Integrity Checking and Maintenance with Active Rules in XML Databases

Henning Christiansen
Programming, Logic and Intelligent Systems
Dep't of Communication, Business and Information Technologies
Roskilde University
P.O.Box 260, DK-4000 Roskilde, Denmark
henning@ruc.dk

Maria Rekouts
Institute for System Programming
Russian Academy of Sciences
Moscow, B. Kommunisticheskaya, 25, Russia
rekouts@ispras.ru

Abstract

While specification languages for integrity constraints for XML data have been considered in the literature, actual technologies and methodologies for checking and maintaining integrity are still in their infancy. Triggers, or active rules, which are widely used in previous technologies for the purpose are still rather untested in XML databases.

We present the first steps towards a methodology for design and verification of triggers that maintain integrity in XML databases. Starting from a specification of the integrity constraints plus a collection of XPath expressions describing the possible updates, the method indicates trigger conditions and correctness criteria to be met by the trigger code supplied by a developer or possibly automatic methods. We show examples developed in the Sedna XML database system which provides a running implementation of XML triggers.

1 Introduction

Active rules or triggers in databases can be difficult to work with due to their complex semantics, but well-designed triggers are very effective for detection and prevention of inconsistencies in a database, including in situations that are not anticipated by the developer. Compared with other approaches that leave to the application program to do the checking, triggers may have advantages in efficiency (executed internally to the database engine) and security (triggers can be verified once and for all, so that inferior application programs cannot destroy the database).

Triggers are an established technology for relational databases with methodologies described in research papers and textbooks and supported by developers' tools. For XML databases, on the other hand, triggers are still a novel technology, only few implemented systems are reported and the body of experience in using them rather is limited.

The semantics of triggers in XML database management systems seems inherently more intricate than what is familiar from relational systems. For XML, updates can affect arbitrarily nested structures so, e.g., deleting one node means to delete all its descendant nodes, each of which may need to be considered by a trigger to have suitable repair actions performed elsewhere in the database. Results and methodologies for traditional technology cannot be adapted in a direct way to the case of XML, and this is the motivation for the present work.

We show the first steps in the direction of a methodology for reasoning about XML triggers and proving them correct with respect to specified integrity constraints. Such methods are in demand for manual development of triggers as well as for (semi-) automatic tools that produce triggers directly from the constraints and perhaps specifications of the possible updates.

We intend to develop methods that are independent of specific update and trigger languages, both to ensure generality and to avoid getting stuck in the morass of XML related notations. XPath expressions are used to designate those parts of a database that are critical for maintenance of given constraints. Combining this with other XPath expressions that limit the possible update requests, we pro-

vide suggestions for the trigger conditions and specify conditions needed for proving their correctness. We consider both checking and repairing triggers.

We envision a semantics for triggers based on predicate transformers as a way to abstract away the complexity of triggers and to be able to rely on established proof techniques. We unfold this for selected classes of "well-behaved" triggers.

The concepts are illustrated by triggers in the native XML database management system Sedna [25] which provides one of the few running implementations of XML triggers. We refer to survey papers [21, 17, 22] for general background on integrity constraints, checking, and maintenance.

The present work is preliminary in the sense that it provides an overall framework and definitions within which a research agenda can be posed, rather that presenting ready-to-use practical methods and tools. Section 2 gives basic definitions related to databases, constraints, and triggers, and section 3 defines correctness and sufficiency for triggers and hint proof methods. Section 4 reviews related work in traditional and XML databases, and section 5 summarizes the contribution and our perspectives for future research.

2 XML Databases, Triggers, and Integrity Constraints

Besides being adopted as the new interchange format for the Internet, XML is finding increasing acceptance as a native data repository language. There is a growing number of native XML database systems [30, 24, 25, 34, 15] fully equipped with database management capabilities: they support XQuery [10] for querying stored data and some XQuery-based update language. Currently W3C is working on XQuery update facility specification and its first draft has recently been released; the mentioned native XML database systems have provided their own update language inspired by research proposals [31, 19, 2]. Now the time is ripe for the XML database vendors to enrich their products with advanced facilities analogous to those that are popular in traditional DBMS (active rules, view maintenance, authorized access control etc.) and to study how these capabilities can be efficiently applied. In the present paper, we contribute to this evolution showing how the well-established notion of triggers is relevant for integrity constraint maintenance in an XML context. Examples given below use Sedna's update language, which is based on [19], and Sedna' XQuery extension for triggers [29].

Our paper can be seen as a parallel to Ceri and Widom's work [8] from 1990, in which triggers for integrity maintenance are designed on the basis of an analysis of constraints and update requests in an SQL like setting. Although the basic problem is the same, their setting makes it much simpler than ours as their "path expressions" point out tables, and the items to be inserted or removed are tuples and not arbitraryly nested structures.

We make our definitions as abstract as possible in order to make our approach independent of specific specification languages and to avoid the overwhelming diversity of XML related notations.

To solve these difficulties, we avoid a detailed analysis of update expressions and constraint specifications and require the developer (or perhaps some automatic process) to present a set of XPath expressions that is sufficient for the purpose. In many cases, e.g., if only single nodes are updated one at a time, these "sufficient" expressions can be read directly out of the code, and an analysis similar to that of [8] can be used.

Definition 2.1. *A* database *is a set of distinctly named XML documents which may or may not have an associated schema. Any database mentioned in the following has the same fixed set of document names.*

A constraint *is a logical condition c which, given a database D, returns a truth value; when c is true in D, we write $D \models c$ and similarly for a set of constraints when they are all true in D. A fixed set of constraints C is assumed, and we say that a database D is* consistent *whenever $D \models C$; otherwise, i.e., $D \not\models C$ we say that D is* inconsistent.

Example 2.2. *Let us consider a simple database D consisting of two documents,* `staff` *and* `offices`*:*

```
staff:
...
 <person pid = "1234">
    <name>Maria</name>
    <office>A7</office>
 </person>

 <person pid = "1235">
    <name>Henning</name>
    <office>A7</office>
 </person>
...

offices:
...
 <office oid = "A7">
  <no_of_persons>2</no_of_persons>
 </office>
...
```

For brevity, we may occasionally leave the document names implicit in XPath expressions appearing in the examples that follow.

The following definition characterizes, by sets of XPath expressions, those nodes in a database which, when updated, may be critical for maintenance of consistence. Critical nodes are those, which, when affected by an update, may

need to be considered by a trigger for checking and possible repair by a trigger. The clue to understand definition 2.3 below is that an update is conceived as a process that gradually modifies leaf nodes, one at a time. So, for example, replacing one subtree T_1 by another T_2 can be seen as first removing all subtrees of T_1 by gradually cutting more and more leaves, then replacing the top node and building T_2 top down by adding more and more leaves.

Below, we refer to *restricted* XPath expressions, which do not contain predicates and only refer to descendant, descendant-or-self, and attribute axes. This means that whether or not a given node n is contained in the value of a restricted XPath expression depends only on the direct path from the root down to n.

Definition 2.3. *Three sets of restricted XPath expressions are assumed,* CritInsXP, CritDelXP, *and* CritRplXP, *called* critical XPath sets, *satisfying the following properties.*

- *Whenever a leaf node n can be inserted into a consistent database D, leading to an inconsistent database D', there is an element in* CritInsXP *whose value in D' includes n.*
- *Whenever a leaf node n can be deleted from a consistent database D, leading to an inconsistent database, there is an element in* CritDelXP *whose value in D includes n.*
- *Whenever a leaf node n in a consistent database D can be replaced by another leaf node, leading to an inconsistent database, there is an element in* CritRplXP *whose value in D includes n.*

It can be argued that a definition referring to only top nodes of "problematic" subtrees will include too many nodes as critical and would thus claim too many triggers necessary: if node n is critical for insertion by such a definition, the any node from the n to the root would also be critical.

Example 2.4. *Consider a tree T which contains* <a> <b/> <b/> *and assume for this example a constraint saying that an* a *node must have one or two* b *subtrees. Deletion of just a single* b *does not introduce inconsistency, but since an XML update can delete several subtrees at a time, it is essential that at least one of the* b *subtrees are considered critical. To see that this is the case, consider a copy of T with the indicated subtree replaced by* <a> <b/> . *Definition 2.3 requires* CritDelXP *to contain an expression $E = \cdots$/a/b; we see that the evaluation of E in T will provide a set of nodes that includes the two* b *nodes.*

Notice that definition 2.3 allows the sets of critical XPath expressions to over-approximate, which may be an advantage when they are otherwise difficult to characterize. However, for the applications of these XPath sets below, it is better to have them as fine-grained and precise as possible, i.e., a large set of specific expressions is better than a smaller one with a higher degree of over-approximation.

For any specific constraint $c \in C$, we use informally notation CritDelXP(c) for those expressions of a CritDelXP set needed to cope with c and similarly for the other two sets.

Example 2.5. *(continues example 2.2) For the running examples, we specify a set C of the three constraints described below: key constraints, a referential integrity constraint and an aggregate constraint. Each constraint is described using the XIC constraint language proposed in [12]. For each one, we indicate its contribution to the critical XPath sets.*

c_1: *Attributes* `pid` *are keys for persons, and attributes* `oid` *are keys for offices:*

$$\forall x, y, id[doc('staff')//person](x) \wedge [doc('staff')//person](y) \wedge [./@pid](x, id) \wedge [./@pid](y, id) \rightarrow x = y$$

$$\forall x, y, id[doc('offices')//office](x) \wedge [doc('offices')//office](y) \wedge [./@oid](x, id) \wedge [./@oid](y, id) \rightarrow x = y$$

$$\text{CritInsXP}(c_1) = \{doc('staff')//person/@pid;\ doc('offices')//office/@oid\}$$

$$\text{CritRplXP}(c_1) = \{doc('staff')//person/@pid;\ doc('offices')//office/@oid\}$$

c_2: *For each* `office` *element in* `staff` *there must exist a corresponding* `office` *in* `offices` *with relevant* `oid`*:*

$$\forall p, oid[doc('staff')//person/office](p, o) \rightarrow \exists o[doc('offices')//office](o)(o) = (p, oid)$$

$$\text{CritInsXP}(c_2) = \{doc('staff')//person/office\}$$

$$\text{CritDelXP}(c_2) = \{doc('offices')//office/@oid\}$$

$$\text{CritRplXP}(c2) = \{doc('staff')//person/office;\ doc('offices')//office/@oid\}$$

In other words, insertion of a new item with a possibly dangling reference might lead to violation of c_2; deletion of an item to which other items could possibly refer to might also lead to violation of c_2.

c_3: *Each* `no_of_persons` *element contains the number of persons in that office:*

$$\forall o[doc('offices')//office/no\_of\_persons](o) \rightarrow [doc('staff')//person](p)(o) = sum(p)$$

$$\text{CritInsXP}(c_3) = \{doc('staff')//person/office\}$$

$$\mathsf{CritDelXP}(c_3) = \{doc('staff')//person/office\}$$
$$\mathsf{CritRplXP}(c_3) = \{doc('staff')//person/office;\ doc('offices')//office/no\_of\_persons;\ doc('offices')//office/@oid\}$$

It may be possible to use instead more elaborate, critical expressions with predicates that actually check whether, say, a referential integrity is broken, but our method cannot employ this at present. Triggers in database systems, whether relational or XML, are often triggered not only by the effect of an update, but also the type of update that produced the effect. Here we will model the conventions used in Sedna, but other choices could have been made as well. In the following definition we have abstracted away most of the syntax of an update language, except the details that affect trigger behavior.

For an update, we informally characterize its *application points* as representing the nodes where an update attaches or detaches something from a tree. For delete and replace operations, the application points typically correspond to the value of an XQuery expression given in the update request, e.g., the value of x in DELETE x or REPLACE x WITH y. We capture the applications point by the sets denoted $\mathsf{Applic}(-)$ in the definition.

Definition 2.6. *An* update u *is an operation that modifies one or more documents in a database D_1, producing another database D_2; we denote this fact $D_1 \overset{\tau:u}{\Rightarrow} D_2$, where τ is a* type $\tau \in \{INSERT, DELETE, REPLACE\}$. *An update of this form is characterized by four sets of nodes,* $\mathsf{Applic}(u)$, $\mathsf{Inserted}(u)$, $\mathsf{Deleted}(u)$, *and* $\mathsf{Replaced}(u)$*;* $\mathsf{Applic}(u)$ *is determined by the semantics of the update language.*

For type $INSERT$, $\mathsf{Deleted}(u)$ *and* $\mathsf{Replaced}(u)$ *are empty, and* $\mathsf{Applic}(u)$ *is a set of nodes in D_2;* $\mathsf{Inserted}(u)$ *is comprised by* $\mathsf{Applic}(u)$ *and any descendant thereof; removing* $\mathsf{Inserted}(u)$ *from D_2 yields D_1.*

For type $DELETE$, $\mathsf{Inserted}(u)$ *and* $\mathsf{Replaced}(u)$ *are empty, and* $\mathsf{Applic}(u)$ *is a set of nodes in D_1;* $\mathsf{Deleted}(u)$ *is comprised by* $\mathsf{Applic}(u)$ *and any descendant thereof; removing* $\mathsf{Deleted}(u)$ *from D_1 yields D_2.*

For type $REPLACE$, $\mathsf{Applic}(u)$ *identifies a set of nodes in D_1;* $\mathsf{Replaced}(u)$ *coincides with* $\mathsf{Applic}(u)$*;* $\mathsf{Inserted}(u)$ *is comprised by the nodes of D_2 in positions matching* $\mathsf{Applic}(u)$ *and any descendant thereof in D_2;* $\mathsf{Deleted}(u)$ *is comprised by the nodes of* $\mathsf{Applic}(u)$ *in D_1 and any descendant thereof; removing* $\mathsf{Inserted}(u)$ *from D_2 and* $\mathsf{Deleted}(u)$ *from D_1 yields identical databases.*

Example 2.7. *The transition from* <a>   *into* <a> <c/> *can be defined by different updates, 1) inserting a* <c> *node under the* *node, 2) replacing the* *node by* <c/> *, and 3) replacing the whole tree by the new one. Although the effect is the same of all three, the* Applic, Inserted, Deleted, *and* Replaced *sets differ (and we may expect that different triggers are called in each case).*

Example 2.8. *(continues examples 2.2 and 2.5) The following two update statements u_1 and u_2 are given in Sedna's update language:*

u_1 :
```
UPDATE
    delete doc("staff")//person[@pid="1234"]
```

u_2 :
```
UPDATE
    replace
       $n in doc("staff")//person[@pid="128"]/name
    with
       <name>Maria P. Rekouts</name>
```

With u_2, each node returned by `doc("staff")//person[@pid="128"]/name` *is replaced with* `<name>Maria P. Rekouts</name>`*; variable* `$n` *is bound to the nodes one at a time (but not referenced in this particular example).*

We assume that update requests are issued by a fixed application program $\mathcal{P}$ and trigger set $\mathcal{T}$. While our goal is to provide guidelines for how triggers should be defined in order to work correctly (according to given constraints C), we assume for ease of notation that the trigger set $\mathcal{T}$ is fixed and analyze in the following requirements to this trigger set. Thus there is a well-defined meaning when below we say "an update $D_1 \overset{\tau:u}{\Rightarrow} D_2$ that may occur (for $(\mathcal{P}, \mathcal{T})$)". The following definition provides an abstract characterization of the possible updates in terms of XPath expressions.

Definition 2.9. *For fixed application program $\mathcal{P}$ and trigger set $\mathcal{T}$, three sets of XPath expressions are assumed,* $\mathsf{InsXP}(\mathcal{P}, \mathcal{T})$, $\mathsf{DelXP}(\mathcal{P}, \mathcal{T})$, *and* $\mathsf{RplXP}(\mathcal{P}, \mathcal{T})$ *with the following properties.*

For any update $D_1 \overset{\tau:u}{\Rightarrow} D_2$ that may occur, it holds that for any n in $\mathsf{Inserted}(u)$*, there is an expression in* $\mathsf{InsXP}(\mathcal{P}, \mathcal{T})$ *whose value in D_2 contains n.*

For any update $D_1 \overset{\tau:u}{\Rightarrow} D_2$ that may occur, it holds that for any n in $\mathsf{Deleted}(u)$ *(resp.* $\mathsf{Replaced}(u)$*), there is an expression in* $\mathsf{DelXP}(\mathcal{P}, \mathcal{T})$ *(resp.* $\mathsf{RplXP}(\mathcal{P}, \mathcal{T})$*) whose value in D_1 contains n.*

It is worth noting that the InsXP, DelXP, and RplXP sets can be much simpler than those XPath expressions found in the actual code. For example, `//a` in InsXP may "subsume" a huge collection of more detailed expressions ending with `/a`, which in some cases may be useful in order to reduce the number of triggers.

These sets can be produced by intellectual manners or automatically as discussed above for critical XPath sets. If the purpose of the trigger collection to be developed is to detect and reject updates that violate integrity, and only that,

InsXP etc. may be identified from an analysis of the application program.

On the other hand, when triggers are intended to repair inconsistencies introduced by updates, there will be a natural feed-back loop in the development process, since InsXP etc. are translated into triggers, which as their instrument for repair introduce yet other updates, which in turn require extensions to InsXP, etc.

Example 2.10. *(continues examples 2.2, 2.5, and 2.8) We consider in this example an application program $\mathcal{P}$ which performs updates similar to u_1 and u_2 but for varying* pids *and names; the trigger set is currently assumed empty. Here we may use the following XPath sets to capture the possible updates.*

$$\mathsf{InsXP}(\mathcal{P},\mathcal{T}) = \{doc('staff')//person/name\}$$
$$\mathsf{DelXP}(\mathcal{P},\mathcal{T}) = \{doc('staff')//person;\ doc('staff')//person/office;\ doc('staff')//person/name;\ doc('staff')//person/@pid\}$$
$$\mathsf{RplXP}(\mathcal{P},\mathcal{T}) = \{doc('staff')//person/name\}$$

Following [18], we define $E_1 \cap E_2 \neq \emptyset$ to be true for XPath expressions E_1 and E_2 if for some database D, E_1 and E_2 evaluate to lists of values with a common element in D; the reference [18] shows that the problem is decidable and provides algorithms. The following sets of XPath expressions determine the trigger expressions for which triggers must be provided; we will later show that a set of triggers for these patterns and satisfying a suitable semantic condition, will be sufficient to maintain consistency.

Definition 2.11. *Three sets of XPath expressions are defined as follows.*

$$\mathsf{CritInsXP}(\mathcal{P},\mathcal{T}) = \{pi \in \mathsf{InsXP}(\mathcal{P},\mathcal{T}) \mid \exists pc \in \mathsf{CritInsXP}:\ pi \cap pc \neq \emptyset\}$$
$$\mathsf{CritDelXP}(\mathcal{P},\mathcal{T}) = \{pd \in \mathsf{DelXP}(\mathcal{P},\mathcal{T}) \mid \exists pc \in \mathsf{CritDelXP}:\ pd \cap pc \neq \emptyset\}$$
$$\mathsf{CritRplXP}(\mathcal{P},\mathcal{T}) = \{pr \in \mathsf{RplXP}(\mathcal{P},\mathcal{T}) \mid \exists pc \in \mathsf{CritRplXP}:\ pr \cap pc \neq \emptyset\}$$

The following important property indicates the relevance of this definition which later will indicate that at least one trigger is called when an inconsistency is introduced; the proof depends on the fact that the indicated XPath sets consist of restricted expressions only.

Proposition 2.12. *Let $D_1 \overset{\tau:u}{\Rightarrow} D_2$ be an update that may occur for some $\mathcal{P}$ and $\mathcal{T}$, with D_1 consistent and D_2 inconsistent. Then at least one of the following properties are true.*

- *There is a $p \in \mathsf{CritInsXP}(\mathcal{P},\mathcal{T})$ whose value in D_1 includes a node of $\mathsf{Inserted}(u)$.*
- *There is a $p \in \mathsf{CritDelXP}(\mathcal{P},\mathcal{T})$ whose value in D_2 includes a node of $\mathsf{Deleted}(u)$.*
- *There is a $p \in \mathsf{CritRplXP}(\mathcal{P},\mathcal{T})$ whose value in D_1 includes a node of $\mathsf{Replaced}(u)$.*

This has an interesting consequence which we state informally: if an update u is reduced to a smallest update u' which introduce inconsistency, then there will still be a match of one of the indicated XPath expressions with a node of $\mathsf{Inserted}(u')$, $\mathsf{Deleted}(u')$, or $\mathsf{Replaced}(u')$. In other words, the XPath expressions characterize the "real cause" of inconsistency, but we cannot exclude that it matches a bit more than this.

We notice the following corollary.

Proposition 2.13. *In case $\mathsf{CritInsXP}(\mathcal{P},\mathcal{T}) = \mathsf{CritDelXP}(\mathcal{P},\mathcal{T}) = \mathsf{CritRplXP}(\mathcal{P},\mathcal{T}) = \emptyset$, any update of a consistent database will preserve consistency.*

This proposition can indicate when no triggers are necessary, but may also apply when triggers are used for maintaining relationships that are not formalized as constraints.

We notice that our approach may possibly be improved here in some case where, say CritInsXP (with no arguments, i.e., derived from the constraint set C only) contains predicates that test whether integrity actually is broken. Here it may be advantageous not only to take a subset of $\mathsf{InsXP}(\mathcal{P},\mathcal{T})$, but to produce a more optimal integration with CritInsXP, e.g., employing predicates in it. Our current version misses such optimizations.

Example 2.14. *(continues examples 2.2, 2.5, 2.8, and 2.10) With the collected assumptions we get the following sets.*

$$\mathsf{CritInsXP}(\mathcal{P},\mathcal{T}) = \emptyset$$
$$\mathsf{CritDelXP}(\mathcal{P},\mathcal{T}) = \{doc('staff')//person/office\}$$
$$\mathsf{CritRplXP}(\mathcal{P},\mathcal{T}) = \emptyset$$

The expression $//person/office$ from CritDelXP *is included in $\mathsf{CritDelXP}(\mathcal{P},\mathcal{T})$ as it appears identically in $\mathsf{DelXP}(\mathcal{P},\mathcal{T})$, i.e., a special case of nonempty intersection.*

3 Correct and sufficient triggers

A trigger is a piece of code controlled by the database management system; it starts performing its actions when an update is registered for a node within the value of its triggering expression. We sketch here the syntax of triggers as they appear in the Sedna system.

```
CREATE TRIGGER <trigger_name>
BEFORE|AFTER (INSERT|DELETE|REPLACE)+
OF XPathExpression (,XPathExpression)*
FOR EACH (NODE|STATEMENT)
DO
{ (XQuery-expression($NEW, $OLD, $WHERE);)* }
}
```

- The `CREATE TRIGGER` clause is used to define a new XML trigger with the specified name.

- `BEFORE | AFTER` clause the triggering time relative to the triggering operation.
- Each trigger is associated with a set of update operations (`INSERT`, `DELETE` and `REPLACE`) adopted from the update extension used in Sedna [19].
- The operation is relative to nodes that match an XPath expressions specified after the `OF` keyword.
- `FOR EACH NODE/STATEMENT` expresses the trigger granularity. A *statement-level* trigger executes once for each set of nodes extracted by evaluating the XPath expressions mentioned above, while a *node-level* trigger executes once for each of those nodes.
- The action is expressed by means of the `DO` clause. It consists of any number of arbitrarily complex update operations and, possibly, a final query operation. For a node-level trigger, the value of the query operation is returned to the calling executor. A node-level trigger fired `BEFORE` an operation has the following options:
  - It can return an empty sequence with the meaning of skipping the operation for the current node. This instructs the executor to reject the node-level operation that invoked the trigger (the insertion or replacement of a particular node).
  - For node-level `INSERT` and `REPLACE` triggers only, the returned node becomes the node that will be inserted or will replace the node being updated. This allows the trigger to modify the node being inserted or updated.
- Trigger action can make update requests to other XML documents, possibly starting other triggers (known as cascading).
- For node-level triggers transition variables `$OLD`, `$NEW` and `WHERE` are accessible in the trigger's action.

However, for reasons of efficiency and scheduling, Sedna has the restriction that within the handling of a given top-level update, cascading must never lead to updates in a document from which a trigger has been called. This restriction is similar to what is found in some RDBMS [35]. The method we propose here, does not depend on this restriction.

When executed, a trigger has access to the value of the triggering XPath expression, and it can access any part of the database, including old and new value of nodes which are subject of a modification. From this, it determines its action as indicated.

Example 3.1. *As an example we provide the following trigger that would partially support c_3 of example 2.5.*

```
CREATE TRIGGER "tr_c3"
AFTER DELETE
OF doc("staff")//person/office
FOR EACH NODE
DO { UPDATE
       replace $n
       in doc("offices")//office[@oid=$OLD]
                          /no_of_persons/text()
       with xs:integer($n)-1 ;
   }
```

This trigger fires, when an `office` *node of any person in the* `doc("staff")` *is deleted, and decreases the value of* `no_of_persons` *element in a corresponding* `office`*. Note, the predefined variable* `$OLD` *is used to refer the deleted* `office`*.*

For proving correctness of a trigger or set of triggers, we consider an update, and the possible combination of updates performed by triggers as a *predicate transformer* [13]. When u is such a piece of code (e.g., an update request), the notation $[C_1]u[C_2]$ means that if a set of constraints C_1 holds before u, then C_2 holds after the update.

Example 3.2. *Consider an update $D_1 \stackrel{\tau:U}{\Rightarrow} D_2$, where*
D_1: `<a> </a> <a> </a>`*,*
D_2: `<a> <b/> </a> <a> </a>`*,*
and let u^{-1} be the reversed update. Here we have $[C_1]u[C_2]$ and $[C_2]u^{-1}[C_1]$ where C_1 is "`a` *nodes have no subtrees", and C_2 "*`a` *nodes have no subtrees, except the first one".*

Correctness of an arbitrary trigger set is specified in the following highly abstract and, at first glance, not very useful way.

Definition 3.3. *For given application program $\mathcal{P}$ and a trigger set $\mathcal{T}$, we say that $\mathcal{T}$ is* correct *for an update u applied to a consistent database, whenever the net effect of the activated triggers (with possibly cascading and change of the requested update) amounts to an action T_u with the following property. Notice that C refers to the given fixed set of constraints.*

$$[C]T_u[C]$$

The definition is relevant since we can show below, different sufficient conditions which apply for specific classes of triggers.

Definition 3.4. *Given an application program $\mathcal{P}$ and a trigger set $\mathcal{T}$, we say that $\mathcal{T}$ is* sufficient *for $(\mathcal{P}, \mathcal{T})$ if at least one trigger is called for any update that may occur in a consistent state and leading to an inconsistent state.*

Proposition 3.5. *A trigger set $\mathcal{T}$ for application program $\mathcal{P}$ is sufficient if it satisfies the following properties.*

- *For any $p \in \mathsf{CritInsXP}(\mathcal{P}, \mathcal{T})$, $\mathcal{T}$ has a trigger of the form CREATE TRIGGER . . . INSERT OF p . . .*

- *For any $p \in \mathsf{CritDelXP}(\mathcal{P}, \mathcal{T})$, $\mathcal{T}$ has a trigger of the form CREATE TRIGGER ... DELETE OF p ...*

- *For any $p \in \mathsf{CritRplXP}(\mathcal{P}, \mathcal{T})$, $\mathcal{T}$ has a trigger of the form CREATE TRIGGER ... REPLACE OF p ...*

Proposition 3.6. *Consider an application program $\mathcal{P}$ and a trigger set $\mathcal{T}$, which is correct and sufficient. Then any sequence of updates produced by $(\mathcal{P}, \mathcal{T})$, starting from a consistent database, will lead to consistent databases only (not counting intermediate states between a top level update request is posed and the last trigger has finished).*

We can now give a (high-level!) recipe for producing a set of triggers $\mathcal{T}$ that is guaranteed to maintain consistency:

1. $\mathcal{T} := \emptyset$,
2. Calculate the three sets of XPath expressions $\mathsf{CritInsXP}(\mathcal{P}, \mathcal{T})$, $\mathsf{CritDelXP}(\mathcal{P}, \mathcal{T})$, and $\mathsf{CritRplXP}(\mathcal{P}, \mathcal{T})$.
3. Add a trigger to $\mathcal{T}$ for each of those and ensure that $\mathcal{T}$ is correct.
4. If step 3 resulted in new critical XPath expressions (due to update requests in trigger bodies), continue with the relevant portions of 2–4.

The big issue is, now, how actually to prove correctness of a trigger set. At present, we have not developed a catalogue of proof techniques, and we provide instead a few informally described propositions stating examples of sufficient conditions.

Proposition 3.7. *Consider an application program $\mathcal{P}$ and a set of BEFORE triggers $\mathcal{T}$ with no cascading, i.e., when a trigger body requests an additional update, this affects only other nodes in a way that does not start other triggers.*

We have that $\mathcal{T}$ is correct, if any $T \in \mathcal{T}$ satisfies the following property:

- *For any update u which activates T, T either rejects u or modifies it into another u' and requests additional update u'' for which $[C]u''[C']u[C]$ holds for some C'.*

This property has the practical advantage that each trigger can be proved correct one at a time. It will be relevant to identify less restricted classes of triggers that preserve this.

For cascading triggers, we may rely on an ordering among the triggers analogously to [9]. Let us detail a special case. We assume, for a given constraint c, that there is a well-founded ordering that measures of the degree of violation of c; for a referential constraint, for example, this can be defined in terms of the number of nodes that contain an unbound reference.

Proposition 3.8. *Consider an application program $\mathcal{P}$ and a set of triggers $\mathcal{T} = \{t_1, \ldots, t_n\}$ such that the set of constraints can be listed as $C = \{c_1, \ldots, c_n\}$ where the following holds for $i = 1, \ldots, n$,*

- *t_i requests additional updates only when c_i is violated, in which case it reduces the degree of violation for c_i; furthermore, it does not increase the degree of violation for any of $c_1, \ldots, c_{i-1}$.*

Then $\mathcal{T}$ is correct.

4 Related work

The use of active rules to support database integrity constraints has been recognized for decades and extensively studied for relational databases. A lot of research has been devoted to the automatic or semi-automatic generation of integrity-preserving rules from specifications of integrity constraint for relational databases. These works may be classified roughly by three main possibilities:

1. Syntactic generation of events (and conditions, if the active rule language allows condition specification). Triggering events and rule conditions usually can be generated from the integrity constraint specification by means of relatively straightforward syntactic analysis [8]. This approach leaves the generation of the action part of the rules to the designer.
2. Syntactic generation of events and condition, declarative specification of action. When multiple repair strategies are possible, the designer may specify, in a declarative style, the integrity constraint together with a repair strategy. Rule generation can then be fully automated [4, 16].
3. Syntactic generation of events and condition, semantic generation of action. A rule generation module may exhaustively consider all possible repair actions for a set of constraints and choose an appropriate combination among them, considering the interactions among constraints and possibly gathering additional information from the designer [7, 26].

Our work considers the first of these possibilities for XML databases, where support for advanced features such as active rules is becoming available. Currently, we are only aware of the XML database Sedna [25] which is sufficiently powerful to support integrity maintenance in way we have presented. We are aware that the native XML database eXist [24] supports triggers, but in a restricted manner: triggers can only be set on documents and collections, and do not cover the XML document hierarchy [23]. We expect that

our results can be integrated with the two other possibilities as XML database technologies get more mature.

However, active rules for XML has been considered in a number of research papers. In [5], an active rule language for XML repositories, called Active XQuery, is presented as an extension of XQuery. The authors describe the syntax and semantics of Active XQuery describing an algorithm to support triggers and a sketchy system architecture. The authors assume that trigger processing can be performed completely at compile-time and trigger supporting module can be implemented on top of the existing XML repository with XQuery support. In [29] we provide an alternative method to implement XML triggers inside an XML repository.

Reference [3] provides an ECA-rule language for XML (that is close Sedna's) and exhaustively covers the problem of statically predicting the active rules' run-time behavior. This work can be used as a good basis for building active rules design tools to facilitate a designer's work of building sufficient set of triggers for constraint maintenance.

Another research direction concerns *repair* of inconsistent databases (XML and other kinds) which has applications in, among other fields, data integration; general and automatic methods are described by, e.g., [6, 33]. An interesting detailed study of particular cases of repairs in a partly manual and partly automatic system is done by [14], which may provide inspiration for extending our approach into an interactive environment for development of triggers.

Instead of using triggers, integrity can also be checked and maintained by tests and actions performed by the application program, before or after an update is performed. This is often based on so-called simplification methods [27] which are systematic or automatic methods, which have been studied mainly for relational and deductive databases, to transform global integrity constraints into specialized and optimal checks for individual updates or update patterns; see [21, 11] for an overview of previous and recent results. Similar techniques are suggested for XML databases by [28], however, seemingly unaware of the work on simplification.

5 Conclusion and future directions

We may envision an interactive development tool which takes as input constraints in some formal language plus perhaps XPaths expressions describing possible updates. Trigger expressions are generated automatically as indicated above, and code templates can be suggested. The system may also embed knowledge about the semantics in terms of a mapping into predicate transformers, and supply assisting proof tools.

Simplification techniques known from relational and deductive databases, e.g., [11], may be adapted to XML, so that proposals for checking code, and alternatives for repair code, may be produced by automatic transformations of constraints; some initial work has been done in this direction, but fully satisfactory solutions are yet to come.

An important detail in our approach which can and must be improved is the derivation of trigger expression from the given XPath sets, definition 2.11. At present, we take subsets of the restricted XPath expressions describing the possible updates, which seems unnecessarily coarse. A better approach may be to invent a sort of intersection operation for XPath, which produces new XPath expressions that capture the intersection of node sets for the critical and the update XPath expressions.

In order to test our approach for applications that cannot be expressed in a natural way using relational technology, we are currently developing a database which mimics a program text editor for a Java-like language. Arbitrarily deep syntax trees are stored as XML trees and triggers are introduced to check context-sensitive syntax constraints relating to declarations, visibility and scope, marking possible violations.

XML is also extensively used in content management applications. Here, the dependencies between the documents in a complex publishing scenarios are usually defined using some domain specific language (for example as it is organized in Cocoon framework [32]). It should be analyzed how our methodology may be applied to generate triggers that maintain constraints specified in such domain-specific languages.

Acknowledgement: This work is supported by the CONTROL project, funded by Danish Natural Science Research Council, and partly supported by grants of RFBR NN 05-07-90204- and 06-07-08073.

References

[1] *16th International Workshop on Database and Expert Systems Applications (DEXA 2005), 22-26 August 2005, Copenhagen, Denmark*. IEEE Computer Society, 2005.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.

[3] J. Bailey, G. Papamarkos, A. Poulovassilis, and P. T. Wood. An event-condition-action language for xml. In Levene and Poulovassilis [20], pages 223–248.

[4] E. Baralis, S. Ceri, and S. Paraboschi. Modularization techniques for active rules design. *ACM Trans. Database Syst.*, 21(1):1–29, 1996.

[5] A. Bonifati and S. Paraboschi. Active xquery. In Levene and Poulovassilis [20], pages 249–274.

[6] A. Calì, D. Calvanese, G. D. Giacomo, and M. Lenzerini. Data integration under integrity constraints. *Inf. Syst.*, 29(2):147–163, 2004.

[7] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM Trans. Database Syst.*, 19(3):367–422, 1994.

[8] S. Ceri and J. Widom. Deriving production rules for constraint maintainance. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *VLDB*, pages 566–577. Morgan Kaufmann, 1990.

[9] S. Ceri and J. Widom. Deriving incremental production rules for deductive data. *Inf. Syst.*, 19(6):467–490, 1994.

[10] D. Chamberlin, M. F. Fernandez, and J. Simeon et al. XQuery 1.0: An XML Query Language, W3C Recommendation 23 January 2007. Available at `http://www.w3.org/TR/xquery/` link checked December 2006.

[11] H. Christiansen and D. Martinenghi. On simplification of database integrity constraints. *Fundamenta Informaticae*, 71:371–417, 2006.

[12] A. Deutsch and V. Tannen. Xml queries and constraints, containment and reformulation. *Theoretical Computer Science*, 336(1):57–87, 2005.

[13] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[14] S. M. Embury, S. M. Brandt, J. S. Robinson, I. Sutherland, F. A. Bisby, W. A. Gray, A. C. Jones, and R. J. White. Adapting integrity enforcement techniques for data reconciliation. *Inf. Syst.*, 26(8):657–689, 2001.

[15] T. Fiebig, C.-C. Kanne, and G. Moerkotte. Natix - ein natives XML-DBMS. *Datenbank-Spektrum*, 1:5–13, 2001.

[16] M. Gertz. Specifying reactive integrity control for active databases. In *RIDE-ADS*, pages 62–70, 1994.

[17] P. Godfrey, J. Grant, J. Gryz, and J. Minker. Integrity constraints: Semantics and applications. In *Logics for Databases and Information Systems*, pages 265–306, 1998.

[18] B. C. Hammerschmidt, M. Kempa, and V. Linnemann. On the intersection of xpath expressions. In *IDEAS*, pages 49–57. IEEE Computer Society, 2005.

[19] P. Lehti. Design and implementation of a data manipulation processor for an XML query language., 2001. Available at `http://www.lehti.de/beruf/diplomarbeit.pdf`.

[20] M. Levene and A. Poulovassilis, editors. *Web Dynamics - Adapting to Change in Content, Size, Topology and Use*. Springer, 2004.

[21] D. Martinenghi, H. Christiansen, and H. Decker. Integrity checking and maintenance in relational and deductive databases – and beyond. In Z. Ma, editor, *Intelligent Databases: Technologies and Applications*, pages 238–285. Idea Group Publishing, 2006.

[22] E. Mayol and E. Teniente. A survey of current methods for integrity constraint maintenance and view updating. In P. P. Chen, D. W. Embley, J. Kouloumdjian, S. W. Liddle, and J. F. Roddick, editors, *ER (Workshops)*, volume 1727 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 1999.

[23] W. Meier. exist XML database documentation. Available at `http://exist.sourceforge.net/`.

[24] W. Meier. Open source native XML database eXist. Available at `http://exist.sourceforge.net/`.

[25] MODIS TEAM at ISP RAS. Open source native XML database Sedna. Available at `http://www.modis.ispras.ru/sedna/`.

[26] G. Moerkotte and P. C. Lockemann. Reactive consistency control in deductive databases. *ACM Trans. Database Syst.*, 16(4):670–702, 1991.

[27] J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.

[28] S. B. N. Praveen Madiraju, Rajshekhar Sunderraman and H. Wang. Semantic integrity constraint checking for multiple xml databases. *Journal of Database Management*, 17:1–19, 2006.

[29] M. Rekouts. Incorporating active rules processing into update execution in XML database systems. In *DEXA Workshops* [1], pages 831–836.

[30] Software AG. XML database Tamino. Available at `www.softwareag.com/tamino`.

[31] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating xml. In *SIGMOD Conference*, pages 413–424, 2001.

[32] The Apache Cocoon Project. Cocoon, web development framework. Available at `http://cocoon.apache.org/`.

[33] J. Wijsen. On condensing database repairs obtained by tuple deletions. In *DEXA Workshops* [1], pages 849–853.

[34] XHive. XML database XHive/DB. Available at `http://www.x-hive.com/products/db/index.html`.

[35] J. Y. Yu-May Chang, Jeff Ullman. Constraints and triggers in Oracle. Available at `http://www.cise.ufl.edu/~jhammer/classes/Oracle/Cons_Triggers.htm`.