# FORMAT EVOLUTION

Ralf Lämmel* Wolfgang Lohmann†

***Abstract***

*A systematic approach to the adaptation of XML documents and their DTDs is developed. The approach facilitates the evolution of XML-based formats. There are two essential ideas. Firstly, changes in the formats of documents are represented as stepwise transformations on the underlying DTDs. Secondly, the corresponding migration of the XML data is largely induced by the DTD transformations. The presentation focuses on concepts of format evolution, namely roles of corresponding transformations, properties of the transformations, and expressiveness to implement the transformations.*

## 1 Introduction

**The XML-age** XML (cf. [31]) is more and more used as interchange format in distributed, client-server, intra- and internet applications. It also finds its way to specific application domains to serve as storage and exchange format, e.g., for abstract syntax trees or intermediate representations in language processing. XML is usually employed back-to-back with DTDs (document type definitions; cf. [31]) in order to constrain the XML documents according to a specific format. In a non-trivial application architecture of the XML-age, various components cooperate based on various interchange formats (i.e., DTDs) serving as contracts between the components. The underlying data might be stored in a database (presumably according to some relational or object-oriented database schema). XML data is obtained via database queries. XML data might also be managed in a database or in a file system in a more native manner. The components of the application query the database, access XML files, or they obtain the (XML) data from other components. The data to be presented to the user as spreadsheets, active web-pages, reports etc. is also encoded as XML data.

**Evolution of XML/DTD** In many application contexts of XML, the formats regulated by the underlying DTDs are repeatedly changed, either due to maintenance requirements, or due to refactoring, or due to independent evolution of one of the components. Also, if different components need to interact, often the corresponding interfaces need to be matched up—especially if the interaction was not

*CWI & VU Amsterdam, The Netherlands, `ralf@cwi.nl`
†Universität Rostock, Germany, `wlohmann@informatik.uni-rostock.de`

anticipated (cf. [7]). In the present paper, we show that the systematic evolution of DTDs can usually be represented by a number of atomic transformation steps which in turn induce a transformation for XML document migration. The resulting *evolution by transformation* approach is outlined in Figure 1. While XML document transformation is an established concept, the intertwined treatment of format change and document migration has not been addressed elsewhere in depth. Such a treatment contributes to *format and document reengineering*. The overall approach is not too much dependent on XML/DTD. The concepts should also be of value for other technical settings of format evolution, e.g., for proprietary formats, or for the setting of SGML (cf. [28]). The approach is also applicable to XML schema languages other than DTD, e.g., XML Schema (cf. [32, 33, 34]) or DSD (cf. [24]).

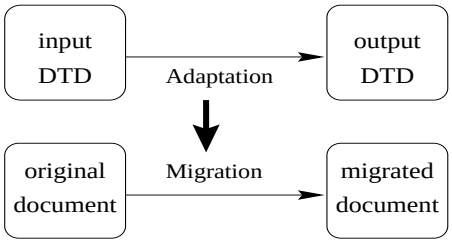


Figure 1. Induced XML document migration

**Databases and XML** We should point out that our approach to format evolution is only sensible if the DTDs and XML documents at hand are the primary artifacts, that is, if XML is used as interchange or storage format in the first place. By contrast, if the XML data originates from database queries, and the underlying DTD corresponds to a (relational or an object-oriented) database schema (cf. [29]), then the database schema and the database instance are the primary artifacts. Consequently, we had to focus on a (database) schema evolution as opposed to a (DTD) format evolution. There is a large body of research on database re- and reverse engineering dealing with the problem of schema evolution and instance mapping (cf. [9]). As an aside, if XML data is solely *managed* in a database according to some mapping (cf. [17]), we still consider DTDs and XML documents as the primary artifacts.

**Structure of the paper** In Section 2, we present our approach in a nutshell, that is, we examine a few scenarios of transforming DTDs, and we indicate in what sense the DTD transformation steps induce a migration for XML documents. In Section 3, we define some terms to reason about DTD and XML transformations. In the subsequent two sections, we systematically examine roles of format evolution, that is, transformations for DTDs and XML documents. There are two groups: In Section 4, transformations for refactoring are studied. In Section 5, structure-extending and -reducing transformations are studied. The assumption underlying our approach is that the evolution of a format can be represented as a number of transformation steps according to the identified roles. The paper is concluded in Section 6 including a discussion of related work.

**Disclaimer** Due to space constraints and the complexity of XML, we can only cover a (nevertheless substantial) fragment of the full expressiveness of XML and DTDs. As for the element type declarations, we only cover the form that children are precisely specified (as opposed to the options `ANY` or `MIXED`), and the form `#PCDATA` (which is a specific instance of `MIXED`). Also, we do not consider entity declarations at all. Furthermore, we only consider the attribute type `CDATA`, and enumerated types. In fact, the treatment of other attribute types is not straightforward. The treatment of `ID`s and `IDREF`s, for example, requires to deal explicitly with references in a document.

**Acknowledgement** Part of the article was presented by the first author at the Dagstuhl Seminar No. 01041 "*Interoperability of Reengineering Tools*". The work of the first author was supported, in part, by *NWO*, in the project "*Generation of Program Transformation Systems*". The work of the second author was supported by Universiteit van Amsterdam through a grant for a research visit. We are grateful for Paul Klint's support of this project. Finally, we would like to thank Rasmus Faust, Jens Jahnke, and Meike Klettke for fruitful discussions on the subject of the paper.

## 2 The approach in a nutshell

We will explain the rationale for our approach to format evolution by a few illustrative transformations on a simple sample DTD, and a corresponding XML document. Figure 2 shows a DTD for music albums, whereas Figure 3 shows an XML document with a particular album. The example has been adopted from [30].

```
<!ELEMENT album (title, artist, recording?, catalogno*, player+, track*, notes)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT artist (#PCDATA)>
<!ELEMENT recording EMPTY>
    <!ATTLIST recording date  CDATA #IMPLIED
                        place CDATA #IMPLIED>
<!ELEMENT catalogno EMPTY>
    <!ATTLIST catalogno label       CDATA                #REQUIRED
                        number      CDATA                #REQUIRED
                        format      (CD | LP | MiniDisc) #IMPLIED
                        releasedate CDATA                #IMPLIED
                        country     CDATA                #IMPLIED>
<!ELEMENT player EMPTY>
    <!ATTLIST player name       CDATA #REQUIRED
                     instrument CDATA #REQUIRED>
<!ELEMENT track EMPTY>
    <!ATTLIST track title  CDATA #REQUIRED
                    credit CDATA #IMPLIED
                    timing CDATA #IMPLIED>
<!ELEMENT notes (#PCDATA | albumref | trackref)*>
    <!ATTLIST notes author CDATA #IMPLIED>
<!ELEMENT albumref (#PCDATA)>
    <!ATTLIST albumref link CDATA #REQUIRED>
<!ELEMENT trackref (#PCDATA)>
    <!ATTLIST trackref link CDATA #IMPLIED>
```

Figure 2. A DTD for music albums

```
<?xml version='1.0'?>
<!DOCTYPE album SYSTEM "album.dtd">
<album>
 <title>Time Out</title>
 <artist>Dave Brubeck Quartet</artist>
 <recording date="June-August 1959" place="NYC"/>
 <catalogno label='Columbia' number='CL 1397' format='mono'/>
 <catalogno label='Columbia' number='CPK 1181' format='LP' country='Korea'/>
 <player name='Dave Brubeck' instrument='piano'/>
 <player name='Eugene Wright' instrument='bass'/>
 <player name='Joe Morello' instrument='drums'/>
 <track title='Blue Rondo &agrave; la Turk' credit='Brubeck' timing='6m42s'/>
 <track title='Strange Meadow Lark' credit='Brubeck' timing='7m20s'/>
 <track title='Take Five' credit='Desmond' timing='5m24s'/>
 <track title='Three To Get Ready' credit='Brubeck' timing='5m21s'/>
 <notes> <trackref link='#3'>Take Five</trackref> is a famous jazz track of
 that period. See also <albumref link='cbs-tfo'>Time Further Out</albumref>.
 </notes>
</album>
```

Figure 3. A music album as XML document

### 2.1 Restructuring

Let us assume that the structure of the `album` element type is going to be somewhat richer than originally declared in Figure 2. In particular, there will be further personnel than just players. To prepare for that enrichment, we *fold* `player+` to obtain an element type `personnel`. The relevant fragment of the adapted DTD is the following:

```
<!ELEMENT album (title, artist, recording?, catalogno*, personnel, track*, notes)>
<!ELEMENT personnel (player+)>
```

At the data level, that is, for XML documents, the element tags for `personnel` have to be inserted accordingly. Thus, we get the following:

```
<personnel>
 <player name='Dave Brubeck' instrument='piano'/>
 <player name='Eugene Wright' instrument='bass'/>
 <player name='Joe Morello' instrument='drums'/>
</personnel>
```

The important thing to notice here is that the XML transformation is *induced* by the DTD transformation: Performing a fold at the DTD level, the necessary migration of XML data to witness folding is (completely) determined.

### 2.2 Enrichment

Let us carry on with the evolution. We want to add some structural component to the element type `personnel`, namely we want to support an optional manager. Thus, we get the following extended definition of `personnel`, and a definition of `manager`:

```
<!ELEMENT personnel (player+, manager?)>
<!ELEMENT manager EMPTY>
    <!ATTLIST manager name CDATA #REQUIRED>
```

Actually, this transformation consists of two steps, namely the introduction of `manager`, and the enrichment of the structure of `personnel` to include an optional manager. The introduction step does not affect the XML level because all original element types are preserved. As for the enrichment step, a decision is due. Do we insert actual managers or not? Since the manager is optional, we can decide to leave the field for `manager` blank. Thus, we can say that the XML transformation induced by the enrichment of `personnel` is the identity mapping.

### 2.3 Attributes to elements conversion

As a final example, let us consider the problem of turning attributes into elements. Consider, for example, the element type declaration for `recording` in Figure 2. A `recording` element does not have children but it might carry attributes. Suppose we want to change the status of the attributes `date` and `place` to become element types. Thus, we want to derive the following DTD:

```
<!ELEMENT recording (date?, place?)>
<!ELEMENT date  (#PCDATA)>
<!ELEMENT place (#PCDATA)>
```

The induced XML transformation is somewhat involved. `CDATA` needs to be coerced to `#PCDATA`. Attribute lists have to be converted to elements. The migrated `recording` element is the following:

```
<recording> <date>June-August 1959</date> <place>NYC</place> </recording>
```

This concludes our introductory list of examples. The purpose of the paper is to systematically study the roles involved in DTD transformations, and the induced XML transformations.

### 2.4 Transformation technology

The most obvious language candidate for XML transformation is XSLT (cf. [36]). In fact, throughout the paper, we will use XSLT to illustrate some of the XML transformations which we employ for format evolution. Lacking an obvious candidate for DTD transformations, the transformations at the DTD level are described verbally. If we resorted to XML-based XML schema representations (as opposed to DTDs) such as XML Schema (cf. [32, 33, 34]), we can also use XSLT (or any other XML transformation language) for encoding the schema transformations.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="*|@*|comment()|processing-instruction()|text()">
 <xsl:copy>
  <xsl:apply-templates select="*|@*|comment()|processing-instruction()|text()"/>
 </xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

Figure 4. The identity transformation in XSLT

In Figure 4, we describe, for example, an identity function on XML documents in XSLT. The program descends into all elements and reconstructs them identically. The program is useful as a starting point for more meaningful XML transformations which are specific about certain patterns but behave like identity most of the time. The identity function is encoded with a single XSLT template which applies to all elements (cf. `match="*|..."`), all attributes (cf. `match="...|@*|..."`) and others (i.e., comments, processing instructions, and text). Tags are copied (cf. `<xsl:copy>`). The template recursively descends into all children (cf. `<xsl:apply-templates ...>`).
Despite the popularity of XSLT, our experiments clearly indicated that XSLT is not really convenient for some transformations involved in format evolution. We will detail this experience in Section 6. We will also discuss other languages for XML transformation in Section 6.

## 3 Properties of transformations

We need a few terms to reason about the transformations for format evolution. Transformations are functions, often partial ones. We are concerned with two levels: DTD transformations are (partial) functions on DTD—the domain of all DTDs. XML transformations are (partial) functions on XML—the domain of all XML documents. For every DTD transformation $f$ we intend to supply an *induced* XML transformation $\overline{f}$. Let us consider required or convenient properties of $f$ and $\overline{f}$.

### 3.1 Well-formedness

All DTDs we process have to be well-formed. By well-formedness we mean that the DTDs are deterministic / unambiguous (cf. [31, 6]). Furthermore, we require a reducedness property in the sense of context-free grammars (cf. [11]), that is, all element names used in element type declarations are defined, and they are reachable from the root element, and all alternatives of all declarations are feasible. As for XML documents, the standard regulates what well-formedness means. We restrict XML to the set of all well-formed XML documents. Without further mentioning, all subsequently discussed DTD and XML transformations are constrained by an implicit postcondition to produce well-formed outputs from well-formed inputs (or to fail otherwise).

### 3.2 Validity

Using XML terminology, an XML document is said to be valid w.r.t. a given DTD if the document is formed according to the DTD. We use the notation $d \vdash x$ to express that an XML document $x \in \text{XML}$ is valid w.r.t. a DTD $d \in \text{DTD}$. Informally, this means that $x$ can be derived from $d$ starting with the root element of $d$.[1] A minimum requirement for an induced XML transformation is that validity is preserved, say that the migrated XML data is valid (w.r.t. the adapted DTD):

[1] Here, we resort to terminology used for string languages, e.g., languages defined via context-free grammars.

**Definition 1** *Let $f$ be a DTD transformation. The induced XML transformation $\overline{f}$* preserves validity *if for all $d \in$ DTD and $x \in$ XML with $d \vdash x$ it holds if $f(d)$ and $\overline{f}(x)$ are defined, then $f(d) \vdash \overline{f}(x)$.*

**Example 1** *The XML document from Figure 3 is valid w.r.t. the DTD in Figure 2. Consider the folding step from Section 2.1. The migrated XML document with the additional tags for* `personnel` *is valid w.r.t. the adapted DTD where* `player+` *has been folded to* `personnel`*. Note that the original XML document is not valid w.r.t. the adapted DTD.*

### 3.3 Totality and partiality

DTD transformations are inherently partial because of applicability constraints. Of course, a DTD transformation should be feasible, that is, there exist DTDs which the transformation is applicable to. We should be able to describe precisely the applicability constraints for a DTD transformation.

**Example 2** *Folding is only feasible if the element name introduced by folding (cf.* `personnel` *in Section 2.1) is fresh in the input DTD, and the particle to be folded (cf.* `player+` *in Section 2.1) does indeed occur. Otherwise, there are no preconditions for folding to succeed at the DTD level.*

Ideally, we want an induced XML transformation to be total. In a sense, the DTD transformation should be responsible for establishing all preconditions, and then the induced XML transformation should be enabled. In this case, we speak of a total induced XML transformation:

**Definition 2** *Let $f$ be a DTD transformation. The induced XML transformation $\overline{f}$ is* total *if for all $d \in$ DTD, $x \in$ XML with $d \vdash x$ it holds that $f(d)$ is defined implies $\overline{f}(x)$ is defined.*

**Example 3** *We continue with the folding scenario from Example 2. The induced XML transformation is indeed total because the insertion of the additional tags for* `personnel` *is feasible without further preconditions.*

In turns out that some induced XML transformations have to establish preconditions specific to the data level. Consequently, these induced XML transformations are partial since they are supposed to fail if the preconditions are not met by the XML document at hand.

**Example 4** *Consider, for example, a DTD with a particle "$n?$" for an optional element. Assume that we want to enforce an obligatory element, that is, we have to replace "$n?$" by "$n$". A candidate for the induced XML transformation is the identity mapping. However, for the sake of preserving validity, we have to refuse XML data where the optional element is omitted.*

### 3.4 Structure preservation

Starting at the DTD level of transformation, we would like to be sure that the structure of the input DTD is preserved by the transformation in some reasonable manner. At the XML level of transformation, we also want to be sure that the structure[2] of the documents is preserved in a similar sense. We indeed have to consider structure preservation at both levels: Given a structure-preserving DTD transformation, the structure-preservation for a validity-preserving XML transformation is not implied.
There are different ways to formalise that a transformation is "well-behaved". One classical approach—often used in transformational programming (cf. [26])—is to identify *semantics-preserving* transformations. For that purpose, one could define the semantics of a DTD as the set of all valid XML documents admitted by the DTD. The problem is that this semantics is too concrete, i.e., there are hardly semantics-preserving transformations in this sense. There is no obvious alternative semantics. Furthermore, it is not clear how to lift semantics-preservation from the DTD level to the XML level. This motivates our simple formalisation of structure preservation based on *reversibility*. This is also a prominent approach in the context of database schema transformations (cf. [2]).

**Definition 3** *A transformation $t$ is called* structure-preserving*, if $t$ is a bijective function. The transformation $t$ is called* structure-extending*, if $t$ is an injective but not a surjective function. The transformation $t$ is called* structure-reducing*, if $t$ is not an injective function.*

**Example 5** *Folding as exemplified in Section 2.1 is injective because there is an inverse at the DTD level, that is, unfolding, and there is also an inverse at the XML level, that is, the additional tags which were inserted to witness folding are simply removed again. Since unfolding is also injective, folding and unfolding are structure-preserving.*

In studying the properties of the transformations for format evolution, we do not simply want to check if a transformation is injective or surjective. We also want to explicitly specify the inverse transformations as in the above example. For transformations which are meant to restructure a given DTD, we plan to point out injective inverses. For transformations which are meant to extend the structure at one or both levels, we should be able to point out the inverse transformations which remove the added structure. This also explains how we can discipline structure-reducing transformations. While they cannot be reversed, they should be conceived as inverses of structure-extending transformations.

## 4 Refactoring

We start our catalog of roles for format evolution with transformations for refactoring DTDs. By refactoring we mean structure-preserving transformations intended to improve the structure of DTDs,

[2] For a uniform terminology, we use the term *structure* at both the DTD and the XML level (as opposed to *content*).

or to "massage" the structure so that it becomes more suitable for subsequent adaptations. Structure-extending and -reducing transformations are considered in Section 5.

### 4.1 Renaming

As a warm-up, we start with renaming which is a fundamental operator applicable to names of abstractions of any language. For DTDs, there are two abstractions which have to be covered, namely elements and attributes. Renaming means to replace the corresponding names consistently. Renaming attribute values is conceivable, too. Replacing text is clearly not to be regarded as proper renaming.

**Renaming elements** Suppose the original element name is $n$, and the new element name is $n'$. At the DTD level, the following replacements are due. The element type declaration <!ELEMENT $n$ $e$> is replaced by <!ELEMENT $n'$ $e$> where $e$ serves as place-holder for the definition of $n$.[3] Within all content particles all occurrences of $n$ have to be located and replaced by $n'$ as well. At the XML level, all tags of the form <$n$>, </$n$>, and <$n$/> (optionally with attributes) have to be updated accordingly.

**Renaming attributes** Renaming attributes is only slightly more interesting. One specific property that we have to cope with is a kind of scope rule. Attributes belong to elements. Renaming an attribute (name) $a$ to $a'$ should include the identification of the corresponding element type $n$. At the DTD level, we have to lookup the attribute list declaration <!ATTLIST $n$ $l$>, and then we replace $a$ by $a'$ in the list $l$. At the XML level, similarly, we have to lookup start tags <$n$ $l$> and empty element tags <$n$ $l$/>, and then replace $a$ by $a'$ in the list $l$ of name-value pairs.

**Example 6** *The following XSLT template renames the attribute* `link` *of the element type* `albumref` *to* `url`. *If we add the template for the identity transformation (cf. Figure 4), we will obtain a complete XSLT program.*

```
<xsl:template match="albumref">
 <xsl:copy>
  <xsl:attribute name="url"> <xsl:value-of select="@link"/> </xsl:attribute>
  <xsl:apply-templates select="*|comment()|processing-instruction()|text()"/>
  <xsl:apply-templates select="attribute::*[not(name(.)='link')]"/>
 </xsl:copy>
</xsl:template>
```

*The template matches with* `albumref`. *The attribute value associated with* `link` *is copied using the new name* `url` *in the name-value pair. All other attributes are copied without changes.*

[3] In XML terminology, $e$ is called *content specification*. If $e$ is of the form that *children* are specified for $n$ (as opposed to #PCDATA), then we say that $e$ is built from *content particles*, namely names, choice and sequence lists of content particles, and content particles postfixed with "?", "+", and "*".

**Properties** Renaming admits several universal properties. For renaming elements, these properties can be explained as follows. It is easy to see that renaming is validity-preserving. Renaming is also reversible. Renaming $n$ to $n'$ is undone by renaming $n'$ to $n$. Thus, renaming is structure-preserving. At the DTD level, renaming is not total because of the precondition that $n'$ must be fresh. The induced XML transformation is total.

### 4.2 Introduction and elimination

We describe two auxiliary structure-preserving transformations. One can *introduce* element type declarations which are not (yet) used in the DTD altogether. Subsequent adaptations are meant to make use of the new element types. Dually, one can *eliminate* element type declarations which are not used (anymore) in the DTD.

**Introduction of element types** There are simple preconditions at the DTD level. The introduced element name must be fresh in the input DTD. The induced XML transformation is the identity mapping without further requirements, since all element type declarations are completely preserved.

**Elimination of element types** At the DTD level, an important constraint is that the element type to be eliminated is not used in other declarations. Otherwise, we would create a "non-terminated" DTD. Furthermore, we require that the element type corresponding to the name of the document type is never eliminated (cf. `album` in `<!DOCTYPE album SYSTEM "album.dtd">` in Figure 2). Note that the root element of a valid XML document must match the name of the document type. These preconditions are sufficient to ensure that actual XML data cannot exercise the eliminated element types. Hence, the identity mapping is the appropriate induced XML transformation.

**Properties** Clearly, the induced identity mappings are in both cases validity-preserving since the DTD transformations do not change reachable element type declarations. The induced transformations are also total. Structure-preservation holds since introduction and elimination are each other inverses.

### 4.3 Folding and unfolding

Most forms of abstractions in specification formalisms and programming languages admit two important dual concepts, that is, folding and unfolding. Refer, for example, to [27], for an in-depth discussion of these concepts in the context of functional and logic programming. Folding and unfolding is particularly useful for restructuring DTDs (and, in general, for grammar formalisms). In an abstract sense, unfolding means to replace the name of an abstraction by the definition of it. Dually,

folding means to introduce a new abstraction according to some identified expression, and then to replace the expression by the new name. As for DTDs, folding and unfolding is all about elements.

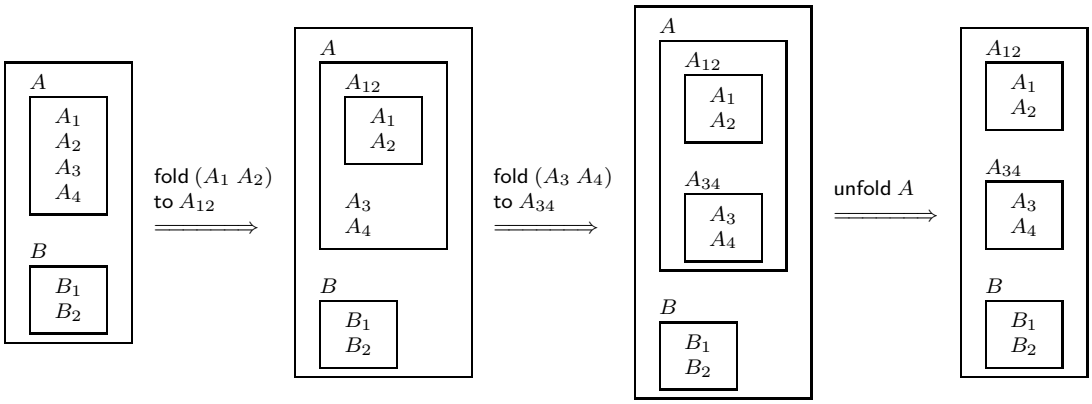


Figure 5. Form restructuring

**Example 7** *In Figure 5, folding and unfolding is illustrated in the context of form restructuring. We start from a form (say DTD) with two top level elements $A$ and $B$ corresponding to the two main regions of the form. We want to refactor the structure underlying the region $A$ in a way that two subregions are identified. The first fold step points out the first subregion $A_{12}$ which encloses $A_1$ and $A_2$. In the same manner, the second fold step points out the second subregion $A_{34}$ enclosing $A_3$ and $A_4$. The unfold step drops the nesting of $A_{12}$ and $A_{34}$ inside $A$. The final form has now three instead of two main regions. The splitting of the region $A$ was achieved by fold-fold-unfold.*

**DTD level** Folding consists of two steps: an introduction step for the new element type, and a replacement step to refer to the new element type instead of using its definition. It is conceivable to define unfolding completely dual to folding, that is, after unfolding the corresponding element type would be eliminated. We do not pursue this approach because, usually, unfolding is only done in the focus of a particular occurrence of the considered element type.[4] Thereby, some references to the unfolded element type will remain after unfolding, and thus elimination will not be feasible. Otherwise unfolding behaves completely inverse to folding, that is, the name of an element type is replaced by its definition.

**XML level** Folding means to identify XML fragments corresponding to the folded expression, and to surround them accordingly with the tags for the new element. We postpone discussing details of the replacement process at both levels until Section 5.2 and Section 5.3. Unfolding is dual to folding, i.e., the tags for the unfolded element are removed.

[4] We assume that all our transformations can be applied in a *focus* as spelled out in [19].

**Example 8** *The following XSLT template describes unfolding for the element type* `personnel`. *Again, if we add the template for the identity transformation (cf. Figure 4), we will obtain a complete XSLT program.*

```
<xsl:template match="personnel">
 <xsl:apply-templates select="*|@*|comment()|processing-instruction()|text()"/>
</xsl:template>
```

*The template applies to elements of type* `personnel`. *As in the case of the identity transformation, we descend into all children. However, there is no mentioning of* `<xsl:copy>` *(as opposed to the template for the identity transformations). Thus, the tags for* `personnel` *are not copied. This directly implements the idea of unfolding.*

**Properties** At the DTD level, folding is constrained by the precondition of element type introduction. Furthermore, we should require that there is indeed an occurrence of the expression to be folded away. Dually, unfolding is only meaningful if we focus on at least one occurrence of the corresponding element type. The induced XML transformations for folding and unfolding are total. Folding can be reversed by unfolding (at both levels). To be precise, unfolding and the replacement step involved in folding are each other's inverses at the DTD level. At the XML level, the described insertion or removal of tags resp. to witness folding and unfolding are obligatory to achieve preservation of validity.

### 4.4 Elements vs. attributes

We only sketch the topic of turning attributes into elements and vice versa. We refer to Section 2.3 for an example of the former direction. The following correspondences can be established to deal with element to attribute conversion and vice versa:

- Implied attributes correspond to optional elements.
- Required attributes correspond to obligatory elements.
- The attribute type `CDATA` corresponds to `#PCDATA` content.

**Turning a content particle into an attribute declaration** A selected content particle $e$ is removed from the content specification of the selected element type $n$. The attribute list declaration for $n$ is extended accordingly. The content particle $e$ has to be of the form "$n'$" or "$n'?$" where $n'$ is the name of an element type. There is no straightforward way to deal with choice lists and with the repetition operators "`+`" and "`*`". Furthermore, the content specification of $n'$ should be `#PCDATA`, since it is not obvious how to turn structured content into an attribute value. The attribute type of $n'$ will be `CDATA`, and the default declaration for $n'$ (`#REQUIRED` vs. `#IMPLIED`) depends on the form of $e$

as pointed out above. Another precondition for the transformation (at the DTD level) is that $n'$ must not yet be used as an attribute name in the attribute list declaration for $n$.

**Turning an attribute declaration into a content particle** A selected attribute declaration $a$ is removed from the attribute list declaration for the selected element type $n$. We extend the content specification of $n$ by a corresponding particle, the location of which needs to be defined explicitly. The attribute name $n'$ of $a$ serves as the name of the element type in the content particle $e$ to be inserted. The particle $e$ will be of the form "$n'?$" if $n'$ is an implied attribute, or the form "$n'$" if $n'$ is a required attribute. There is no obvious way to deal with attribute types for $n'$ other than CDATA.[5] We assume that the introduction of the corresponding element type for $n'$ (with #PCDATA as content specification) is done separately before the attribute to element conversion.

**XML level** An element is turned into a name-value pair by regarding #PCDATA content as a CDATA value, and by interpreting the element name as an attribute name; similarly for the other direction. If the element is not present, then we omit the (implied) attribute.

**Example 9** *The following XSLT template implements an attributes to elements conversion for elements of type* recording. *In fact, all attribute values are turned into corresponding elements. Therefore, we perform iteration over all attributes (cf.* <xsl:for-each select="@*">*). The template is again to be completed by the template for identity transformation (cf. Figure 4). The code only provides an approximative solution as it ignores the problem that the order of the actual name-value pairs might differ from the order prescribed in the content specification which was derived from the attribute list declaration.*

```
<xsl:template match="recording">
 <xsl:copy>
  <xsl:for-each select="@*">
   <xsl:element name = "{name(.)}"> <xsl:value-of select="."/> </xsl:element>
  </xsl:for-each>
  <xsl:apply-templates/>
 </xsl:copy>
</xsl:template>
```

## 5 Construction and destruction

As a matter of fact, format evolution is hardly restricted to pure restructuring. New structure (say content) needs to be included. Dually, existing structure (say content) might become obsolete. Two classes of adaptations are identified for these purposes, namely *construction* and *destruction*. We will discuss corresponding roles back-to-back since the latter behave as the inverses of the former. In format evolution, steps for refactoring, construction, and destruction alternate.

[5] We can cope with enumerated types for attributes the values of which are then represented as #PCDATA however.

### 5.1 Generalisation and restriction of expressions

**Generality of content particles** In the process of extending a format, we need to generalise content particles; dually for restricting a format. Contrast that with renaming, folding, and unfolding where we deal with tags only. To reason about generality of content particles we define a partial order "$<$" on content particles:

| | | | |
|---|---|---|---|
| $e_1$ | $<$ | $e_3$ | if $e_1 < e_2$ and $e_2 < e_3$ |
| $e_1$ | $\not<$ | $e_2$ | if $e_2 < e_1$ |
| $e$ | $<$ | $e?$ | |
| EMPTY | $<$ | $e?$ | |
| $e^+$ | $<$ | $e^*$ | |
| $e?$ | $<$ | $e^*$ | |
| $e$ | $<$ | $e^+$ | |
| $(e_1, \ldots, e_n)$ | $<$ | $(e'_1, \ldots, e'_n)$ | if $1 \leq i \leq n, e_i < e'_i, e_j = e'_j$ for $j = 1, \ldots, n, j \neq i$ |
| $(e_1 \vert \cdots \vert e_n)$ | $<$ | $(e'_1 \vert \cdots \vert e'_n)$ | if $1 \leq i \leq n, e_i < e'_i, e_j = e'_j$ for $j = 1, \ldots, n, j \neq i$ |
| $(e_1 \vert \cdots \vert e_n)$ | $<$ | $(e_1 \vert \cdots \vert e_{i-1} \vert e_{i+1} \vert \cdots \vert e_n)$ | if $1 \leq i \leq n$ |

The first two properties state transitivity and asymmetry. Then, we have a number of properties to deal with optionals, star- and plus-notation. Afterwards, sequence lists (with associative concatenation) are addressed. Finally, we address choice lists. It is said, for example, that a choice with less branches admits less structure than a choice with more branches. Generality of content particles can be lifted to generality of DTDs. We say that $d \in \text{DTD} < d' \in \text{DTD}$ if for all $x \in \text{XML}$, $d \vdash x$ implies $d' \vdash x$ but not vice versa.

**DTD level** A replacement of a content particle $e$ by another content particle $e'$ is called a (DTD) *generalisation* if $e < e'$. In the opposite case (i.e., $e' < e$), the replacement is called a (DTD) *restriction*. The specification of "$<$" clearly indicates how we can generalise; dually for restriction: A child can be turned into an optional one. EMPTY can be replaced by an optional. From $e$ we might also generalise to plus-lists of $e$. Finally, we can add alternatives to choice lists.

**XML level** Initially, we choose the identity mapping as induced XML transformation for both generalisation and restriction. For the former, no further preconditions are to be ensured since a generalisation simply relaxes the format. For the latter, we have to check that the generality of the original particle is never exercised. Consider, for example, a restriction of "$n$+" to "$n$". If we encounter an XML document where we have non-trivial lists of elements of type $n$, we have to refuse the document (for the sake of preservation of validity). This is a serious limitation. Thus, we give up the idea of the identity mapping. Instead, we need to perform an XML transformation to simplify the XML data so that the generality which is removed by the DTD restriction is not exercised anymore. Such a simplifying XML transformation $h$ (say XML restriction) can be characterised via the "$<$" relation: For all $d \in \text{DTD}$, there exists some $d' \in \text{DTD}$ with $d' < d$ such that for all $x \in \text{XML}$ with $d \vdash x$ it holds that $h(x)$ is defined implies $d' \vdash h(x)$. Now, if the $d'$ always corresponds to the adapted DTD according

to a DTD restriction, both levels of restriction are properly intertwined. In practice, we will specify the DTD transformation by a focused replacement of one content particle $e$ by a more restrictive one $e'$. We might also specify the XML restriction $h$ by just a function on content of the form $e$ to content of the form $e'$. The function operating on XML can be derived by lifting.

**Properties** XML generalisations preserve validity because of the way how "$<$" is defined for DTDs. XML generalisations are structure-extending simply because the co-domain of the transformation is not exhausted. The above definition of an XML restriction directly enforces preservation of validity. XML restrictions are structure-reducing simply because XML data exercising the obsolete generality will be mapped to a restricted format.

### 5.2 First intermezzo: Replacing content particles

Replacement at the DTD level is not just relevant for generalisation and restriction where the original and the resulting content particles are related by "$<$". We also encountered replacement in the context of folding where a content particle had to be replaced by the name of a newly introduced element type. Further examples are unfolding and renaming. We want to explain the general concept of replacement in some technical detail.

Replacement can be performed as follows. Given two content particles $u$ and $v$, we want to lookup the occurrences of $u$ in the input DTD, and we want to replace them by $v$ to derive the output DTD. If there are multiple matches for $u$, we process them in some order. For certain transformations, we might require unique matches. We assume that replacement is always done in a certain focus, for example, for a certain element type declaration, or for a certain occurrence of the particle $u$. It is important to notice that the operands $u$ and $v$ can be in general sequence lists of content particles rather than single particles. In the context of a fold, for example, $u$ is potentially a list of content particles to be folded, and $v$ is the new element name. Thus, the lookup problem comes down to associative list pattern matching (cf. [3]) where we try to match part of a list $l = (e_1, \ldots, e_m)$ of content particles from the input DTD against $u$. The list $l$ might occur at the top level of a content specification for an element type $n$, or in a nested occurrence, e.g., as an alternative of a choice list. Matching means that we want to find $p$ and $q$ with $1 \leq p \leq q \leq m$ such that $(e_p, \ldots, e_q) = u$. The list $l$ is to be replaced by the concatenation of $(e_1, \ldots, e_{p-1})$, $v$, and $(e_{q+1}, \ldots, e_m)$.

### 5.3 Second intermezzo: Matching at the XML level

There is no (completely) determined way how replacement at the XML level would be induced by a DTD transformation. However, the general matching problem preceding specific replacements (and also tests for preconditions), that is, the problem to find the content fragments corresponding to the

focus of a DTD transformation is interconnected to the matching problem (i.e., part of the replacement problem) at the DTD level. There are two cases:

(1.) The sequence list $l$ matching with the operand $u$ occurred at the top level of the element type declaration for $n$. We can easily locate the elements of type $n$ by looking for matches with <$n$>$f$</$n$>. Thus, the list $l$ corresponds directly to the fragment $f$, that is, $f$ is supposed to match the content specification $l$. We split up the fragment $f$ into $f_p\, f_u\, f_q$ according to the associative list pattern matching at the DTD level. The fragment $f_u$ matches $u$, and the surrounding fragments $f_p$ and $f_q$ correspond to the prefix $(e_1, \ldots, e_{p-1})$ and the postfix $(e_{q+1}, \ldots, e_m)$ of $l$. Note that there is, of course, no 1-1 correspondence of particles to fragments because some elements might be optional, others might be lists. The kind of matching to determine $f_p$, $f_u$, and $f_q$ corresponds to parsing, or—more precisely—to regular expression pattern matching (cf. [14, 12]).

(2.) Matching rather happens at some level of nesting. Thus, we first need to descend into $f$ to navigate to the relevant sequence list. This is again a kind of regular expression pattern matching, i.e., we parse $f$ according the element definition for $n$, and we select the right parse subtree $f'$ corresponding to $l$. Then, we proceed as in the 1st case, i.e., we split $f'$ into $f_p\, f_u\, f_q$ and so on.

Once matching is done, one can observe $f_u$, for example, to check a certain precondition. One can also perform an actual replacement, that is, $f_u$ is replaced by content for $v$ in the context $f_p\, f_u\, f_q$. In the case of unfolding $n'$, for example, we know that $f_u$ must be of the form <$n'$>$f'_u$</$n'$>, and we simply replace $f_u$ by $f'_u$, i.e., we omit the tags for $n'$. In the case of a partial identity transformation complementing a DTD restriction at the XML level (cf. Section 5.1), we can ensure the precondition that $f_u$ is covered by the restricted particle by a simple inspection of $f_u$.

**Example 10** *Suppose we want to turn an optional* `manager` *into an obligatory one, i.e., we need to replace* `manager?` *by* `manager`. *The obligation at the XML level is that we make sure that all elements of type* `personnel` *contain a manager. This basically means that we locate all XML content fragments corresponding to the particle* `manager?`*, and then we only have to examine the matched fragment not to be empty.*

### 5.4 Structure enrichment and removal

In a sense, generalisation means to add possible "branches" to the format, vice versa for restriction. There is another mode of construction and destruction, that is, if we want to enrich the content model or to remove part of it by means of inserting or removing content particles.

**DTD level** Enrichment can be conceived as a replacement as follows. Given a content particle $u$ (typically a sequence list $(e_1, \ldots, e_m)$), we want to replace it by another content particle $v$ which subsumes $u$ in the following sense. The new particle $v$ is of the form $(e'_1, \ldots, e'_{m'})$ with $m' > m$, and

there are $i_1, \ldots, i_m$ such that $1 \leq i_1 < \cdots < i_m \leq m'$[6] and $e_1 = e'_{i_1}, \ldots, e_m = e'_{i_m}$. For removal, the roles of $u$ and $v$ are flipped. This explanation makes immediately clear that enrichment and removal can be implemented by means of the replacement procedure discussed in Section 5.2.

**XML level** To evolve the XML data, we need to insert or to remove elements corresponding to the additional or removed particles. In the case of removal, the induced XML transformation is merely a projection. Matching is sufficient to select those fragments which should be kept. Insertion is more delicate. For new particles, we have to decide how suitable content is obtained. As for optional elements and star-lists of elements, this is easy since the empty content is valid in this case; refer to Section 2.2 for the example dealing with an optional manager. Otherwise, we have to augment the transformation for enrichment to supply suitable content. This problem is somewhat similar to the problem of an XML restriction which had to be provided explicitly to enforce a format (say DTD) restriction. Due to space constraints, we omit a thorough discussion.

**Properties** At the DTD level, there is a way to conceive enrichment and removal as each other's inverse (and hence they would be structure-preserving). For that purpose, we do not directly consider DTDs but DTDs with a focus. Then, we can define an equivalence on DTDs (with a focus), where $d \equiv d'$ if $d$ and $d'$ only differ in the focused particle. This point of view emphasizes that the structure-extending or -reducing behaviour of enrichment and removal essentially happens at the XML level. Removal of content is structure-reducing because projections are not injective. By duality, enrichment cannot be surjective, and hence it is structure-extending.

### 5.5 Attributes

In the same way as we can generalise and restrict choice lists for element type declarations (cf. Section 5.1), we can also extend or shrink enumeration types for attributes. There is also a counterpart for structure enrichment and removal discussed above, namely attribute addition and removal.

**Example 11** *Let us consider a simple scenario in the running example. We want to enable DVD albums. Thus, we need to add another alternative (namely,* `DVD`*) to the* `format` *attribute type in the attribute list declaration for* `catalogno`*:*

```
<!ELEMENT catalogno EMPTY>
 <!ATTLIST catalogno label       CDATA                       #REQUIRED
                     number      CDATA                       #REQUIRED
                     format      (CD | LP | MiniDisc | DVD) #IMPLIED
                     releasedate CDATA                       #IMPLIED
                     country     CDATA                       #IMPLIED>
```

[6] If we omitted this condition, we consider permutations of sequence lists. For the sake of orthogonality, permutation should constitute a separate operation.

## 6 Concluding remarks

First, we summarise the contribution of the paper. Then, related work is discussed. The paper is concluded with the formulation of a challenge in transformational programming to effectively support the proposed style of format evolution.

### 6.1 Contribution

The paper illustrates how DTD transformations and induced XML counterparts can be employed to master format evolution problems. The use of transformations supports traceability, and it automates document migration. The roles of format evolution were inspired by our previous research on grammar adaptation (cf. [19, 20]). DTD transformations, in a sense, can be conceived as grammar transformations. In [20], we describe how to use term rewriting technology to describe grammar transformations rigorously. This approach would also applicable to DTD transformations since DTDs are to some extent grammars. Of course, DTDs are not quite context-free grammars or extended BNFs. We have to cope, for example, with tags, attributes, and references. Also, DTDs and context-free grammars require different semantic treatments. Consider, for example, fold/unfold manipulations. They do not affect the string language generated by a context-free grammar. By contrast, folding/unfolding a DTD changes the content model. The original contribution of the present paper is the consideration of XML transformations *induced* by the DTD transformations, and the investigation of schema transformations in the application context of XML.

### 6.2 Related work

**One-level vs. two-level transformation** Previous work on XML transformation considers only the scenario, where a transformational program consumes XML documents of one kind and produces some output—often some XML document again. In the optimal case, this scenario is applied in a typeful manner, that is, both inputs and outputs are required to be valid w.r.t. some DTD, and validity is then enforced dynamically or statically. DTDs are considered as given a-priori. We call this (common) transformation setup *one-level transformation*. Another well-known example of one-level transformation is *refactoring* of object-oriented programs (cf. [25, 8]). By contrast, we are concerned with *two-level transformation* (in the context of format evolution). In our case, transformation starts at the DTD level. The induced XML transformations are not typed by fixed DTDs. Conceptually, the induced XML transformations are generic in the sense that they are feasible for all input DTDs which satisfy the preconditions of the underlying DTD transformation.

**Limitations of XSLT** XSLT (Version 1.0) is not optimally suited for XML transformations which are *conceptually* based on DTD patterns. Most transformations for format evolution indeed are con-

cerned with XML fragments according to DTD patterns as described in Section 5.2 and Section 5.3. By contrast, the kind of selection or matching supported by XSLT is based on patterns and axes in the sense of XPath (cf. [35]). XSLT allows one to constrain types and ancestors of nodes to be matched. One can also constrain the position in the list of children. The conditions in matching and selection may involve various axes to select from different parts of the tree relative to the current node (generally the node that the XSLT template matches). All this expressiveness, however, does not allow us to describe matching according to the patterns (consisting of content particles) occurring in element type declarations. The need for language support to process an XML document according to the DTD patterns in a typeful fashion has already been realised by others. In [14, 12], *regular expression pattern matching* is proposed as a language construct.

```
<xsl:template match="player">
 <xsl:choose>
  <xsl:when
   test="preceding-sibling::*[1][self::player]"/>
  <xsl:otherwise>
   <personnel>
    <xsl:apply-templates select="."
     mode="players"/>
   </personnel>
  </xsl:otherwise>
 </xsl:choose>
</xsl:template>

<xsl:template match="player" mode="players">
 <xsl:copy>
  <xsl:apply-templates select="*|@*|..."/>
 </xsl:copy>
 <xsl:apply-templates
  select="following-sibling::*[1][self::player]"
  mode="players"/>
</xsl:template>
```

```
<xsl:template match ="player[position()=1]">
 <xsl:text disable-output-escaping="yes">
  <![CDATA[<personnel>]]>
 </xsl:text>
 <xsl:copy>
  <xsl:apply-templates select="*|@*|..."/>
  </xsl:copy>
</xsl:template>

<xsl:template match ="player[position()=last()]">
 <xsl:copy>
  <xsl:apply-templates select="*|@*|..."/>
 </xsl:copy>
 <xsl:text disable-output-escaping="yes">
  <![CDATA[</personnel>]]>
 </xsl:text>
</xsl:template>
```

Figure 6. Two inconvenient implementations of the fold problem from Section 2.1

**Example 12** *Let us implement the fold step from the beginning of the paper (cf. Section 2.1). The plan would be to perform a match with* `player+` *(to focus on the lists of players), and to surround the matched players with* `<personnel>` *and* `</personnel>`. *By contrast, XSLT matching works element-wise. In Figure 6, we show two XSLT encodings for the folding problem. The left program catches the first player, and then switches to a different mode to copy all the players. The additional tags are inserted accordingly (cf.* `<personnel>` *...* `</personnel>`*). In a sense, we* parse *the list of players. The right program in Figure 6 is somewhat simpler but at the expense of a hopelessly untyped approach. The first and the last player are caught by two independent templates. The open and close tags are smuggled in as text (cf.* `<![CDATA[<personnel>]]>` *and* `<![CDATA[</personnel>]]>`*).*

Any XSLT encoding of replacements which have to do with more complex patterns than single elements requires the simulation of regular expression pattern matching. Such parsers are encoded by means of several templates, auxiliary modes, variables to accumulate substructures, or what have you.

A less fundamental but annoying problem of available XSLT implementations is that *valid* XML documents are not enforced statically. Given an XSLT script, there is no guarantee that the input document will be properly queried, and that a valid output document will be generated. It is not a fundamental problem. In [23], it is shown that type-checking XML transformations based on a model covering the essentials of XSLT, namely $k$-pebble tree transducers, is decidable.

**Other XML transformation frameworks** There is an abundance of XML query languages applicable for extraction and restructuring XML data. Refer to [5] for some recent comparative analysis of some prominent representatives. We want to point out some language proposals originating from the functional programming field which have something to offer in the context of our specific kind of transformations. The typed functional language XDuce (cf. [13]) is a statically typed programming language in the spirit of mainstream functional languages but specialised to the domain of XML processing. XDuce supports regular expression types and the aforementioned regular expression pattern matching to be applicable to XML-kind types. Another functional language for typeful XML transformation is XM$\lambda$ (cf. [22]). The formula underlying the functional approach is the following: DTDs are types. XML document fragments are values. Valid XML documents are enforced statically by type-checking. XML transformations are DTD-typed functions.
A convenient property underlying the XSLT language model, is the genericity which allows us to achieve easily a full traversal of the input document by means of default templates as in the identity transformation (cf. Figure 4). One way to recover this genericity in a functional language setting is to resort to an untyped generic XML representation, and to use a suitable combinator library (cf. [30]) for manipulation. Another way could be based on (typeful) generic programming (cf. [16, 10]).

**Database schema evolution** There is a large body of related research addressing the problem of (relational or object-oriented) database schema evolution (cf. [1]). Schema evolution is useful, for example, in database re- and reverse-engineering (cf. [9]). The schema transformations themselves can be compared with our format transformations only at a superficial level because of the different formalisms involved (ER model etc. vs. DTD). However, database schema evolution provides another instance of a two-level transformation setup since one usually requires an instance mapping for a schema evolution (cf. [2]). The formal underpinnings of schema transformations have been studied in great detail, e.g., schema equivalence (cf. [18]), or reversibility of transformations (cf. [9]). There exist formal frameworks for the definition of schema transformations (cf. [21]), and different catalogs have been developed (cf. [1, 4]). Numerous formalisms have been proposed. One appealing method which also covers the intertwined character of schema transformation and instance mapping is based on graph transformations, say graph grammars (cf. [15]). This formal approach is reasonably accessible because of the use of a graphical notation.

### 6.3 A challenge

For a *fixed* DTD, the *description* of both a DTD transformation (for format evolution), and the induced XML transformation is merely a matter of choice of some convenient transformation language. We have examined typeful functional XML transformation languages, term rewriting systems, combinator libraries, and logic programming. However, the *coupled* treatment of DTD transformations and induced XML transformations in a *typeful* and *generic* manner, poses a challenge for formal reasoning, type systems, and language design. Consider, for example, a generic fold function, which takes the particle to be folded, the name of the element type to be introduced, and which returns two functions, namely a DTD transformation, and a dependently typed XML transformation. How can we describe such a function so that properties like preservation of validity are implied by the type of the function? How can the induced character of the XML transformation be made explicit? We might conceive a DTD transformation as a *type transformation* $t$. The induced XML transformation in turn had to be regarded as a function where source and target are related via $t$. In [22], it is pointed out that, indeed, an important topic for future research are *generic* document transformations, and generic functional programming (cf. [16, 10]) is considered as a promising framework in this respect. As yet, it is not clear how generic programming can be made fit to cope with the kind of type and value transformations relevant in format evolution, especially if we think of a complete coverage of XML.

## References


[1] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD Record (Proc. Conf. on Management of Data)*, 16(3):311–322, May 1987.

[2] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual database design*. Benjamin/Cummings, Redwood City, US, 1992.

[3] D. Benanav, D. Kapur, and P. Narendran. Complexity of Matching Problems. *Journal of Symbolic Computation*, 3(1–2):203–216, Feb.–Apr. 1987.

[4] M. Blaha and W. Premerlani. A catalog of object model transformations. In *Third Working Conference on Reverse Engineering*. IEEE, 1996.

[5] A. Bonifati and S. Ceri. Comparative Analysis of Five XML Query Languages. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(1):68–81, 2000.

[6] A. Brüggemann-Klein. *Formal Models in Document Processing*. Habilitation, Institut für Informatik, Universität Freiburg, Freiburg, Germany, 1993.

[7] M. G. Burke et al. XML Transformation: Matching & Reconciliation, 2001. available at http://www.research.ibm.com/hyperspace/mr/.

[8] M. Fowler. *Refactoring—Improving the Design of Existing Code*. Addison Wesley, 1999.

[9] J.-L. Hainaut, C. Tonneau, M. Joris, , and M. Chandelon. Schema Transformation Techniques for Database Reverse Engineering. In *Proc. of the 12th Int. Conf. on ER Approach*, Arlington-Dallas, 1993. E/R Institute.

[10] R. Hinze. A new approach to generic functional programming. In T. W. Reps, editor, *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, January 19-21*, pages 119–132, Jan. 2000.

[11] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, N. Reading, MA, 1980.

[12] H. Hosoya and B. Pierce. Regular Expression Pattern Matching. In *POPL '01. Proc. Conference on Principles of Programming Languages*, New York, NY, USA, 2001. ACM Press.

[13] H. Hosoya and B. C. Pierce. XDuce: A Typed XML Processing Language (Preliminary Report), May 2000. WebDB workshop.

[14] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22, Sept. 2000.

[15] J.-H. Jahnke and A. Zündorf. Applying Graph Transformations To Database Re-Engineering. In *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific, 1999.

[16] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[17] M. Klettke and H. Meyer. Managing XML documents in object-relational databases. In *XML Europe*, 2000.

[18] I. Kobayashi. Losslessness and semantic correctness of database schema transformation: another look of schema equivalence. *Information Systems*, 11(1):41–59, 1986.

[19] R. Lämmel. Grammar Adaptation. In *Proc. Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*. Springer-Verlag, 2001.

[20] R. Lämmel and G. Wachsmuth. Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. In M. van den Brand and D. Parigot, editors, *Proc. LDTA'01*, volume 44 of *ENTCS*, pages 1–25. Elsevier Science, Apr. 2001.

[21] P. McBrien and A. Poulovassilis. A Formal Framework for ER Schema Transformation. In D. W. Embley and R. C. Goldstein, editors, *Conceptual Modeling - ER '97, 16th International Conference on Conceptual Modeling, Los Angeles, California, USA, November 3-5, 1997, Proc.*, volume 1331 of *LNCS*, pages 408–421. Springer-Verlag, 1997.

[22] E. Meijer and M. Shields. XM$\lambda$: A Functional Language for Constructing and Manipulating XML Documents. (Draft), 1999.

[23] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. In *Proc. 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*, pages 11–22. ACM, 2000.

[24] A. M. Nils Klarlund and M. I. Schwartzbach. DSD 1.0 Specification, 2000. `http://www.brics.dk/DSD/`.

[25] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[26] H. A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.

[27] A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, June 1996.

[28] Information processing – Text and office systems – Standard Generalized Markup Language (SGML), 1986. ISO 8879:1986 (JTC 1/SC 34).

[29] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proc. VLDB Conference*, Sept. 2000.

[30] M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *International Conference on Functional Programming (ICFP'99), Paris, France, ACM SIGPLAN*, Sept. 1999.

[31] Extensible Markup Language (XML) 1.0 (second edition), Oct. 2000. `http://www.w3.org/TR/REC-xml`.

[32] XML Schema Part 0: Primer, Oct. 2000. `http://www.w3.org/TR/xmlschema-0/`.

[33] XML Schema Part 1: Structures, May 2001. `http://www.w3.org/TR/xmlschema-1/`.

[34] XML Schema Part 2: Datatypes, May 2001. `http://www.w3.org/TR/xmlschema-2/`.

[35] XML Path Language (XPath) Version 1.0, Nov. 1999. `http://www.w3.org/TR/xpath`.

[36] XSL Transformations (XSLT) Version 1.0, Nov. 1999. `http://www.w3.org/TR/xslt`.