A Schema-only Approach to Validate XML Schema Mappings

(Preliminary Version)

Guillem Rull	grull@essi.upc.edu
Carles Farré	farre@essi.upc.edu
Ernest Teniente	teniente@essi.upc.edu
Toni Urpí	urpi@essi.upc.edu

March 2010

Report ESSI-TR-10-3

Departament d'Enginyeria de Serveis i Sistemes d'Informació

ABSTRACT

Since the emergence of the Web, the ability to map XML data between different data sources has become crucial. Defining a mapping is however not a fully automatic process. The designer needs to figure out whether the mapping is what was intended. Our approach to this validation consists of defining and checking certain desirable properties of mappings. We translate the XML schemas and the mapping into first-order logic formalism and apply a reasoning mechanism to check the desirable properties automatically, without assuming any particular instantiation of the schemas.

1. INTRODUCTION

Schema mappings are specifications that model a relationship between two data schemas, and are key elements in any system that requires the interaction of heterogeneous data and applications [11, 14]. The need of mappings has become more important with the emergence of the Web. The increasing number of data sources, each one independently developed and with a particular way of representing data, has made the ability to map between different data schemas, and in particular the ability to map not only relational but also XML data [17], a crucial one.

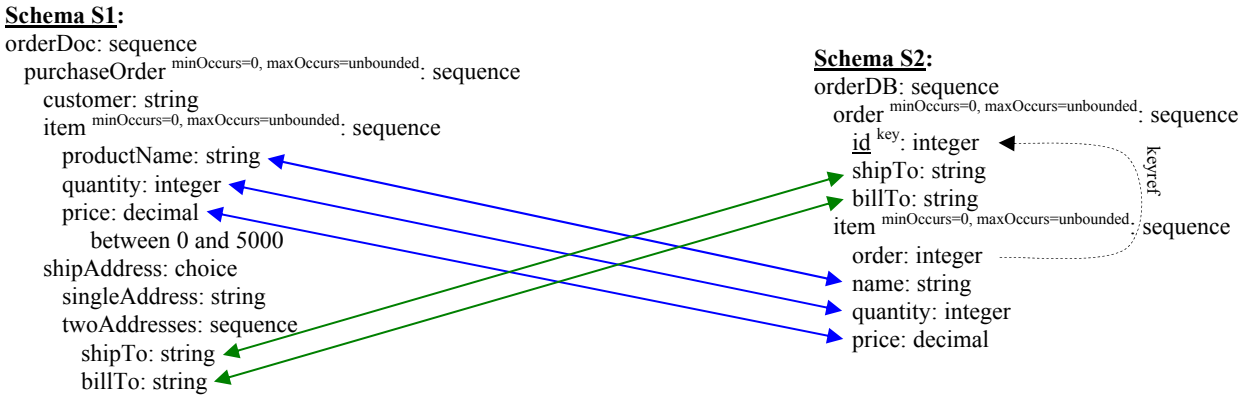
The process of defining a mapping is however not fully automatic in the general case. It requires human feedback at some point, fundamentally to resolve semantic heterogeneities. The designer thus needs to check whether the mapping he produced is in fact what was intended. That is, he must find a way to validate the mapping.

Our approach to mapping validation consists of the definition of certain desirable properties that the mapping should satisfy and providing a reasoning mechanism that allows checking them automatically. Fulfillment of these properties will provide information on whether the mapping adequately matches the intended needs and requirements. In this paper, we focus on the validation of mappings between XML schemas. Since the relational setting can be seen as a particular case of XML, our approach can also be used to validate any combination of relational and XML.

As example, consider the mapping scenario depicted in Figure 1. Schema *S1* models documents about purchase orders in which the information about items is nested inside the corresponding order. Schema *S2* has a relational structure; it represents orders and items separately, related by means of a `keyref` constraint (i.e. a foreign key). Mapping *M* is defined through a single assertion $Q^{S1} = Q^{S2}$, where Q^{S1} and Q^{S2} are queries over the schemas *S1* and *S2*. *M* maps item, shipping and billing information of all purchase orders in *S1* having two addresses with those orders in *S2* that only have items with a price greater than 5000.

This mapping may seem correct because it relates orders in *S1* with orders in *S2*. However, only those orders in *S1* that have no items can satisfy the assertion, because the price of all items in those orders is lower or equal than 5000. That is, the restriction in the range of the price in *S1* contradicts with the "where" clause of the Q^{S2} query in the mapping assertion. Our approach helps the designer discover this kind of semantic flaws. In particular, testing the strong mapping satisfiability property we propose in Section 4.1 would reveal this problem.

We consider schemas defined by means of a subset (see Section 2.1) of the XML Schema Definition language (XSD) [22]. In particular, we allow cardinality constraints, range restrictions and the use of the `<choice>` construct, which are not considered on nested relational. In this way, we extend the expressivity of previous work on XML mapping validation [1, 6, 7] that focuses on nested relational schemas and [3] that considers DTDs.



Mapping M between S1 and S2: $M = \{Q^{S1} = Q^{S2}\}$

Q^{S1}: for $po in //purchaseOrder[.//twoAddresses]
return <order>{$po//shipTo, $po//billTo, for $it in $po/item return $it}</order>

Q^{S2}: for $o in /orderDB/order
where not(/orderDB/item[./order/text() = $o/id/text() and ./price/text() <= 5000])
return <order>{ $o/shipTo, $o/billTo,
for $it in //item[./order/text() = $o/id/text()] return
<item><productName>{$it/name/text()}</productName>{$it/quantity, $it/price}</item>}
</order>

Figure 1: Example mapping scenario.

Our mappings are global-and-local-as-view (GLAV) mappings [14], where queries are expressed in a subset of the XQuery language [23]. So, negations and order comparisons are allowed in the mapping definition as shown in Figure 1. This is also a significant extension with regards to previous work which focuses only on tuple-generating dependencies [6, 7], nested mappings [1] and implications (tgds) of tree patterns [3].

In our approach, we only reason from the mapping and the mapped schemas without assuming any particular instantiation of the schemas. On the contrary, existing approaches [1, 6, 7] require users to provide a given set of schema instances to perform the validation. However, the fact that a certain property is not satisfied in a particular instance does not necessarily imply the inexistence of an instance in which the property is satisfied. Furthermore, if the property is definitely not satisfiable, the user could successively provide several instantiations without having the certainty that the property does not hold.

Our approach is based on translating the validation problem from the XML setting into first-order logic formalism. In this way, we extend our previous work on validating relational mappings [18] while taking advantage of it. First, we translate the XML schemas and the mapping into logic, which results in a logic database schema. Then, for each desirable property, the outcome of the translation is a distinguished query, defined in terms of the former logic schema. The desirable property will hold if and only if the distinguished query is satisfiable over the schema, and that can be checked by means of the *CQC method* [12]. Our XML-to-logic translation is not done from scratch since it incorporates previous proposals to translate parts of the XML schemas and the mapping.

The intuition behind our approach is that the database schema that results from the proposed translation is equivalent to the original XML mapping scenario, that is, there is a consistent database instance for each consistent instantiation of the mapping scenario and vice versa. And, similarly, the distinguished query is equivalent to the definition of the property being tested. Then, if the query is satisfiable over the logic schema, that means there is an instantiation of the mapping scenario that exemplifies the satisfaction of the tested property. If there is no such example, then the property does not hold and the query cannot be satisfied.

2. PRELIMINARIES

2.1 XML Schemas and Mappings

We consider XML schemas defined by means of a subset of the XML Schema Definition language (XSD) [22]. Basically, these schemas consist in a root element definition followed by a collection of type definitions. Complex types can be defined either as a `<sequence>` of element definitions, or as a `<choice>` among element definitions. Elements can also be defined as of simple type. We consider the integrity constraints `key` and `keyref`, and the possibility to restrict the range of a simple type for a certain element definition. What we do not consider is the order of the elements inside a type definition.

In order to avoid the verbose XML representation, we use a more compact notation. We represent an *XML schema* as a tuple (r, T, Σ), where r is an element definition (the root), T is a set of type definitions, and Σ is a set of integrity constraints. An *element definition* is also a tuple (*name*, *type_name*, *min_occurs*, *max_occurs*), where the components are: the element's name, the element's type, and the cardinality constraints. A *type definition* is in the form of (*type_name*, `sequence/choice`, $\{e_1, ..., e_n\}$), where the second component indicates whether the type is defined as a sequence of element definitions $e_1, ..., e_n$ ($n \geq 1$), or as a choice among element definitions $e_1, e_2, ... e_n$ ($n \geq 2$). The root's definition must be like r = (*root_name*, *root_type*, 0, 1), where *root_type* must be defined in T. We assume minOccurs = 0 for the root in order to allow the empty instance to be a valid instance of the schema.

Regarding integrity constraints, the XSD's `key` and `keyref` constraints are represented as (`key`, *name*, *selector*, ($field_1$, ..., $field_n$)) and (`keyref`, *name*, *selector*, ($field_1$, ..., $field_n$), *referenced_key*), where *selector* and $field_i$ are XPath expressions. The *selector* indicates the element that we want to identify, and $field_1$, ..., $field_n$ the elements that form the key. `keyref` is just a referential constraint to an existing key.

The XSD language allows deriving new simple types by restricting existing simple types. In particular, the range of valid values of a simple type can be restricted by means of the facets: minInclusive, minExclusive, maxInclusive and maxExclusive. We model this feature as an integrity constraint: (`restriction`, *selector*, *valid_range*), where *selector* is an XPath expression that indicates the element of basic type we are going to restrict its range, and *valid_range* is the new range of values.

As an example, the XML schema *S1* in Figure 1 would be represented as follows:

$S1 = (r_1, T_1, \Sigma_1)$, where r_1 = (orderDoc, orderDocType, 0, 1),

T_1 = {(orderDocType, `sequence`, {(purchaseOrder, purchaseOrderType, 0, `unbounded`)}),
(purchaseOrderType, `sequence`, {(costumer, `string`, 1, 1), (item, itemType, 0, `unbounded`), (shipAddress, shipAddressType, 1, 1)}),
(itemType, `sequence`, {(productName, `string`, 1, 1), (quantity, `integer`, 1, 1), (price, `decimal`, 1, 1)}),
(shipAddressType, `choice`, {(singleAddress, `string`, 1, 1), (twoAddresses, twoAddressesType, 1, 1)}),
(twoAddressesType, `sequence`, {(shipTo, `string`, 1, 1), (billTo, `string`, 1, 1)})}

Σ_1 = {(`restriction`, //price, ./text() >= 0 and ./text() <= 5000)}.

We consider an *XML schema mapping* to be defined as a set of assertions $M = \{m_1, ..., m_k\}$ that specify a relationship between two XML schemas. Each assertion m_i is of the form $Q^{S1}{}_i \; op_i \; Q^{S2}{}_i$, where $Q^{S1}{}_i$ and $Q^{S2}{}_i$ are queries expressed in a subset of the XQuery language [23], *S1* and *S2* are the two mapped schemas, and op_i is $\sqsubseteq$ or $=$.

We say that two instances of the XML schemas being mapped are *consistent with the mapping* if all the mapping assertions are evaluated true. A mapping assertion $Q^{S1} \sqsubseteq (=) Q^{S2}$ is true if the answer to Q^{S1} is contained in (equivalent to) the answer to Q^{S2} when the queries are evaluated over the pair of mapped schema instances.

We assume that the two queries of a certain mapping assertion return a result of the same type, that is, the answers to these queries are XML documents with the same structure and element names. Therefore, and for the sake of simplicity, we skip the concrete element names in the return clause and just indicate the structure. The syntax is the following:

MappingAssertion ::= $Query_1$ ($\sqsubseteq$ | =) $Query_2$
Query ::= **for** (*Var* **in** *Path*)+ (**where** *Cond*)? **return** '**[**' *Result*+ '**]**'
Result ::= $Path_1$/**text()** | *Query* | *Const* | '**[**' *Result*+ '**]**'
Path ::= (*Var* | '**.**')? ((/ | //) $ElementName_2$ ('**[**' *Cond* '**]**')?)+
Cond ::= ($Path_1$/**text()** | $Const_1$) (**eq** | **ne** | **lt** | **le** | **gt** | **ge**)
($Path_2$/**text()** | $Const_2$) |
Path | $Cond_1$ **and** $Cond_2$ | $Cond_1$ **or** $Cond_2$ |
not *Cond* | '**(**' *Cond* '**)**'

where *Var* denotes a variable, *Const* denotes a constant, *ElementName* must match one of the elements defined in the corresponding schema, and "/text()" must be applied to a path that returns one simple-type node.

We consider containment and equivalence of nested structures under set semantics [10, 15]. The *answer to a query* will be thus a set of records $\{[R_{1,1}, ..., R_{1,m}], ..., [R_{n,1}, ..., R_{n,m}]\}$, where each $R_{i,j}$ is either a simple type value, a record, or a set of records.

The *containment* of two nested structures R_1, R_2 of the same type T, i.e. $R_1 \sqsubseteq R_2$, can be defined by induction on T as follows [15]:
(1) If T is a simple type, $R_1 \sqsubseteq R_2$ iff $R_1 = R_2$.
(2) If T is a record type, $R_1 = [R_{1,1}, ..., R_{1,n}] \sqsubseteq R_2 = [R_{2,1}, ..., R_{2,n}]$ iff $R_{1,1} \sqsubseteq R_{2,1} \wedge ... \wedge R_{1,n} \sqsubseteq R_{2,n}$.
(3) If T is a set type, $R_1 = \{R_{1,1}, ..., R_{1,n}\} \sqsubseteq R_2 = \{R_{2,1}, ..., R_{2,n}\}$ iff $\forall i\, \exists j\, R_{1,i} \sqsubseteq R_{2,j}$.

Equivalence can be defined similarly:
(1) If T is a simple type, $R_1 = R_2$.
(2) If T is a record type, $[R_{1,1}, ..., R_{1,n}] = [R_{2,1}, ..., R_{2,n}]$ iff $R_{1,1} = R_{2,1} \wedge ... \wedge R_{1,n} = R_{2,n}$.
(3) If T is a set type, $\{R_{1,1}, ..., R_{1,n}\} = \{R_{2,1}, ..., R_{2,n}\}$ iff $\forall i\, \exists j\, R_{1,i} = R_{2,j} \wedge \forall j\, \exists i\, R_{2,j} = R_{1,i}$.

2.2 Logic Database Schemas and the CQC Method

A *logic database schema* is a tuple (*DR*, *IC*), where *DR* is a finite set of deductive rules, and *IC* is a finite set of constraints. A *deductive rule* has the form:

$$p(\bar{X}) \leftarrow r_1(\bar{X}_1) \wedge \ldots \wedge r_n(\bar{X}_n) \wedge \neg r_{n+1}(\bar{Y}_1) \wedge \ldots \wedge \neg r_m(\bar{Y}_s) \wedge C_1 \wedge \ldots \wedge C_t,$$

and a *constraint* takes one of the following two forms:

$$r_1(\bar{X}_1) \wedge \ldots \wedge r_n(\bar{X}_n) \rightarrow C_1 \vee \ldots \vee C_t,$$
$$r_1(\bar{X}_1) \wedge \ldots \wedge r_n(\bar{X}_n) \wedge C_1 \wedge \ldots \wedge C_t \rightarrow \exists \bar{V}_1\, r_{n+1}(\bar{Z}_1) \vee \ldots \vee \exists \bar{V}_s\, r_m(\bar{Z}_s).$$

Symbols p, r_i are *predicates*. Tuples $\bar{X}$, $\bar{X}_i$, $\bar{Y}_i$ contain *terms*, which are either variables or constants. Each $\bar{V}_i$ is a tuple of distinct variables. Each C_i is a *built-in literal* in the form of $t_1\ \theta\ t_2$, where t_1 and t_2 are terms and the operator θ is $<$, $\leq$, $>$, $\geq$, $=$ or $\neq$. Atom $p(\bar{X})$ is the *head* of the rule, and $r_i(\bar{X}_i)$, $\neg r_i(\bar{Y}_i)$ are positive and negative *ordinary literals* (those that are not built-in). Every rule and constraint must be *safe* [21], that is, every variable that occurs in $\bar{X}$, $\bar{Y}_i$, C_i must also appear in some $\bar{X}_j$. Each variable in $\bar{Z}_i$ is either from $\bar{V}_i$ or from some $\bar{X}_j$. Variables in $\bar{X}_i$ are assumed to be universally quantified over the whole formula. Predicates that appear in the head of a rule are *derived predicates* (views or queries). The rest are *base predicates* (tables). They correspond to tables. A database *violates* a constraint $L_1 \wedge ... \wedge L_n \rightarrow L_{n+1} \vee ... \vee L_m$ if the implication $(L_1 \wedge ... \wedge L_n \rightarrow L_{n+1} \vee ... \vee L_m)\sigma$ is false for some ground substitution σ.

Given a query Q defined by means of a set of deductive rules, and a logic database schema S, we can use the *CQC method* [12] to check whether Q is satisfiable in S, that is, whether there exists an instance of S in which Q returns a non-empty answer.

The *CQC method* is a constructive method. It tries to build a database instance that does not violate any constraint in the schema, and that serves as example of the given query Q being satisfiable. To instantiate the tuples in this instance, the method uses a set of *Variable Instantiation Patterns* (VIPs) according to the syntactic properties of the schema and the query. If the method ends and no instance has been built, that means query Q is not satisfiable.

3. TRANSLATING XML SCHEMAS AND MAPPINGS INTO LOGIC

To validate XML schema mappings, we translate the problem from the initial XML setting into a first-order logic formalism. The main goal of this section is to define such a translation for both the original XML schemas and the mapping.

3.1 Translating the Nested Structure of the Mapped Schemas

Each element definition e = (*name*, *type*, ...) in a schema S is translated into a base predicate along the lines defined in [24]. If e is a root element, it is translated into the predicate *name*(*id*). Otherwise, the predicate will be either *name*(*id*, *parentId*) if *type* is complex, or *name*(*id*, *parentId*, *value*) if it is simple. The terms of the predicates represent: the id of the XML node, the id of the parent node, and the simple type value, when required. For instance, the translation until the second level of nesting of *S1* is:

orderDoc(id), purchaseOrder(id, parentId), customer(id, parentId, value), item(id, parentId) and shipAddress(id, parentId).

Moreover, to ensure the original semantics of element definitions at the logical level, we have to define a set of integrity constraints that make explicit some of the XML assumptions and structure.

First of all, we must guarantee that there cannot be two different instances of an element definition with the same *id*. For example, we need the constraint:

purchaseOrder(id, pid1) $\wedge$ purchaseOrder(id, pid2) $\rightarrow$ pid1 = pid2,

to make term *id* unique in the context of *purchaseOrder*.

We also need additional constraints to make explicit the parent-child relationship between element definitions. Consider two element definitions e_1, e_2 and a type t_1, such that $e_1 = (n_1, t_1, min_1, max_1)$, $t_1 = (tn, sc, E)$, $e_2 = (n_2, t_2, min_2, max_2)$, and $e_2 \in E$. Then, we define a referential constraint from the *parentId* term of predicate n_2 to the *id* term of predicate n_1. For example:

customer(id, pid, val) $\rightarrow \exists$id2 purchaseOrder(pid, id2).

Ensuring enforcement of the maxOccurs and minOccurs facets of each element definition is also achieved by additional constraints. Consider again the two generic element definitions e_1 and e_2. In order to enforce the maxOccurs facet of e_2, and if $max_2 \neq$ `unbounded`, we must define the following constraint:

$n_1(id_{n1}, ...) \wedge n_2(id_{n2,1}, id_{n1}, ...) \wedge ... \wedge n_2(id_{n2,max2+1}, id_{n1}, ...) \rightarrow id_{n2,1} = id_{n2,2} \vee ... \vee id_{n2,1} = id_{n2,max2+1} \vee ... \vee id_{n2,max2} = id_{n2,max2+1}$.

The enforcement of minOccurs facet of e_2 depends however on whether type t_1 is defined as a sequence or a choice. If sc = `sequence` and $min_2 > 0$, we must define the constraint:

$n_1(id, ...) \rightarrow min^{seq}{}_{n2}(id)$, where

$min^{seq}{}_{n2}(pid) \leftarrow n_2(id_1, pid, ...) \wedge ... \wedge n_2(id_{min2}, pid, ...) \wedge id_1 \neq id_2 \wedge ... \wedge id_1 \neq id_{min2} \wedge ... \wedge id_{min2-1} \neq id_{min2}$.

Instead, if sc = `choice`, $min_2 > 1$ and all element definitions in E have minOccurs > 0, the following constraint is needed:

$n_1(id_{n1}, ..) \wedge n_2(id_{n2,1}, id_{n1}, ...) \rightarrow min^{choice}{}_{n2}(id_{n1}, id_{n2,1})$, where

$min^{choice}{}_{n2}(pid, id_1) \leftarrow n_2(id_2, pid, ...) \wedge ... \wedge n_2(id_{min2}, pid, ...) \wedge id_1 \neq id_2 \wedge ... \wedge id_1 \neq id_{min2} \wedge ... \wedge id_{min2-1} \neq id_{min2}$.

For example, the implicit facet maxOccurs=1 of the *singleAddress* element definition in schema *S1* would be translated as follows:

shipAddress(id, pid) $\wedge$ singleAddress(id1, id, val1) $\wedge$ singleAddress(id2, id, val2) $\rightarrow$ id1 = id2.

Additionally, to make explicit the semantics of the `<choice>` construct, we must guarantee that one and only one element definition is chosen. Consider an element definition e = (n, t, *min*, *max*), where $t_1 = (tn, sc, E)$, sc = `choice` and $E = \{e_1 = (n_1, t_1, min_1, max_1), ..., e_k = (n_k, t_k, min_k, max_k)\}$. In order to state that at least one element definition from E must be chosen, and if all element definitions in E have minOccurs > 0, we need the constraint:

$$n(id, ...) \rightarrow \exists id_1\, n_1(id_1, id\ ...) \vee ... \vee \exists id_k\, n_k(id_k, id...).$$

In order to state that no more than one element definition from E can be chosen, the following constraints are required:

$name_1(id_1, pid_1, ...) \wedge name_2(id_2, pid_2, ...) \rightarrow pid_1 \neq pid_2$,
..., $name_1(id_1, pid_1, ...) \wedge name_k(id_k, pid_k, ...) \rightarrow pid_1 \neq pid_k$,
..., $name_{k-1}(id_{k-1}, pid_{k-1}, ...) \wedge name_k(id_k, pid_k, ...) \rightarrow pid_{k-1} \neq pid_k$.

For example, the *shipAddress* choice in *S1* would be translated:

shipAddress(id, pid) $\rightarrow \exists$(id1, value) singleAddress(id1, id, value)$\vee \exists$(id2) twoAddresses(id2, id),

singleAddress(id1, pid1, val) $\wedge$ twoAddresses(id2, pid2) $\rightarrow$ pid1 $\neq$ pid2.

Finally, we also have to make explicit that there must be only one instance of each root element. We do that by means of two constraints. In our example:

orderDoc(id1) $\wedge$ orderDoc(id2) $\rightarrow$ id1 = id2,
orderDB(id1) $\wedge$ orderDB(id2) $\rightarrow$ id1 = id2.

It is worth mentioning that since a mapping scenario involves two schemas, we assume, for the sake of clarity, that all element definitions have a different name. If this is not the case, we need to rename, without loss of generality, the colliding elements before applying our approach. For instance, in our example, both mapped schemas have an element named price. Those elements could easily be renamed as $price_{S1}$ and $price_{S2}$, respectively.

3.2 Translating the Integrity Constraints

A schema S may contain `key` and `keyref` integrity constraints as well as range restrictions on simple type elements. Each constraint in S is defined by means of a certain XPath expression that specifies the elements to constrain. Since we assume all elements in S have unique names, we can remove all "//" axis (i.e. all "descendant" axis) appearing in the XPath expression. Then, all resulting XPath expressions will have the form: $/name_1[cond_1]/name_2[cond_2]/ \ldots /name_n[cond_n]$, where $n \geq 1$, and $[cond_i]$ is a condition that may or may not appear in the original expression.

We translate each XPath expression *path* into a derived predicate along the lines suggested in [9]. The main difference is that we allow conditions with negations and order comparisons, which are not handled in [9]. The translated path of *path*, denoted by T-path(*path*, *id*), is defined by means of the predicate $P_{path}(id_n)$ according to the equivalence: T-path(*path*, *id*) = $P_{path}(id_n)$. Now, $P_{path}(id_n)$ is the derived predicate we obtain as a result of our translation, and it is defined by the following rule:

$$P_{/name1[cond1]/name2[cond2]/.../namen[condn]}(id_n) \leftarrow name_1(id_1) \wedge \text{T-cond}(cond_1, id_1) \wedge name_2(id_2, id_1) \wedge \text{T-cond}(cond_2, id_2) \wedge \ldots \wedge name_n(id_n, id_{n\text{-}1}, \ldots) \wedge \text{T-cond}(cond_n, id_n).$$

If the path ends with "/text()", the literal about $name_n$ should be $name_n(id_n, id_{n\text{-}1}, value)$, and the argument in the head of the rule should be *value* instead of id_n. In the formula, T-cond stands for the translation of a condition in *path*. It is defined according to the following rules:

(1) T-cond($cond_1$ **and** $cond_2$, *pid*) = T-cond($cond_1$, *pid*) $\wedge$ T-cond($cond_2$, *pid*).

(2) T-cond($cond_1$ **or** $cond_2$, *pid*) = $aux_{cond1\ or\ cond2}$(*pid*), where
$aux_{cond1\ or\ cond2}$(*pid*) $\leftarrow$ T-cond($cond_1$, *pid*),
$aux_{cond1\ or\ cond2}$(*pid*) $\leftarrow$ T-cond($cond_2$, *pid*).

(3) T-cond(**not** *cond*, *pid*) = $\neg aux_{cond}$(*pid*), where aux_{cond}(*pid*) $\leftarrow$T-cond(*cond*, *pid*).

(4) T-cond($path_1$/text() ***op*** $path_2$/text(), *pid*) = T-relpath($path_1$/ text(), *pid*, $value_1$) $\wedge$T-relpath($path_2$/text(), *pid*, $value_2$) $\wedge$ $value_1$ ***op*** $value_2$, where $value_1$ and $value_2$ are the simple-type results of the relative path expressions.

(5) T-cond(*path*, *pid*) = T-relpath(*path*, *pid*, *res*).

The relative paths that may appear in the conditions have the following translation:

(1) T-relpath($./name_1[cond_1]/ \ldots /name_n[cond_n]$, *pid*, id_n) = $name_1(id_1, pid) \wedge \text{T-cond}(cond_1, id_1) \wedge \ldots \wedge name_n(id_n, id_{n\text{-}1}) \wedge \text{T-cond}(cond_n, id_n)$.

(2) T-relpath($./name_1[cond_1]/ \ldots /name_n[cond_n]$/text(), *pid*, *value*) = $name_1(id_1, pid) \wedge \text{T-cond}(cond_1, id_1) \wedge \ldots \wedge name_n(id_n, id_{n\text{-}1}, value) \wedge \text{T-cond}(cond_n, id_n)$.

As an example, the path expression:

/orderDoc/purchaseOrder[not(./item[./price/text()< 1000])]/customer

would be translated as:

$P_{/orderDoc/purchaseOrder[not(./item[./price/text()<1000])]/customer}(id) \leftarrow orderDoc(id1) \wedge purchaseOrder(id2, id1) \wedge \neg aux_{./item[./price/text() < 1000]}(id2) \wedge costumer(id, id2)$
$aux_{./item[./price/text() < 1000]}(id2) \leftarrow item(id3, id2) \wedge price(id4, id3, val) \wedge val < 1000.$

Once the XPath expressions have been translated, the integrity constraints are translated in logic as follows:

- A key constraint (`key`, *name*, *selector*, ($field_1$, ..., $field_n$)) is translated into:

  T-path(*selector*, id_1) $\wedge$ T-path(*selector*, id_2) $\wedge$ T-relpath($field_1$, id_1, val_1) $\wedge$ T-relpath($field_1$, id_2, val_1) $\wedge \ldots \wedge$ T-relpath($field_n$, id_1, val_n) $\wedge$ T-relpath($field_n$, id_2, val_n) $\rightarrow$ $id_1 = id_2$.

- Let (`keyref`, *name*, *selector*, ($field_1$, ..., $field_n$), *key_name*) be a keyref constraint, and let (`key`, *key_name*, *ref_selec*, (ref_field_1, ..., ref_field_n)) be the referenced key. The keyref constrain is translated as follows:

  T-path(*selector*, *id*) $\wedge$ T-relpath($field_1$, *id*, val_1) $\wedge \ldots \wedge$ T-relpath($field_n$, *id*, val_n) $\rightarrow$ $aux_{name}(val_1, \ldots, val_n)$,
  $aux_{name}(val_1, \ldots, val_n) \leftarrow$ T-path(*ref_selec*, *id*) $\wedge$ T-relpath(ref_field_1, *id*, val_1) $\wedge \ldots \wedge$ T-relpath(ref_field_n, *id*, val_n).

- Finally, a range restriction like (`restriction`, /orderDoc/ purchaseOrder/item/ price, ./text() >= 0 and ./text() <= 5000) is translated into:

  T-path(/orderDoc/purchaseOrder/item/price, value) $\rightarrow$ value ≥ 0
  T-path(/orderDoc/purchaseOrder/item/price, value) $\rightarrow$ value ≤ 5000

3.3 Translating the Mapping Assertions

A mapping assertion consists in two nested XML queries related by means of an = or $\sqsubseteq$ operator. We will translate each of these queries as a collection of "flat" queries. Following the lines of [15], there will be one flat query for each nested block. For example, let us consider the query Q^{S1} in Figure 1. It has two "for ... return ..." blocks. We translate the outermost block as follows:

$Q^{S1}{}_0(po, st, bt) \leftarrow$ T-path(//purchaseOrder[.//twoAddresses], po) $\wedge$ T-relpath(.//shipTo/text(), po, st) $\wedge$ T-relpath(.//billTo/text(), po, bt)

Each flat query will have one term in its head for each variable in the "for" clause, plus one term for each simple type expression in the "return" clause of the block. If the return clause contains record type expressions, those must be flattened before the translation. For example, let V be the following query:

V: for \$po in //purchaseOrder[.//twoAddresses]
return [\$po/customer/text(),
[\$po//shipTo/text(), \$po//billTo/text(), for... return...]].

Its outermost block would be translated as follows:

V_0(po, c, st, bt) ← T-path(//purchaseOrder[.//twoAddresses], po) ∧
T-relpath(./customer/text(), po, c) ∧
T-relpath(.//shipTo/text(), po, st) ∧
T-relpath(.//billTo/ text(), po, bt).

The translation of an inner block requires taking into account its inherited variables, e.g. *\$po* in Q^{S1}. We use access patterns [8] to deal with this kind of variables. In particular, we consider derived predicates with "input-only" terms. We denote these predicates $Q<t_1, ..., t_n>(t_{n+1}, ..., t_m)$, where $t_1, ..., t_n$ are the input-only terms, and $t_{n+1}, ..., t_m$ are the usual "input-output" terms. This way, the inner block of Q^{S1} is translated:

$Q^{S1}{}_1$<po>(it, pn, q, p) ← T-relpath(./item, po, it) ∧ T-relpath(./productName, it, pn) ∧ T-relpath(./quantity, it, q) ∧ T-relpath(./price, it, p).

The input-only variables that appear in the head of a deductive rule are not required to appear in a positive ordinary literal in the rule's body. However, the input-only variables that appear in a body's literal and do not appear in the head are still required to appear in some positive ordinary literal in the same body. The last condition allows the *CQC method* to deal with these predicates as if they were normal derived predicates.

The translation of the "where" clause of a query block is very similar to the translation of a path condition. The only difference is that a path condition involves a single variable that denotes the node which the condition is applied to, while a where clause potentially involves all the variables in the "for" clause (plus the variables inherited from its ancestor blocks). For example, consider the outermost block of the query Q^{S2} in Figure 1. It would be translated as follows:

$Q^{S2}{}_0$(o, st, bt) ← T-path(/orderDB/order, o) ∧ ¬aux_{where}<o>() ∧ T-relpath(.//shipTo/text(), po, st) ∧ T-relpath(.//billTo/text(), po, bt)
aux_{where}<o>() ← T-path(/orderDB/item, it) ∧ T-relpath(./order/text(), it, id1) ∧ T-relpath(./id/text(), o, id2) ∧id1 = id2 ∧
T-relpath(./price/text(), it, pr) ∧ pr <= 5000

To translate a mapping assertion $Q_1 \sqsubseteq (=) Q_2$ we have to express the containment (equivalence) definition from Section 2.1 in our logic formalism. We will rely on the flat queries that result from the translation of Q_1 and Q_2.

Let Q_A, Q_B be two generic (sub-)queries with the same return type:

Q_A: for $\$v_1$ in $path_1$, ..., $\$v_{na}$ in $path_{na}$ where *cond*
return [A_1, ..., A_m, B_1, ..., B_k]

Q_B: for $\$v_1$' in $path_1$',...,$\$v_{nb}$' in $path_{nb}$' where *cond*'
return [A_1',..., A_m', B_1',..., B_k'],

where A_i, A_i' are simple-type expressions, and B_i, B_i' are sub-queries. Let us assume the outermost block of Q_A is translated into predicate $Q_{A0}<x_1, ..., x_{ka}>(v_1, ..., v_{na}, r_1, ..., r_m)$, where $x_1, ..., x_{ka}$ denote the variables inherited from the ancestor query blocks, $v_1, ..., v_n$ denote the variables in the "for" clause, and $r_1, ..., r_m$ denote the simple-type values returned by the block. Similarly, let us also assume the outermost block of Q_B is translated into $Q_{B0}<x_1', ..., x_{kb}'>(v_1', ..., v_{nb}', r_1', ..., r_m')$.

The translation of $Q_A \sqsubseteq Q_B$ into first-order logic is:

T-containment(Q_A, Q_B, $\{i_1,...,i_h\}$) =
$\forall(v_1,...,v_{na}, r_1,...,r_m)\ (Q_{A0}<x_1,...,x_{ka}>(v_1,...,v_{na}, r_1,...,r_m) \rightarrow \exists(v_1',...,v_{nb}')\ (Q_{B0}<x_1',...,x_{kb}'>(v_1',...,v_{nb}', r_1,...,r_m)$
$\wedge$ T-containment(B_1, B_1', $\{i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m, v_1',...,v_{nb}'\}$)
$\wedge$ ... $\wedge$ T-containment(B_k, B_k', $\{i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m, v_1',...,v_{nb}'\}$))),
where $\{i_1,...,i_h\}$ is the union of the inherited variables of Q_A and Q_B, and $\{x_1,...,x_{ka}\} \cup \{x_1',...,x_{kb}'\} \subseteq \{i_1,...,i_h\}$.

The above expression, however, does not fit the syntactic requirements of the class of logic database schemas the *CQC method* works on (see Section 2.2). To address that, the first thing we need to do is get rid of the universal quantifiers. To do so, we perform a double negation on T-containment, and move one of the negations inwards:

$\neg\neg$T-containment(Q_A, Q_B, $\{i_1,...,i_h\}$) =
$\neg\exists(v_1,...,v_{na}, r_1,...,r_m)\ (Q_{A0}<x_1,...,x_{ka}>(v_1,...,v_{na}, r_1,...,r_m) \wedge \neg\exists(v_1',...,v_{nb}')\ (Q_{B0}<x_1',...,x_{kb}'>(v_1',...,v_{nb}', r_1,...,r_m)$
$\wedge$ $\neg\neg$T-containment(B_1, B_1', $\{i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m, v_1',...,v_{nb}'\}$)
$\wedge$ ... $\wedge$ $\neg\neg$T-containment(B_k, B_k', $\{i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m, v_1',...,v_{nb}'\}$))).

Now, we fold each existentially quantified (sub-)expression and get the following constraint:

$$\text{T-noncontainment}(Q_A, Q_B, \varnothing) \rightarrow \bot$$

where $\bot$ denotes any contradiction, e.g. 1 = 0, and

$$\text{T-noncontainment}(Q_A, Q_B, \{i_1,...,i_h\}) = Q_A\text{-}not\text{-}contained\text{-}in\text{-}Q_B\langle i_1,...,i_h\rangle()$$

where Q_A-*not-contained-in-*Q_B is a derived predicate defined by the following deductive rules:

$$Q_A\text{-}not\text{-}contained\text{-}in\text{-}Q_B\langle i_1,...,i_h\rangle() \leftarrow Q_{A0}\langle x_1,...,x_{ka}\rangle(v_1,...,v_{na}, r_1,...,r_m) \wedge \neg aux^{Q_A \not\subseteq Q_B}\langle i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m\rangle()$$

$$aux^{Q_A \not\subseteq Q_B}\langle i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m\rangle() \leftarrow Q_{B0}\langle x_1',...,x_{kb}'\rangle(v_1',...,v_{nb}', r_1,...,r_m)$$
$$\wedge \neg\text{T-noncontainment}(B_1, B_1', \{i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m, v_1',...,v_{nb}'\})$$
$$\wedge ... \wedge \neg\text{T-noncontainment}(B_k, B_k', \{i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m, v_1',...,v_{nb}'\}).$$

As an example, consider the mapping assertion $Q^{S1} \sqsubseteq Q^{S2}$, where Q^{S1}, Q^{S2} are the queries in Figure 1. It would be translated into the constraint Q^{S1}-not-contained-in-Q^{S2}<>() $\rightarrow \bot$, where

Q^{S1}-not-contained-in-Q^{S2}<>() $\leftarrow$ $Q^{S1}{}_0$(po, st, bt) $\wedge \neg$ aux$^{QS1 \not\subseteq QS2}$<po, st, bt>(),

aux$^{QS1 \not\subseteq QS2}$<po, st, bt>() $\leftarrow$ $Q^{S2}{}_0$(o, st, bt) $\wedge \neg Q^{S1}{}_1$-not-contained-in-$Q^{S2}{}_1$<po, st, bt, o>(),

$Q^{S1}{}_1$-not-contained-in-$Q^{S2}{}_1$<po, st, bt, o>() $\leftarrow$ $Q^{S1}{}_1$<po>(it, pn, q, p) $\wedge \neg$ aux$^{QS11 \not\subseteq QS21}$<po, st, bt, o, it, pn, q, p>(),

aux$^{QS11 \not\subseteq QS21}$<po, st, bt, o, it, pn, q, p>() $\leftarrow$ $Q^{S2}{}_1$<o>(it', pn, q, p).

Similarly, the translation of an equivalence assertion $Q_1 = Q_2$ results in two constraints:

$$\text{T-nonequivalence}(Q_1, Q_2, \varnothing) \rightarrow \bot$$

$$\text{T-nonequivalence}(Q_2, Q_1, \varnothing) \rightarrow \bot$$

where T-nonequivalence is generically defined as follows:

$$\text{T-nonequivalence}(Q_A, Q_B, \{i_1,...,i_h\}) = Q_A\text{-}not\text{-}eq\text{-}to\text{-}Q_B\langle i_1,...,i_h\rangle()$$

and Q_A-*not-eq-to-*Q_B is a derived predicate defined by the rules:

$$Q_A\text{-}not\text{-}eq\text{-}to\text{-}Q_B\langle i_1,...,i_h\rangle() \leftarrow Q_{A0}\langle x_1,...,x_{ka}\rangle(v_1,...,v_{na}, r_1,...,r_m) \wedge \neg aux^{Q_A \neq Q_B}\langle i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m\rangle()$$

$$aux^{Q_A \neq Q_B}\langle i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m\rangle() \leftarrow Q_{B0}\langle x_1',...,x_{kb}'\rangle(v_1',...,v_{nb}', r_1,...,r_m)$$
$$\wedge \neg\text{T-nonequivalence}(B_1, B_1', \{i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m, v_1',...,v_{nb}'\}),$$
$$\wedge \neg\text{T-nonequivalence}(B_1', B_1, \{i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m, v_1',...,v_{nb}'\})$$
$$\wedge ... \wedge \neg\text{T-nonequivalence}(B_k, B_k', \{i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m, v_1',...,v_{nb}'\})$$
$$\wedge \neg\text{T-nonequivalence}(B_k', B_k, \{i_1,...,i_h, v_1,...,v_{na}, r_1,...,r_m, v_1',...,v_{nb}'\}).$$

As an example, consider the mapping assertion $Q^{S1} = Q^{S2}$ from Figure 1. It would be translated into Q^{S1}-not-eq-to-Q^{S2}<>() $\rightarrow \bot$ and Q^{S2}-not-eq-to-Q^{S1}<>() $\rightarrow \bot$, where:

Q^{S1}-not-eq-to-Q^{S2}<>() $\leftarrow$ $Q^{S1}{}_0$(po, st, bt) $\wedge \neg$ aux$^{QS1 \neq QS2}$<po, st, bt>()

aux$^{QS1 \neq QS2}$<po, st, bt> $\leftarrow$ $Q^{S2}{}_0$(o, st, bt) $\wedge \neg Q^{S1}{}_1$-not-eq-to-$Q^{S2}{}_1$<po, st, bt, o>() $\wedge \neg Q^{S2}{}_1$-not-eq-to-$Q^{S1}{}_1$<po, st, bt, o>()

$Q^{S1}{}_1$-not-eq-to-$Q^{S2}{}_1$<po, st, bt, o>() $\leftarrow$ $Q^{S1}{}_1$<po>(it, pn, q, p) $\wedge \neg$ aux$^{QS11 \neq QS21}$<po, st, bt, o, it, pn, q, p>()

aux$^{QS11 \neq QS21}$<po, st, bt, o, it, pn, q, p>() $\leftarrow$ $Q^{S2}{}_1$<o>(it', pn, q, p)

$Q^{S2}{}_1$-not-eq-to-$Q^{S1}{}_1$<po, st, bt, o>() $\leftarrow$ $Q^{S2}{}_1$<o>(it', pn', q', p') $\wedge \neg$ aux$^{QS21 \neq QS11}$<po, st, bt, o, it', pn', q', p'>()

aux$^{QS21 \neq QS11}$<po, st, bt, o, it', pn', q', p'>() $\leftarrow$ $Q^{S1}{}_1$<po>(it, pn', q', p')

Q^{S2}-not-eq-to-Q^{S1}<>() $\leftarrow$ $Q^{S2}{}_0$(o, st', bt') $\wedge \neg$ aux$^{QS2 \neq QS1}$<o, st', bt'>()

aux$^{QS2 \neq QS1}$<o, st', bt'>() $\leftarrow$ $Q^{S1}{}_0$(po, st', bt') $\wedge \neg Q^{S2}{}_1$-not-eq-to-$Q^{S1}{}_1$<po, st', bt', o>() $\wedge \neg Q^{S1}{}_1$-not-eq-to-$Q^{S2}{}_1$<po, st', bt', o>()

4. CHECKING DESIRABLE PROPERTIES OF XML MAPPINGS

Our approach to validation of XML mappings is aimed at providing the designer with a set of desirable properties that the mapping should satisfy. For each property to be tested, a query that formalizes the property is defined. Then, the *CQC method* [12] is used to determine whether the property is satisfied, i.e. whether the query is satisfiable. In addition to the query stating the property, the *CQC method* requires also the logic database schema for which satisfiability of the query should be tested.

4.1 Strong Mapping Satisfiability

A mapping is *strongly satisfiable* if there is a pair of schema instances that make all mapping assertions true in a non-trivial way. In the relational setting [18], the trivial case is that in which all queries in the assertion return an empty answer. In XML, however, queries may

Instance 1 of S1:

```
<orderDoc>
 <purchaseOrder>
  <customer>Andy</customer>
  <shipAddress>
   <singleAddress>Address1
   </singleAddress>
  </shipAddress>
 </purchaseOrder>
 <purchaseOrder>
  <customer>Mary</customer>
  <item>
   <productName>product1
   </productName>
   <quantity>2</quantity>
   <price>50</price>
  </item>
  <shipAddress>
   <twoAddresses>
    <shipTo>Address2</shipTo>
    <billTo>Address3</billTo>
   </twoAddresses>
  </shipAddress>
 </purchaseOrder>
</orderDoc>
```

Instance 2 of S1:

```
<orderDoc>
 <purchaseOrder>
  <customer>Joan</customer>
  <shipAddress>
   <singleAddress>Address4
   </singleAddress>
  </shipAddress>
 </purchaseOrder>
 <purchaseOrder>
  <customer>Mary</customer>
  <item>
   <productName>product1
   </productName>
   <quantity>2</quantity>
   <price>50</price>
  </item>
  <shipAddress>
   <twoAddresses>
    <shipTo>Address2</shipTo>
    <billTo>Address3</billTo>
   </twoAddresses>
  </shipAddress>
 </purchaseOrder>
</orderDoc>
```

Instance of S2:

```
<orderDB>
 <order>
  <id>0</id>
  <shipTo>Address2</shipTo>
  <billTo>Address3</billTo>
 </order>
 <item>
  <order>0</order>
  <name>product1</name>
  <quantity>2</quantity>
  <price>50</price>
 </item>
</orderDB>
```

Figure 2: Counterexample for mapping losslessness.

return a nested structure. Therefore, testing this property must make sure that all levels of nesting can be satisfied non-trivially, as shown in our example in the introduction. Then, strong satisfiability of XML schema mapping must be formalized as follows:

Definition 1. *An XML schema mapping* M *between schemas* S1, S2 *is strongly satisfiable if* $\exists I^{S1}$, I^{S2} *instances of* S1 *and* S2, *respectively*, *such that* I^{S1} *and* I^{S2} *satisfy the assertion in* M, *and for each assertion* $Q^{S1}{}_i$ op_i $Q^{S2}{}_i$ *in* M, *the answer to* $Q^{S1}{}_i$ *in* I^{S1} *is a strong answer. We say that* R *is a strong answer if*: (1) R *is a simple type value*, (2) R *is a record* $[R_1, ..., R_n]$ *and* $R_1, ..., R_n$ *are all strong answers*, *or* (3) R *is a non-empty set* $\{R_1, ..., R_n\}$ *and* $R_1, ..., R_n$ *are all strong answers*.

The query that specifies strong satisfiability of a mapping *M* is defined as follows:

$$Q_{stronglySat} \leftarrow \text{StrongSat}(Q^{S1}{}_1, \varnothing) \wedge ... \wedge \text{StrongSat}(Q^{S1}{}_n, \varnothing),$$

where StrongSat is a function generically defined as follows. Let *V* be a generic (sub-)query in *M*:

V: for $\$v_1$ in $path_1$, ..., $\$v_s$ in $path_s$ where *cond*
return $[A_1, ..., A_m, B_1, ..., B_k]$,

where $A_1, ..., A_m$ are simple-type expressions and $B_1, ..., B_k$ are query blocks, and let V_0 be the translation of the outermost block of *V* (obtained as explained in Section 3.3). Then,

$$\text{StrongSat}(V, inheritedVars) = V_0\langle x_1,...,x_r\rangle(v_1,...,v_s, r_1,...,r_m) \wedge \text{StrongSat}(B_1, inheritedVars \cup \{v_1,...,v_s, r_1,...,r_m\}) \wedge ... \wedge \text{StrongSat}(B_k, inheritedVars \cup \{v_1,...,v_s, r_1,...,r_m\}),$$

where $\{x_1,...,x_r\} \subseteq inheritedVars$.

The logic schema *DB* that must be considered to check satisfiability of $Q_{stronglySat}$ is obtained as follows. Let DR_M, IC_M be the deductive rules and denial constraints that result from the translation of the assertions of mapping $M = \{Q^{S1}{}_1\ op_1\ Q^{S2}{}_1, ..., Q^{S1}{}_n\ op_n\ Q^{S2}{}_n\}$. Let DR_{S1}, IC_{S1} and DR_{S2}, IC_{S2} be the rules and constraints from the translation of mapped schemas *S1* and *S2*, respectively. Then, $DB = (DR_{S1} \cup DR_{S2} \cup DR_M, IC_{S1} \cup IC_{S2} \cup IC_M)$.

As an example, consider the mapping *M* in Figure 1. Strong satisfiability of this mapping is defined by the query: $Q_{stronglySat} \leftarrow Q^{S1}{}_0(po, st, bt) \wedge Q^{S1}{}_1\langle po\rangle(it, pn, q, p)$. Note that the second literal in the body of this query may never be satisfied because every possible instantiation either violates the range restriction on price element of schema *S1* or it violates the mapping assertion (more specifically, the definition of Q^{S2}). Therefore, and as we have also mentioned in the introduction, *M* is not strongly satisfiable. Such unsatisfiability is determined by applying the *CQC method* to $Q_{stronglySat}$.

4.2 Mapping Losslessness

The *mapping losslessness* property [18] allows the designer to provide a query defined over one of the mapped schemas and check whether all the data needed to answer that query is mapped. It can be used, for example, to know whether a mapping that may be partial or incomplete suffices for the intended task, or to be sure that certain private information is not made public by the mapping.

Definition 2. *Let* Q *be a query posed on schema* S1. *Let* M *be an XML mapping between schemas* S1, S2 *with assertions*: $\{Q^{S1}{}_1\ op_1\ Q^{S2}{}_1, ..., Q^{S1}{}_n\ op_n\ Q^{S2}{}_n\}$. *We say that* M *is* lossless with respect to Q *if* $\forall I^{S1}{}_1, I^{S1}{}_2$ *instances of* S1 *both*

(1) $\exists I^{S2}$ *instance of* S2 *such that* $I^{S1}{}_1$ *and* $I^{S1}{}_2$ *are both mapped into* I^{S2}, *and*

(2) $\forall\ Q^{S1}{}_i\ op_i\ Q^{S2}{}_i$ *mapping assertion from* M, *the answer of* $Q^{S1}{}_i$ *over* $I^{S1}{}_1$ *is equal to the answer of* $Q^{S1}{}_i$ *over* $I^{S1}{}_2$,

imply that the answer of Q *over* $I^{S1}{}_1$ *is equal to the answer of* Q *over* $I^{S1}{}_2$.

In other words, mapping M is lossless w.r.t. Q if the answer to Q is determined by the extension of the $Q^{S1}{}_i$ queries, where these extensions must be the result of evaluating the queries over an instance of *S1* that is mapped into some consistent instance of *S2*.

As an example, consider the mapping M in Figure 1. Suppose that we have changed "./price/text() <= 5000" by "./price/text() > 5000" in the definition of Q^{S2} in order to make M strongly satisfiable. Consider also the following query Q:

Q: for $sa in //singleAddress return [$sa/text()].

Intuitively, mapping M is not lossless w.r.t. Q because it maps the purchase orders with *twoAddresses*, but not the ones with *singleAddress*. More formally, we can find a counterexample that shows M is lossy w.r.t. Q. This counterexample is depicted in Figure 2, and it consists in two instances of *S1* that have the same extension for Q^{S1}, that are both mapped to a consistent instance of *S2*, and that have different answers for Q.

Let $M = \{Q^{S1}{}_1\ op_1\ Q^{S2}{}_1, ..., Q^{S1}{}_n\ op_n\ Q^{S2}{}_n\}$ be a mapping between schemas *S1* and *S2*, and let Q be a query over *S1*. The query that specifies losslessness of mapping M with respect to query Q is defined as follows:

$Q_{lossy} \leftarrow$ T-noncontainment(Q, Q', $\varnothing$),

where Q' is a copy of Q in which each element name n has been renamed n'.

The logic schema DB that must be considered to check satisfiability of Q_{lossy} is defined as follows. Let DR_{S1}, IC_{S1} and DR_{S2}, IC_{S2} be the rules and constraints from the translation of *S1* and *S2*, respectively; let DR_{S1}', IC_{S1}' be a copy of DR_{S1}, IC_{S1} in which each predicate p has been renamed p'; and let DR_L, IC_L be the result of translating the assertions: $Q^{S1}{}_1 = Q^{S1}{}_1$', ..., $Q^{S1}{}_n = Q^{S1}{}_n$'. Then, $DB = (DR_{S1} \cup DR_{S2} \cup DR_M \cup DR_{S1}' \cup DR_L, IC_{S1} \cup IC_{S2} \cup IC_M \cup IC_{S1}' \cup IC_L)$.

If the *CQC method* can build an instance of DB in which Q_{lossy} is true, this instance can be partitioned in three instances: one for *S1*, one for *S1*', and one for *S2*. Since *S1* and *S1*' are actually two copies of the same schema, we can say that we have two instances of *S1*. Both are map to the instance of *S2* (because IC_M), and share the same answer for the $Q^{S1}{}_i$ queries in mapping M (because IC_L). Also, since Q_{lossy} is true and its definition requires that $Q \not\equiv Q$', the two instances of *S1* have different answers for query Q. In conclusion, we have got a counterexample that shows M is lossy w.r.t. query Q.

4.3 Mapping Inference

The *mapping inference* property [16] checks whether a given mapping assertion is inferred from a set of others assertions. It can be used, for instance, to detect redundant assertions or to test equivalence of mappings.

Definition 3. *Let* M *be an XML mapping between schemas* S1, S2. *Let* F *be a mapping assertion between* S1 *and* S2. *We say that* F *is inferred from* M *if* $\forall I^{S1}, I^{S2}$ *instances of schemas* S1 *and* S2, *respectively*, *such that* I^{S1}, I^{S2} *satisfy the assertions in* M, *then* I^{S1}, I^{S2} *also satisfy assertion* F.

The query that specifies the mapping inference property with respect to a given assertion F is defined as follows:

- If F is a containment assertion, i.e. $Q_1 \sqsubseteq Q_2$, query $Q_{notInferred}$ will be defined by a single rule:
  $Q_{notInferred} \leftarrow$ T-noncontainment(Q_1, Q_2, $\varnothing$).
- Otherwise, if F is like $Q_1 = Q_2$, there will be two rules:
  $Q_{notInferred} \leftarrow$ T-nonequivalence(Q_1, Q_2, $\varnothing$)
  $Q_{notInferred} \leftarrow$ T-nonequivalence(Q_2, Q_1, $\varnothing$).

The logic schema DB to be used to test the satisfiability of query $Q_{notInferred}$ is $DB = (DR_{S1} \cup DR_{S2} \cup DR_M, IC_{S1} \cup IC_{S2} \cup IC_M)$.

As an example, let F be $Q_1 = Q_2$, and let Q_1, Q_2 be the following queries defined over the schemas shown in Figure 1:

```
Q1: for $po in //purchaseOrder
    return [for $sa in $po/shipAddress/singleAddress return [$sa/text()],
            for $ta in $po/shipAddress/twoAddresses return
                [$ta/shipAddress/text(), $ta/billTo/text()]
            for $it in $po/item return
                [$it/productName/text(), $it/quantity/text(), $it/price/text()] ]

Q2: for $o in /orderDB/order where
    not(/orderDB/item[./order/text() = $o/id/text() and ./price/text() > 5000])
    return [for $st in $o/shipTo, $bt in $o/billTo where $st/text() = $bt/text()
                return [$st/text()],
            for $st in $o/shipTo, $bt in $o/billTo where $st/text() ≠ $bt/text()
                return [$st/text(), $bt/text()],
            for $it in //item[./order/text() = $o/id/text()]
                return [$it/name/text(), $it/quantity/text(), $it/price/text()] ]
```

Assertion F maps both the purchase orders that have a *twoAddresses* node, and also those with a *singleAddress* node. It fixes thus the problem of mapping M not being lossless w.r.t. the *singleAddress* information (see Section 4.2). Let us suppose that we want to see whether F is inferred from M. We apply the *CQC method* over $Q_{notInferred}$ and we obtain a counterexample, which consists in a pair of

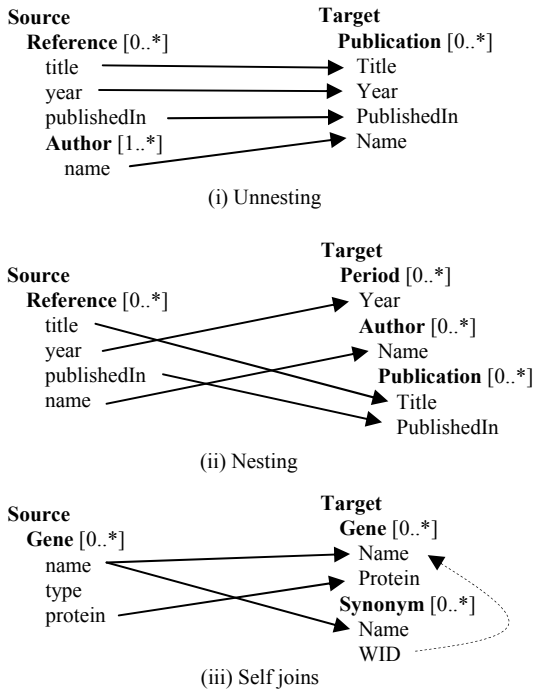


Figure 3: Mapping scenarios taken from the STBenchmark [2].

	strong map. satisfiability		mapping inference		mapping losslessness	
	#constraints	#rules	#constraints	#rules	#constraints	#rules
unnesting	50	28	50	43	78	62
nesting	51	33	51	37	76	57
self joins	46	30	46	38	68	66

Table 1: Size of the logic database schemas that result from the translation of the mapping scenarios in Figure 3.

schema instances that satisfy M (because IC_M), that is, they share the *twoAddresses* nodes, but do not satisfy F (because the definition of $Q_{notInferred}$), that is, they do not have the same *singleAddress* nodes. Therefore, F is not inferred from M.

5. EXPERIMENTS

To show the feasibility of our approach, we perform a series of experiments and report the results in this section. We perform the experiments on an Intel Core2 Duo machine with 2GB RAM and Windows XP SP3.

The mapping scenarios we use for the experiments are adapted from the *STBenchmark* [2]. From the basic mapping scenarios proposed in this benchmark, we consider those that can be easily rewritten into the class of mapping scenarios described in Section 2.1 and that have at least one level of nesting. These scenarios are the ones called: *unnesting* and *nesting*. We also consider one of the flat relational scenarios, namely the one called *self joins*, to show that our approach generalizes the relational case. These mapping scenarios are depicted in Figure 3.

For each one of these three mapping scenarios we validate the three properties discussed in the paper, i.e., strong mapping satisfiability, mapping losslessness and mapping inference. In order to do this, we apply the translation presented in this paper to transform each mapping scenario into a logic database schema and the mapping validation test into a query satisfiability test over the logic schema. Note that although [2] expresses the mappings in the global-as-view formalism, they can be easily rewritten into the formalism we consider in this paper as mapping assertions in the form of $Q_{source} \sqsubseteq Q_{target}$. Since we have not yet implemented the automatic XML-to-logic translation, we perform it manually. The number of constraints and deductive rules in the resulting logic schemas are shown in Table 1.

To execute the corresponding query satisfiability tests, we use the implementation of the *CQC method* that is the core of our existing relational mapping validation tool (MVT) [19].

We perform two series of experiments, one in which the three properties hold for each mapping scenario, and one in which they do not. The results of these series are shown in Figure 4(a) and 4(b), respectively.

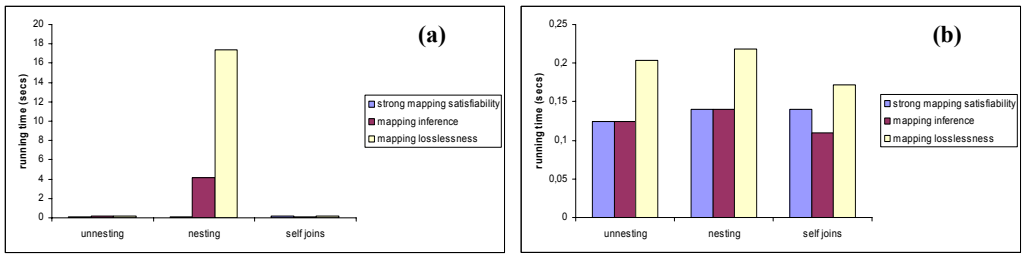


Figure 4: Experiment results when (a) the mapping properties hold and when (b) they do not.

In Figure 4(a), since the properties of mapping inference and mapping losslessness must be checked with respect to a user-provided parameter, and given that we want the mappings to satisfy these properties, we check whether a "strengthened" version of one of the mapping assertions is inferred from the mapping in each case, and whether each mapping is lossless with respect to a strengthened version of one of its mapping queries. These strengthened queries and assertions are built by taking the original ones and adding an additional arithmetic comparison. Similarly, in Figure 4(b), we strengthen the assertions/queries in the mapping and use one of the original ones as parameter for mapping inference and mapping losslessness, respectively. Regarding strong mapping satisfiability, we introduce two contradictory range constraints, one in each mapped schema, in order to ensure the property will "fail".

We can see in Figure 4(a) that the three properties are checked fast in the *unnesting* and *self joins* scenarios, while mapping inference and mapping losslessness require much more time to be tested in the *nesting* scenario. This is not unexpected since the mapping queries of the nesting scenario have two levels of nesting, while those from the other two scenarios are flat. To understand why mapping inference and mapping losslessness are the most affected by the increment in the level of nesting, we must recall how the properties are reformulated in terms of query satisfiability. In particular, the query to be tested for satisfiability in both mapping losslessness and mapping inference reformulation encodes the negation of a query containment assertion that depends on the parameter query/assertion, as shown in Section 4. Therefore, an increment of the level of nesting of the mapping scenario is likely to cause an increment of the level of nesting of the tested query, which is what happens in the *nesting* scenario; and a higher level of nesting means a more complex translation into logic, involving multiple levels of negation, as shown in Section 3.3.

In Figure 4(b), we can see that all three properties run fast and that there is no much difference between the mapping scenarios. It is also remarkable the performance improvement of the *nesting* scenario with respect to Figure 4(a). To understand these results we must remember that mapping inference and mapping losslessness are both check by means of searching for a counterexample. That means its checking can stop now as soon as the counterexample is found, while, in Figure 4(a), all relevant counterexample candidates had to be evaluated. The behavior of strong mapping satisfiability is exactly the opposite; however, the results of this property in this series of experiments are very similar to those in Figure 4(a). The intuition to this is that strong satisfiability requires all mapping assertions to be non-trivially satisfied; thus, as soon as one of them cannot be so, the checking process can stop.

6. RELATED WORK

Existing approaches for validating mappings are [1, 6, 7], which focus on mappings between nested relational schemas. In this paper, we consider a more general class of XML schemas since we also allow the use of `<choice>`, range restrictions, and mapping assertions with negations and order comparisons. The mapping formalism used in [6, 7] is the one of tuple-generating dependencies (TGDs). These are logic formulas in the form of $\forall \bar{X}\,(p(\bar{X}) \rightarrow \exists \bar{Y}\, q(\bar{X}, \bar{Y}))$, where p and q are conjunctive queries. In [1], the more recent formalism of nested mappings [13] is considered. This formalism allows nesting TGDs, which results in more expressive and compact mappings. In both cases, the mappings can be reformulated into the class of mapping assertions that we consider in this paper, in particular into assertions of the form of $Q_1 \sqsubseteq Q_2$.

As an example, consider the nested relational schemas in Figure 5(a) and the following nested mapping (taken from [13]):

N: for p in projs exists d' in depts
where d'.dname = p.dname ∧ d'.emps = E[p.dname]
∧ (for e in p.emps exists e' in d'.emps
where e'.ename=e.ename ∧ e'.salary=e.salary)

Notice the use of the Skolem function *E* to express that employees must be grouped in the target by department name. A straightforward reformulation can be done as follows. First, we extend the mapped schemas to the ones shown in Figure 5(b). Then, we define mapping assertion $Q_{source} \sqsubseteq Q_{target}$ as follows:

Q_{source}: for $p in //proj
return [$p/dname/text(), //E[./input/text()=$p/dname/text()]/output/text(), for $e in $p/emp
return [$e/ename/text(), $e/salary/text()]]

Q_{target}: for $d' in //dept
return [$d'/dname/text(), $d'/empsSetId/text(), for $e' in $d'/emps/emp
return [$e'/ename/text(), $e'/salary/text()]]

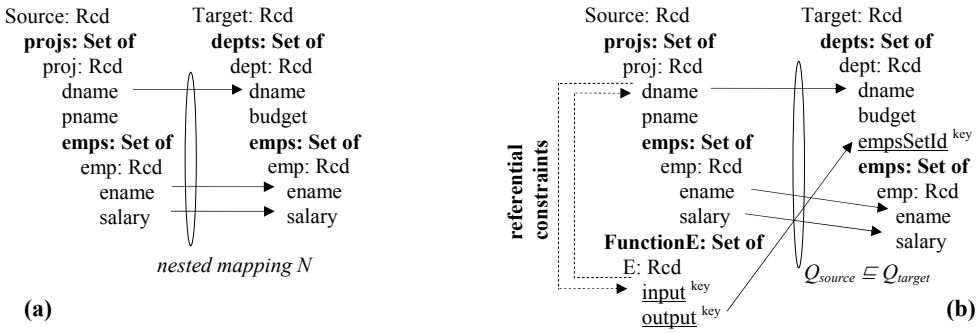


Figure 5: (a) A nested mapping scenario and (b) its reformulated version.

Notice that the implicit semantics of Skolem functions has been made explicit in Figure 5(b) by means of the introduction of new elements and constraints into the schemas.

Outside the context of mapping validation, the compilation of an XML mapping into a relational one has been used to solve the problem of query reformulation [9] (i.e. rewriting a query through a mapping) using a generic relational schema encoding based on predicates such as *child*(x, y) and *desc*(y, z) that model the parent-child and ancestor-descendant relationships, respectively. However, this generic encoding alone does not model the entire mapped schemas, so [9] assumes that any other information in these schemas is provided already translated in the form of a set of disjunctive embedded dependencies (DEDs), which are formulas like $\forall \bar{X}\ (p(\bar{X}) \rightarrow \exists \bar{Y}_1\ q_1(\bar{X}, \bar{Y}_1) \vee ... \vee \exists \bar{Y}_n\ q_n(\bar{X}, \bar{Y}_n))$, where p and q_i are conjunctive queries with (in)equalities. Alternatively, our translation is more focused on how to translate the schemas into a class of constraints (see Section 2.2) that includes that of DEDs. Regarding the mapping formalism, [9] deals with global-as-view and local-as-view queries, that is, a mapping is represented by a set of queries. Although these queries return a nested structure as the ones we consider, they do not allow order comparisons or negations.

Information preservation in XML mappings has been studied in [4, 5]. The property of query preservation requires that, for a particular query language, all queries on the source schema can be answered on the target. This property, although related with our mapping losslessness property, is not the same property. In fact, our losslessness property assumes that mappings may be partial or incomplete, and thus, not query preserving.

Mapping satisfiability has been studied in [3] for DTD schemas and XML mappings expressed as implications (TGDs) of tree patterns. However, negation, arithmetic comparisons or integrity constraints are not considered in [3]. Since it is not obvious how the results of [3] could be applied to the class of schemas and mappings we consider in this paper, we can say that our approach complements the work of [3].

As we widely discussed in [18, 19], our approach of testing desirable properties is complementary to the existing approaches to mapping validation [1, 5, 7].

7. CONCLUSIONS AND FURTHER WORK

We have proposed an approach to the validation of XML schema mappings which is based on the translation of the schemas and the mapping into a first-order logic formalism. In this way, we can take advantage of our previous work on validating relational mappings [18] and check certain desirable properties of mappings automatically. We do that by means of reasoning over the mapping definition itself rather than relying on specific instances that may not reveal all the potential pitfalls.

As further work, we plan to implement this work into our mapping validation tool [19], which currently can only deal with relational mappings. It would also be interesting to consider mapping assertions relating XQuery and SQL/XML [20] queries. That would allow us to validate mappings between XML and relational schemas directly, without first having to rewrite SQL queries into XQuery.

ACKNOWLEDGEMENTS

This work has been supported in part by Microsoft Research through the European PhD Scholarship Programme, and by the Ministerio de Ciencia e Innovación under project TIN2008-03863.

8. REFERENCES

[1] Bogdan Alexe, Laura Chiticariu, Renée J. Miller, Wang Chiew Tan: Muse: Mapping Understanding and deSign by Example. ICDE 2008: 10-19.
[2] Bogdan Alexe, Wang Chiew Tan, Yannis Velegrakis: STBenchmark: towards a benchmark for mapping systems. PVLDB 1(1): 230-244 (2008)
[3] Shun'ichi Amano, Leonid libkin, Filip Murlak: XML Schema Mappings. PODS 2009:33-42.

[4] Denilson Barbosa, Juliana Freire, Alberto O. Mendelzon: Information Preservation in XML-to-Relational Mappings. XSym 2004: 66-81.
[5] Philip Bohannon, Wenfei Fan, Michael Flaster, P. P. S. Narayan: Information Preserving XML Schema Embedding. VLDB 2005: 85-96.
[6] Angela Bonifati, Giansalvatore Mecca, Alessandro Pappalardo, Salvatore Raunich, Gianvito Summa: Schema mapping verification: the spicy way. EDBT 2008: 85-96.
[7] Laura Chiticariu, Wang Chiew Tan: Debugging Schema Mappings with Routes. VLDB 2006: 79-90.
[8] Alin Deutsch, Bertram Ludäscher, Alan Nash: Rewriting queries using views with access patterns under integrity constraints. Theor. Comput. Sci. 371(3): 200-226 (2007).
[9] Alin Deutsch, Val Tannen: XML queries and constraints, containment and reformulation. Theor. Comput. Sci. 336(1): 57-87 (2005).
[10] Xin Dong, Alon Y. Halevy, Igor Tatarinov: Containment of Nested XML Queries. VLDB 2004: 132-143.
[11] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, Lucian Popa: Data Exchange: Semantics and Query Answering. ICDT 2003: 207-224.
[12] Carles Farré, Ernest Teniente, Toni Urpí: Checking query containment with the CQC method. Data Knowl. Eng. 53(2): 163-223 (2005).
[13] Ariel Fuxman, Mauricio A. Hernández, C. T. Howard Ho, Renée J. Miller, Paolo Papotti, Lucian Popa: Nested Mappings: Schema Mapping Reloaded. VLDB 2006: 67-78.
[14] Maurizio Lenzerini: Data Integration: A Theoretical Perspective. PODS 2002: 233-246.
[15] Alon Y. Levy, Dan Suciu: Deciding Containment for Queries with Complex Objects. PODS 1997: 20-31.
[16] Jayant Madhavan, Philip A. Bernstein, Pedro Domingos, Alon Y. Halevy: Representing and Reasoning about Mappings between Domain Models. AAAI/IAAI 2002: 80-86.
[17] Lucian Popa, Yannis Velegrakis, Renée J. Miller, Mauricio A. Hernández, Ronald Fagin: Translating Web Data. VLDB 2002: 598-609.
[18] Guillem Rull, Carles Farré, Ernest Teniente, Toni Urpí: Validation of mappings between schemas. Data Knowl. Eng. 66(3): 414-437 (2008).
[19] Guillem Rull, Carles Farré, Ernest Teniente, Toni Urpí: MVT: a schema mapping validation tool. EDBT 2009: 1120-1123.
[20] SQLX: SQL/XML. http://www.sqlx.org/.
[21] Jeffrey D. Ullman: Principles of Database and Knowledge-Base Systems, Volume II Computer Science Press 1989.
[22] W3C: XML Schema. http://www.w3.org/TR/xmlschema-0/.
[23] W3C: XQuery Language. http://www.w3.org/TR/xquery/.
[24] Cong Yu, H. V. Jagadish: XML schema refinement through redundancy detection and normalization. VLDB J. 17(2): 203-223 (2008).