATA)>
 </manager>                             <!ELEMENT numOfEmps (#PCDATA)>
 <emp> <name>Jones</name>               <!ELEMENT salary (#PCDATA)>
       <salary>8000</salary> </emp>
</dept>
```

Fig. 4. An XML document and its DTD schema

actions: insertions, updates, deletions and denials. If the action is a denial of the operation, a *stop sign* is placed in this region (Figure 3e). For the other three kinds of action, a tree in $\mathcal{A}$ expresses the selection of the XML fragments which must be affected. Such tree can be either rooted in a *Root Element*, or in a *Rectangular Element* bound to a node in $\mathcal{S}$. If the action of the rule is an update or an insertion, then the target of the action is the node reached by the action arrow (described below), which is tagged with the type of operation. If the action is a deletion, then a red cross identifies the element to be deleted. ActiveXQBE triggers can only perform one action, so at most one action arrow can reach $\mathcal{A}$, either originating from $\mathcal{S}$ or $\mathcal{C}$.

Figure 3b shows the syntax for specifying the target of the event: a blue oval surrounds the target node (which can be either an Element, an attribute or a PCDATA node), and a tag specifies the type of event.

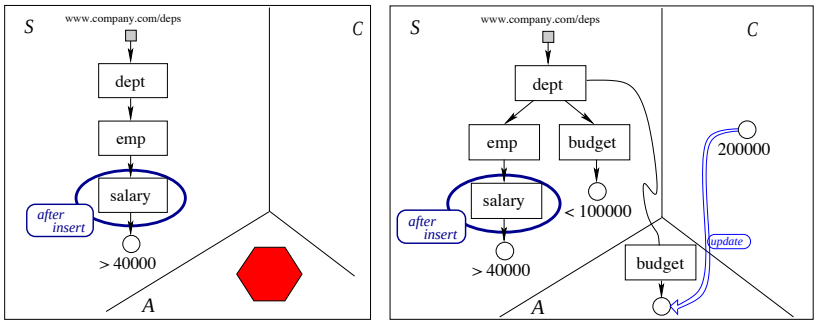
The syntax for actions is shown in Figure 3c, d, and e: an *action arrow* is used to *insert* or *update* an element with data extracted from the original document, or built on purpose. The arrow starts from the element to be inserted and reaches the element that will be updated (or below which the new item will be inserted). The tag on the arrow specifies the kind of action to be performed (insert-before, insert-after[3], update). A *red cross* is used to mark a node that must be deleted. A *stop sign* is used to indicate that the event should be prevented. In other words, if the trigger is evaluated *before* the event, then the action is stopped so that the update that activates the trigger is not even executed; if the trigger is evaluated *after* the event, then a rollback is performed that cancels the effect of the event itself.

In order to describe our approach, we show some triggers that apply to an XML data set exemplified in Figure 4.

Example 1 (Rollback).

As a first example, consider the trigger of Figure 5(a), which blocks insertion of salaries greater than 40000$ to an employee. In this example the Construct Part is empty, since there is no need to build any new elements; the symbol in the action part states that the insertion is to be undone.

[3] As in the XML update language proposal XUpdate [16], the "-before" and "-after" suffixes are to be interpreted with respect to the position of the pointed node.



(a) (b)

Fig. 5. (a) Trigger 1: denying too high salaries (b) Trigger 2: updating budget

The `salary` node is the subject of the event: as stated by the label attached to the oval surrounding it, the trigger is evaluated *after* a `salary` node is inserted as a child of `emp`. Then, the action (in this case, a rollback of the insertion) is performed only when all the conditions of the source graph apply, i.e., only when the value of the `salary` is greater than 40000$. Note that the trigger would also be evaluated after the insertion, for instance, of an `emp` element, because a `salary` sub-element would be inserted too.

Example 2 (Update).

The example of Figure 5(b) extends the previous one with the addition of a condition and an action; in particular, when a salary greater than 40000$ is inserted for an employee belonging to a department with low budget (say, less than 100000$), this budget is updated to 200000$.

Note that the `dept` element in the action part is bound to the `dept` element in $\mathcal{S}$: the binding edge toward the action part transfers the context from $\mathcal{S}$, so, when a high salary is set to an employee, only the budget of his department is updated. The Construct Part of this trigger is used to build a new constant value of 200000$, that is used to update the budget of the department by means of an action arrow.

Example 3 (Adding complexity).

In Figure 6 we compare the newly inserted salary of an employee with the manager's salary. If the manager's salary is lower, then the employee's salary is updated to be the same as the manager's. The peculiarities of this trigger are the *join node* used to compare salaries (the small rhombus in Figure 6) and the action arrow, which comes from the Source Part. The Construct Part is unused.

Note that the action updates the newly inserted salary (i.e., the salary is removed and inserted again). This causes the trigger to be evaluated again, but the second time the comparison between salaries definitely fails, since the

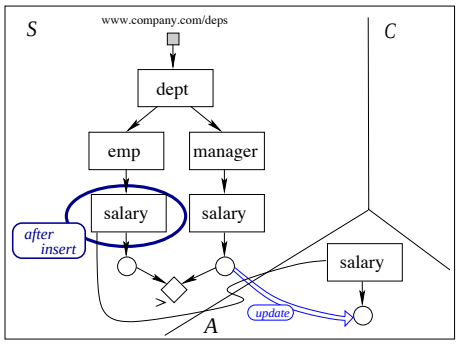


Fig. 6. Trigger 3: updating salary

contents of the elements selected by the paths `emp/salary` and `manager/salary` are equal.

4 Graph translation algorithm

ActiveXQBE triggers are not interpreted by a specific engine; rather, they are translated into *Active XQuery* triggers and evaluated by existing rule engines.

As with many declarative languages, there are many ways to express the same operation in Active XQuery. Here we first define a *canonical Active XQuery form* and then show how to translate an ActiveXQBE graph into such form. Note that every well-formed ActiveXQBE graph can be translated into canonical Active XQuery, while nothing can be stated about the contrary.

Active XQuery triggers comply with the following syntax:

```
1. CREATE TRIGGER Trigger-Name [WITH PRIORITY Signed-Integer-Number]
2. (BEFORE|AFTER) (INSERT|DELETE|REPLACE|RENAME) OF
      XPathExpression (,XPathExpression)*
3. [FOR EACH (NODE|STATEMENT)]
4. [XQuery-Let-Clauses]
5. [WHEN XQuery-Where-Clause]
6. DO XUpdate-UpdateOp
```

The trigger can be divided into five main blocks: lines 1 and 3 are a sort of *header*, where the name and priority[4] of the trigger are defined, as well as its granularity[5]; line 2 describes the triggering *event*; in line 4 the *variables* useful to express conditions and actions can be defined; line 5 contains the trigger

[4] When the same event fires multiple rules, their actions are executed in priority order; within the same priority level, an implicit creation order is followed. This behavior is independent of ActiveXQBE, since it is enforced by the Active XQuery rule engine.

[5] I.e., whether the trigger must react on a per-statement basis, or once for each node affected by the event. In the latter case, the corresponding `$new` and `$old` variables will be available.

conditions; line 6 describes the *action* to be performed when the rule is triggered, which can either be an update statement written in XUpdate.

We say that a trigger is in *canonical form* when (a) the triggering event is monitored for target nodes which are identified by a *single path expression* and (b) the triggering conditions are expressed by stating the *non-emptiness* of *node sequences*, defined by means of *XQuery let-clauses*. Any trigger can be expressed in canonical form. We define the semantics (and perform the translation) of ActiveXQBE graphs in terms of canonical Active XQuery triggers. The translation algorithm operates with the steps described below.

Source graph reduction. The first step is to locate and mark some nodes in the *source* graph which are said to be *relevant*. These nodes are the **event target** (the node on which the event occurs); the nodes with **binding edges**, both towards the construct and the action part; the nodes involved in an **action**. A variable will be generated for each such relevant node in the textual counterpart of the trigger.

Variable generation. The event target represents the node on which the event occurs and is implicitly associated to a variable named *$new* or *$old*, according to the type of the event; this straightforwardly corresponds to the notion of "transition data" in SQL triggers. This variable, available for use in any part of the trigger, implicitly and automatically fixes the context of evaluation for other expressions.

All the other *relevant* nodes will be associated to a variable by means of suitable *let clauses*, which in turn can be used within other let-clauses, when-conditions, and during action specification (see below). The construction of such let clauses relies on topological analysis of the nodes surrounding the relevant ones.

As the Source Part of an XQBE query is always a directed acyclic graph, a partial order is implicitly defined over the set of relevant nodes. The nodes are considered in an order which is compliant with such partial order (namely, as encountered in a left-pre-ordered traversal that starts from the *initial* nodes, considered in left-to-right order). Each relevant node is associated to a path expression which corresponds to a path in the graph. For the first nodes considered, such path expression is rooted in one of the initial nodes (those corresponding to root nodes in the XML documents). For the other nodes such paths are rooted in the nearest ancestor chosen between the already considered relevant nodes (which are all already bound to a variable). The selection conditions of the XQuery statement that constitutes the body of the let clause associated to each variable (namely: the conjuncts of a where clause) are generated considering all the nodes reachable from the node currently under consideration, up to the not yet visited relevant nodes. In other words, starting from each relevant node, a set of conditions is generated, considering a subgraph that contains the current relevant node and is limited by the other relevant nodes. Some of the considered branches may have bifurcations at some point; in this case, an internal variable is generated for the node with the bifurcation, which is local to one specific let

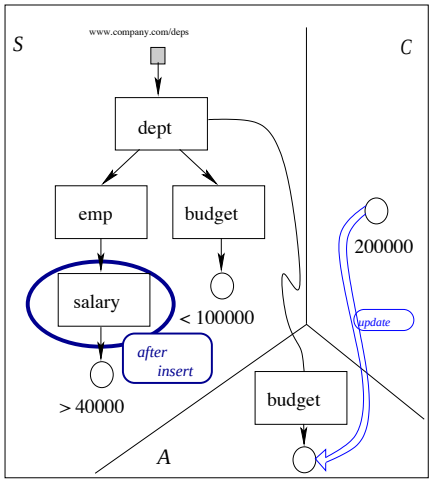
clause. Nodes with bifurcations, though, are not considered as relevant, and they are not involved in any separated let-clauses.

WHEN clauses. All the conditions on the triggering of the rule, as represented by the query structures, have already been considered in the variable generation process; therefore, the WHEN clause can be reduced to a conjunction of non-emptiness statements over the node sequences returned by the previously defined let-clauses. Note that only the conjunction of *all* these statements ensures that all the conditions expressed in the Source Part will be correctly evaluated.

Action specification. The last part of the trigger is the action to be performed. The Active XQuery language allows for any update language inside the `DO` block. Here, we choose XUpdate.

If the action part of ActiveXQBE contains special directives, such as a ROLLBACK, the corresponding textual command is simply generated within the action block. In general, when the action part contains a graph, an XUpdate variable is generated to select the target elements; moreover, if the action is not a deletion, another variable is generated to describe the elements to be inserted or updated. These elements are obtained either from the source or the Construct Part, based on the origin of the action arrow.

Figure 7 shows the trigger of figure 5(b) and its translation into Active XQuery.




```
CREATE TRIGGER t
AFTER INSERT OF
  doc("...")/dept/emp/salary
FOR EACH NODE
let $dept_s := (
  for $dept in doc("...")/dept
  where $dept/emp/salary = $new
  return $dept )
let $new_s := (
  for $new in $new
  where $new/text() > 40000
  return $new )
let $budget_s := (
  for $budget in $dept_s/budget
  where $budget/text() < 100000
  return $budget )
WHEN
  not(empty($dept_s)) AND
  not(empty($new_s)) AND
  not(empty($budget_s))
DO (
  <XUpdate:update select="$budget_s">
    200000
  </XUpdate:update>  )
```

Fig. 7. An ActiveXQBE trigger and its Active XQuery translation

5 Related work

Querying XML documents content has been extensively studied within the database and semi-structured data communities and, ultimately, within the W3C.

Once established, query languages have a natural extension in supporting content-based updates or in extracting views of XML documents. XQuery has been extended to support updates as a result of a research work [20], and the first working draft on the XQuery Update Facility has been recently published[6]. XQuery update operations include deletion, insertion, replacement and renaming of XML data. The XUpdate language is described in [16]. An XQuery-based XML update language is described in [19].

Active rules to enforce correctness of update operations and to automatically maintain views over data have been extensively studied in database systems [9]. Several research projects have provided substantial contributions to the field of active databases (e.g., Starburst [21], Hipac [15], Reach [8], Sentinel [10]).

Active XQuery [4], our target language, aims at emulating the trigger definition and execution model of SQL3 with respect to the XML data model. It builds on the XML update language and model defined in [20]. Other XML trigger languages are XChange [7] and ECA rules for XML [2]. None of these languages offers a graphical approach.

XQBE[5] comes after a long stream of research on visual and graph-based languages, started many years ago with QBE [22]. The first graphical query languages were G and G+ [13, 14]. Graphlog [12] is a direct descendant of G+. A uniform notation for object databases where nodes represent objects and edges represent relationships was used in Good [17]. A Good-like notation was used by G-Log [18], a logic-based graphical language that allows one to represent and query complex objects by means of directed labeled graphs. An evolution of this language is XML-GL [11]. XQBE can be considered a successor of XML-GL, albeit with several new features.

6 Conclusion

In this paper we presented a framework for the visual specification of active rules for XML data. We showed that ActiveXQBE is a suitable tool for visually designing triggers in an intuitive fashion, as was demonstrated through a number of examples. These triggers can then be translated to Active XQuery and, thus, executed by implemented engines.

Among the possible future directions of research, we are studying the creation of an XQBE-based tool, to be integrated with ActiveXQBE, for the automatic or semi-automatic generation of optimized versions of the visual triggers that respond to given events and integrity constraints. Such tool would be designed along the lines of optimization frameworks for integrity constraints in deductive databases, but based on graph grammars to define rewrite rules for graphs.

[6] http://www.w3.org/TR/2006/WD-xqupdate-20060127/

References

1. Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, pages 38–49, 1998.
2. James Bailey, Alexandra Poulovassilis, and Peter T. Wood. Analysis and optimisation of event-condition-action rules on xml. *Computer Networks*, 39(3):239–259, 2002.
3. Michael Benedikt, Glenn Bruns, Julie Gibson, Robin Kuss, and Amy Ng. Automated Update Management for XML Integrity Constraints. In *PLAN-X*, 2002.
4. A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active xquery. In *Proc. of the 18th ICDE, IEEE Computer Society Press, San José, California*, Feb. 2002.
5. D. Braga, A. Campi, and S. Ceri. XQBE (XQuery By Example): a visual interface to the standard XML query language. *ACM TODS*, 30(2):398–443, 2005.
6. François Bry and Massimo Marchiori. Ten theses on logic languages for the semantic web. In *W3C WS on Rule Languages for Interoperability*. W3C, 2005.
7. François Bry and Paula-Lavinia Patranjan. Reactivity on the web: paradigms and applications of the language xchange. In *SAC*, pages 1645–1649, 2005.
8. A. P. Buchmann, H. Branding, T. Kudrass, and J. Zimmermann. Reach: A real-time, active and heterogeneous mediator system. *IEEE Data Eng. Bull.*, 15(1-4):44–47, 1992.
9. S. Ceri, R.J. Cochrane, and J. Widom. Practical applications of triggers and constraints: Successes and lingering issues. In *VLDB*, pages 254–262, 2000.
10. S. Chakravarthy, E. Anwar, and L. Maugis. Design and implementation of active capability for an object-oriented database. Technical report, Univ. Florida, 1993.
11. S. Comai, E. Damiani, and P. Fraternali. Computing graphical queries over xml data. *ACM TOIS*, 19(4):371–430, 2001.
12. M. P. Consens and A. O. Mendelzon. The g+/graphlog visual query system. In *Proc. of the 1990 ACM SIGMOD, Atlantic City, NJ, May 23-25*, page 388, 1990.
13. I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *Proc. of the ACM SIGMOD*, pages 323–330, 1987.
14. I. F. Cruz, A. O. Mendelzon, and P. T. Wood. G+: Recursive queries without recursion. In *2nd Int. Conf. on Expert Database Systems*, pages 355–368, 1988.
15. U. Dayal, A. P. Buchmann, and S. Chakravarthy. *Active Database Systems*, chapter The HiPAC Project, pages 177–205. Morgan Kauffmann, 1996.
16. Andreas Laux and Lars Matin. XUpdate working draft. Technical report, `http://www.xmldb.org/xupdate`, October 2000.
17. J. Paredaens, J. Van den Bussche, M. Andries, M. Gemis, M. Gyssens, I. Thyssens, D. Van Gucht, V. Sarathy, and L. V. Saxton. An Overview of GOOD. *SIGMOD Record*, 21(1):25–31, 1992.
18. J. Paredaens, P. Peelman, and L. Tanca. G-log a declarative graph-based language. *IEEE Trans. on Knowledge and Data Eng.*, 7(3):436–453, 1995.
19. Gargi Sur, Joachim Hammer, and Jerome Siméon. UpdateX - An XQuery-Based Language for Processing Updates in XML. In *PLAN-X 04*, pages 40–53, 2004.
20. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *ACM SIGMOD*, pages 413–424, 2001.
21. J. Widom. The starburst active database rule system. *IEEE TKDE*, (4):583–595,1996.
22. Moshé M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.