WPI-CS-TR-04-19 July 2004

Updating XML Views Published over Relational Databases: Towards the Existence of a Correct Update Mapping

by

Ling Wang
Elke A. Rundesnteiner and Murali Mani

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Updating XML Views Published over Relational Databases: Towards the Existence of a Correct Update Mapping

Ling Wang, Elke A. Rundesnteiner and Murali Mani
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609
Tel.: (508) 831–5857, Fax: (508) 831–5776
{lingw, rundenst, mmani}@cs.wpi.edu

Abstract

XML data management using relational database systems has been intensively studied in the last few years. However, in order for such systems to be viable, they must support not only queries, but also updates over virtual XML views that wrap the relational data. While view updating is a long-standing difficult issue in the relational context, the flexible XML data model and nested XML query language both pose additional challenges for view updating.

This paper addresses the question, if for a given update over an XML view, a correct relational update translation exists. First, we propose a clean extended-source theory as criteria for determining whether a given translation mapping is correct. To determine the existence of such a correct mapping, we classify a view update as either untranslatable, conditionally or unconditionally translatable under a given update translation policy. This classification depends on several features of the XML view and the update: (a) granularity of the update at the view side, (b) properties of the view construction, and (c) types of duplication appearing in the view. These features are represented in the Annotated Schema Graph. This is further utilized by our Schema-driven Translatability Reasoning algorithm (STAR) to classify a given update into one of the three above update categories. The correctness of the algorithm is proven using our clean extended-source theory. This technique represents a practical approach that can be applied by any existing view update system in industry and academia for analyzing the translatability of a given update statement before translation of it is attempted. To illustrate the working algorithm, we provide a concrete case study on the translatability of XML view updates.

Keywords: XML View, XQuery, Update Translatability.

1 Introduction

XML has become the standard for interchanging data between web applications. Recent XML management systems [9, 14, 28] combine the strengths of the XML data model with the maturity of relational database technology to provide both reliable persistent storage as well as flexible query processing and publishing. To bridge the gap between relational databases and XML applications, such systems typically support the creation of XML views that wrap the relational base and querying against them. Update operations against such wrapper views, however, are not well supported in most of these systems [9, 14].

Motivation. The problem of updating XML views published over relational data comes with new challenges beyond those of updating relational [1, 2, 12, 13, 16, 17] or even object-oriented [4] views. The first challenge relates to the *update translatability* problem. That is, the mismatch between the hierarchical XML view model and the flat relational base model raises the question whether the given view update is even mappable into SQL updates. For instance, the nested structure imposed by an XML view may be in conflict with the constraints of the underlying relational schema, so that updates that are valid on the view may not be achieved on the base. The second challenge concerns the *translation strategy* issue. That is, assuming the view update is indeed identified as being translatable, we need to devise a strategy to identify the *minimal* mapping update. This mapping has to best bridge the two query and update languages (updates with diverse granularity on the XML view versus flat tuple-based SQL updates on the relational base).

The *translation strategy* issue has been explored to some degree in recent work. [22] studies the execution performance of translated updates. They assume in their work that the given update is indeed translatable, i.e., that the update translatability question has already been resolved positively. Two, one fixed loading strategy, namely, inlining loading strategy [15], is assumed. Third, this work assumes that XML updates have already been translated into SQL updates over a relational database, and they focus only on how to make such a set of SQL statements efficient. XML view management systems such as SQL-Server2000 [21], Oracle [3] and DB2 [10] provide system-specific solutions for limited update support, again under the assumption of updates always being translatable. For instance, instead of using update statements, [21] uses a before and after image of the view to compute the corresponding SQL statements.

However, the first issue, *update translatability* question, remains largely unexplored. Studying update translatability is important in terms of both correctness and performance. Without translatability checking, blindly translating a given view update into relational updates can be dangerous. Such blind translation may result in unintended view side effects. To identify this, the view before the update and after the update would have to be compared. To adjust for such an error, the view update would have to be rejected and the relational database would have to be recovered for example by rolling back. This would be time consuming and depends on the size of the database. However, by performing an update translatability analysis, such ill-behaved updates could be identified early on and rejected at compile time. The latter would be less costly, depending only on the view query size.

State-of-Art. Update translatability checking has been a long standing difficult issue even in the relational context. [12, 2] propose a complementary theory that requires a correct mapping to avoid view side effects as well as database side effects. That is, for a translation to be considered correct, it cannot affect any part of the database that is "outside" the view. This correctness criteria, however, is too restrictive to be practical. [13] relaxes this condition to only requires that no view side effect occurs. In other words, a translation is correct as long as it corresponds exactly to the specified update, and it does not affect anything else in the view. Using the concept of "clean source", it also characterizes the schema conditions under which an update of a relational view is translatable. Under this relaxed criteria, [16, 17, 1] study the view update translation mechanism for SPJ queries on relations that are in BCNF.

Using the complementary theory, in our earlier work [25], we study the update translatability of XML views over the relational database in the special "round-trip" case, which is characterized by a pair of reversible lossless mappings for (i) first loading the XML document into the relational database for storage, and (ii) extracting an XML view identical to the original XML document back out of it. We prove that any *valid* update operation over such XML views, given a pair of round-trip mappings, is always translatable.

To the best of our knowledge, however, no result in the literature focuses on a general method to assess the translatability of updates on an *arbitrary* XML view published over relational data. That is, given a relational database and an XML view definition on top of it, we now propose to tackle the question whether a given update over the virtual XML view can be mapped into relational SQL updates without any view side effect.

Our Approach on *S*chema-driven Update *T*r*A*nslatability *R*easoning (STAR) . With the hierarchical structure of the XML data model in consideration, our work first extends the concepts of a "clean source" for relational databases [13] into the concept of a "clean extended-source" suitable for XML. We propose a clean-extended source theory for determining the existence of a correct relational update translation for a given XML view update.

Based on this theoretical foundation, a practical algorithm is developed to address the update translatability issue specific to the XML views. Given an *update translation policy*, we classify updates over an XML view as *un-translatable*, *conditionally* translatable or *unconditionally translatable*. This classification depends on several features of the XML view and of the update: (a) granularity of the update at the view side, (b) properties of the view *construction*, and (c) types of *duplication* appearing in the view. These features are represented in the *Annotated Schema Graphs*(ASG). These ASGs are utilized by our **STAR** algorithm to classify a given update into one of the three update categories. First, a *STAR marking step* pre-codes each node in the ASG with a pair of labels (UContext|UPoint) to indicate its update properties. This analysis is performed only once at compile time. The *STAR checking step* will use resulting labels classify the given view update into one of the three translatability categories by a. Our STAR algorithm depends only on the view and database schema knowledge instead of on the actual database content. It rejects un-translatable updates, requests additional conditions for conditionally translatable updates, and passes unconditionally translatable updates to the later update translation step. The correctness of our STAR approach is proven utilizing our *clean extended-source theory*.

Contributions. The preliminary work in [26] has the following contributions: (1) We characterize the *update translatability* problem for XML views and identify key factors affecting the translatability. (2) We propose the *clean extended-source theory* for determining whether a correct view update translation exists. (3) We present a *schema-driven translatability reasoning* algorithm for deciding if an update on an XML view is un-translatable, conditionally or unconditionally translatable, when the construction consistency is the major concern. That is, the conflict between the view hierarchy and the relational foreign key constraints.

Beyond this work, we now have made the following additional contributions in this full manuscript:

- We now propose a complete *schema-driven translatability Reasoning* algorithm (STAR) that handles updates (especially delete) on XML views with both content and structural duplication.
- We prove the correctness of our *clean extended source theory*, which is criteria for determining whether a given translation mapping is correct. This theory now serves as a solid foundation for our STAR approach.
- Using the clean extended source theory, we prove our *schema-driven translatability reasoning* algorithm (STAR) for deciding whether an update on an XML view is translatable.
- We provide case studies to apply proposed algorithm to analyze the translatability of updates in various XML views.

Outline. The view update problem is formalized in Section 2. In Section 3 we propose the "clean extended-source" theory as theoretical foundation of our proposed solution. Section 4 analyzes the factors deciding the translatability of updates on XML views. Our schema-driven translatability reasoning algorithm is depicted in Section 5. its correctness is proven in Section 6. Section 7 provides an evaluation of our STAR algorithm. Section 8 reviews related work while Section 9 provides our conclusions and future directions.

2 Formalizing the Problem of XML View Updatability

Intuitively, the translatability of XML view update question can be described as follows. Given a relational database and an XML view definition over it, can we decide whether an update against the XML view is translatable into corresponding updates against the underlying relational database without violating any consistency? Intuitively, by consistency, we mean that (i) the requested view update is *valid*. That is, it agrees with the implied *XML view schema*. (ii) The translated updates against the relational database comply with the *relational schema*, namely, to keep the relational database consistent by update propagation if necessary. (iii) The XML view reconstructed on the updated relational database using the view definition is exactly the same as the result that would be generated by directly updating the materialized view, namely, without *view side-effects*. In this section, we now formally model this problem.

2.1 XML View over Relational Database

The structure of a relation is described by a **relation schema** $\mathcal{R}(\mathcal{N}, \mathcal{A}, \mathcal{F})$, where $\mathcal{N}$ is the name of the relation, $\mathcal{A} = \{a_1, a_2, ..., a_m\}$ is its attribute set, and $\mathcal{F}$ is a set of constraints. A **relation** R is a finite subset of $dom(\mathcal{A})$, a *product* of all the attribute domains. A **relational database**, denoted as D, is a set of n relations $R_1, ..., R_n$. A **relational update operation** $u^R \in \mho^R$ is a deletion, insertion or replacement on a database D. A **sequence** of relational update operations, denoted by $U^R = \{u_1^R, u_2^R, ..., u_p^R\}$ is also modeled as a function $U^R(D) = u_p^R(u_{p-1}^R(..., u_2^R(u_1^R(D))))$. Fig. 1 depicts an example of a relational schema and database which contains a list of books by titles and their (optional) prices.

book

	bookid	title
t_1	98001	TCP/IP Illustrated
t_2	98002	Programming in Unix
t_3	98003	Data on the Web

Legend: Primary Key / Non Key

price

	pid	amount	website
t_1	98001	63.70	www.amazon.com
t_2	98003	56.00	www.amazon.com
t_3	98003	45.60	www.bookpool.com

```
CREATE TABLE book(
  bookid VARCHAR2(20),
  title VARCHAR2(100) NOT NULL,
  CONSTRAINTS BookPK
      PRIMARYKEY (bookid))

CREATE TABLE price(
  bookid VARCHAR2(20),
  amount DOUBLE,
  website VARCHAR2(100),
  CONSTRAINTS PricePK
      PRIMARYKEY (bookid, website),
  FOREIGNKEY (bookid)
      REFERENCES book (bookid))
```

Fig. 1. Relational database

```
<DB>
  <book>
    <row>
      <bookid>98001</bookid>
      <title>TCP/IP Illustrated</title>
    </row> ...
  </book>
  <price>
    <row>
      <bookid>98001</bookid>
      <amount>63.70</amount>
      <website>www.amazon.com</website>
    </row> ...
  </price>
<DB>
```

Fig. 2. Default XML view of database shown in Figure 1

Recent XML systems (XPERANTO [9], SilkRoute [14] and Rainbow [28]) use a basic XML view, called *default XML view*, to define the one-to-one XML-to-relational mapping. Fig. 2 is the default XML view of the relational database shown in Fig. 1.

Table 1. Notations for XML view update problem

D	relational database	$\mathcal{R}(\mathcal{N}, \mathcal{A}, \mathcal{F})$	schema of relation
R	relation	$\mho^R$	domain of relational update operations
u^R	relational update operation	U^R	sequence of relational update operations
V	XML view	DEF^V	XML view definition
u^V	view update	$\mho^V$	domain of view update operations

On top of this default XML view, a (*virtual*) **XML view** V is defined by a **view definition** DEF^V to publish user-specific data. In our case, DEF^V is an XQuery expression (e.g. [23]) called a *view query.* Fig. 3 lists some example view queries used as running examples in this paper. The domain of the view is denoted by $dom(V)$.

For the view definition DEF^V, we do not consider any aggregation and recursion. These operations make views non-updatable, as enunciated in [16]. Also, the predicate used in the view query expression is a conjunction of *non-correlation* or *correlation* predicates defined as below. Given a predicate p of the form $a\ \theta\ b$, where $\theta \in \{=, \neq, <, \leq, >, \geq\}$. We say that p is a *non-correlation predicate* if b is a literal, otherwise, p is said to be a *correlation predicate*. For example, \$*price/website* = *"www.amazon.com"* is a non-correlation predicate while \$*book/bookid* = \$*price/bookid* is an equi-correlation predicate.

Each XML view is associated with an **XML view schema** generated by analyzing both the view definition DEF^V and the schema of the underlying relations [5, 6].

This view schema can also be represented by a graph similar to SilkRoute [14] and other XML publishing work [7, 8]. Fig. 4 shows the view schema for the view query $V1$ as well as its graph representation. Let *rel* be a function to extract the relations in D referenced by DEF^V, then $rel(DEF^V) = \{R_{i_1}, R_{i_2}, ..., R_{i_p}\} \subseteq D$. For example, let DEF^V be the query $Q1$, then $rel(DEF^V) = \{book, price\}$. For each node in the schema graph, we define two functions *Update Context Binding*, denoted by $UCBinding()$, and *Update Point Binding*, denoted by $UPBinding()$. *UCBinding(n)* will extract all the relations referred up to n in DEF^V. *UPBinding(n)* includes all the relations referred below n. The UCBinding and UPBinding for different schema nodes are shown in Fig. 4. Note that $UCBinding(n) \subseteq rel(DEF^V)$ and $UPBinding(n) \subseteq rel(DEF^V)$.

Further, the domain constraints of the XML view schema are consistent with the relational schema. For example, *amount* has the data type decimal, thus takes the domain of decimal. The hierarchical constraints are extracted from the view query. The cardinality constraints such as "minOccurs=1" and "maxOccurs=1" are extracted by combining both the view query and the relational database schema. For instance, *bookid* has "minOccurs=1" and "maxOccurs=1" since it is the key of the underlying relational table *book*. While $price\_info$ has "maxOccurs=unbounded" since the view query defines $price\_info$ elements inside of $book\_info$ element. Recent works [5, 6] study the XML view schema publishing over the relational database. We thus omit the discussion of how to publish the view schema given a view query and the underlying relational database schema. Instead

Q1

```
<bib>
FOR $book IN document("default.xml")/book/row
RETURN {
   <book_info>
      $book/bookid, $book/title,
      FOR $price IN document("default.xml")/price/row
      WHERE $book/bookid = $price/bookid
      RETURN {
         <price_info>
            $price/amount, $price/website
         </price_info>}
   </book_info>
</bib> }
```

V1

```
<bib>
   <book_info>
      <bookid>98001</bookid>
      <title> TCP/IP Illustrated </title>
      <price_info>
         <amount>63.70</amount>
         <website> www.amazon.com</website>
      </price_info>
   </book_info>
   <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
      <price_info>
         <amount>56.00</amount>
         <website> www.amazon.com</website>
      </price_info>
      <price_info>
         <amount>45.60</amount>
         <website>
            www.bookpool.com
         </website>
      </price_info>
   </book_info>
</bib>
```

(a) $V1$ defined by $Q1$

Q2

```
<bib>
FOR $book IN document("default.xml")/book/row,
      $price IN document("default.xml")/price/row
WHERE $book/bookid = $price/bookid
RETURN {
   <price_info>
         $price/amount, $price/website,
         <book_info>
            $book/bookid, $book/title
         </book_info>
   </price_info>
</bib> }
```

V2

```
<bib>
   <price_info>
      <amount>63.70</amount>
      <website> www.amazon.com </website>
      <book_info>
         <bookid>98001</bookid>
         <title>TCP/IP Illustrated</title>
      </book_info>
   </price_info>
   <price_info>
      <amount>56.00</amount>
      <website> www.amazon.com </website>
      <book_info>
         <bookid>98003</bookid>
         <title>Data on the Web</title>
      </book_info>
   </price_info>
   <price_info>
      <amount>45.60</amount>
      <website> www.bookpool.com </website>
      <book_info>
         <bookid>98003</bookid>
         <title>Data on the Web</title>
      </book_info>
   </price_info>
</bib>
```

(b) $V2$ defined by $Q2$

Q3

```
<bib>
FOR $book IN document("default.xml")/book/row,
      $price IN document("default.xml")/price/row
WHERE $book/bookid = $price/bookid
RETURN {
   <book_info>
         $book/bookid, $book/title,
         <price_info>
            $price/amount, $price/website
         </price_info>
   </book_info>
</bib>}
```

V3

```
<bib>
   <book_info>
      <bookid>98001</bookid>
      <title> TCP/IP Illustrated </title>
      <price_info>
         <amount>63.70</amount>
         <website>  www.amazon.com  </website>
      </price_info>
   </book_info>
   <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
      <price_info>
         <amount>56.00</amount>
         <website>  www.amazon.com </website>
      </price_info>
   </book_info>
   <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
      <price_info>
         <amount>45.60</amount>
         <website> www.bookpool.com </website>
      </price_info>
   </book_info>
</bib>
```

(c) $V3$ defined by $Q3$

Q4

```
<bib>
FOR $book IN document("default.xml")/book/row
RETURN {
   <book_info>
      $book/bookid, $book/title,
      FOR $price IN document("default.xml")/price/row
      WHERE $book/bookid = $price/bookid
      RETURN {
         <price_info>
            $book/bookid, $price/amount, $price/website
         </price_info>}
   </book_info>
</bib> }
```

V4

```
<bib>
   <book_info>
      <bookid>98001</bookid>
      <title> TCP/IP Illustrated </title>
      <price_info>
         <bookid>98001</bookid>
         <amount>63.70</amount>
         <website>  www.amazon.com  </website>
      </price_info>
   </book_info>
   <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
      <price_info>
         <bookid>98003</bookid>
         <amount>56.00</amount>
         <website>  www.amazon.com </website>
      </price_info>
      <price_info>
         <bookid>98003</bookid>
         <amount>45.60</amount>
         <website> www.bookpool.com </website>
      </price_info>
   </book_info>
</bib>
```

(d) $V4$ defined by $Q4$

Q5

```
<bib>
FOR $book1 IN document("default.xml")/book/row
RETURN {
   <book_info>
      $book/bookid, $book/title,
   </book_info>},
FOR $book2 IN document("default.xml")/book/row
RETURN {
   <book_info>
      $book/bookid, $book/title,
   </book_info>}
</bib> }
```

V5

```
<bib>
   <book_info>
      <bookid>98001</bookid>
      <title> TCP/IP Illustrated </title>
   </book_info>
   <book_info>
      <bookid>98002</bookid>
      <title>Programming in Unix </title>
   </book_info>
   <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
   </book_info>
   <book_info>
      <bookid>98001</bookid>
      <title> TCP/IP Illustrated </title>
   </book_info>
   <book_info>
      <bookid>98002</bookid>
      <title>Programming in Unix </title>
   </book_info>
   <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
   </book_info>
</bib>
```

(e) $V5$ defined by $Q5$

Fig. 3. View V1 to V5 defined by XQuery Q1 to Q5 respectively

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="price_info_type">
    <xs:sequence>
      <xs:element name="amount" type="xs:decimal"/>
      <xs:element name="website" type="xs:string"/>
    <xs:sequence>
  </xs:complexType>
  <xs:complexType name="book_info_type">
    <xs:sequence>
      <xs:element name="“bookid" type="xs:string" minOccurs="1" maxOccurs="1"/>
      <xs:element name="title" type="xs:string" minOccurs="1"/>
      <xs:element name="price_info" type="price_info_type" maxOccurs="unbounded"/>
    </xs:sequence>
  <xs:complexType>
  <xs:element name="book_info" type="book_info_type" maxOccurs="unbounded"/>
</xs:schema>
```

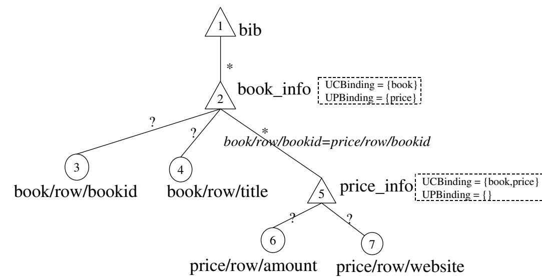


Fig. 4. View Schema and its graph representation for V1 in Fig. 3(a)

we concentrate on the influence that the view schema has on the existence of the view update translation. Some constraints in the view schema, e.g., domain constraints, can be used to check if the given view updates are valid. Other constraints may affect the update translatability of the view, in particular, when the constraints extracted from view definition and relational schema are not consistent with each other.

2.2 XML View Update Problem

2.2.1 Update Operations over XML View

Let $u^V \in \mho^V$ be an update on the view V. An *insertion* adds while a *deletion* removes an element from the XML view. A *replacement* replaces a view element with another view element. Although W3C is adding update capabilities to the XQuery language [23], to date there is no one standard update XQuery syntax. For illustration purposes, our work adopts the update XQuery syntax introduced in [22]. Fig. 5 shows several examples of view updates expressed in this XQuery like language from [22].

A **valid view update** is an insert, delete or replace operation that satisfies all constraints in the view schema. All updates in Fig. 5 are valid updates since they agree with all constraints of the view schema. Figure 6 contains several examples of invalid updates which violate either the Null constraints or the domain constraints. Assuming we have constraints for the relational database in Figure 1 as shown in Figure 6(a), then u_{13} in Figure 6(b) is not a valid update because it conflicts with the NOT NULL constraints for the “title” attribute in the “book” relation. u_{13} in Figure 6(c) is not a valid update either because it disagrees with the domain constraints in the “price” relation.

2.2.2 Update Translation Policy

Clearly, the *update translation policy* chosen for a given translation system is essential for the decision of view update translatability. A given update may be translatable under one policy, while not under another one. We

u^V_1
```
FOR $root IN document("V1.xml"),
      $book IN $root/book_info
WHERE $book/title/text() = " TCP/IP Illustrated"
UPDATE $root {
      DELETE $book }
```

u^V_2
```
FOR $root IN document("V2.xml"),
      $book IN $root/price_info/book_info
WHERE $book/title/text() = " TCP/IP Illustrated"
UPDATE $root {
      DELETE $book }
```

u^V_3
```
FOR $root IN document("V3.xml"),
      $book IN $root/book_info
WHERE $book/title/text() = " Data on the Web"
AND $book/price_info/website = " www.amazon.com"
UPDATE $root {
      DELETE $book }
```

u^V_4
```
FOR $root IN document("V4.xml"),
      $book IN $root/book_info
WHERE $book/title/text() = "Data on the Web"
AND $book/price_info/website = " www.amazon.com"
UPDATE $root {
      DELETE $book/price_info}
```

u^V_5
```
FOR $root IN document("V2.xml"),
      $price IN $root/price_info
WHERE $price/book_info/title/text() = "Data on the Web"
         AND $price/website = "www.amazon.com"
UPDATE $root {
      DELETE $price }
```

u^V_6
```
FOR $root IN document("V3.xml"),
      $book IN $root/book_info
WHERE $book/title/text() = " Data on the Web"
AND $book/price_info/website = " www.amazon.com"
UPDATE $root {
      DELETE $book/price_info }
```

u^V_7
```
FOR $root IN document("V4.xml"),
      $book IN $root/book_info
WHERE $book/title/text() = "Data on the Web"
AND $book/price_info/website = " www.amazon.com"
UPDATE $root {
      DELETE $book }
```

u^V_8
```
FOR $root IN document("V5.xml"),
      $book IN $root/book_info
WHERE $book/title/text() = "Data on the Web"
UPDATE $root {
      DELETE $book }
```

Fig. 5. Update operations on XML views defined in Fig.3

```
Constraints for Book relation:
   title VARCHAR2(100) NOT NULL

Constraints for price relation:
   amount DOUBLE CHECK (amount > 0.00)
```
(a)

u^V_{13}
```
FOR $root IN document("V1.xml"),
      $book IN $root/book
UPDATE $root {
      DELETE $book/title
}
```
(b)

u^V_{14}
```
FOR $root IN document("V1.xml")
UPDATE $root {
   INSERT
      <book>
         <bookid>"98004"<bookid>
         <title>" Data on the Web "</title>
         <price>
            <amount> -100.00 </amount>
            <website>www.ebay.com</website>
         </price>
      </book>
}
```
(c)

Fig. 6. Example of valid update checking

now enumerate common policies observed in the literature [4, 22, 20] and in practice [28].

Policies for update type selection. (1) Same type. The translated update always has the same update type as the given view update. 2) Mixed type. The type of the translated updates could be different from the original view update.

Policies for maintaining referential integrity of the relational database under deletion. (1) Cascade. The directly translated relational updates cascade to update the referencing relations as well. 2) Restrict. The relational update is restricted to the case when there are no referencing relations. Otherwise, reject the view update. 3) Set Null. The relational update is performed as required, while the foreign key is set to be NULL in each dangling tuple.

In our discussion throughout the paper, when not stated otherwise, we will pick the most commonly used policy [11]. That is, (i) translated updates have the same type as the view update and (ii) delete cascading is applied in maintaining referential integrity of the relational database. If a different translation policy were to be used, such as a *restrict deletion* policy in maintaining referential integrity, then our discussion would need to be adjusted accordingly. However, the overall approach will still stay the same irrespective of the policies used.

2.2.3 Update Translatability

A correct translation means the "rectangle" rules shown in Fig. 7 hold. Intuitively, it implies the translated relational updates "exactly" perform the view update, that is, without view side effects.

Definition 1 *Let D be a relational database and V be a virtual view defined over D by the view definition DEF^V. A relational update sequence U^R is a* **correct translation** *of a valid update u^V iff $u^V(DEF^V(D)) = DEF^V(U^R(D))$.*

If a correct translation does not exist under the current update translation policy, u^V is said to be **un-translatable**. Otherwise, it is said to be **translatable**. If additional conditions are required to hold true for finding a correct translation, u^V is said to be **conditionally translatable**. A potential extra condition may be translated update minimization. A valid update u^V is said to be **unconditionally translatable** otherwise.

The problem of **XML view update translatability** is to determine whether an update can be classified as either *unconditionally translatable*, *conditionally translatable* or *un-translatable*. This typical partition of the view update domain $\mho^V$ into classes of update translatable types is shown in Fig. 8.

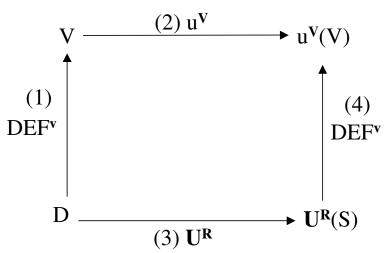


Fig. 7. Correct translation of view update to relational update

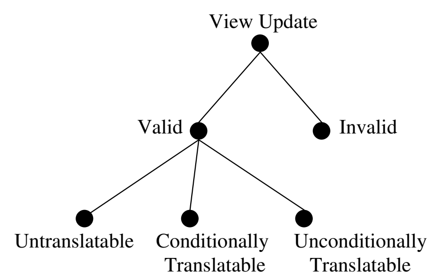


Fig. 8. The partition of view update domain $\mho^V$

3 Theoretical Foundation for Translatability of XML Views

Much work has been done on the existence of a correct translation for various classes of view specifications [2, 13] in the relational context. Especially, Dayal and Bernstein [13] use the concept of "clean source" to characterize the schema conditions under which a relational view over a single relational table is updatable.

However, the relational view update translatability problem addressed in [13] is different from the XML view update translatability problem for two different reasons.

First, update operation can be specified on any element of the XML view, so called different *update granularity*. This flexibility in turn causes more concerns in the translatability study. For example, consider $V3$ in Fig. 3(c), we can specify an update on a *book_info* element. We also can specify it on a *price_info* element. Deleting *price_info* will not be translatable since it will cause the *book_info* to disappear from the view.

Second, Dayal and Bernstein [13] only consider the functional dependencies inside a single relation. However, we notice that the constraints existing in a relational database schema are a main cause of making a view update un-translatable. Especially the integrity constraints such as foreign keys deserve careful consideration since (i) in most practical cases, nesting in XML views is done through the Join operation between Key and Foreign Key constrained hierarchies and (ii) the update propagation through the foreign key, which is used to maintain the referential integrity, is one major reason causing view side effect. Considering integrity constraints makes the view update problem harder than considering only updates over a single relation, for such *propagated updates* may again cause view side effects.

In this section, we propose a *clean extended source* theory to whether a correct translation of a given XML view update exists.

3.1 Extended Source and Clean Extended Source

The key concepts used by our *clean extended source theory* include *extended source* and *clean extended source*.

Recall the *UCBinding()* function for a node in the view schema graph introduced in Section 2. The tuples in the UCBinding relation set, which are used to generate a view element, form its *generator* as defined below.

Definition 2 *Given a relational database D and an XML view V over D. Let v be a view element of V, whose schema node is n. Let $g = \{T_k \mid 1 \leq k \leq \mid UCBinding(n) \mid\}$, where $T_k = \{t_i \mid t_i \in R_k\}$ and $DEF^V(g)$ generates v. We say g is the* **generator** *of v, and $\forall t_i \in g$ is a* **source-tuple** *in D of v.*

As an example, in $V1$ of Fig. 3, consider the *book_info* element with bookid=98001, $UCBinding(book\_info) = \{book\}$. Its generator is given by $g = \{book.t_1\}$, where $book.t_1$ is the book tuple *(98001,TCP/IP Illustrated)*. And $book.t_1$ is a source-tuple. The generator for *price_info* element is $\{book.t_1, price.t_1\}$.

Definition 3 *Let V^0 be a set of view elements in a given XML view V. Let $G(V^0)$ be the set of generators of V^0 defined by $G(V^0) = \{g \mid g$ is a generator of a view-element in $V^0\}$. For each $g \in G(V^0)$, let $H(g)$ be some nonempty subset of g. Then $\cup_{g \in G(V^0)} H(g)$ is a* **source** *in D of V^0. If $G(V^0) = \emptyset$, then V^0 has no source in D.*

Definition 4 *Let S be a source of V^0. Let E be the set of tuples $\{t_j\}$ from the relations $rel(DEF^V)$, where $\exists t_i \in S$ such that t_j refers to t_i through foreign key constraint(s). We say $S_e = S \cup E$ is an* **extended source** *in D of V^0.*

A *source* includes the underlying relational part of a set of view elements V^0, which is sufficient to decide the appearance of V^0. For example, in V1 in Fig. 3, consider V^0 as all the *price_info* elements of the second *book_info* (bookid=98003). We have $G(V^0) = \{g_1, g_2\}$, where $g_1 = \{book.t_3, price.t_2\}$, $g_2 = \{book.t_3, price.t_3\}$. Then $H(g_1)_1 = \{book.t_3\}$, $H(g_1)_2 = \{price.t_2\}$, $H(g_1)_3 = \{book.t_3, price.t_2\}$. And $H(g_2)_1 = \{book.t_3\}$, $H(g_2)_2 = \{price.t_3\}$, $H(g_2)_3 = \{book.t_3, price.t_3\}$. Any combination of $H(g_1)_i$ and $H(g_2)_j$ will be a source of V^0, for example, $S_1 = \{book.t_3\}$ and $S_2 = \{price.t_2, price.t_3\}$. The extended source of S_1 is given by $S_{e1} = \{book.t_3, price.t_2, price.t_3\}$, while $S_{e2} = S_2$.

Definition 5 *Let $D = \{R_1, ..., R_n\}$ be a relational database. Let V^0 be a set of view elements in a given XML view V and S_e be an extended source in D of V^0. S_e is a* **clean extended source** *in D of V^0 iff $(\forall v \in V - V^0)$, $(\exists g)$ such that g is a generator in $(R_1 - S_{e1}, ..., R_n - S_{en})$ of v. Or, equivalently, S_e is a clean extended source in D of V^0 iff $(\forall v \in V - V^0)(S_e \cap g = \emptyset)$, where g is the generator of v.*

A *clean extended source* defines an extended source that is only referenced by the given view element itself. For instance, in V1 in Fig. 3, consider V^0 as the second *book_info* element (bookid=98003). Its source $S = \{book.t_3\}$, and its extended source is $S_e = \{book.t_3, price.t_2, price.t_3\}$. S_e is a clean extended source of V^0. For the *book_info* element (bookid = 98001) in $V2$ (Fig. 3), its extended source is $\{book.t_1, price.t_1\}$, which is not a *clean extended source* since it is also part of the generator of its parent *price_info* element.

3.2 Clean Extended Source Theory

We now establish a connection between *clean extended source* and update translatability by introducing series of theorems. The following theorems form an **clean extended source theory**. This serves as the base theory for identifying whether an update is *translatable*. Although somewhat similar to [13], the theorems below differ in several important ways. Most notably, (i) the key concepts, such as the *generator, source, extended source* and *clean extended source*, now follow the new definitions from Section 3.1 and (ii) XML view elements have different granularity, instead of just the uniform granularity for relational view tuples. For example, in V1 in Fig. 3, one can delete *book_info* element, which also deletes its child *price_info* elements. Or, delete only the *price_info* element.

We assume view update u^V is a valid view update and the update translation policy is the same as before, that is, same type update translation and delete cascading.

Lemma 1, 2 and 3 below are used to prove Theorem 1 and Theorem 2. Let relational database $D = \{R_1, ..., R_n\}$. Let V^0 be a set of XML view elements in a given XML view V and $v \in V^0$ indicates that v is a view-element inside V^0.

Lemma 1

(a) S_e is an extended source in D of V^0 iff $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) \subseteq V - V^0$.

(b) S_e is a clean extended source in D of V^0 iff $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) = V - V^0$.

Lemma 2 *Given a view V defined by DEF^V over D. Let u^V and U^R be updates on V and D (respectively). Let $v \in V$. Then (U^R deletes an extended source of v and U^R does not insert an source-tuple of v) iff $v \notin DEF^V(U^R(D))$.*

Lemma 3 *Let u^V, U^R, V, D be as in Lemma 2. Let $v \in dom(V) - V$. Then U^R inserts source-tuples of v iff $v \in DEF^V(U^R(D))$.*

The following theorems form the core of the clean-extended source theory. The intuition behind is that the relational update sequence correctly translates a view update if and only if it deletes or inserts a clean extended source of the view element. In other words, a deletion or insertion is translatable as long as there is a clean extended source of the view element being deleted or inserted.

Theorem 1 *Let u^V be the deletion of a set of view elements $V^d \subseteq V$. Let τ be a translation procedure, $\tau(u^V, D) = U^R$. Then τ* **correctly translates** *u^V to D iff U^R deletes a clean extended source of V^d.*

By Definition 1, a correct delete translation is one without any view side effect. This is exactly what deleting a clean extended-source guarantees by Definition 5. Thus Theorem 1 follows.

Theorem 2 *Let u^V be the insertion of a set of view elements V^i into V. Let $V^- = V - V^i$, $V^u = V^i - V$. Let τ be a translation procedure, $\tau(u^V, D) = U^R$. Then τ* **correctly translates** *u^V to D iff (i) $(\forall v \in V^u)$(U^R inserts a source tuple of v) and (ii) $(\forall v \in dom(V) - (V^u \cup V^-))$($U^R$ does not insert a source tuple of v).*

Since $dom(V) - (V^u \cup V^-) = (dom(V) - (V^i \cup V)) \cup (V^i \cap V)$, Theorem 2 indicates a correct insert translation is one without any duplicate insertion (insert a source of $V^i \cap V$) and any extra insertion (insert a source of $dom(V) - (V^i \cup V)$). That is, it inserts a clean extended source for the new view-element. Duplicate insertion is not allowed by BCNF, while extra insertion will cause a view side effect. For example, for u_8^V in Fig. 5, let $u_1^R = \{$*Insert (98003,Data on the Web) into book*$\}$, $u_2^R = \{$*Insert (98003,56.00,www.ebay.com) into price*$\}$. Then $U^R = \{u_1^R, u_2^R\}$ is not a correct translation since it inserts a duplicate source tuple into *book*. On the other hand, $U^{R'} = \{u_2^R\}$ is a correct translation.

4 Deciding Factors for Translatability of XML View Updates

Using examples, we now illustrate what factors affect the update translatability, and in particular which features of XML specifically cause new view update translation issues. We only consider insertion and deletion in our discussion. A replacement can be treated as a deletion followed by an insertion and is not specifically discussed in our update translatability study.

In our discussion below, when not stated otherwise, we will pick the most commonly used policy. That is, (i) translated updates have the same type as the view update and (ii) delete cascading is applied in maintaining referential integrity of the relational database. If a different translation policy is used, such as a *restrict deletion* policy in maintaining referential integrity, then the discussion on update translatability below can be adjusted accordingly. Also, we do not indicate the order of the translated relational updates. However, for a given execution strategy, the correct order can be easily decided [16, 17, 1, 22, 25].

Q1

```
<bib>
FOR $book IN document("default.xml")/book/row
RETURN {
  <book_info>
    $book/bookid, $book/title,
    FOR  $price  IN document("default.xml")/price/row
    WHERE $book/bookid = $price/bookid
    RETURN {
      <price_info>
        $price/amount, $price/website
      </price_info>}
  </book_info>
</bib> }
```

V1

```
<bib>
    <book_info>
        <bookid>98001</bookid>
        <title> TCP/IP Illustrated </title>
        <price_info>
            <amount>63.70</amount>
            <website> www.amazon.com</website>
        </price_info>
    </book_info>
    <book_info>
        <bookid>98003</bookid>
        <title>Data on the Web</title>
        <price_info>
            <amount>56.00</amount>
            <website> www.amazon.com</website>
        </price_info>
        <price_info>
            <amount>45.60</amount>
            <website>
                www.bookpool.com
            </website>
        </price_info>
    </book_info>
</bib>
```

(a) The XML view $V1$ defined by Query $Q1$

u^V_1

```
FOR $root IN document("V1.xml"),
      $book IN $root/book_info
WHERE $book/title/text() = " TCP/IP Illustrated"
UPDATE $root {
      DELETE $book }
```

(b) View update u_1 over $V1$

```
<bib>
    <book_info>
        <bookid>98003</bookid>
        <title>Data on the Web</title>
        <price_info>
            <amount>56.00</amount>
            <website>www.amazon.com</website>
        </price_info>
        <price_info>
            <amount>45.60</amount>
            <website>www.bookpool.com</website>
        </price_info>
    </book_info>
</bib>
```

(c)$V1'$: The user expected updated XML view

u_1^R:

```
DELETE FROM book
WHERE book.ROWID IN (
   SELECT DISTINCT book.ROWID FROM book
   WHERE (book.title = 'TCP/IP Illustrated') )
```

u_2^R:

```
DELETE FROM price
WHERE price.ROWID IN (
   SELECT DISTINCT price.ROWID FROM book,price
   WHERE (book.title = 'TCP/IP Illustrated') AND
         (book.bookid = price.pid) )
```

(d) U^R: The translated update sequence

book

bookid	title
98002	Programming in Unix
98003	Data on the Web

Legend: Primary Key / Non Key

price

pid	amount	website
98003	56.00	www.amazon.com
98003	45.60	www.bookpool.com

(e) D': The updated relational database

(f) $Q1(D')$: The regenerated view is the same as (c).

Fig. 9. Example of Translating Update u_1^V on $V1$.

Q2

```
<bib>
FOR $book IN document("default.xml")/book/row,
      $price IN document("default.xml")/price/row
WHERE $book/bookid = $price/bookid
RETURN {
   <price_info>
        $price/amount, $price/website,
        <book_info>
           $book/bookid, $book/title
        </book_info>
   </price_info>
</bib> }
```

V2

```
<bib>
   <price_info>
      <amount>63.70</amount>
      <website> www.amazon.com </website>
      <book_info>
         <bookid>98001</bookid>
         <title>TCP/IP Illustrated</title>
      </book_info>
   </price_info>
   <price_info>
      <amount>56.00</amount>
      <website> www.amazon.com </website>
      <book_info>
         <bookid>98003</bookid>
         <title>Data on the Web</title>
      </book_info>
   </price_info>
   <price_info>
      <amount>45.60</amount>
      <website> www.bookpool.com </website>
      <book_info>
         <bookid>98003</bookid>
         <title>Data on the Web</title>
      </book_info>
   </price_info>
</bib>
```

(a) The XML view $V2$ defined by Query $Q2$

u^V_2

```
FOR $root IN document("V2.xml"),
      $book IN $root/price_info/book_info
WHERE $book/title/text() = " TCP/IP Illustrated"
UPDATE $root {
      DELETE $book }
```

(b) An view update u_2^V over $V2$

```
<bib>
   <price_info>
      <amount>63.70</amount>
      <website>www.amazon.com</website>
   </price_info>
   <price_info>
      <amount>56.00</amount>
      <website>www.amazon.com</website>
      <book_info>
         <bookid>98003</bookid>
         <title>Data on the Web</title>
      </book_info>
   </price_info>
   ...
</bib>
```

(c) $V2'$: The user expected updated XML view

```
u1R: DELETE FROM book
     WHERE book.ROWID IN (
        SELECT DISTINCT book.ROWID FROM book
        WHERE (book.title = 'TCP/IP Illustrated') )
```

```
u2R: DELETE FROM price
     WHERE price.ROWID IN (
        SELECT DISTINCT price.ROWID FROM book,price
        WHERE (book.title = 'TCP/IP Illustrated') AND
              (book.bookid = price.pid) )
```

(d) U^R: The translated update

book

bookid	title
98002	Programming in Unix
98003	Data on the Web

Legend: Primary Key, Non Key

price

pid	amount	website
98003	56.00	www.amazon.com
98003	45.60	www.bookpool.com

(e) D': The updated relational database

```
<bib>
   <price_info>
      <amount>56.00</amount>
      <website>www.amazon.com</website>
      <book_info>
         <bookid>98003</bookid>
         <title>Data on the Web</title>
      </book_info>
   </price_info>
   ...
</bib>
```

(f) $Q2(D')$: The regenerated view

Fig. 10. Example of Translating Update u_2^V on $V2$.

Q3

```
<bib>
FOR $book IN document("default.xml")/book/row,
     $price IN document("default.xml")/price/row
WHERE $book/bookid = $price/bookid
RETURN {
   <book_info>
        $book/bookid, $book/title,
        <price_info>
           $price/amount, $price/website
        </price_info>
   </book_info>
</bib>}
```

V3

```
<bib>
   <book_info>
      <bookid>98001</bookid>
      <title> TCP/IP Illustrated </title>
      <price_info>
         <amount>63.70</amount>
         <website>  www.amazon.com  </website>
       </price_info>
   </book_info>
   <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
      <price_info>
         <amount>56.00</amount>
         <website>  www.amazon.com </website>
      </price_info>
   </book_info>
   <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
      <price_info>
         <amount>45.60</amount>
         <website> www.bookpool.com </website>
      </price_info>
   </book_info>
</bib>
```

(a) The XML view $V3$ defined by Query $Q3$

u^V_3

```
FOR $root IN document("V3.xml"),
     $book IN $root/book_info
WHERE $book/title/text() = " Data on the Web"
AND $book/price_info/website = " www.amazon.com"
UPDATE $root {
     DELETE $book }
```

(b) An view update u^V_3 over $V3$

```
<bib>
   <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
      <price_info>
         <amount>56.00</amount>
         <website>  www.amazon.com </website>
      </price_info>
   </book_info>
   <book_info>
      <bookid>98003</bookid>
      <title>Data on the Web</title>
      <price_info>
         <amount>45.60</amount>
         <website> www.bookpool.com </website>
      </price_info>
   </book_info>
</bib>
```

(c) $V3'$: The user expected updated XML view

```
u1R:  DELETE FROM book
      WHERE book.ROWID IN (
         SELECT DISTINCT book.ROWID FROM book
         WHERE (book.title = 'Data on the Web') )
```

```
u2R:  DELETE FROM price
      WHERE price.ROWID IN (
         SELECT DISTINCT price.ROWID FROM book,price
         WHERE (book.title = 'Data on the Web') AND
                  (book.bookid = price.bookid)  AND
                  (price.website = " www.amazon.com") )
```

(d) U^R: The directly translated update sequence

```
u2R:  DELETE FROM price
      WHERE price.ROWID IN (
         SELECT DISTINCT price.ROWID FROM book,price
         WHERE (book.title = 'Data on the Web') AND
                  (book.bookid = price.pid)  AND
               (price.website = " www.amazon.com") )
```

(e) $U^{R'}$: The translated update sequence after condition checking

book

bookid	title
98001	TCP/IP Illustrated
98002	Programming in Unix
98003	Data on the Web

price

pid	amount	website
98001	63.7	www.amazon.com
98003	45.60	www.bookpool.com

Legend:
- **Primary Key**
- **Non Key**

(f) D': The updated relational database

(g) $Q3(D')$: The regenerated view is the same as (c).

Fig. 11. Example of Translating Update u^V_3 on $V3$.

Example 1 : View construction consistency.

Given the two XQuery view definitions $Q1$ *and* $Q2$ *in Fig. 3. Both define views representing all books with its price (if any), though each with a different XML view hierarchy. Two view updates* u_1^V *and* u_2^V *(Fig. 5) delete a "book_info" element from* $V1$ *and* $V2$ *respectively.*

(i) u_1^V *is unconditionally translatable as shown by Fig. 9. The translated relational update sequence* U^R *in Fig. 9(d) will delete the first book from the "book" relation using* u_1^R*, and its price information from the "price" relation through* u_2^R*. By re-applying the view query* $Q1$ *on the updated database* D' *in Fig. 9(e), the user would get the updated XML view in Fig. 9(f). This regenerated XML view is equal to the expected updated view* $V1'$ *in Fig. 9(c). Hence* U^R *in Fig. 9(d) is a correct translation of* u_1^V*.*

(ii) u_2^V *is un-translatable as shown by Fig. 10. First, the relational update* u_1^R *in Fig. 10(d) is generated to delete the book (bookid=98001) from the "book" relation. We note the existing foreign key from the "price" relation to the "book" relation as shown by the relational database schema (Fig. 1). Then according to our selected update translation policy, the second update operation* u_2^R *will be generated by the update translator to keep the relational database consistent. That is, the corresponding price of the deleted book will be deleted as well. By reapplying* $Q2$ *on this updated database in Fig. 10(e), we will produce the updated view in Fig. 10(f). This view is different than the expected updated view* $V2'$ *in Fig. 10(c). Also, it can be shown that no other translation is available which could preserve consistency either, since the translated update can not be further minimized. Thus* u_2^V *is said to be un-translatable.*

As can be seen here, this difference in the existence of a correct translation is caused by the mismatch between the XML hierarchical view model and the underlying flat relational base model. This view construction consistency property, namely, whether the XML view hierarchy agrees with the hierarchical structure implied by the base relational schema, is one of the key factors for deciding XML view update translatability.

Example 2 : Content duplication.

Next we compare the two virtual XQuery views $V1$ *and* $V3$ *in Fig. 3. The book (bookid=98003) with two prices is exposed twice in* $V3$*, while only once in* $V1$*. The update* u_3^V *in Fig. 11 will delete the second "book_info" element, which is from the website "www.amazon.com", while keeping the third "book_info" element from "www.bookpool.com" (Fig. 11(c)). A direct translation for this update is given in Fig. 11(d), that is, to delete both the book tuple as well as the price tuple. However, the book tuple (bookid=98003) in the "book" table is still being referenced to by another part of the view (the third "book_info" element). Thus deleting this book tuple will cause view side effects. By using an additional condition, such as an extra translation rule like "No underlying tuple is deleted if it is still referenced by any other part of the view", we can minimize* U^R *so that it deletes the price only (Fig. 11(e)). The updated relational database is shown in Fig. 11(f). The regenerated view (Fig. 11(g)) will be the same as expected by the* user. Thus $U^{R'}$ now makes the update u_3^V translatable. We thus say that u_3^V is conditionally translatable.

The "duplication" causing this ambiguity is introduced by the XQuery "FOR" expressions. We call it content duplication. While not unique to XML, it is rather common here due to XML views being generated by nested XQuery expressions. It may appear in relational views as well, for example in Join views. Some extra deletion rule such as the zero reference deletion in our example is also commonly used for handling content duplication in the relational context.

Q4

```
<bib>
FOR $book IN document("default.xml")/book/row
RETURN {
  <book_info>
    $book/bookid, $book/title,
    FOR $price IN document("default.xml")/price/row
    WHERE $book/bookid = $price/bookid
    RETURN {
      <price_info>
        $book/bookid, $price/amount, $price/website
      </price_info>}
  </book_info>
</bib> }
```

V4

```
<bib>
  <book_info>
    <bookid>98001</bookid>
    <title> TCP/IP Illustrated </title>
    <price_info>
      <bookid>98001</bookid>
      <amount>63.70</amount>
      <website> www.amazon.com </website>
    </price_info>
  </book_info>
  <book_info>
    <bookid>98003</bookid>
    <title>Data on the Web</title>
    <price_info>
      <bookid>98003</bookid>
      <amount>56.00</amount>
      <website> www.amazon.com </website>
    </price_info>
    <price_info>
      <bookid>98003</bookid>
      <amount>45.60</amount>
      <website> www.bookpool.com </website>
    </price_info>
  </book_info>
</bib>
```

(a) The XML view $V4$ defined by Query $Q4$

u_4^V

```
FOR $root IN document("V4.xml"),
    $book IN $root/book_info
WHERE $book/title/text() = "Data on the Web"
AND $book/price_info/website = " www.amazon.com"
UPDATE $root {
    DELETE $book/price_info}
```

(b) An view update u_4^V over $V4$

```
<bib>
  <book_info>
    <bookid>98003</bookid>
    <title>Data on the Web</title>
    <price_info>
      <bookid>98003</bookid>
      <amount>56.00</amount>
      <website> www.amazon.com </website>
    </price_info>
  </book_info>
  <book_info>
    <bookid>98003</bookid>
    <title>Data on the Web</title>
    <price_info>
      <bookid>98003</bookid>
      <amount>45.60</amount>
      <website> www.bookpool.com </website>
    </price_info>
  </book_info>
</bib>
```

(c) $V4'$: The user expected updated XML view

```
u1R: DELETE FROM book
     WHERE book.ROWID IN (
        SELECT DISTINCT book.ROWID FROM book
        WHERE (book.title = 'Data on the Web') )
```

```
u2R: DELETE FROM price
     WHERE price.ROWID IN (
        SELECT DISTINCT price.ROWID FROM book,price
        WHERE (book.title = 'Data on the Web') AND
              (book.bookid = price.bookid) AND
              (price.website = " www.amazon.com") )
```

(d) U^R: The directly translated update sequence

```
u2R: DELETE FROM price
     WHERE price.ROWID IN (
        SELECT DISTINCT price.ROWID FROM book,price
        WHERE (book.title = 'Data on the Web') AND
              (book.bookid = price.pid) AND
              (price.website = " www.amazon.com") )
```

(e) $U^{R'}$: The translated update sequence after condition checking

book

bookid	title
98001	TCP/IP Illustrated
98002	Programming in Unix
98003	Data on the Web

Legend: Primary Key; Non Key

price

pid	amount	website
98001	63.7	www.amazon.com
98003	45.60	www.bookpool.com

(f) D': The updated relational database

(g) $Q4(D')$: The regenerated view is the same as (c).

Fig. 12. Example of Translating Update u_4^V on $V4$

Example 3 : Structural duplication.

As shown in Fig. 12, the bookid element is exposed twice by Q4. Hence each "price_info" within the "book_info" element will also have a "bookid" element. The update u_4^V *deletes the first price of the specified book is ambiguous in translation. Since the update operation touches the primary key "book.bookid", a directly translation, as shown by* U^R *in Fig. 12(d), will delete the corresponding book tuple. This will cause the side effect, since the bookid and title in the view will also disappear. However, with an additional condition, such as knowledge of the user intention about the update, or having the assumption of a zero reference deletion, the new update translation* $U^{R'}$ *in Fig. 12(e) would only delete the price tuple, leaving the book tuple untouched. This update is classified as conditionally translatable. Structural duplication, as illustrated above, is another case causing update translatability problem.*

Example 4 : Update granularity.

In some sense, the importance of the update granularity on translatability has been illustrated by the examples above. The following example now highlights it specifically.

Compared with the failure of translating u_2^V *in Example 1, the update* u_5^V *in Fig. 5 on the same view* $V2$ *is conditionally translatable as shown by Fig. 13.* u_5^V *deletes the whole "price_info" element instead of just the sub-element "book_info" from* $V2$ *(Fig. 13(c)). The directly translated relational update sequence* U^R *in Fig. 13(d) deletes the book (bookid=98003) and its price from amazon. This will cause a view side effect since the book is also referenced by another book_info element with a price from bookpool. By using an additional condition, such as an extra translation rule like "No underlying tuple is deleted if it is still referenced by any other part of the view", we can minimize* U^R *as deleting the price only (Fig. 13(e)). The updated relational database is shown in Fig. 13(f). The regenerated view (Fig. 13(g)) will be the same as user expected. Thus* $U^{R'}$ *now makes the update* u_5^V *translatable. Thus, for the same view* $V2$*, updates on different elements have different translatability.*

Here the difference in translatability is not just caused by the shape of the view structure as in Example 1, but also by the granularity of the update operation. That is the effects that updates have on the view. The update u_5^V *has a larger granularity than* u_2^V*, covering the "top" element of the XML view. It thus "resolves" the inconsistency between the two hierarchies respectively from the view and the relational schema mentioned by Example 1. The XML hierarchical structure offers an opportunity for different update granularity. This is an issue that does not arise for relational views.*

5 STAR: Schema-driven Update Translatability Reasoning

As depicted in Section 4, several factors can affect translatability of updates on XML views. In this section, we propose a **schema-driven update translatability reasoning** (STAR) algorithm to identify these factors

Q2

```
<bib>
FOR $book IN document("default.xml")/book/row,
    $price IN document("default.xml")/price/row
WHERE $book/bookid = $price/bookid
RETURN {
  <price_info>
      $price/amount, $price/website,
      <book_info>
         $book/bookid, $book/title
      </book_info>
  </price_info>
</bib> }
```

V2

```
<bib>
   <price_info>
      <amount>63.70</amount>
      <website> www.amazon.com </website>
      <book_info>
         <bookid>98001</bookid>
         <title>TCP/IP Illustrated</title>
      </book_info>
   </price_info>
   <price_info>
      <amount>56.00</amount>
      <website> www.amazon.com </website>
      <book_info>
         <bookid>98003</bookid>
         <title>Data on the Web</title>
      </book_info>
   </price_info>
   <price_info>
      <amount>45.60</amount>
      <website> www.bookpool.com </website>
      <book_info>
         <bookid>98003</bookid>
         <title>Data on the Web</title>
      </book_info>
   </price_info>
</bib>
```

(a) The XML view $V2$ defined by Query $Q2$

u^V_5

```
FOR $root IN document("V2.xml"),
    $price IN $root/price_info
WHERE $price/book_info/title/text() = "Data on the Web"
    AND $price/website = "www.amazon.com"
UPDATE $root {
    DELETE $price }
```

(b) An view update u^V_5 over $V5$

```
<bib>
   <price_info>
      <amount>63.70</amount>
      <website> www.amazon.com </website>
      <book_info>
         <bookid>98001</bookid>
         <title>TCP/IP Illustrated</title>
      </book_info>
   </price_info>
   <price_info>
      <amount>45.60</amount>
      <website> www.bookpool.com </website>
      <book_info>
         <bookid>98003</bookid>
         <title>Data on the Web</title>
      </book_info>
   </price_info>
</bib>
```

(c) $V2'$: The user expected updated XML view

```
u1R: DELETE FROM book
     WHERE book.ROWID IN (
        SELECT DISTINCT book.ROWID FROM book
        WHERE (book.title = 'Data on the Web') )
```

```
u2R: DELETE FROM price
     WHERE price.ROWID IN (
        SELECT DISTINCT price.ROWID FROM book,price
        WHERE (book.title = 'Data on the Web') AND
              (book.bookid = price.bookid) AND
              (price.website = " www.amazon.com") )
```

(d) U^R: The directly translated update sequence

```
u2R: DELETE FROM price
     WHERE price.ROWID IN (
        SELECT DISTINCT price.ROWID FROM book,price
        WHERE (book.title = 'Data on the Web') AND
              (book.bookid = price.pid) AND
              (price.website = " www.amazon.com") )
```

(e) $U^{R'}$: The translated update sequence after condition checking

book

bookid	title
98001	TCP/IP Illustrated
98002	Programming in Unix
98003	Data on the Web

Legend: Primary Key; Non Key

price

pid	amount	website
98001	63.7	www.amazon.com
98003	45.60	www.bookpool.com

(f) D': The updated relational database

(g) $Q2(D')$: The regenerated view is the same as (c).

Fig. 13. Example of Translating Update u^V_5 on $V2$

and their effects on the update translatability based on the clean extended source theory introduced in Section 3. The STAR algorithm includes two major steps. A *STAR Marking Step* first pre-codes each node in the ASG with a pair of labels (UContext|UPoint) to indicate its update properties. This analysis is performed only once at compile time. The resulting labels will be used to classify the given view update into one of the three translatability categories by a *STAR checking step*.

We assume the relational database is in the BCNF form and no cyclic dependency caused by integrity constraints among relations exists. The reason for this requirement is that in our approach we use the functional dependencies and integrity constraints of the relational database (such as keys, foreign keys, etc.) to determine the update propagation.

5.1 The Annotated Schema Graph

Intuitively, the update translatability problem is closely related with constraints existing in the underlying relational database or enforced by the view definition. In other words, the constraint knowledge will help us to decide whether a given update is valid, or even translatable. The *non-correlation predicates* in the view query form *local constraints*, while the other constraints in the view query, such as *correlation predicates*, *cardinality constraints* and *hierarchical structure* form *global constraints*.

In the relational schema, the *local constraints* include all the constraints specified over one single relation, such as domain constrains, NOT NULL constraints, Key, UNIQUE and Check constraints for the domain. The *global constraints* include constraints specified over multiple relations in the relational schema, in our case these are foreign key constraints.

Intuitively, constraints that affect only one tuple of a base relation or one view element are called *local constraints*. Otherwise, they are called *global constraints*. Figure 14 lists all the constraints considered affecting the update translatability.

We use *Annotated Schema Graphs* (ASG) to model these constraints. The constraints can be analyzed at compile time, as soon as the view query and relational schema are available. Thereafter they can be reused for any future update checking specified over this same view. The local constraints are used to decide whether the given update is valid. Since our focus in this paper is on whether a valid update is translatable, for simplicity, we thus omit the local constraints in our graph representation. For details of utilizing constraints please refer to [27].

The **view ASG**, denoted by $\mathcal{G}_V$, is a forest representing the hierarchical structure of the *XML view*. Let $N_{\mathcal{G}_V}$ and $E_{\mathcal{G}_V}$ respectively denote the nodes and edges of $\mathcal{G}_V$. Computing $\mathcal{G}_V$ is done similarly as in SilkRoute [19]. An internal node, $n \in N_{\mathcal{G}_V}$, represented by a triangle $\triangle$, identifies a view element or attribute labeled by its XML tag name. A leaf node $n \in N_{\mathcal{G}_V}$, represented by a small circle $\circ$, is an atomic type, labeled by both the XPath

	View Query	Relational Database Schema
Local Constraints	• Non-correlation predicates	• Domain constraints • Not Null • Key constraints • Unique constraints • Check constraints
Global Constraints	• Hierarchical structure • Cardinality between elements • Correlation predicates	• Foreign key constraints • *Assertion*

Fig. 14. Constraints considered in update translatability study

expressed in the view query and the name of its corresponding relational column.

Given two nodes $n_1, n_2 \in N_{\mathcal{G}_V}$, the edge $e(n_1, n_2) \in E_{\mathcal{G}_V}$ represents that n_1 is a parent of n_2 in the view hierarchy. Each edge is annotated by its cardinality type (inferred from the view query) and its condition (if any), extracted from the correlation predicate in the view query. The cardinality type is one of types from the enumeration domain $\{?, *\}$, representing $1:\{0,1\}$ (single), and $1:n$ (multiple) respectively. Figures 15(a) to 15(e) depict the view ASGs for $V1$ to $V5$ respectively.

Note that we assume there is always a root tag to enclose the FLWR expression in the view query. Otherwise, we would add a "dummy" root node instead. Also, without loss of generality, given an edge (n_1, n_2), we assume that $|UCBinding(n_2) - UCBinding(n_1)| \leq 1$. That is there is at most one more relation referred for defining n_2 than n_1. Otherwise, dummy nodes are added to split the edge and guarantee this property.

The **Base ASG** $\mathcal{G}_D$ is a graph that captures the hierarchical and cardinality constraints inferred from the key and foreign key constraints. Let $N_{\mathcal{G}_D}$ denote the nodes and $E_{\mathcal{G}_D}$ denote the edges. $\mathcal{G}_D$ is computed as follows.

For each leaf node in the view ASG, there exists a corresponding relational attribute. The union of all these relational attributes forms the leaf nodes of the base ASG. A leaf node labeled by the primary key attribute of a relation is called a *key node* (depicted by a black circle •). For a leaf node n_l labeled with $R.a$, we introduce a node n corresponding to R and an edge (n, n_l).

For any two nodes n_1, n_2 that correspond to relations R, S respectively, we introduce an edge (n_1, n_2), if there is a foreign key from S to R. The base ASGs of the view queries $Q1$ to $Q4$ are identical as shown in Fig. 16(a), since all of them are defined over the same base relations. Fig. 16(b) is the $\mathcal{G}_D(V)$ of V5.

5.2 Closure

We use the concept of closure in $\mathcal{G}_V$ and $\mathcal{G}_D$ to indicate the effect of an update on the view and on the relational database respectively. For simplicity, the cardinality of ? are omitted.

The **closure** of a node n in view ASG $\mathcal{G}_V$, denoted by n^+, is defined as follows: (1) If n is a leaf node, $n^+ = \{n\}$. (2) Otherwise, n^+ is the union of its children's closures grouped by their hierarchical relationship and

Q1

```
<bib>
FOR $book IN document("default.xml")/book/row
RETURN {
   <book_info>
      $book/bookid, $book/title,
      FOR  $price  IN document("default.xml")/price/row
      WHERE $book/bookid = $price/bookid
      RETURN {
         <price_info>
            $price/amount, $price/website
         </price_info>}
   </book_info>
</bib> }
```

Q2

```
<bib>
FOR $book IN document("default.xml")/book/row,
       $price IN document("default.xml")/price/row
WHERE $book/bookid = $price/bookid
RETURN {
   <price_info>
          $price/amount, $price/website,
          <book_info>
             $book/bookid,  $book/title
          </book_info>
   </price_info>
</bib> }
```

Q3

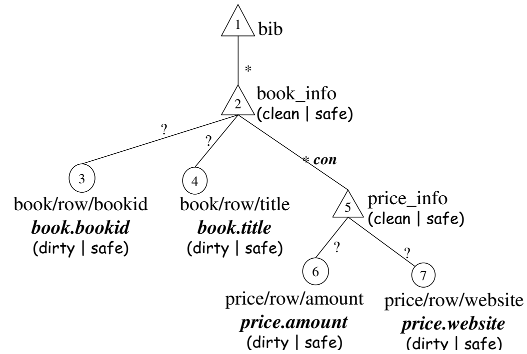
```
<bib>
FOR $book IN document("default.xml")/book/row,
       $price IN document("default.xml")/price/row
WHERE $book/bookid = $price/bookid
RETURN {
   <book_info>
          $book/bookid, $book/title,
          <price_info>
             $price/amount, $price/website
          </price_info>
   </book_info>
</bib>}
```

Q4

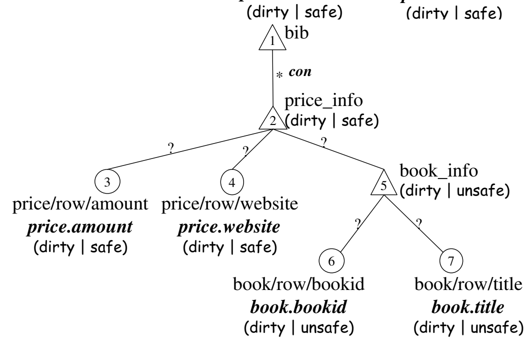
```
<bib>
FOR $book IN document("default.xml")/book/row
RETURN {
   <book_info>
      $book/bookid,  $book/title,
      FOR  $price  IN document("default.xml")/price/row
      WHERE $book/bookid = $price/bookid
      RETURN {
         <price_info>
            $book/bookid, $price/amount, $price/website
         </price_info>}
   </book_info>
</bib> }
```

Q5

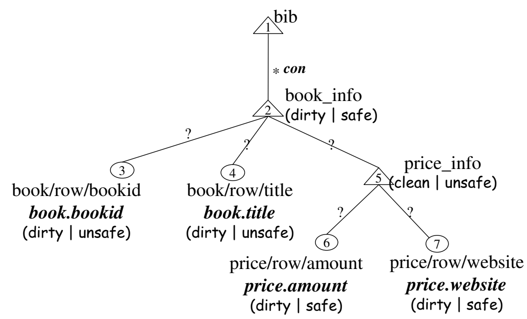
```
<bib>
FOR $book1 IN document("default.xml")/book/row
RETURN {
   <book_info>
      $book/bookid, $book/title,
   </book_info>},
FOR $book2 IN document("default.xml")/book/row
RETURN {
   <book_info>
      $book/bookid, $book/title,
   </book_info>}
</bib> }
```



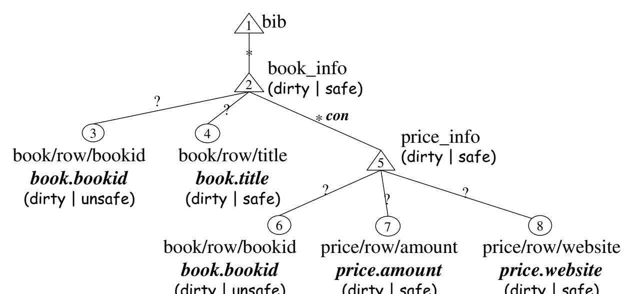
(a) $\mathcal{G}_V$ of V1



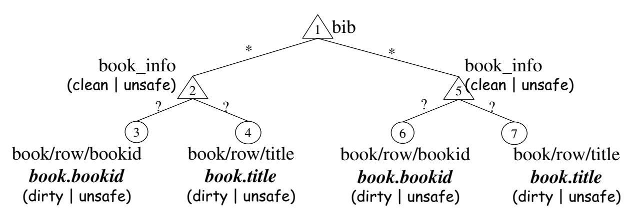
(b) $\mathcal{G}_V$ of V2



(c) $\mathcal{G}_V$ of V3



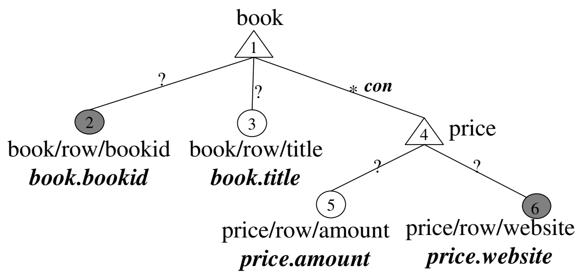
(d) $\mathcal{G}_V$ of V4



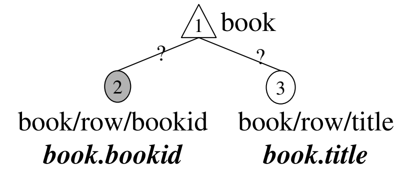
(e) $\mathcal{G}_V$ of V5

*Note: con = (book/row/bookid=price/row/bookid)

Fig. 15. $\mathcal{G}_V$ of V1 to V5 as shown by (a) to (e)



(a) *Note: con = (book.bookid=price.bookid)



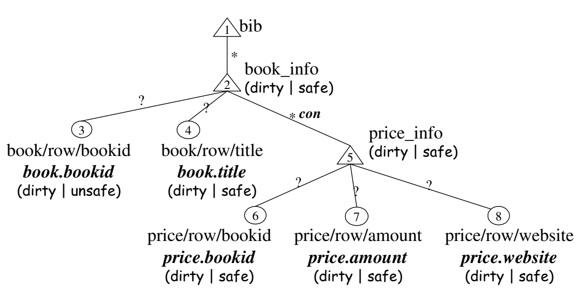
(b)

Fig. 16. (a) $\mathcal{G}_D$ of V1 – V4 (b) $\mathcal{G}_D$ of V5

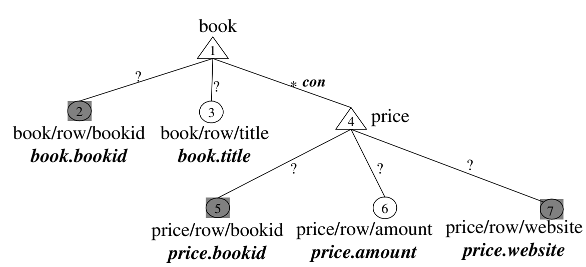
marked by their cardinality. For example, in Figure 15(a), $(n_3)^+_{\mathcal{G}_V} = \{n_3\}$, while $(n_5)^+_{\mathcal{G}_V} = \{n_6, n_7\}$, $(n_2)^+_{\mathcal{G}_V} = \{n_3, n_4, (n_6, n_7)^*\}$. To reduce the closure definition, the group mark "()" can be eliminated if its cardinality mark is "?". For example, in Figure 15(c), $(n_2)^+_{\mathcal{G}_D} = \{n_3, n_4, (n_6, n_7)\} = \{n_3, n_4, n_6, n_7\}$. Especially when we compute the closure of the root, if there is a single * edge $e = (root, n_k)$ starting from the root, that is without any predicate condition, then we define $root^+ = n_k^+$. For example, in Fig. 15(a), $n_1^+ = n_2^+$.

The **closure** of an internal node in base ASG $\mathcal{G}_D$ is defined as the union of its children leaf nodes and the closure of its non-leaf child nodes. For instance, in Fig. 16(a), since $(n_4)^+_{\mathcal{G}_D} = \{n_5, n_6\}$, we have $(n_1)^+_{\mathcal{G}_D} = \{n_2, n_3, (n_5, n_6)^*\}$. As the relational database is a tuple-based data model, the closure of a leaf node is the same as the closure of its parent node. For example, $(n_2)^+_{\mathcal{G}_D} = (n_3)^+_{\mathcal{G}_D}$.

Note that this closure definition is based on the pre-selected update policy: *same type* and *delete cascade*. In the *same type* policy, a delete (insert) can only be translated to one or more deletes (inserts). An alternative is the *mixed type* policy, where a delete (insert) can also be translated into replacements. For example, the delete on n_6 in Fig. 15(a) can be translated to replace the *price.amount* with NULL. In this *mixed type* policy, the closure definition in the base ASG $\mathcal{G}_D$ needs to be adjusted as follows: "the leaf node with property as key has the same closure as its parent, otherwise, its closure includes only itself." For example, in Fig. 16(a), under the mixed policy $(n_3)^+_{\mathcal{G}_D} = \{n_3\}$, while $(n_2)^+_{\mathcal{G}_D} = \{n_2, n_3, (n_5, n_6)^*\}$.



(a) *Note: con = (book/row/bookid=price/row/bookid)



(b) *Note: con = (book.bookid=price.bookid)

Fig. 17. (a) An Example View ASG $\mathcal{G}_V$ and (b) The Base ASG $\mathcal{G}_D$ for View in (a)

Similarly, for the *Set Null* policy, the closure of an internal node is the union of its children leaf nodes and other descendant nodes which would be set as NULL. For example, consider a view defined by view ASG in Fig. 17(a), its base ASG is shown in Fig. 17(b). Node n_5 represents the *price.bookid* attribute and will be set NULL when the foreign key constraint is maintained. According to the integrity maintenance policy *Set Null*, we would now have $n_1^+ = \{n_2, n_3, n_5\}$.

Note that the policy used affects only the closure definitions of the base ASG, while the rest of our steps for translatability checking remain the same.

Closure Comparison. Let getNodes() be a function to extract all the nodes from a given closure, while Distinct() removes duplicates. We define $C_1 \subseteq C_2$, if C_1 appears in C_2. In Fig. 16(a), $n_4^+ = \{n_5, n_6\}$ and $n_1^+ = \{n_2, n_3, (n_5, n_6)^{*con}\}$, thus $n_4^+ \subseteq n_1^+$. Two closures C_1 and C_2 are **equal**, denoted by $C_1 \equiv C_2$, if $C_1 \subseteq C_2$ and $C_1 \supseteq C_2$. In Fig. 16(a), $n_5^+ \equiv n_6^+$.

Mapping Closure. While above we defined the closure for a single node n, we now define the closure of a set of nodes N, denoted by N^+, as $N^+ = \bigsqcup_{(n_i \in N)} n_i^+$, where $\bigsqcup$ is a "Union-like" operation that combines the nodes but eliminates duplicates. That is, if $n_k^+ \subseteq n_j^+$, $N^+ = \bigsqcup_{(n_i \in N, i \neq k)} n_i^+$. For instance, in Fig. 16, $(n_1, n_4)^+ = (n_1)^+ \bigsqcup (n_4)^+ = (n_1)^+ = \{n_2, n_3, (n_5, n_6)^{*con}\}$.

For a node n in the view ASG $\mathcal{G}_V$, we define its *mapping closure* in $\mathcal{G}_D$ as follows. First we compute $C_V = n^+$ in $\mathcal{G}_V$. Let $N = Distinct(getNodes(C_V))$. For each node $n_i \in N$, its mapping leaf node n_i' in $\mathcal{G}_D$ is the one with the same annotation as n_i. Let N' denote the set of mapping nodes for all nodes in N. Let $C_D = N'^+$ in $\mathcal{G}_D$. We call C_D as the **mapping closure** of n. For example, the mapping closure of n_2 in Fig. 15(a) is $C_D = \{n_2, n_3, (n_5, n_6)^{*con})\}$.

5.3 STAR Marking Step

As shown by our motivation example in Section 4, the construction consistency and duplication are two major reasons causing view side effect. The update translatability algorithm needs to detect the appearance of these factors.

The *closure* defined in both ASGs captures the construction consistency feature. Duplication in the view, however, exists in different formats. First, a single view element instance might include duplications inside. Second, two instances of the same view element might include duplicate sub-elements. Third, two instances of different view elements might also include duplicate sub-elements.

We use two marks, namely, the *update point type* and the *update context type* to indicate the appearance of both *construction consistency* and *duplication* in the view. Our STAR marking step encodes each node in $\mathcal{G}_V$ using a pair of labels (UPoint|UContext). This mark is then used to determine the translatability of updates specified on the nodes. In this paper, we only mark view ASG nodes according to the translatability of delete operations specified over each node. Our algorithm can also be adjusted to mark the view ASG according to the translatability of insert operations.

5.3.1 Update Context Type

The **update context type** (UContext) of a node in $\mathcal{G}_V$ determines whether view side effect might appear when deleting this node. A node is said to be *safe* if deleting its instance will not cause any view side effect. Otherwise it is said to be *unsafe*.

The following rules are used to determine the UContext of a node. Rules 1 and 2 are used to identify unsafe leaf nodes. Rule 3 is used to identify unsafe internal nodes according to unsafe leaf nodes. Rule 4 identifies the unsafe internal nodes caused by the second type of duplications. That is the duplication between two instances of the same view element. Rule 5 identifies the unsafe internal nodes caused by the third type of duplications. That is the duplication between two instances of different view elements. Finally Rule 6 marks all the remaining nodes.

Rule 1:*Consider two leaf nodes* $n_1, n_2 \in N_{\mathcal{G}_V}$ *with* $n_1 \neq n_2$. *If* $n_1^+ \equiv n_2^+$ *then* $UContext(n_1) = unsafe$ *and* $UContext(n_2) = unsafe$.

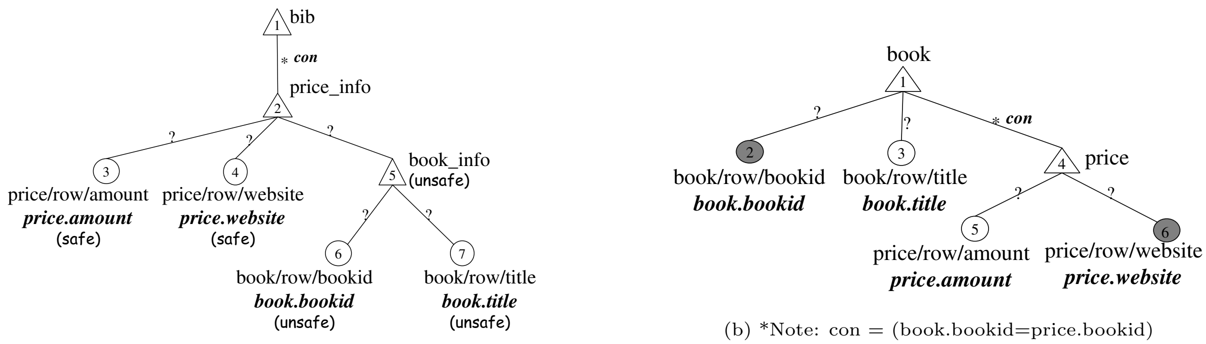
Rule 1 indicates that if two leaf nodes have exactly the same closure in $\mathcal{G}_V$, then they both are unsafe. This means the relational data is explicitly exposed more than once in the view query. For instance, in Fig. 15(e), nodes n_3, n_4, n_6, n_7 are all unsafe.

In Fig. 18(a), we can not delete a $book\_info$ element, because the same book could appear multiple times in the view. Such cases are captured by our Rule 2 and Rule 3. First, we define an operation over closure as "distribution", denoted by Ш, to distribute the cardinality mark and condition into each of the individual nodes in it. The equivalence rules used in distribution are listed in Table 2. Further we denote the occurrence of each node n in a distribution as $Ш_n$. As an example, in Fig. 15(b), $Ш(n_1^+) = Ш((n_3, n_4, (n_6, n_7))^{*con}) = n_3^{*con}, n_4^{*con}, n_6^{*con}, n_7^{*con}$.

Table 2. Distribution Rules

$C = a^{*con}$ where $a = (b, c) \Rightarrow Ш(C) = (b^{*con}, c^{*con})$
$C = a^{con}$ where $a = (b, c) \Rightarrow Ш(C) = (b^{con}, c^{con})$
$C = (n^{*con_1})^{*con_2} \Rightarrow Ш(C) = (n^{*con_1 \wedge con_2})$
$C = (n^{*con_1})^{con_2} \Rightarrow Ш(C) = (n^{*con_1 \wedge con_2})$
$C = (n^{con_1})^{*con_2} \Rightarrow Ш(C) = (n^{*con_1 \wedge con_2})$
$C = (n^{con_1})^{con_2} \Rightarrow Ш(C) = (n^{con_1 \wedge con_2})$

Rule 2: *Let C_V denote the closure of the root node in $\mathcal{G}_V$ and C_D denote its mapping closure in $\mathcal{G}_D$. For each leaf node n_l in $\mathcal{G}_V$ and its mapping node n'_l in $\mathcal{G}_D$, if $\amalg_{n_l} \not\equiv \amalg_{n'_l}$, then $UContext_{n_l} = unsafe$.*



(a) *Note: con = (book/row/bookid=price/row/bookid)

(b) *Note: con = (book.bookid=price.bookid)

Fig. 18. (a) View ASG $\mathcal{G}_V$ of V2 and (b) Base ASG $\mathcal{G}_D$ of V2

For instance, in Fig. 18(b), $C_V = r^+ = bib^+ = (price.amount, price.website, (book.bookid, book.title))^{*con}$, and $\amalg(bib^+) = (price.amount)^{*con}, price.website^{*con}, book.bookid^{*con}, book.title^{*con}$.
Let $getNodes(C_V) = \{price.amount, price.website, book.bookid, book.title\}$.
We have (C_D =$(book.bookid, book.title, (price.amount, price.website)^{*con}$).
And, $\amalg(C_D)$ =$book.bookid, book.title, (price.amount)^{*con}, (price.website)^{*con}$. Thus as shown in Fig. 18(a), we say n_6 (book.bookid) and n_7 (book.title) are both unsafe.

Rule 3: *Given an internal node $n \in N_{\mathcal{G}_V}$. Let C denote the set of its children. If $\forall n_k \in C$, n_k is unsafe, then $UContext_n = unsafe$.*

Rule 3 means that if all the children of a given internal node are unsafe, that is, are part of a duplication, then the current internal node will also be part of the duplication, thus unsafe. For example, in Fig. 18(a), after we identified that n_6 and n_7 are unsafe, using Rule 5, we can reason that n_5 is also unsafe.

Given an internal node $n \in N_{\mathcal{G}_V}$ and its parent node $p \in N_{\mathcal{G}_V}$. We define the *Current Relations* of n as *CR(n) = UCBinding(n) - UCBinding(p)*. Also we define a proper Join condition $R_i.a = R_j.b$ on an edge $e = (n_1, n_2)$ as below, that ensures no duplicates are introduced for n_2 by this Join. This Join condition is said to be proper if (i) $R_j \in CR(n_2)$ and (ii) $R_i.a$ is a unique identifier of $R_i \in CR(p)$.

Rule 4: *Let $e = (n_1, n_2)$ be an edge in $\mathcal{G}_V$ with type "*" and n_1 is not the root of $\mathcal{G}_V$. UContext of any node in the subtree rooted at n_2 is* **unsafe** *if e is not associated with a proper Join condition (as defined above).*

Rule 4 can be used to identify any missing Join condition, which causes duplications. As an example of applying this rule, assume that we ignore the WHERE clause in $Q1$. That is, the edge (n_2, n_5) in Fig. 15(a) is not annotated with any condition any more. It is easy to tell that n_5, n_6, n_7 are each unsafe since the whole price table is now nested inside of each individual book, even if unrelated.

This rule can also identify any "improper" Join conditions, which cause duplications. As an example, assume the WHERE clause of V1 in Fig. 3 is replaced by a correlated predicate "$\$book/title = \$price/website$". Then the edge (n_2, n_5) in Fig. 15(a) is annotated with a Join condition *book.title* = *price.website*. Since neither *book.title* nor *price.website* is UNIQUE, we will need to mark all the nodes in the subtree of n_5 as unsafe.

Rule 4 above identifies unsafe internal nodes, which could have duplicate instances in the view. Now assume all * edges in $\mathcal{G}_V$ are annotated with a proper Join condition (however, we still assume that they do not start from the root of $\mathcal{G}_V$). Is it still possible for a side effect to appear? The answer is yes. Rule 5 below is used to identify unsafe internal nodes, which could decide the appearance of its non-descendant nodes. As an example, again consider n_5 in 19(a). Rule 5 below will mark *price_info* node as unsafe, because it determines the appearance of its parent *book_info* node. Recall the *UCBinding* defined in Section 2.1. Given a relation R, we also define $extend(R) \subseteq rel(DEF^V)$ as a set of relations that refer to R through foreign key constraint(s).

Rule 5: *Given an internal node $n \in N_{\mathcal{G}_V}$ and its parent node p. Let CR = UCBinding(n) - UCBinding(p). If $\nexists R \in CR$ such that $\forall n' \in N_{\mathcal{G}_V}$ that is a non-descendant node of n, $extend(R) \cap UCBinding(n') = \emptyset$, then $UContext_n = unsafe$.*

The *UCBinding* difference between the node to be deleted and its parent, denoted by CR in Rule 5, indicates the minimum searching space for a *clean extended source*. Deleting from any relation of this searching space will all achieve the desired operation. However, only if all the potential *extended sources* are identified to be dangerous, a view side effect might appear.

As an example, again consider n_5 in 19(a). We have *UCBinding(n_5)* = {*book,price*} and *UCBinding(n_2)* = {*book,price*}. Thus $CR = \emptyset$. The potential searching space is empty. Deleting a price node itself will cause a view side effect. Namely, the book will also disappear or appear from the view. Thus $UContext_{n_5} = unsafe$, as marked in Fig. 19(b).

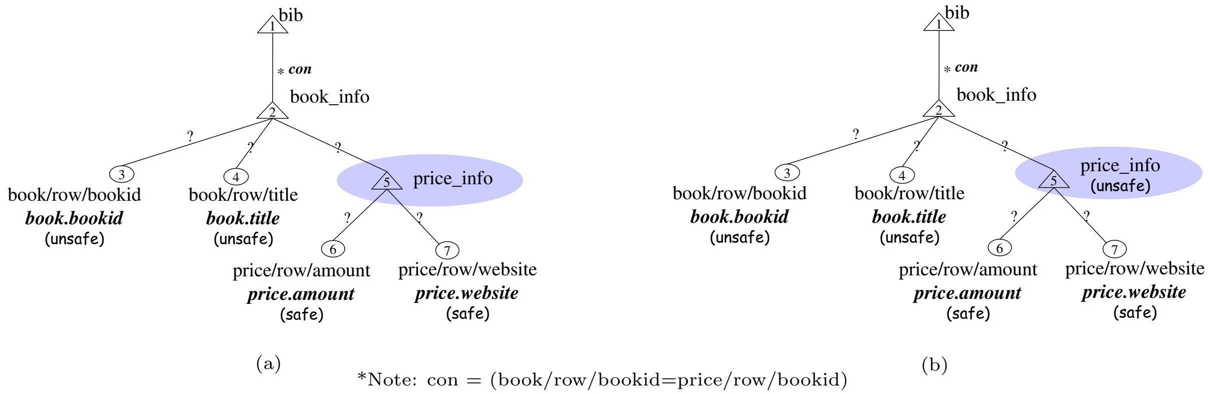


Fig. 19. (a) $\mathcal{G}_V$ of V3 after applying Rule 1 to 3 and (b) $\mathcal{G}_V$ of V3 after applying Rule 5

5.3.2 Update Point Type

As we will shown in Section 5.4, the *update context type* of a node determines whether there exists a clean extended source for a schema node. Below, we define the *update point type* (UPoint) of a node that determines whether the mapping closure is the clean extended source we are looking for.

Definition 6 *Let n be a node in $\mathcal{G}_V$, C_V is its closure in $\mathcal{G}_V$, and C_D be its mapping closure in $\mathcal{G}_D$. Let $UPoint(n)$ denote its update point type. We define $UPoint(n)$ =* **clean** *if $C_V \equiv C_D$. Otherwise, $UPoint(n)$ =* **dirty**.

For example, in Fig. 15(b), $UPoint(n_5) = dirty$. In this case, deleting a book affects the prices that reference to this book (a potential view side effect). In Fig. 15(b), we mark $UPoint(n_5) = clean$. For each price, there is no duplication among its descendants.

5.3.3 The Algorithm for STAR Marking Step

Every node in $\mathcal{G}_V$ now has been marked by a pair of labels (UPoint|UContext), as shown in Fig. 15. Algorithm 1 incorporates those label pairs into the on-the-fly update translatability checking component. This analysis is done once at compile time. Thereafter it can be reused to check the translatability of any update operation over the view. Procedure *markASG()* in Algorithm 1 is the main function for this purpose. It first computes the closure for all the nodes inside $\mathcal{G}_V$ and $\mathcal{G}_D$ using the function *computeClosure()*. Then the function *markUPoint()* is used to mark the update point type for each node in $\mathcal{G}_V$. This marking is based on Definition 6. It distinguishes a node as being dirty and clean using the closure comparison in Definition 6. The function *markUContext()* is then used to mark the node context for the nodes in $\mathcal{G}_V$. This is based on the rules defined in Section 5.3.1. It applies Rules in the order from 1 to 5. After this the remaining nodes are all marked as *safe*.

Algorithm 1 Algorithm for marking $\mathcal{G}_V$ with the (UPoint|UContext)

/*Mark (UPoint|UContext) for $\mathcal{G}_V$*/
PROCEDURE markViewASG ($\mathcal{G}_V$, $\mathcal{G}_D$)
computeClosure($\mathcal{G}_V$, $\mathcal{G}_D$)
markUPoint($\mathcal{G}_V$, $\mathcal{G}_D$)
markUContext($\mathcal{G}_V$, $\mathcal{G}_D$)

/*Compute closure of node in $\mathcal{G}_V$ and its mapping closure in $\mathcal{G}_D$*/
PROCEDURE computeClosure ($\mathcal{G}_V$, $\mathcal{G}_D$)
Initiate $C_{\mathcal{G}_V}$ and $C_{\mathcal{G}_V}$ empty
while $N_{\mathcal{G}_V}$ has more nodes **do**
  Get the next node $n \in N_{\mathcal{G}_V}$
  Initiate C_V and C_D empty
  C_V = **computeNodeClosure(n, $\mathcal{G}_V$)**
  while C_V has more nodes **do**
    Get the next node $n_i \in C_V$
    $C_D = C_D \cup$ **computeNodeClosure(n_i, $\mathcal{G}_D$)**
  end while
  Add C_V into $C_{\mathcal{G}_V}$
  Add C_D into $C_{\mathcal{G}_D}$
end while

/* Mark $UPoint_n$ as inconsistent, clear, dirty-c or dirty-s*/
PROCEDURE markUPoint($\mathcal{G}_V$, $\mathcal{G}_D$)
while $N_{\mathcal{G}_V}$ has more nodes **do**
  Get the next node $n \in N_{\mathcal{G}_V}$
  Get C_V = **getClosure**(n, $\mathcal{G}_V$)
  Get C_D = **getClosure**(C_V, $\mathcal{G}_D$)
  if $C_V \equiv C_D$ **then**
    $UPoint(n) = clean$
  else
    $UPoint(n) = dirty$
  end if
end while

/* Mark $UContext_n$ as safe or unsafe*/
PROCEDURE markUContext ($\mathcal{G}_V$, $\mathcal{G}_D$)
Initiate rules set S for update context checking
Add rules 1 to 5 into S
while S has more rules to be evaluated **do**
  Get the next rule r from S
  evaluateRule(r,$\mathcal{G}_V$, $\mathcal{G}_D$)
end while
while $N_{\mathcal{G}_V}$ has more unmarked nodes **do**
  Get the next node $n \in N_{\mathcal{G}_V}$
  $UContext(n) = safe$
end while

/*Structural Duplication Checking (Rule 1)*/
PROCEDURE evaluateRule(rule1, $\mathcal{G}_V$, $\mathcal{G}_D$)
while $N_{\mathcal{G}_V}$ has nodes left **do**
  Get the next node $n \in N_{\mathcal{G}_V}$
  if n is not marked **then**
    C_R = **getClosure**(n, $C_{\mathcal{G}_V}$)
    E = **getMappingClosureNode**(n, $C_{\mathcal{G}_V}$)
    if E is not empty **then**
      Mark all nodes $n_i \in E$ as unsafe
    end if
  end if
end while

/*Content Duplication by Correlation Predicate (Rule 2)*/
PROCEDURE evaluateRule(rule2, $\mathcal{G}_V$, $\mathcal{G}_D$)
Get the root r of $\mathcal{G}_V$
Get C_R = **getClosure**(r, $C_{\mathcal{G}_V}$)
Distribute C_R into $\amalg(C_R)$
Get C_T = **getClosure**($C_R, C_{\mathcal{G}_D}$)
Distribute C_T into $\amalg(C_T)$
N = **getNode**$(\amalg(C_R)) \cap$ **getNode**$(\amalg(C_T))$
while N has nodes left **do**
  Get the next node $n \in N$
  if n is not marked for safety **then**
    if (**getDistribution**$(n, \amalg(C_R))$ $\not\equiv$ **getDistribution**$(n, \amalg(C_T))$) **then**
      Mark n as unsafe
    end if
  end if
end while

/*Content Duplication by Internal Node Reasoning (Rule 3)*/
PROCEDURE evaluateRule(rule3, $\mathcal{G}_V$, $\mathcal{G}_D$)
while $N_{\mathcal{G}_V}$ has more internal nodes **do**
  Get the next internal node $n \in N_{\mathcal{G}_V}$
  if n is not marked **then**
    if All children of n are unsafe **then**
      $UContext(n) = unsafe$
    end if
  end if
end while

/*Missing or Improper Join Condition (Rule 4)*/
PROCEDURE evaluateRule(rule4, $\mathcal{G}_V$, $\mathcal{G}_D$)
while $E_{\mathcal{G}_V}$ has edges left **do**
  Get the next edge $e \in E_{\mathcal{G}_V}$
  Get end nodes of e into n_1,n_2
  if $(n_1 \neq root) \wedge (n_2 \neq root)$ **then**
    if (e is a * edge without condition) **then**
      Mark the subtree rooted from n_2 as unsafe
    else
      if !(IS_PROPER_JOIN(n_1)) **then**
        Mark the subtree rooted from n_2 as unsafe
      end if
    end if
  end if
end while

/*View Side Effect (Rule 5)*/
PROCEDURE evaluateRule(rule5, $\mathcal{G}_V$, $\mathcal{G}_D$)
while $N_{\mathcal{G}_V}$ has more internal nodes **do**
  Get the next internal node $n \in N_{\mathcal{G}_V}$ and its parent p
  if n is not marked **then**
    $CR = UCBinding(n) - UCBinding(p)$
    DNS = n's decendant nodes set
    CNS = ($N_{\mathcal{G}_V}$ - DNS)
    $SafeFlag = false$
    while CR has more relations **do**
      Get the next relation $R \in CR$
      $CleanFlag = true$
      while CNS has more nodes **do**
        Get the next internal node $n' \in CNS$
        if $extend(R) \cap UCBinding(n') \neq \emptyset$ **then**
          $CleanFlag = false$
          Exit while loop
        end if
      end while
      if $CleanFlag = true$ **then**
        $SafeFlag = true$
        Exit While Loop
      end if
    end while
    if $SafeFlag = false$ **then**
      Mark n as unsafe
    end if
  end if
end while

5.4 STAR Checking Step

Once the given view is analyzed and marked by pairs of its update point type and its update context labels, Observations 1, 2 can now be used to decide the update translatability as well as additional conditions required (if any).

Observation 1 *A deletion on a safe node is translatable. An deletion on an unsafe node is un-translatable.*

Observation 2 *A deletion on a (clean | safe) node is unconditionally translatable. A deletion on a (dirty | safe) node is conditionally translatable. The condition required is translation minimization. That is, in the update translation procedure, the generated SQL statements have to be minimized. This condition guarantees the translated update sequence to avoid the view side effect from content duplication.*

As shown below, we provide a concrete case study on translatability of deletions over the XML view, when the challenge factors introduced in Section 4 appear.

Example 5 u_2^V *is untranslatable since it deletes the schema node* $book\_info$ *in Fig. 15(b), which is unsafe.* u_6^V *is untranslatable since it deletes the schema node* $price\_info$ *in Fig. 15(c), which is unsafe.* u_8^V *is untranslatable since it deletes the schema node* $book\_info$ *in Fig. 15(e), which is unsafe.*

Example 6 u_1^V *is unconditionally translatable since it deletes the schema node* $book\_info$ *in Fig. 15(a), which is safe and clean.*

Example 7 u_5^V *is conditionally translatable since it deletes the schema node* $price\_info$ *in Fig. 15(b), which is safe but dirty. Translation minimization is required for this update to be translatable. Thus* $U^{R'}$ *in Fig. 13(c) is the correct translation. As another example,* u_3^V *is conditionally translatable since it deletes the schema node* $book\_info$ *in Fig. 15(c), which is safe but dirty. Again translation minimization is required for this update to be translatable. The correct translation is shown in Fig. 11(c).* u_7^V *is conditionally translatable since it deletes the schema node* $book\_info$ *in Fig. 15(d), which is safe but dirty.* u_4^V *is conditionally translatable since it deletes the schema node* $price\_info$ *in Fig. 15(d), which is safe but dirty.*

In addition, we do not emphasize the update operation for leaf nodes. Principally the leaf node is not updatable, since the view is defined over a relational database. However, we still mark the leaf node for the purpose of reasoning some internal nodes and for optimization purpose. In addition, when the view is defined over an XML document, instead of the relational database, marking leaf nodes is essential.

6 Correctness of Algorithm

To prove the correctness of our STAR algorithm, we use the *clean-extended source* theory from Section 3.

The intuition behind Rule 5 is that if an element of a safe node is updated, no instance of any other node will be affected. The correctness of this rule is proven by the following lemma. For a node $n \in \mathcal{G}_V$, we denote its set of instances in V by $I(n)$. For example, in Fig. 3(a) $I(n_2)$ consists of two *book_info* elements in view $V1$.

Lemma 4 *Given an XML view V, its view ASG $\mathcal{G}_V$. Let $n \in \mathcal{G}_V$ and $V^0 \subseteq I(n)$. If $UContext_n = safe$ then $\exists S_e$ be an extended source of V^0 such that $(S_e \cap g = \emptyset)$ holds, where $\forall n' \in \mathcal{G}_V$ and $n' \neq n$, $\forall v \in I(n')$, g is the generator of v.*

The intuition behind Rule 4 is that if an element of a safe node is updated, no any other instance of the same node will be affected. The correctness of this rule is proven by the following lemma.

Lemma 5 *Let $V, D, \mathcal{G}_V, n, V^0$ be the same as lemma 4. If $UContext_n = safe$ then $\exists S_e$ be an extended source of V^0 such that $(S_e \cap g = \emptyset)$ holds, where $\forall v \in I(n) - V^0$, g is the generator of v.*

Since we consider only relational sources, updates of individual leaf nodes is outside of our scope; we therefore consider only updates of internal nodes. In other words, we show the correctness of our algorithm from Rules 4 and 5. Note that Rule 3 does not affect the correctness of our algorithm; it makes it more efficient. Theorem below prove the connection between the safe node identified by our checking rules and the existence of a clean extended source.

Theorem 3 *Let $V, D, \mathcal{G}_V, n, V^0$ be the same as in lemma 4. There exists a clean extended source in D of V^0 if $UContext_n = safe$.*

Theorem 3 indicates that a given view element v has a clean extended source if its schema node in $\mathcal{G}_V$ is marked as safe. As indicated by Theorems 1 and 2, the existence of a clean extended source for a given XML view element implies that the update touching this element is translatable. We thus proved the correctness of our algorithm 1.

7 Discussion and Evaluation

The update translatability problem is well-known to be a difficult issue since even relational views are very often not updatable. By analyzing the properties of the XML view, we now identify what kind of XML view is handled by our schema-driven update translatability reasoning solution.

The XML views can be divided as *Tree View*, *DAG View* and *Recursive View*. The view ASG for a tree view is a tree or a forest, and a directed acyclic graph for a DAG view. The recursive view can not be represented by a view ASG.

We define an XML tree view to be a *well-formed* XML view if each correlated predicate is a key-foreign key join. Any view beyond those with key-foreign key joins involves complex duplication handling, which is a major contribution of [18]. Further, a well-formed view is *non-resized* if neither any aggregate function, such as max(), min(), count(), nor any distinct-value function is used. These operations make views non-updatable, as enunciated in [16]. The view considered in STAR is assumed to be a *non-resized well-formed* XML view.

Various internal representations are used by XML-to-SQL systems to support queries or updates through an XML view expressed over a relational database, such as SilkRoute's *view forest* [14], XPERANTO's *XQGM* [9] and *Query Trees* in [7, 8]. XPERANTO's *XQGM* can express most queries in XQuery. The view forest from SilkRoute is capable of expressing any query in the XQueryCore [24]. Query trees from [7, 8] are adapted from the view forest. The Annotated Schema Graph used in STAR has the same capabilities and limitations as the view forest, that is, it is also not capable of expressing if/then/else expressions, order and user-defined functions.

We conduct an evaluation on the expressiveness of our view ASG in order to be able to handle W3C use cases. The evaluation result is shown in Table 3. Note that W3C "SEQ" use cases focus on order queries, "STRING" use cases focus on string comparisons, "NS" use cases focus on metadata queries, "PARTS" use cases focuses on recursive queries, "STRONG" use cases includes queries that exploit strongly typed data. We thus omit evaluation of those use case groups.

The W3C's "R" user case group is the one used to access data stored in relational databases through an XML view. As we can see, 4 out of 18 queries can be represented by our view relationship graph. These 4 then in turn can be checked by our schema-reasoning solution for update translatability checking. Most queries which cannot be handled include aggregation functions, such as max(), count(), avg().

Other two test case groups "XMP" and "TREE" are defined over native XML documents and DTD, instead of over relational databases. We here assume that the inline loading strategy was used to build an underlying relational database. An extraction query is also used to extract exactly the same XML document, which is the basis of the view query within the use case groups "XMP" and "TREE". The evaluation of the expressiveness of view ASGs thus includes the combination of the extraction query and view query. As we can see, 50% of queries in "XMP" and one third of "TREE" can be expressed by ASGs, and thus handled by our update translatability checking solution. Again, most failed cases are due to aggregation or distinct functions.

Table 3. Evaluation of W3C User Case

View Query	Expressiveness	Reason
XMP-{Q1-Q3, Q5, Q7-Q9, Q11}	√	
XMP-{Q4,Q10}	×	distinct-value()
XMP-Q6	×	Aggregrate Function — Count()
XMP-Q12	×	Self-Join
TREE-Q1	√	user-defined function is not recursive
TREE-Q2	√	
TREE-{Q3,Q4,Q5,Q6}	×	Aggregation Function — Count()
R-{Q1,Q3,Q4,Q17}	√	
R-{Q2,Q5,Q6-Q15}	×	Aggregation Function — max(),avg(),count()
R-Q16	×	if-then-else
R-Q18	×	distinct-value()

8 Related Work

[1, 16, 17] study the view update translation mechanism for SPJ queries on relations that are in BCNF. These works have been further extended for object-based views in [4].

[22] presents an XQuery update grammar. It also studies the performance of executing the translated updates, assuming that the update is indeed translatable and has in fact already been translated into updates over a relational database. Our work now addresses a different aspect of the view update problem, namely, the view update translatability instead of the update translation strategy.

One of the earlier works [13] studies the view update translatability problem in the relational context. Based on the notion of a *clean source*, it presents an approach for determining the existence of update translations by performing a careful semantic analysis of the view definition. The XQuery update problem discussed in our paper is more complex than that of a pure relational view update. Not only do all the problems in the relational context still exist in XML semantics, but we also have to address the new update issues introduced by the XML hierarchical data model and the flexible update language. Work in [26] extended [13] as a *clean-extended source theory.* It serves as theoretical foundation for the schema-based XML view update translatability study. Our work in this paper provides a practical approach with the flexibility of XML views and XQuery updates being considered.

Recent works [7, 8] study the XML view update problem using a *nested relational algebra.* They assume the view is always *well-nested*, that is, joins are through keys and foreign keys, and nesting is controlled to agree with the integrity constraints and to avoid duplication. The update over such a view is thus always translatable. Our work is *orthogonal* to this work by addressing new challenges related to the decision of translation existence when no restrictions have been placed on the defined view. That is, in general, conflicts are possible and a view cannot always be guaranteed to be well-nested (as assumed in this prior work).

Commercial database systems, such as Oracle, DB2 and SQL-Server, also provide XML support. **Oracle** XML DB [3] provides SQL/XML as an extension to SQL, using functions and operators to query and access XML content as part of normal SQL operations, and also to provide methods for generating XML from the result of an SQL Select statement. The **IBM DB2** XML Extender [10] provides user-defined functions to store and retrieve XML documents in XML columns, as well as to extract XML elements or attribute values. However, neither IBM nor Oracle support update operations. [21] introduces XML view updates in **SQL-Server2000**, based on a specific *annotated schema* and update language called *updategrams*. Instead of using update statements, the user provides a before and after image of the view. The system computes the difference between the image and generates the corresponding SQL statements to reflect changes on the relational database.

9 Conclusion

In this paper, we have formalized the *XML view update problem* and identified the typical factors deciding the *update translatability*. A *theoretical foundation* for translation existence is proposed based on the *extended clean-source* concept. A schema-driven update translatability reasoning algorithm for identifying the conditions, under which an update over XML views is translatable, has been presented. Its correctness has also been proven. A concrete case study about the update translatability of XML views is provided. As proof of viability, a system framework solving the XQuery view update problem has been implemented within the XML data management system *Rainbow* [28] using the XQuery update language proposed in [22]. Several experiments have been conducted to assess various performance characteristics of our update solution [25].

Future Work. Our view update translatability checking solution is based on schema reasoning utilizing only view schema knowledge. We note that the translated updates might still conflict with the real base data. For example, even if an update inserting a book (bookid = 98002) is said to be unconditionally translatable by our schema check procedure, conflicts with the base data in Fig. 1 may still arise. Depending on the selected update translation policy, the translated update can then be either rejected or executed by replacing the existing tuple with the newly inserted tuple. This run-time update translatability issue can only be resolved at execution time by examining the actual data. Future work includes studying this run-time data-driven checking technique. Further, we need to conduct a more detailed analysis for non-delete operations as well as for updates over multiple elements.

Acknowledgment. We would like to thank Professor Susan B. Davidson from University of Pennsylvania and Vanessa P. Braganholo From UFRGS for fruitful discussion and feedback. We would also like to thank the colleagues from Database System Research Group from Worcester Polytechnic Institute.

References

[1] A. M. Keller. The Role of Semantics in Translating View Updates. *IEEE Transactions on Computers*, 19(1):63–73, 1986.

[2] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. In *ACM Transactions on Database Systems*, pages 557–575, Dec 1981.

[3] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, and R. Murthy. Oracle8i - The XML Enabled Data Management System. In *ICDE*, pages 561–568, 2000.

[4] T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating Relational Databases through Object-Based Views. In *SIGMOD*, pages 248–257, 1991.

[5] M. Benedikt, C. Y. Chan, W. Fan, and R. Rastogi. DTD-Directed Publishing with Attribute Translation Grammars. In *VLDB*, pages 838–849, 2002.

[6] P. Bohannon, P. Buneman, B. Choi, and W. Fan. Incremental Evaluation of Schema-Directed XML Publishing. In *SIGMOD*, pages 503–514, 2004.

[7] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. On the Updatability of XML Views over Relational Databases. In *WEBDB*, pages 31–36, 2003.

[8] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. From XML view updates to relational view updates: old solutions to a new problem. In *VLDB*, pages 276–287, 2004.

[9] M. J. Carey, J. Kiernan, J.Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *The VLDB Journal*, pages 646–648, 2000.

[10] J. M. Cheng and J. Xu. XML and DB2. In *ICDE*, pages 569–573, 2000.

[11] C.J.Date. *An Introduction to Database Systems*. Addison-Wesley, 1997.

[12] S. S. Cosmadakis and C. H. Papadimitriou. Updates of Relational Views. *Journal of the Association for Computing Machinery*, pages 742–760, Oct 1984.

[13] U. Dayal and P. A. Bernstein. On the Correct Translation of Update Operations on Relational Views. In *ACM Transactions on Database Systems*, volume 7(3), pages 381–416, Sept 1982.

[14] M. F. Fernandez, A. Morishima, D. Suciu, and W. C. Tan. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2):12–19, 2001.

[15] J. Shanmugasundaram et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, pages 302–314, September 1999.

[16] A. M. Keller. Algorithms for Translating View Updates to Database Updates for View Involving Selections, Projections and Joins. In *Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 154–163, 1985.

[17] A. M. Keller. Choosing a View Update Translator by Dialog at View Definition Time. In *VLDB*, pages 467–474, 1986.

[18] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. Unraveling the Duplicate-Elimination Problem in XML-to-SQL Query Translation. In *WebDB*, 2004.

[19] M. Fernandez et al. SilkRoute: A Framework for Publishing Relational Data in XML. *ACM Transactions on Database Systems*, 27(4):438–493, 2002.

[20] R. Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1997.

[21] M. Rys. Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Build Loosely-Coupled Systems. In *VLDB*, pages 465–472, 2001.

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

[23] W3C. XQuery: A Query Language for XML. http://www.w3.org/TR/xquery/, February 2001.

[24] W3C. XQuery 1.0 Formal Semantics. http://www.w3.org/TR/query-semantics/, June 2003.

[25] L. Wang, M. Mulchandani, and E. A. Rundensteiner. Updating XQuery Views Published over Relational Data: A Round-trip Case Study. In *XML Database Symposium*, pages 223–237, 2003.

[26] L. Wang and E. A. Rundensteiner. On the Updatability of XQuery Views Publised over Relational Data. In *ER*, 2004.

[27] L. Wang, E. A. Rundensteiner, and M. Mani. UFilter: A Lightweight View Update Checker. Technical Report WPI-CS-TR-TBA, Computer Science Department, WPI, 2004.

[28] X. Zhang, K. Dimitrova, L. Wang, M. EL-Sayed, B. Murphy, L. Ding, and E. A. Rundensteiner. RainbowII: Multi-XQuery Optimization Using Materialized XML Views. In *Demo Session Proceedings of SIGMOD*, page 671, 2003.

Appendix

Below we list all the proofs for lemmas and theorems used by this paper.

Proof of Lemma 1.
(a) *If.* Suppose $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) \subseteq V - V^0$ but S_e is not an extended source in D of V^0.

Let $G(V^0)$ be the set of generators of V^0. From definition 4, $\exists(t_1, ..., t_p) \in G(V^0)$ be a generator of $v \in V^0$, such that $(\forall t_i \in R_x) \Rightarrow t_i \notin S_{e_x}$. That is, $t_i \in R_x - S_{e_x}$. Thus $v \in DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n})$. But, $(t_1, ..., t_p)$ is a generator of $v \in V^0$. That is $v \notin V - V^0$. Hence, we have $v \in DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n})$ and $v \notin V - V^0$, a contradiction with the hypothesis that $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) \subseteq V - V^0$.

Only if. Suppose S_e is an extended source in D of V^0 but $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) \not\subseteq V - V^0$.

Then, $\exists v$ such that $(v \in DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n})) \wedge (v \in V^0)$. This implies that there is a generator $(t_1, ..., t_p)$ of $v \in V^0$ such that $\{t_i \mid t_i \in R_x \text{ and } R_x \in rel(DEF^V)\} \cap S_e = \emptyset$, contradicting the hypothesis that S_e is an extended source in D of V^0.

(b) *If.* Suppose $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) = V - V^0$ but S_e is not a clean extended source in D of V^0.

From (a), S_e is an extended source in D of V^0. By Definition 5, $(\exists v \in V - V^0)$ such that there is no generator $g \in \prod_{R_x \in rel(DEF^V)} (R_x - S_{e_x}$ of v, and hence $v \notin DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n})$, a contradiction.

Only if. Assume that S_e is a clean extended source in D of V^0.

By (a), $DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}) \subseteq V - V^0$. Assuming $V - V^0 \not\subseteq DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n})$, that is, $(\exists v \in V - V^0)$ such that $(v \notin DEF^V(R_1 - S_{e_1}, ..., R_n - S_{e_n}))$. Then there is no generator $g \in \prod_{R_x \in rel(DEF^V)} (R_x - S_{e_x})$ of v. Hence, by Definition 3, there is no source in $(R_1 - S_{e_1}, ..., R_n - S_{e_n})$ of $v \in V - V^0$, which contradicts the hypothesis that S_e is a clean extended source in D of V^0. □

Proof of Lemma 2.

Let $R'_x = U^R(R_x)$ be one of the updated relation $R_x \in rel(DEF^V)$. Let $T = D - U^R(D)$.

U^R deletes an extended source of $v \in V$
$\iff T$ is an extended source in D of v
$\iff DEF^V(R_1 - T_1, ..., R_n - T_n) \subseteq V - v$ (lemma 1)
$\iff v \notin DEF^V(R_1 - T_1, ..., R_n - T_n)$
$\iff v \notin DEF^V(R_1 \cap R'_1, ..., R_n \cap R'_n)$ since $R_x - T_x = R_x \cap R'_x$
$\overset{(1)}{\iff}$ There is no extended generator of v in $(R_1 \cap R'_1, ..., R_n \cap R'_n)$.

U^R does not insert an extended source-tuple of $v \in V$
$\overset{(2)}{\iff} \forall R_x \in rel(DEF^V)\ \forall t_i \in R'_x - R_x$, there is no $t_j \in R'_y - R_y$ where $R_y \in rel(DEF^V)$, $x \neq y$, such that $(t_1, ..., t_p)$ is an extended generator of v.

(1) and (2) hold iff there is no extended-generator in $U^R(D)$ of v. The proposition then follows. □

Proof of Lemma 3.

U^R inserts source-tuples of v
$\iff (\exists R_x \in rel(DEF^V), \exists t \in R'_x - R_x)(t$ is a source tuple in $U^R(D)$ of $v)$
$\overset{(1)}{\iff} (\exists g = (t_1, ..., t_p) \in \prod_{R_x \in rel(DEF^V)} R'_x)(g$ is a generator of $v)$.
$\iff v \in DEF^V(R'_1, ..., R'_n) = DEF^V(U^R(D))$.

(1) is proven as below:
If. Follow directly from Definition 3.
Only If. Assume that $g = (t_1, ..., t_p)$ is a generator of v, but $\forall R_x \in rel(DEF^V), t_i \in R_x$. Then $g \in \prod_{R_i \in rel(DEF^V)} R_i$ and so $v \in DEF^V(R_1, ..., R_n) = DEF^V(D)$, a contradiction. □

Proof of Theorem 1.

By lemma 1(b), U^R deletes a clean source of V^d
$\iff DEF^V(R_1 - T_1, ..., R_n - T_n) = V - V^d = u^V(V)$
$\iff DEF^V(U^R(D)) = u^V(V)$
$\iff DEF^V(U^R(D)) = u^V(V)$, since $R_i - T_i = R'_i$
$\iff \tau$ correctly translates u^V to U^R. □

Proof of Theorem 2.

By Lemma 3, condition (i) iff $V^u \subseteq DEF^V(U^R(D))$.
Also, since $type(u^V) = insert$ and $type(U^R) = type(u^V)$, $DEF^V(U^R(D)) \supseteq V \supseteq V^-$.
Hence, $V^u \cup V^- \subseteq DEF^V(U^R(D))$.

By Lemma 3, condition (ii) iff $(dom(V) - (V^u \cup V^-)) \cap (DEF^V(U^R(D))) = \emptyset$.
Hence, $DEF^V(U^R(D)) \subseteq V^u \cup V^-$.

Thus, condition (i) and condition (ii) iff $DEF^V(U^R(D)) = V^- \cup V^u = u^V(V)$, that is τ correctly translates u^V to U^R. □

Proof of Lemma 4.

$UContext_n = safe$
$\Longrightarrow (\exists R \in CR)(\forall n' \in \mathcal{G}_V,\ extend(R) \cap UCBinding(n') = \emptyset)$ by Rule 5
$\stackrel{(1)}{\Longrightarrow} (\exists S_e)$ such that $(\forall v \in I(n'),\ g$ be a generator of $v)\ (S_e \cap g = \emptyset)$
(1) is proven as below:
Let SE be the set of extended sources of V^0. According to the definition of $extend(R)$ in Section 5, we have $\exists S_e \in SE$ such that $rel(S_e) = extend(R)$, where $rel(S_e)$ is the set of relations that tuples in S_e belong to. Similarly, let $rel(g)$ be the set of relations that tuples in g belong to. We also have $rel(g) = UCBinding(n')$ by the definition of generator in Section 5. Then (1) holds trivially. □

Proof of Lemma 5.

The proof is based on the induction on the depth of a node n in $\mathcal{G}_V$, denoted by d.

Let p be the parent node of n. Without loss of generality, we assume that $|CR(n)| \leq 1$. That is there is at most one more relation referred for defining n than p. We denote this relation as R. Let a be the eldest ancestor of p such that $UCBinding(a) = UCBinding(p)$. Let $CR(a) = R'$.

Base Step. For $d = 1, 2$, the proposition is true.

Induction Hypothesis. Assume the proposition is true for all $d < k$.

Induction Step. We shall demonstrate the proposition is true for $d = k$.

$UContext_n = safe \Longrightarrow e = (p, n)$ is a $*$ edge with a proper Join condition as stated by Rule 4. Otherwise n would be marked as unsafe for the following reasons: (i) if e is a ? edge, n would be marked as unsafe by Rule 5; (ii) if e is a $*$ edge without a Join condition or even an "improper" Join condition, n would be marked as unsafe by Rule 4.

According to the induction hypothesis, the proposition holds for node a. $\forall v_1', v_2' \in I(a)$, consider $t_1', t_2' \in R'$ from their respective generators, we have $t_1' \neq t_2'$. According to Rule 4, the proper Join condition e will guarantee that $\forall v_1, v_2 \in I(n)$, let $t_1, t_2 \in R$ be from their respective generators, then we also have $t_1 \neq t_2$.

Consider the source S of $V^0 \subseteq I(n)$ such that $\forall t_i \in S,\ t_i \in R$. The extended source S_e corresponding to S satisfy the proposition. □

Proof of Theorem 3.

$\exists S_e$ be a clean extended source of V^0
$\Longleftrightarrow (\forall v \in V - V^0,\ g$ be a generator of $v)(S_e \cap g = \emptyset)$ by Definition 4
$\Longleftrightarrow$ (i) $(\forall v \in I(n),\ g$ be a generator of $v)(S_e \cap g = \emptyset)$
and (ii) $(\forall n' \in \mathcal{G}_V,\ n' \neq n)(\forall v \in I(n'),\ g$ be the generator of $v)\ (S_e \cap g = \emptyset)$. (1)

(1) holds if $UContext_n = safe$ by lemma 4 and 5. □