Prolog-basierte Modellierung von Format-Evolution für semi-strukturierte Daten

Entwurf Diplomarbeit
Universität Rostock, Fachbereich Informatik

Vorgelegt von
Faust, Rasmus

Geboren am 9. Februar 1976 in Rostock

Betreuer:
Prof. Dr. G. Riedewald
Dr. R. Lämmel
W. Lohmann
Dr. M. Klettke

Abgabedatum:
30. September 2001

Zusammenfassung

Hier steht die Zusammenfassung über das Format-Evolution-Thema.

Abstract

Here goes the abstract about this format evolution thing.

CR-Klassifikation

1.3, 2.3

Schlüsselwörter

XML, Re-Engineering

Inhaltsverzeichnis

Danksagungen

Bedanken möchte ich mich besonders bei Dr. Ralf Lämmel und Wolfgang Lohmann sowie meinen weiteren Betreuern Prof. G. Riedewald und Dr. M. Klettke, die mir mit Ideen, Kritik und Motivation bei der Arbeit zur Seite standen; bei Christian Romberg, der mir spezifische Fragen zu XMLSchema beantworteten konnte, sowie bei allen, die ein Stück ihrer Lebenszeit hergaben, um diese Arbeit korrekturzulesen. Ein weiterer Dank gilt den Erfindern von LaTeX, ohne deren Software diese Arbeit vermutlich weniger formal geraten wäre.

Kapitel 1

Einleitung

Im Bereich der Software-Technik ist das Problem, dass Software immer wieder geändert und an neue Bedingungen angepasst werden muss, seit langem bekannt und gut untersucht [Boe81, Boe88, GKM95, Bal98, Fow99]. Analog dazu besteht das Problem, Dateiformate zu ändern – ein Grund kann sein, neue Eigenschaften der geänderten Software speichern zu müssen, eine andere Ursache sind direkt neue Anforderungen der Benutzer. Einen großen Anteil an den Kosten der Format-Änderung hat die Entwicklung von Konvertern, die die vorhanden Alt-Dateien in das neue Format übertragen müssen. Bei manueller Entwicklung besteht hier die Gefahr von Implementationsfehlern, deren Folgen wegen dem möglichen Verlust oder der Verfälschung von Daten hohe Kosten verursachen können. Die automatische Erzeugung von entsprechenden korrekten Konvertern aus den Format-Änderungen ist daher ein entscheidender Schritt, um den Aufwand für die Umsetzung von Format-Änderungen zu senken.

Diese Problematik wurde für relationale und Objekt-Datenbanken bereits erkannt [HTJ^{+}93, BKKK87], für semistrukturierte Daten ist die Diskussion allerdings noch nicht so weit fortgeschritten. Diese Arbeit konkretisiert die in [LL01] entwickelte Idee der Format-Evolution, das heißt die Weiterentwicklung eines Datei-Formats, unter Erhaltung der vorhandenen Dokumente (soweit möglich). Die eventuell nötige Anpassung der Dokumente wird dabei durch Konverter abgewickelt, die automatisch aus den Format-Änderungen erzeugt werden.

Es werden Operationen definiert, mit deren Hilfe sich Formate ändern lassen. Aus diesen Operationen lassen sich jeweils Dokument-Konverter ableiten. Die Arbeit demonstriert die Verwendung dieser Operationen für XML, wobei als Format-Spezifikation XMLSchema verwendet wird. Die Operationen selbst sind so allgemein, dass sie auch in einem anderen Kontext als XML verwendet werden können.

Kapitel 2

Grundlagen

Dieses Kapitel gibt zunächst einen kurzen Überblick über die relevaten Technologien. Anschließend werden bestehende andere Ansätze zur Dokument- und Format-Modifikation vorgestellt und die benötigten Begriffe definiert.

2.1 Verwendete Format-Notationen

Diese Arbeit setzt die entwickelten Formalismen im Kontext von XML um. Als Dokumentformat-Beschreibung wird XMLSchema verwendet. Beide Formalismen werden in diesem Abschnitt vorgestellt. Zusätzlich wird am Ende eine verkürzte Notation für die Formatbeschreibung, wie sie in den Beispielen in dieser Arbeit verwendet wird, definiert.

2.1.1 XML

Die *Extensible Markup Language* XML [W3C00] stellt einen allgemeinen Rahmen für ein Vielzahl von Sprachen dar. Auf der Grundlage von XML existieren beispielsweise Sprachen zur Programmierung von Transformationen (XSLT, siehe Abschnitt 2.3.1, S. 12), zur Beschreibung von Syntax (XMLSchema, siehe Abschnitt 2.1.2, S. 9), und zur Speicherung von technischen Dokumenten [OAS01]. Sie hat in den letzten Jahren zudem für den Datenaustausch von strukturierten und semistrukturierten Daten an Bedeutung gewonnen, da sie hinreichend generisch ist, um standardisierte Werkzeuge (Parser, Editoren etc.) entwickeln zu können, und flexibel genug, um sie bequem an spezielle Anwendungen anpassen zu können. Zudem lassen sich XML-Dokumente einfach im *World Wide Web* publizieren, da die dort verwendete Sprache HTML mit XML verwandt ist, so dass sich für die Transformation die vorhandene Software-Infrastruktur nutzen lässt.

XML basiert auf der *Standard Generalized Markup Language* SGML [ISO86], hat aber gegenüber dieser einige Vorteile, die ihre Verbreitung förderten:

Einfachheit Die syntaktische Analyse (das Parsen) von XML-Dokumente ist möglich, ohne dass die konkrete Syntax-Ausprägung bekannt ist. Dadurch vereinfacht sich der Aufbau von generischen Parsern, so dass diese preiswerter und effizienter werden.

Verfügbarkeit des Standards Im Gegensatz zu der ISO-Norm ist die Spezifikation von XML kostenlos und in elektronischer Form über das Internet verfügbar.

Diese beiden Aspekte führten dazu, dass der Aufwand für die Entwicklung entsprechender Werkzeuge sank und somit mehr und preiswertere Software als für SGML zur Verfügung steht.

XML-Dokumente speichern hierarchisch organisierte Daten in einer Baumstruktur. Die Knoten dieses Baumes werden als Elemente bezeichnet, besitzen Attribute und können wiederum Unterknoten enthalten. Elemente werden syntaktisch durch sogenannte *Tags* repräsentiert. Elemente, die weder Unterknoten noch Text enthalten werden als leer bezeichnet und mit einem speziellen Tag notiert:

```
<name attr="val"/>.
```

Alternativ wird der Unterbaum des betreffenden Elementes durch ein *Start Tag* und ein *End Tag* eingeschlossen. Die Attribute des Elementes werden dabei im *Start Tag* notiert:

```
<name attr="val"> sub-tree </name>.
```

Für ein XML-Dokument lassen sich mittels einer *Document Type Description* DTD nun die benötigten konkreten Elemente und deren mögliche Struktur in einem Dokument deklarieren. Diese Deklaration umfasst neben den Attributen eines Dokumentes auch die zulässigen Unterelemente. Letztere werden durch einen regulären Ausdruck spezifiziert.

Die DTD [W3C00] lässt es lediglich für Attribute zu, einen sogenannten Default-Wert zu spezifizieren (dieser Default-Wert wird „aktiv“, wenn in einem konkreten Dokument kein Wert für das Attribut angegeben ist). Dies ist dann in XMLSchema [W3C01a, W3C01b, W3C01c] (siehe Abschnitt 2.1.2, S. 9) auch für Elemente möglich. Dasselbe gilt für den fixierten Wert eines Attributes (DTD, XMLSchema) bzw. Elementes (XMLSchema). Hat ein Attribut/Element einen fixierten Wert, so darf im Dokument entweder kein Wert definiert sein, oder aber dieser Wert muss mit dem fixierten Wert übereinstimmen.

Das folgende Beispiel zeigt zunächst einen Eintrag aus einer Filmdatenbank, die XML zur Speicherung ihrer Daten verwendet:

```
<movie title="Vector Space" timing="110min">
  <scifi>
    <bibliography>
      <book title="Visual Basic for construction workers"/>
    </bibliography>
    <stuntman name="W. Miller"/>
    <guru name="Soda-Largo"/>
    <maiden name="Enginarella"/>
  </scifi>
</movie>
```

Auch ein entsprechendes Fragment der DTD soll hier gezeigt werden, das die Elemente `movie`, `scifi` und `stuntman` sowie deren Attribute deklariert:

- `movie` wird so definiert, dass sein Unterelement entweder `scifi`, `action` oder `kungfu` sein kann.

- `scifi` enthält die Abfolge der Elemente `bibliography`, einem oder mehreren Vorkommen von `stuntman`, und schließlich `guru` und `maiden`.

- `stuntman` wird so definiert, dass dieses Element weder Unterelemente noch Text enthält (`EMPTY`).

Es folgen dann die Attributdeklarationen für `movie` und `stuntman`. Dabei bezeichnet `CDATA` das Vorkommen von Text in dem betreffenden Attribut; `#REQUIRED` besagt, dass das entsprechende Attribut im Dokument angegeben werden muss, `#IMPLIED`, dass kein Default-Wert existiert:

```
<!ELEMENT movie ( scifi | action | kungfu )>
<!ELEMENT scifi ( bibliography, stuntman+, guru, maiden )>
<!ELEMENT stuntman EMPTY>
<!ATTLIST movie
    title CDATA #REQUIRED
    timing CDATA #IMPLIED>
<!ATTLIST stuntman
    name CDATA #REQUIRED>
```

2.1.2 XMLSchema

Der von der DTD bereitgestellte Spezifikationsmechanismus ist für zahlreiche Anwendungen zu stark eingeschränkt: beispielsweise darf jedes Element global nur einmal deklariert werden – es ist also nicht möglich, den Unterbaum je nach Kontext verschieden zu gestalten. Zudem lässt DTD keine nähere Spezifikation von Datentypen zu. Diese Problematik hat dazu geführt, dass zahlreiche Ansätze für Format-Spezifikations-Formalismen entwickelt und diskutiert wurden [LC00a, LMM00, Mur00]. Favorit dürfte dabei XMLSchema sein [W3C01a, W3C01b, W3C01c], weil es einerseits sehr umfassend ist (der mächtigste Formalismus der in [LC00a] diskutierten) und andererseits von einer anerkannten Autorität vertreten wird. In der Tat ist die Spezifikation von XMLSchema äußerst komplex [Alt00], und auch ein Versuch der mathematischen Formalisierung [BFRW01] dieses Standards ist nicht vollständig.

XMLSchema verwendet eine XML-basierte Notation. Es ermöglicht, Attribute oder textuellen Elementinhalt mit einem Datentyp (Datums-, Zahlen-, Zeichenketten-Typen etc.) zu versehen. Es lassen sich zudem Datentypen vom Nutzer selbst deklarieren. Hierbei wird das aus der objektorientierten Programmierung entlehnte Konzept der Vererbung verwendet.

Auch für strukturierten Elementinhalt (das heißt, wenn ein Element Unter-Elemente besitzt), lässt sich für die Struktur dieses Unterbaumes explizit ein Datentyp definieren. Zudem ist es möglich, den Typ lokal zu spezifizieren. Auf diese Weise wird es möglich, je nach Kontext verschiedene Typen mit gleich benannten Elementen zu verbinden.

Eine weitere Eigenschaft ist, dass XMLSchema die Deklaration eines Default- bzw. fixierten Wertes nicht nur (wie DTD) für Attribute, sondern auch für Elemente zulässt. Diese Eigenschaft wird in dieser Arbeit eine wichtige Rolle spielen.

Nun ein Beispiel für eine Syntax-Spezifikation mit XMLSchema. Es werden hier dieselben Elemente mit denselben Attributen wie im DTD-Beispiel definiert. Es fällt auf, dass die XMLSchema-Spezifikation gegenüber der DTD weniger kompakt ist. Außerdem wird XMLSchema selbst in einer XML-Sprache notiert (DTD verwendete eine spezielle Notation). Ein Unterschied ist hier, dass `scifi` lokal innerhalb von `movie` deklariert wird. Auch die Attribute werden innerhalb des sie enthaltenden Elements deklariert. `action` und `kungfu` werden (dieses ist dann äquivalent zur DTD) per Referenz auf eine globale Deklaration eingefügt. Ein weiterer Unterschied zeigt sich bei der Verwendung von `stuntman` in `scifi`: Anstelle des „`+`“-Quantors wird in XMLSchema die mögliche Vorkommensanzahl durch eine untere („`minOccurs`“) und obere Grenze („`maxOccurs`“) angegeben. Default-Wert ist jeweils Eins. Der Wert „`unbounded`“ besagt, dass die Anzahl nach oben hin nicht begrenzt ist.

```
<element name="movie">
  <complexType>
    <choice>
      <element name="scifi">
        <complexType>
          <sequence>
            <element ref="bibliography"/>
```

```
            <element ref="stuntman" maxOccurs="unbounded"/>
            <element ref="guru"/>
            <element ref="maiden"/>
          </sequence>
        </complexType>
      </element>
      <element ref="action"/>
      <element ref="kungfu"/>
    </choice>
    <attribute name="title" type="string" use="required"/>
    <attribute name="timing" type="string"/>
  </complexType>
</element>
<element name="stuntman">
  <complexType>
    <attribute name="name" type="string" use="required"/>
  </complexType>
</element>
```

2.1.3 Verkürzte Format-Notation

Sowohl DTD als auch XMLSchema sind relativ unübersichtlich und lang. Im Weiteren wird daher folgende kurze Notation zur Spezifikation von Formaten verwendet:

```
ElementId → Attrs Content
```

Hierbei bedeuten

- `ElementId` Der Name des betreffenden Elements. Bei lokalen Definitionen ist der Name des Elters hier enthalten, beispielsweise `movie/scifi`. Der konkrete `Tag Name` des Elements ergibt sich aus dem letzten Bestandteil der `ElementId`.
- `Attrs` ist die Menge der Attributdeklarationen. Eine Attributdeklaration hat dabei folgenden Aufbau:

  ```
  <Attribut: Use>
  ```

  `Attribut` ist dabei der Name des Attributes, `Use` entweder `required` oder `implied`.
- `Content` ist ein regulärer Ausdruck, der die Struktur der Unterelemente des Elements bestimmt. Im Gegensatz zu XMLSchema werden hier wieder Quantoren anstelle von unterer und oberer Grenze der Vorkommensanzahl verwendet.

Diese Notation ist nicht vollständig (beispielsweise wird der Attributtyp hier ignoriert und als Text angenommen), deckt aber die in den Beispielen in dieser Arbeit verwendeten Eigenschaften ab. Für die Formalisierung wird eine vollständigere (allerdings auch aufwändigere) Notation verwendet. Das folgende Beispiel ist die Übertragung des XMLSchema-Beispiels in diese kurze Notation:

```
movie →{(title: required), (timing: implied)} ( movie/scifi | action
| kungfu )
movie/scifi →{} ( bibliography stuntman+ guru maiden )
stuntman →{(name: required)} ( ϵ )
```

2.2 Begriffsdefinitionen

Zusammenfassend sollen im Folgenden die weiterhin benötigten Begriffe definiert werden. Begonnen werden soll mit den regulären Ausdrücken, die in den betrachteten Formalismen zur Syntax-Spezifikation benutzt werden:

Definition 2.1 (Regulärer Ausdruck über Σ) *Sei Σ eine Menge von Symbolen, die die Basis für den regulären Ausdruck bilden.*

- *ϵ ist ein regulärer Ausdruck, der das leere Wort erzeugt.*
- *$a \in \Sigma$ ist ein regulärer Ausdruck, der das aus dem Symbol a bestehende Wort erzeugt.*
- *$E*$ ist ein regulärer Ausdruck, der kein Vorkommen oder die beliebige Wiederholung der von E erzeugten Wörter bezeichnet. E ist dabei ein regulärer Ausdruck.*
- *$(E|F)$ ist ein regulärer Ausdruck, der die Disjunktion der regulären Ausdrücke E und F bezeichnet. Das bedeutet, ein Wort wird von $(E|F)$ erzeugt, wenn es von E, F oder von beiden erzeugt wird.*
- *$E\,F$ ist ein regulärer Ausdruck, der die Konkatenation (Aneinanderreihung) von Wörtern aus E und F in genau dieser Reihenfolge bezeichnet. E und F sind reguläre Ausdrücke.*
- *(E) ist ein regulärer Ausdruck, der dem regulären Ausdruck E entspricht (Gruppierung).*

Beispielsweise erzeugt der reguläre Ausdruck $(a\,b)*$ Folgen von sich abwechselnden as und bs. Das Wörter $acba$, $acccda$ und aba werden alle von dem regulären Ausdruck $a\,c*\,(b|d)\,a$ erzeugt.

Obige Definition spezifiziert auch die Basis-Notation für reguläre Ausdrücke. Ein regulärer Ausdruck, der nur mit den Operatoren bzw. Symbolen ϵ, $(|)$, $*$, Konkatenation, Elementen aus Σ und Klammern notiert wird, wird im Folgenden „regulärer Ausdruck in Basis-Notation" genannt. Es gibt für reguläre Ausdrücke verschiedene verkürzende Notationen. Sie erweitern die Basisdefinition dann um entsprechende Operatoren. Wie aus den angegebenen Äquivalenzen deutlich wird, lassen sich alle diese verkürzten Notationen auch (mit entsprechendem Aufwand) in der Basis-Notation spezifizieren. Das bedeutet, dass die zusätzlichen Operatoren die Ausdrucksstärke der regulären Ausdrücke nicht ändern.

Zunächst wird die in DTDs verwendete Notation vorgestellt:

Definition 2.2 (Regulärer Ausdruck über Σ in DTD-Notation) *Reguläre Ausdrücke in DTD-Notation sind als Erweiterung von regulären Ausdrücken in Basis-Notation definiert:*

- *Jeder reguläre Ausdruck über Σ in der Basis-Notation ist ein regulärer Ausdruck über Σ in DTD-Notation.*
- *$E?$ ist ein regulärer Ausdruck in DTD-Notation, der das optionale Auftreten des regulären Ausdrucks in DTD-Notation E bezeichnet: $E? \equiv (E|\epsilon)$.*
- *$E+$ ist ein regulärer Ausdruck in DTD-Notation, der einmalige oder öftere Wiederholung des regulären Ausdrucks in DTD-Notation E bezeichnet: $E+ \equiv E\,E*$.*

In XMLSchema wird die Notation gegenüber DTDs nochmals erweitert:

Definition 2.3 (Regulärer Ausdruck über Σ in XMLSchema-Notation) *Reguläre Ausdrücke in XMLSchema-Notation sind als Erweiterung von regulären Ausdrücken in DTD-Notation definiert:*

- *Jeder reguläre Ausdruck über Σ in DTD-Notation ist ein regulärer Ausdruck über Σ in XMLSchema-Notation.*
- *$E_1 \& \cdots \& E_j$ ist ein regulärer Ausdruck in XMLSchema-Notation, der die sogenannte Konjunktion der regulären Ausdrücke in XMLSchema-Notation $E_1, \ldots, E_j$ bezeichnet. „Konjunktion" bedeutet die Konkatenation von Wörtern aus $E_1, \ldots, E_j$ in beliebiger Reihenfolge (siehe auch Abschnitt 3.3.3, S. 30).*

Für die Umwandlung eines Attributs in ein Element und umgekehrt ist ein im Folgenden „Silbe" genannter Teilausdruck von Bedeutung:

Definition 2.4 (Silbe) *Eine* Silbe *eines regulären Ausdrucks oder eines daraus abgeleiteten Ausdrucks E ist ein Teilausdruck $E_i, i \in \{1..j\}$, falls E eine Sequenz von Ausdrücken $E_1 \cdots E_j$ ist, oder ansonsten E selbst.*

Definition 2.5 (Phrase) *Ein Teil eines regulären Ausdrucks oder eines daraus abgeleiteten Ausdrucks wird als* Phrase *bezeichnet.*

Definition 2.6 (Unterstruktur-Typ (*content model*)) *Der Teil einer Typdeklaration eines XML-Elementes, der die Anforderungen an die Syntax der Liste der Kind-Elemente des betreffenden Elementes formuliert, heißt* Unterstruktur-Typ*.*

2.3 XML-Transformationen

In diesem Abschnitt sollen verschiedene bestehende Ansätze zur XML-Transformation vorgestellt werden.

2.3.1 XSLT

Die *Extensible Stylesheet Language* XSL besteht aus drei Teilen: XSL Transformations (XSLT) [W3C99c], XML Path Language XPath [W3C99b] sowie den XML Formatting Objects. Ihr ursprünglicher Zweck liegt darin, XML-Dokumente für die Ausgabe zu formatieren. Ihr Transformationsteil XSLT (zusammen mit dem von XSLT benutzten XPath) ist die wahrscheinlich am häufigsten verwendete Transformationssprache für XML und vermutlich auch umgekehrt der am häufigsten benutzte Teil von XSL. Da zahlreiche Implementationen nur XSLT implementieren, jedoch nicht die Formatting Objects, gibt es die Diskussion, den Transformationsteil anders zu benennen und aus der XSL-Spezifikation auszulagern [Pre01].

In [BMN00, MN00] wird ein Teil von XSLT formalisiert. Die Autoren stellen dabei fest, dass XSLT vielen anderen XML-Anfrage- und Transformations-Mechanismen an Ausdrucksstärke überlegen ist.

XSLT ermöglicht es, bestimmte Aktionen auf Elementen des XML-Dokuments auszuführen. Die Elemente lassen sich durch entsprechende Muster auswählen und dann modifizieren. Weiterhin ist das gezielte Traversieren ihres Unterbaumes möglich. Hinzu kommen die Möglichkeit, Variablen zu deklarieren. Dadurch lassen sich sogenannte `Joins` durchführen, das bedeutet zwei verschiedene Elemente über die Werte ihrer Attribute bzw. Unterelemente in Beziehung setzen. Zudem ist es dadurch, dass bei der Verarbeitung verschiedene Modi verwendet werden können, möglich bei verschiedenen Durchläufen durch den Baum auf denselben Elementen verschiedene Aktionen auszuführen [BMN00].

Auch XSLT verwendet zu seiner Notation eine XML-basierte Sprache.

Das folgende Beispiel zeigt zwei Regeln („*Templates*“, die bei zur Transformation der Filmdatenbank nach HTML verwendet werden. Die erste Regel transformiert den Wurzelknoten der Datenbank (`match"/"`), indem sie den Rahmen für das Ausgabedokument erzeugt, nämlich die `<html>`, `<body>` und `<table>`-Tags. Die `<xsl:apply-templates/>`-Anweisung bedeutet, dass an dieser Stelle der Unterbaum des Knotens (hier also der Wurzel) traversiert werden soll.

Die zweite Regel erzeugt für jeden `<movie>`-Eintrag eine Tabellenzeile in der Ausgabetabelle. An geeigneter Stelle wird dabei mittels `<xsl:value-of/>` der Wert von `title`- und `timing`-Attribut eingefügt. Auch hier wird durch das dann folgende `<xsl:apply-templates/>` die Verarbeitung der Unterelemente des `<movie>`-Knotens ausgelöst. Diese geschieht dann mit weiteren entsprechenden Regeln, die hier nicht mehr dargestellt sind.

```
<xsl:template match="/">
  <html>
    <body>
      <table>
        <xsl:apply-templates/>
      </table>
    </body>
  </html>
</xsl:template>

<xsl:template match="movie">
  <tr>
    <td colspan="2">
      <b><xsl:value-of select="@title"/></b>
      (<xsl:value-of select="@timing"/>)
    </td>
  </tr>
  <xsl:apply-templates/>
</xsl:template>
```

Das Ergebnis der Transformation ist dann der folgende HTML-Code:

```
<html>
  <body>
    <table>
      <tr>
        <td colspan="2">
          <b>Vector Space</b>
          (110min)
        </td>
      </tr>
      <tr>
        <td>Bibliography</td>
        <td>&quot;Visual Basic for construction workers&quot; (Book)</td>
      </tr>
      <tr>
        <td>Stuntman</td>
        <td>W. Miller</td>
      </tr>
        <td>Guru</td>
        <td>Soda-Largo</td>
      </tr>
        <td>Maiden</td>
        <td>Enginarella</td>
      </tr
```

```
    </table>
  </body>
</html>
```

2.3.2 YaTL

YaTL [CS00] dient der Abfrage und Transformation von XML mittels einem Ansatz, der auf funktionaler Programmierung beruht. Elemente werden anhand eines Suchmusters ausgewählt und können dann noch, je nach dem Muster, dem sie entsprechen, verschieden behandelt werden. Da es möglich ist, Funktionen zu deklarieren, kann bei der Transformation von Bäumen auch Rekursion verwendet werden. Obwohl YaTL keine Typisierung der XML-Daten unterstützt, können in den Transformationen und Anfragen Datentypen der zugrundeliegenden funktionalen Sprache (Object Caml) wie boolesche Werte, Zahlen etc. verwendet werden.

Ein XML-Element

```
<name attr="value"> <sub1/> ... <subn/> </name>
```

wird in YaTL

```
name[ @attr[ "value" ], sub1, ..., subn ]
```

notiert. Diese Notation wird in ähnlicher Form auch in XDuce verwendet (siehe Abschnitt 2.3.3, S. 15), das ebenfalls auf einer funktionalen Programmiersprache basiert.

Auffällig ist die Notation von Attributen, die lediglich durch das „`@`“-Präfix von Unterelementen unterschieden werden.

Der Filmdatenbank-Eintrag sieht in dieser Notation wie folgt aus:

```
movie[ @title[ "Vector Space" ], @timing[ "110min" ],
  scifi[
    bibliography[
      book[ @title[ "Visual Basic for construction workers" ] ]
    ],
    stuntman[ @name[ "W. Miller" ] ],
    guru[ @name[ "Soda-Largo" ] ],
    maiden[ @name[ "Enginarella" ] ]
  ]
]
```

Das folgende Transformation gibt einen XML-Baum als eine Hierarchie von Listen nach HTML aus. Der `*`-Operator steht dabei für eine Liste von Elementen, `$` definiert eine neue Variable. Die `concat()`-Funktion konkateniert die ihr übergebenen Zeichenketten. In dem Beispiel wird also ein Knoten als Parameter in der Variablen `T` übergeben und dann in der `case`-Anweisung analysiert. Wenn sein Unterbaum leer ist, er also die Form `Atom[]` hat, wird einfach der Name (der an die Variable `Atom` zugewiesen wurde) ausgegeben. Ansonsten wird die zweite Alternative benutzt. Diese gibt zunächst den Namen des Knoten, gefolgt von einem Doppelpunkt, aus, und erzeugt dann eine Liste der Unterelemente dieses Knotens. Die einzelnen Unterelemente (in der Variablen `Child` gespeichert) werden dann wiederum durch rekursiven Aufruf von `html()` transformiert.

```
define html($T)=
  case T of
  | $Atom[]
    -> Atom
  | $Atom[ * $Child ]
```

```
    -> [ concat( Atom, " :" ),
         ul[ * li[ html( Child ) ] ]
       ]
```

Die Anwendung der Funktion `html()` auf den `stuntman`-Eintrag ergibt dann folgenden Term:

```
[
  "stuntman :",
  ul[
    li[
      "@name :",
      ul[ li[ "W. Miller" ] ]
    ]
  ]
]
```

Das entspricht in XML-Notation

```
stuntman :
<ul>
  <li>
    @name :
    <ul> <li> W. Miller </li> </ul>
  </li>
</ul>
```

2.3.3 XDuce

XDuce [HP00] basiert wie YaTL auf einer funktionalen Programmiersprache. Im Gegensatz zu YaTL lässt sich hier jedoch der Typ mittels einem regulären Ausdruck spezifizieren. Es wird zudem eine Untertyp-Relation benutzt, beispielsweise sind `X` und ϵ Untertypen (Spezialisierungen) von `X?`. Umgekehrt ist beispielsweise `(Y|A)` ein Obertyp (eine Verallgemeinerung) von `Y`.

Das folgende Beispiel demonstriert die Typvereinbarung mit XDuce. Wie auch bei YaTL wird ein Element durch `name[ sub1, ..., subn ]` repräsentiert. Der Inhalt von `Addrbook`-Elementen wird mit einer Art regulärem Ausdruck restriktiviert:

```
type Addrbook = addrbook[ ( Name, Addr, Tel? )* ]
type Name = name[ String ]
```

Im folgenden noch ein Beispiel für *Pattern Matching* mit XDuce. Auch hier werden die Funktionsparameter typisiert.

```
fun mkTelList : ( Name, Addr, Tel? )* -> ( Name, Tel )* =
    name[ n:String ], addr[ a:String ], tel[ t:String ],
        rest:( Name, Addr, Tel? )*
      -> name[ n ], tel[ t ], mkTelList( rest )
  | name[ n:String ], addr[ a:String ], rest:( Name, Addr, Tel? )*
      -> mkTelList( rest )
  | ()
      -> ()
```

Analog zu dem Beispiel in YaTL werden hier durch Muster verschiedene Alternativen identifiziert. Die erste Alternative behandelt den Fall, dass eine Telefonnummer vorhanden ist und gibt dann den Namen, die Telefonnummer und das Ergebnis der rekursiven Verarbeitung der Restliste zurück. Falls keine Telefonnummer vorhanden ist, gibt die zweite Alternative nur das Verarbeitungsergebnis der Restliste zurück. Die dritte Alternative behandelt die leere Liste.

2.3.4 XML Script

XML Script [Dec00] basiert auf imperativen Programmiersprachen. Der Nutzer definiert eine Vorlage, die die Ziel-XML-Datei beschreibt sowie Verarbeitungsanweisungen (Name der Quell-Datei, Iteration über Quell-Elemente etc.) enthält. An den Stellen, an denen Elemente des Quell-Dokuments eingefügt werden sollen, können entsprechende Referenzen spezifiziert werden. Die Ausgabe einer Anweisung kann von weiteren Anweisungen weiterverarbeitet werden. Zudem ist es möglich, Variablen zu deklarieren (diese werden in derselben Weise wie auch die Elemente des XML-Dokumentes adressiert). Das folgende Beispiel deklariert zunächst die Konstanten `IssueMonth`, ..., `Maturity` und erzeugt dann eine Liste von `<PaymentDate/>`-Elementen. Hierzu wird mit dem `<_for/>`-*Tag* eine Schleife programmiert:

```
<XST __maxCR="1" __XMLscript='1.1' >

  # \.IssueMonth := "April" #
  # \.IssueDay := "15" #
  # \.IssueYear := "2003" #
  # \.Maturity := "10" #

  <FixedRateBond Coupon="5.5" Currency="USD" >
    <_for from="1" to="\.Maturity" >
      <PaymentDate ># \.IssueDay # # \.IssueMonth #
                    # \.IssueYear + $!index #</PaymentDate>
    </_for>
  </FixedRateBond>

</XST>
```

Dieses Script erzeugt daraus die folgende Ausgabe:

```
<FixedRateBond Coupon="3.5" Currency="USD" >
        <PaymentDate >15 April 2004</PaymentDate>
        <PaymentDate >15 April 2005</PaymentDate>
        <PaymentDate >15 April 2006</PaymentDate>
        <PaymentDate >15 April 2007</PaymentDate>
        <PaymentDate >15 April 2008</PaymentDate>
        <PaymentDate >15 April 2009</PaymentDate>
        <PaymentDate >15 April 2010</PaymentDate>
        <PaymentDate >15 April 2011</PaymentDate>
        <PaymentDate >15 April 2012</PaymentDate>
        <PaymentDate >15 April 2013</PaymentDate>
</FixedRateBond>
```

Kapitel 3

Operationen

Der in dieser Arbeit verfolgte Ansatz ist, die Format-Evolution mit einer Menge von gewissen Operationen durchzuführen. Diese Operationen sind so formuliert, dass sie meist Änderungen auf dem Format spezifizieren. Daraus werden dann auch die Dokument-Transformationen abgeleitet.

In diesem Kapitel sollen diese Operationen zur Format-Evolution vorgestellt werden. Das Kapitel beginnt zunächst mit einem Beispiel für die Format-Evolution einer Filmdatenbank. Darauf folgt ein Überblick über die über die Operationsmenge. Nach der Diskussion verschiedener Aspekte des Designs werden die Operationen formalisiert. Das Kapitel schließt mit der Beschreibung der Umsetzung der Transformationen für Format und Dokumente.

3.1 Beispiel

Zur Notation von Formaten wird im Folgenden die Abschnitt 2.1.3, S. 10 definierte vereinfachte Notation verwendet.

Das folgende Beispiel führt umfangreiche Änderungen auf der bereits eingeführten Filmdatenbank durch. Die im Folgenden verwendete Datenbank unterscheidet sich in einigen Details von den vorigen Beispielen (beispielsweise existiert hier kein `timing`-Attribut im `movie`-Element). Dafür werden jetzt weitere Einträge hinzugenommen. Zunächst ein Ausschnitt der Format-Spezifikation dieser Datenbank (die Definitionen von Elementen, die lediglich Text enthalten, wurden weggelassen):

```
movie → {(title: required)} ( action | scifi | kungfu )
action → {(bodycount: required)} carchase* stuntman+ maiden
kungfu → {} shout+ stuntperson+ maiden?
scifi → {} bibliography stuntman+ guru maiden
stuntperson → {(name: required), (sex: required)} ( ϵ )
```

Das für dieses Beispiel verwendete Dokument dieser Filmdatenbank enthält folgende Einträge:

```
<movie title="Vector Space">
  <scifi>
    <bibliography>
      <book title="Visual Basic for construction workers"/>
    </bibliography>
    <stuntman name="W. Miller"/>
    <guru name="Soda-Largo"/>
    <maiden name="Enginarella"/>
  </scifi>
</movie>
```

```
<movie title="Steel-Fist Dragon">
  <kungfu>
    <shout soundslike="yeeee-ha! ha!"/>
    <stuntperson name="W. Lee" sex="female"/>
    <stuntperson name="L. Chu" sex="male"/>
  </kungfu>
</movie>

<movie title="Gefahr im Verzug">
  <action bodycount="3.14">
    <carchase duration="50min">
      <vehicle type="Tank" crashed="yes"/>
    </carchase>
    <stuntman name="K.-H. Schulze"/>
    <maiden name="Hilderose"/>
  </action>
</movie>
```

Diese Datenbank soll nun renoviert werden, da sich das bisherige Format als unzureichend herausgestellt hat und viele Felder in allen Alternativen vorkommen. Beispielsweise soll ein französischer Action-Film in die Datenbank aufgenommen werden, der keine Figur enthält, für die das `maiden`-Feld adäquat wäre. Außerdem sollen die Stunt-Leute und die `maiden`-Felder in den `movie`-Eintrag verschoben werden. Das `guru`-Feld soll entfernt werden, weil es von den Benutzern der Datenbank niemals abgefragt wurde. Das angestrebte neue Format ist dann das folgende:

```
movie → {(title: required)} ( action_ | scifi_ | kungfu_ ) stuntperson+
maiden?
action_ → {(bodycount: required)} carchase*
scifi_ → {} bibliography
kungfu_ → {} shout+
stuntperson → {(name: required), (sex: required)} ( ϵ )
```

Die Renovierung beginnt damit, dass die obligatorischen `maiden`-Felder in den `action`- und `scifi`-Elementen optional gemacht werden. Dies wird mit einer speziellen Verallgemeinerungs-Operation durchgeführt:

```
generalize maiden to maiden? in {action, scifi}
```

Der Teil hinter „`in`" spezifiziert den Fokus der Transformation, also die Elemente, mit denen die Transformation durchgeführt werden soll.

Die Verallgemeinerung bedeutet, dass das geänderte Format nachwievor die existierenden Dokumente akzeptiert (daher ist in diesem Fall keine Dokument-Transformation nötig), jedoch nun auch zusätzlich Dokumente, die dem ursprünglichen Format nicht entsprechen. Eine Definition der Verallgemeinerung wird in Abschnitt 3.3.2, S. 28 vorgenommen.

Zahlreiche der vorgeschlagenen Operationen sind in dieser Art definiert, nämlich als Phrasenersetzung zu einem bestimmten Zweck. Im Gegensatz zu einer uneingeschränkten Phrasenersetzung ermöglicht dieser Stil neben Plausibilitätsüberprüfungen auch eine klarere Formalisierung. Auch die Parallelität von Format- und Dokument-Transformation wird so besser nachvollziehbar.

Nach dieser Ersetzung sind `action` und `scifi` nun wie folgt definiert:

```
action → {(bodycount: required)} carchase* stuntman+ maiden?
scifi → {} bibliography stuntman+ guru maiden?
```

Als nächstes soll das `guru`-Feld gelöscht werden. Hierzu wird eine Lösch-Funktion mit der `editData`-Operation auf die betreffenden Stellen des Dokumentes angewendet. `editData` modifiziert das Dokument, allerdings muss hierbei das Format erhalten bleiben.

Das bedeutet konkret, dass das Format sowohl das Dokument in seinem Zustand *vor* dem Löschen der betreffenden Felder akzeptieren muss, als auch in dem Zustand danach. In dem Filmdatenbank-Beispiel ist das noch nicht der Fall. Es muss daher zunächst entsprechend erweitert werden. Dies geschieht, indem das `guru`-Feld (wieder mittels Generalisierung) optional gemacht wird.

Nachdem die Einträge dann aus dem Dokument entfernt wurden, kann das optionale Feld nun auf „kein Vorkommen“ eingeschränkt werden. Damit ist `guru` nun sowohl aus den Dokumenten als auch aus dem Format entfernt.

Zusammenfassend sind also die entsprechenden Operationen mit ihren jeweiligen Auswirkungen folgende (die Operationen werden durch ein vorangestelltes ▷ gekennzeichnet):

```
▷ generalize guru to guru? in {scifi}

scifi → {} bibliography stuntman+ guru? maiden?

▷ editData guru? by set([]) in {scifi}


<scifi>
  <bibliography>
    <book title="Visual Basic for construction workers"/>
  </bibliography>
  <stuntman name="W. Miller"/>
  <maiden name="Enginarella"/>
</scifi>

▷ restrict guru? to ϵ in {scifi}

scifi → {} bibliography stuntman+ maiden?
```

Als nächstes wird die Deklaration von `stuntman` so erweitert, dass sie zu `stuntperson` kompatibel wird, damit die beiden Definitionen dann zusammengefasst werden können. Dazu ist in `stuntman` ein neues Attribut `sex` erforderlich, dessen Werte im Dokument für alle `stuntman`-Einträge auf „`male`“ gesetzt werden. Da keine speziellen Operationen zur Manipulation von Attributen existieren, wird zunächst ein entsprechendes Element erstellt und dann modifiziert, um es dann schließlich in ein Attribut zu konvertieren. Zum Einfügen dieses `sex`-Elementes wird `generalize` verwendet – analog zum Löschen von Elementen mittels `restrict`.

Zum Setzen der „`male`“-Werte im Dokument kommt dann wieder `editData` zum Einsatz. Da das `sex`-Attribut später obligatorisch sein soll, muss auch das Element auf ein notwendiges Vorkommen restriktiviert werden (dieser Transformation erfordert allerdings, dass auch im Dokument an jeder Stelle ein Wert für `sex` existiert). Schließlich konvertiert `element2attr` das `sex`-Element in ein Attribut.

Zusammenfassend nun die Operationen für diese Transformation:

```
▷ generalize ϵ to sex? in {stuntman}

stuntman → {(name:required)} sex?

▷ editData sex? by set([sex["male"]]) in {stuntman}
```

```
<stuntman name="W. Miller"> <sex>male</sex> </stuntman>
<stuntman name="K.-H. Schulze"> <sex>male</sex> </stuntman>
```

```
▷ restrict sex? to sex in {stuntman}

stuntman → {(name:required)} sex
```

```
▷ element2attr in {stuntman}
stuntman → {(name:required), (sex:required)} ( ε )
```

```
<stuntman name="W. Miller" sex="male"/>
<stuntman name="K.-H. Schulze" sex="male"/>
```

Nun sind die Deklarationen von `stuntman` und `stuntperson` identisch, so dass die `stuntman`-Einträge in `stuntperson` umgewandelt werden können. Dies geschieht mit der Operation `cast`:

```
▷ cast stuntman to stuntperson in {action, scifi}

action → {(bodycount: required)} carchase* stuntperson+ maiden?
scifi → {} bibliography stuntperson+ maiden?
```

```
<stuntperson name="W. Miller" sex="male"/>
<stuntperson name="K.-H. Schulze" sex="male"/>
```

Da `stuntman` nun nicht mehr verwendet wird, kann seine Deklaration aus dem Format entfernt werden (eine Dokumentmodifikation ist nicht mehr nötig, da bereits alle Vorkommen von `stuntman` in `stuntperson` umgewandelt wurden):

```
eliminate stuntman
```

Der Status von Format und Dokument ist nun dieser:

```
movie → {(title: required)} ( action | scifi | kungfu )
action → {(bodycount: required)} carchase* stuntperson+ maiden?
kungfu → {} shout+ stuntperson+ maiden?
scifi → {} bibliography stuntperson+ maiden?
stuntperson → {(name: required), (sex: required)} ( ε )
```

```
<movie title="Vector Space">
  <scifi>
    <bibliography>
      <book title="Visual Basic for construction workers"/>
    </bibliography>
    <stuntperson name="W. Miller" sex="male"/>
    <maiden name="Enginarella"/>
  </scifi>
</movie>

<movie title="Steel-Fist Dragon">
  <kungfu>
    <shout soundslike="yeeee-ha! ha!"/>
    <stuntperson name="W. Lee" sex="female"/>
    <stuntperson name="L. Chu" sex="male"/>
  </kungfu>
</movie>

<movie title="Gefahr im Verzug">
  <action bodycount="3.14">
```

```
      <carchase duration="50min">
        <vehicle type="Tank" crashed="yes"/>
      </carchase>
      <stuntperson name="K.-H. Schulze" sex="male"/>
      <maiden name="Hilderose"/>
    </action>
  </movie>
```

Nun sollen die gemeinsamen Elemente in das `movie`-Element heraufgezogen werden. Dazu muss der Inhalt der `action`, `kungfu` und `scifi`-Elemente direkt im `movie`-Element ausgepackt werden. Da die entsprechenden Elemente, darunter auch `action`, aufgelöst werden, muss dessen `bodycount`-Attribut „gerettet“ werden, indem es in ein Element umgewandelt wird. Diese beiden Transformationen werden wie folgt notiert:

```
▷ attr2element bodycount in action

action → {} action/bodycount carchase* stuntperson+ maiden?
action/bodycount → {} τ

<action>
  <bodycount>3.14</bodycount>
  <carchase duration="50min">
    <vehicle type="Tank" crashed="yes"/>
  </carchase>
  <stuntperson name="K.-H. Schulze" sex="male"/>
  <maiden name="Hilderose"/>
</action>

▷ unfold action in {movie}

movie → {(title: required)}
       ( action/bodycount carchase* stuntperson+ maiden?
       | scifi | kungfu )

<movie title="Gefahr im Verzug">
  <bodycount>3.14</bodycount>

  <carchase duration="50min">
    <vehicle type="Tank" crashed="yes"/>
  </carchase>
  <stuntperson name="K.-H. Schulze" sex="male"/>
  <maiden name="Hilderose"/>
</movie>

▷ unfold kungfu in {movie}

movie → {(title: required)}
       ( action/bodycount carchase* stuntperson+ maiden?
       | scifi
       | shout+ stuntperson+ maiden? )
scifi → {} bibliography stuntperson+ maiden?

<movie title="Steel-Fist Dragon">
  <shout soundslike="yeeee-ha! ha!"/>
  <stuntperson name="W. Lee" sex="female"/>
  <stuntperson name="L. Chu" sex="male"/>
</movie>

▷ unfold scifi in {movie}
movie → {(title: required)}
```

```
       ( action/bodycount carchase* stuntperson+ maiden?
       | bibliography stuntperson+ maiden?
       | shout+ stuntperson+ maiden? )

<movie title="Vector Space">
  <bibliography>
    <book title="Visual Basic for construction workers"/>
  </bibliography>
  <stuntperson name="W. Miller" sex="male"/>
  <maiden name="Enginarella"/>
</movie>
```

Da `action`, `kungfu` und `scifi` nicht mehr benötigt werden, werden ihre Deklarationen nun gelöscht.

```
eliminate action
eliminate kungfu
eliminate scifi
```

Nun werden Teile, die in den Zweigen der Disjunktion jeweils verschieden sind, in die Elemente `action_`, `kungfu_` und `scifi_` gefaltet und `bodycount` wieder in ein Attribut umgewandelt (seine Elementdeklaration wird dann nicht mehr benötigt):

```
▷ fold (action/bodycount carchase*) to action_ in {movie}

movie → {(title: required)}
       ( action_ stuntperson+ maiden?
       | bibliography stuntperson+ maiden?
       | shout+ stuntperson+ maiden? )
action_ → {} action/bodycount carchase*

<movie title="Gefahr im Verzug">
  <action_>
    <bodycount>3.14</bodycount>
    <carchase duration="50min">
      <vehicle type="Tank" crashed="yes"/>
    </carchase>
  </action_>
  <stuntperson name="K.-H. Schulze" sex="male"/>
  <maiden name="Hilderose"/>
</movie>

▷ fold bibliography to scifi_ in {movie}

movie → {(title: required)}
s        ( action_ stuntperson+ maiden?
       | scifi_ stuntperson+ maiden?
       | shout+ stuntperson+ maiden? )
scifi_ → {} bibliography

<movie title="Vector Space">
  <scifi_>
    <bibliography>
      <book title="Visual Basic for construction workers"/>
    </bibliography>
  <scifi_>
  <stuntperson name="W. Miller" sex="male"/>
  <maiden name="Enginarella"/>
</movie>
```

▷ fold shout+ to kungfu_ in {movie}

```
movie → {(title: required)}

       ( action_ stuntperson+ maiden?
       | scifi_ stuntperson+ maiden?
       | kungfu_ stuntperson+ maiden? )
kungfu_ → {} shout+
```

```
<movie title="Steel-Fist Dragon">
  <kungfu>
    <shout soundslike="yeeee-ha! ha!"/>
  </kungfu>
  <stuntperson name="W. Lee" sex="female"/>
  <stuntperson name="L. Chu" sex="male"/>
</movie>
```

▷ element2attr in {action_}

```
action_ → {(bodycount:required)} carchase*
```

```
<action_ bodycount="3.14">
  <carchase duration="50min">
    <vehicle type="Tank" crashed="yes"/>
  </carchase>
</action_>
```

▷ eliminate action/bodycount

Die gemeinsamen Elemente können nun aus der Auswahl herausgezogen werden. Zu beachten ist hier, dass bei `extract` im Gegensatz zu den anderen Operationen nicht die momentan existierende Phrase, sondern das gewünschte Ergebnis der Transformation angegeben wird. Dies vereinfacht in diesem Fall die Spezifikation (eine nähere Erläuterung findet sich später bei dem Überblick über die Operationen). `extract` wurde so eingeschränkt, die Reihenfolge der Elemente bei dieser Extraktion erhalten bleiben muss. Deshalb ist dann keine Dokumentmodifikation mehr nötig.

▷ extract ((action_ | scifi | kungfu_) stuntperson+ maiden?) in {movie}

```
movie → {(title: required)} ( action_ | scifi_ | kungfu_ ) stuntperson+
maiden?
```

Damit ist die Umformung nun abgeschlossen. Die Ergebnisse sind:

```
movie → {(title: required)} ( action_ | scifi_ | kungfu_ ) stuntperson+
maiden?
action_ → {(bodycount: required)} carchase*
scifi_ → {} bibliography
kungfu_ → {} shout+
stuntperson → {(name: required), (sex: required)} ( ϵ )
```

```
<movie title="Vector Space">
  <scifi_>
    <bibliography>
      <book title="Visual Basic for construction workers"/>
    </bibliography>
  </scifi_>
  <stuntperson name="W. Miller" sex="male"/>
  <maiden name="Enginarella"/>
```

```
</movie>

<movie title="Steel-Fist Dragon">
  <kungfu_>
    <shout soundslike="yeeee-ha! ha!"/>
  </kungfu_>
  <stuntperson name="W. Lee" sex="female"/>
  <stuntperson name="L. Chu" sex="male"/>
</movie>

<movie title="Gefahr im Verzug">
  <action_ bodycount="3.14">
    <carchase duration="50min">
      <vehicle type="Tank" crashed="yes"/>
    </carchase>
  </action_>
  <stuntperson name="K.-H. Schulze" sex="male"/>
  <maiden name="Hilderose"/>
</movie>
```

3.2 Überblick Operationen

In diesem Abschnitt soll die Operationsmenge motiviert und vorgestellt werden. Die Auswahl der Operationen orientiert sich an [Läm01a] und [LL01]. Sie wurde unter folgenden Aspekten entwickelt:

Orthogonalität Die Operationen sollten atomar sein, das bedeutet, keine Funktionalität enthalten, die auch durch andere Operationen bereitgestellt wird.

Semantikerhaltung Diese Forderung kommt aus dem Programmiersprachenbereich. [LL01] diskutiert die Problematik, Semantikerhaltung für Transformationen auf Daten zu definieren. In diesem Zusammenhang wird der Begriff der Strukturerhaltung diskutiert. Auch diese Eigenschaft muss abgeschwächt werden, wenn Daten transformiert werden. Als Lösung wird schließlich die Umkehrbarkeit (Bijektivität) der Operationen vorgeschlagen. Aber von diesem Prinzip müssen natürlich (beispielsweise für löschende Operationen) Ausnahmen gemacht werden. Diesem Dilemma wird hier versucht zu begegnen, indem die unter Umständen nicht umkehrbaren Modifikationen in einer Operation, nämlich editData, konzentriert sind.

Gültige Ergebnisse Die Operationen sollen jede für sich gültige Ergebnisse liefern, so dass keine Bedingungen an eventuelle Nachfolgeoperationen gestellt werden. Ein Konzept wie die aus dem Datenbanken-Bereich bekannten Transaktionen, die ungültige Zwischenstände erlauben, solange das Endergebnis einer solchen Transaktion korrekt ist, ist nicht vorgesehen.

Die Operationsmenge enthält keine expliziten Operationen zur Attributmodifikation. Da in XMLSchema jedoch Elemente auch alle Informationen und Eigenschaften von Attributen haben können (siehe Abschnitt 3.7.2, S. 42), lassen sich Attributmodifikationen problemlos durchführen, indem das zu modifizierende Attribut mittels attr2element in ein Element konvertiert, dieses dann modifiziert und schließlich mit element2attr zurück gewandelt wird.

Die Stellen, an denen Ersetzungen durchgeführt werden sollen, können über einen Fokus spezifiziert werden. Dieser Fokus besteht aus einer Menge von Element-Identifikatoren (Z*). Diese Fokussierung ermöglicht eine präzise Kontrolle der Modifikationen, hat jedoch ungünstige Auswirkungen auf Identitäts-Constraints (siehe Abschnitt 3.8.2, S. 47).

Bei anderen Operationen ist ein Fokus nicht nötig, da sie auf das Format wirken (introduce, eliminate und rename).

Im folgenden soll die Operationsmenge zunächst informal vorgestellt werden. Eine formale Definition erfolgt später. In der Darstellung werden folgende Symbole benutzt:

- A, B, C: Phrasen.
- X, Y, Z, Z*: Referenzen auf Elemente. Z ist dabei ein fokussiertes Element, Z* eine Menge von Elementen, die Teil eines Fokus' sind. X und Y sind Elementreferenzen, die durch die entsprechende Operation transformiert werden sollen.
- L: Attributname.
- M: Dokument-Modifikator.

fold B to X in Z* Ersetzt die Vorkommen der Phrase B durch ein neues Element X. Der Inhalt von X wird als B deklariert.

unfold X in Z* Ersetzt das Vorkommen des Elementes X durch seinen Inhalt. Hier ist zu beachten, dass diese Operation nur durchgeführt werden kann, wenn X nicht fixiert ist, keine Attribute enthält und sein Default-Wert der leere String ist (Default- bzw. fixierter Wert sowie die Attribute würden ansonsten verloren gehen – was nicht semantikerhaltend ist).

permute BC to 2,1 in Z* Permutiert Folgen von Ausdrücken einer Sequenz. „2,1" gibt die neue Reihenfolge an: zweites Element (C) zuerst, dann das ursprünglich erste (B).

distribute (B|C)A in Z* Verschiebt die Phrase A in die Zweige der Auswahl (B|C). Um die Wohlgeformtheit des Ausdrucks zu erhalten, ist es erforderlich, dass A direkt auf die Auswahl folgt. Diese Bedingung führt auch dazu, dass keine Modifikation des Dokumentes nötig ist. Abschnitt 3.7.1, S. 41 diskutiert weitere Gründe der Einschränkung der Distribution.

extract (B|C)A in Z* Zieht die gemeinsame Phrase A aller Zweige der Auswahl aus dieser heraus. Dies ist die Umkehroperation von distribute. Auch hier ist keine Dokumentmodifikation nötig. Zu beachten ist hier, dass (B|C)A im Gegensatz zu den anderen Operationen nicht die tatsächlich vorkommende Phrase, sondern das gewünschte Ziel der Extraktion darstellt. Auf diese Weise lassen sich die zu extrahierenden Bestandteile recht einfach spezifizieren. Grundsätzlich *alle* gemeinsamen letzten Phrasen aus den Zweigen der Auswahl herauszuziehen, wäre unter Umständen unerwünscht: (BAC|DAC) $\rightarrow$ (BA|DA)C.

preserve B as C in Z* Ändert den Typ von A in den äquivalenten Typ B. Dies betrifft größtenteils die verkürzenden Notationen für reguläre Ausdrücke: $A? \equiv (A|\epsilon)$, $A\,A* \equiv A+$, $(A\&B) \equiv (AB \mid BA)$ etc.

generalize B to B' in Z* Verallgemeinert den Typ des Teilausdrucks B zu B'. Eine Dokumentmodifikation ist nicht nötig, da der neue Typ per Definition auch alle Instanzen des alten Typs enthalten muss (siehe Abschnitt 3.3.2, S. 28). Mit dieser Operation ist auch das Einfügen von neuen Elementen möglich: generalize ϵ to B? (B? wird dabei den fokussierten Ausdrücken als erste Silbe vorangestellt).

restrict B' to B in Z* Restriktiviert den Typ des Teilausdrucks B' zu B und stellt damit die Umkehroperation zu generalize dar. Eine Dokumentmodifikation wird nicht vorgenommen, da erwartet wird, dass das Dokument bereits dem neuen Typ entspricht. Die Transformation scheitert für solche Dokumente, bei denen dies nicht der Fall ist. Mit dieser Operation ist auch das Entfernen von (in Dokumenten nicht mehr vorkommenden) Teilausdrücken möglich: restrict B? to ϵ. Abschnitt 3.3.2, S. 28 enthält weitere Informationen zur Typ-Restriktion.

cast X to Y in Z* Ersetzt an den fokussierten Stellen die Vorkommen des Elementes X in das (bis auf den Namen identisch deklarierte) Element Y. Das X und Y identisch sind, können Attributwerte und Unterelemente im Dokument von X dabei problemlos nach Y übernommen werden. Zu beachten ist, dass mit dieser Operation rekursive Deklarationen nicht ersetzt werden können (das ist ein Unterschied zu der unify-Operation aus [Läm01a]). Im folgenden Beispiel sind die Deklarationen von $\{X, Y\}$ und $\{X', Y'\}$ bis auf die Namen identisch, können jedoch mit cast nicht ineinander überführt werden:

$$\begin{array}{ll} X \to Y\,Z & X' \to Y'\,Z \\ Y \to (X|Z) & Y' \to (X'|Z) \end{array}$$

editData B by M in Z* Diese Operation ist die einzige, die Information zu den Dokumenten hinzufügen oder aus diesen entfernen kann. Sie ist insofern eingeschränkt, dass sie den Typ des Dokuments nicht verändern darf (ansonsten scheitert die Transformation). M ist eine Methode, die die gewünschte Modifikation auf denjenigen Daten implementiert, die der Phrase B entsprechen. Zu erwähnen ist auch, dass diese Operation Auswirkungen auf die über Referenzen (Identitäts-Constraints, siehe Abschnitt 3.8.1, S. 45) definierten Strukturen haben kann, also (trotz der Typ-Prüfung) nicht strukturerhaltend ist.

element2attr in Z Wandelt die erste oder einzige Silbe des Unterstruktur-Typ-Ausdrucks, die ein obligatorischer oder optionaler Element-Ausdruck sein muss, in ein Attribut um. Voraussetzung dafür ist, dass dieses Element selbst keine Attribute besitzt und einen einfachen Typ hat. Dieser Operation kommt eine entscheidende Bedeutung zu, da sie zur Manipulation von Attributen benötigt wird.

attr2element L in Z Umkehroperation zu element2attr. Wandelt das Attribut L in ein (gegebenenfalls optionales) Element um und fügt es als erste Silbe in den Unterstruktur-Typ-Ausdruck ein. Es wird eine dem Attributtyp entsprechende Element-Deklaration angelegt. Dazu ist es erforderlich, dass der aus Z und L zusammengesetzte Identifikator dieser Deklaration im Format noch nicht verwendet wird.

setFixed to (true|false) in Z Setzt den Fixierungs-Status der fokussierten Elemente Z*. Auch hier wird keine Dokumentmodifikation vorgenommen, da erwartet wird, dass der Wert der konkret vorkommenden Elemente dem fixierten Wert entspricht (ansonsten scheitert die Transformation für das betreffende Dokument). Soll ein Element fixiert werden, ist es erforderlich, dass es einen nicht-leeren Default-Wert spezifiziert.

setDefault to Value in Z Setzt den Default- oder fixierten Wert des Elementes Z. Wenn das Element fixiert ist, werden die konkreten Werte dieses Elementes in den Dokumenten auf den neuen Wert geändert, ansonsten erfolgt keine Dokumentmodifikation. Diese Operation scheitert, wenn versucht wird, den fixierten Wert des Elementes auf eine leere Zeichenkette zu setzen.

introduce X as (A,(true|false),Value) Deklariert das neue Element X mit dem Unterstruktur-Typ A, einem Fixierungsstatus und einem Default- bzw. fixiertem Wert Value.

Attribute müssen per element2attr erzeugt werden. Diese Operation scheitert, wenn X bereits im Format deklariert ist.

eliminate X Entfernt die Element-Deklaration X aus dem Format. X darf innerhalb des Formats nicht verwendet werden (ansonsten scheitert die Transformation).

rename X by Y Benennt das Element X innerhalb des Formats und der Dokumente konsistent in Y um. Dies betrifft sowohl seine Deklaration als auch alle Vorkommen in Unterstruktur-Typ-Ausdrücken. Y darf im Format noch nicht als Element-Identifikator verwendet werden.

3.3 Diskussion

In diesem Abschnitt sollen bestimmte Aspekte der Operationen diskutiert werden. Die Diskussion wird unter anderem auch die Auswirkungen der betreffenden Operationen auf Identitätsbedingungen umfassen.

3.3.1 Ersetzungsstrategien

Die ausgewählte Operation muss immer auf alle im Fokus spezifizierten Elemente anwendbar sein.

Wirkt sie dabei nicht auf die ganze Element-Defintion (wie beispielsweise setFixed, rename), sondern auf einzelne Stellen des Unterstruktur-Typs, so werden die Ersetzungsstellen nicht-überlappend von außen nach innen und von links nach rechts ausgewählt. Eine erneute Suche in modifizierten Ausdrücken erfolgt dann nicht. Dadurch ist die Terminierung der Suche garantiert, allerdings sind die Modifikationen nicht erschöpfend.

Eine Ausnahme stellen hierbei attr2element und element2attr dar, bei denen nur genau eine Ersetzung vorgenommen wird. Das ist sinnvoll, da an diesen Operationen jeweils ein Attribut beteiligt ist und Attribute innerhalb eines Elementes nur höchstens einmal vorkommen dürfen.

Kommt eine Phrase in einem Ausdruck mehrfach vor, soll aber nicht an allen Stellen ersetzt werden, lassen sich die gewünschten Stellen durch eine fold-Operation zunächst isolieren. Daraufhin kann dann in dem als Ergebnis der Faltung erzeugten Element die eigentliche Ersetzung stattfinden. Anschließend wird das Hilfs-Element wieder entfaltet (unfold):

```
family → {(marriage: implied)} human human human*
human → {(name: required), (sex: required)} petName*
child → {(name: required), (sex: required)} petName*
male → {(name: required), (sex: required)} petName*
petName → {} τ

▷ fold human* to x in {family}
family → {(marriage: implied)} human human x
x → {} human*

▷ cast human to child in {x}

x → {} child*

▷ unfold x in {family}
family → {(marriage: implied)} human human child*

▷ eliminate x
family → {(marriage: implied)} human human child*
```

```
human → {(name: required), (sex: required)} petName*
child → {(name: required), (sex: required)} petName*
male → {(name: required), (sex: required)} petName*
petName → {} τ
```

Diese Technik lässt sich entsprechend auch dazu benutzen, die Elemente, die *nicht* von der Transformation betroffen sein sollen, aus dem betreffenden Element auszuklammern:

```
▷ fold human child* to x in {family}
family → {(marriage: implied)} human x

▷ cast human to male in {family}

family → {(marriage: implied)} male x

▷ unfold x in {family}
family → {(marriage: implied)} male human child*

▷ eliminate x
family → {(marriage: implied)} male human child*
human → {(name: required), (sex: required)} petName*
child → {(name: required), (sex: required)} petName*
male → {(name: required), (sex: required)} petName*
petName → {} τ
```

3.3.2 Verallgemeinerung und Restriktion

Für die Generalisierungs- und Restriktivierungsoperationen wird die Verallgemeinerungs-Relation „>“ ($E_1 > E_2$: „E_1 ist allgemeiner als E_2“ bzw. „E_2 ist ein Spezialfall von E_1“) benötigt. Sie wird im Folgenden analog zu [LL01] und [HP00] definiert.

Der grundsätzliche Ansatz ist dabei, dass ein Ausdruck E_1 allgemeiner als ein Ausdruck E_2 ist, wenn E_1 alle Wörter erzeugt die auch E_2 erzeugt, und das umgekehrt nicht gilt. Daraus folgt, dass es sich um eine irreflexive Relation handelt. Das bedeutet, die von E_2 erzeugte Sprache $L(E_2)$ muss eine echte Teilmenge der von E_1 erzeugten Sprache $L(E_2)$ sein.

Die bei der Verschmelzung benutzte Funktion $\mathit{Special}(X)$ ist der Abschluss aller Spezial-Fälle von X, oder, falls X nicht spezialisiert werden kann, X selbst:

$$\mathit{Special}(X) = \begin{cases} \{X\} & \text{wenn } \nexists Y : X > Y \\ \{\, Y \mid (X > Y) \wedge \nexists Z : Y > Z & \text{sonst} \end{cases}$$

Nun die Definition:

$$E_1 > E_2 \Leftrightarrow L(E_1) \supset L(E_1)$$

Transitivität

$$(\ E_1 > E_2\ \wedge\ E_2 > E_3\ ) \Rightarrow\ E_1 > E_3$$

Quantoren

$$\begin{aligned} E* &> E+ \\ E* &> E\,? \\ E\,? &> E \\ E\,? &> \epsilon \\ E+ &> E \end{aligned}$$

Auswahl-Operator

$$(\,E_1 \mid \cdots \mid E_j\,) > (\,E_1 \mid \cdots \mid E_{g-1} \mid E_{g+1} \mid \cdots \mid E_j\,)$$

Konjunktion

$$(\,E_1\,\&\,\cdots\,\&\,E_j\,) > E_1 \cdots E_j$$

Verschmelzung

$$\begin{aligned} i \in \{1,\ldots,j\} \wedge \\ (\forall i : (E_i > E) \vee (E_i \equiv E)) \wedge \\ \bigcup_i \mathit{Special}(E_i)\ \setminus\ \mathit{Special}(E) \cup \{\epsilon\}\ =\ \emptyset \Rightarrow \\ E_1 \cdots E_j < E* \text{ wenn } \forall i : E_i > \epsilon \\ E_1 \cdots E_j < E+ \text{ sonst} \end{aligned}$$

Lifting $(g, i \in \{1,\ldots,j\})$

$$\exists\, g : \forall\, i : E_i > E'_i\ \vee (i \neq g \wedge E_i \equiv E'_i) \Rightarrow \begin{array}{l} E_1 \cdots E_j > E'_1 \cdots E'_j \\ (\,E_1 \mid \cdots \mid E_j\,) > (\,E'_1 \mid \cdots \mid E'_j\,) \\ (\,E_1\,\&\,\cdots\,\&\,E_j\,) > (\,E'_1\,\&\,\cdots\,\&\,E'_j\,) \end{array}$$

Bei den Quantoren ergeben sich weitere Beziehungen per Transitivität, beispielsweise $E* > E\,? > \epsilon$. Die Regeln für den ?-Quantor und die Konjunktion lassen sich über deren Definitionen auf die Regel für den Auswahl-Operator zurückführen.

Die Formulierung in den Lifting-Regeln bedeutet, dass mindestens eines der Elemente von Sequenz- bzw. Auswahl-Operator eine echte Verallgemeinerung sein muss, für die anderen Elemente kann dann auch Gleichheit vorliegen.

Verschmelzen von Ausdrücken Die Verschmelzungsoperation bereitet Probleme, da sie in vielen Fällen nicht wohlgeformte Phrasen betrifft[1]:

$$E*\,E?\ <\ E*$$

Speziell gibt es keine wohlgeformte Sequenz mit mehr als einem Bestandteil, die sich zu $E*$ verallgemeinern lässt. Zur Implementierung wäre daher ein spezielles Trennsymbol sowohl im regulären Ausdruck als auch in den Dokumenten nötig:

$$\begin{array}{c} E* \,\|\, E? \,\|\, (E+ \,|\epsilon)\ <\ E* \\ \|\, e \,\| eeee \overset{\text{generalize}}{\Longrightarrow} eeeee \end{array}$$

Dieses Trennsymbol würde die Formalisierung allerdings komplizierter machen. Beispielsweise sind in diesem Fall bei der Generalisierungs-Operation (im Gegensatz zum sonstigen Verhalten) Änderungen am Dokument nötig, um die Trennsymbole zu entfernen. Weitere Probleme betreffen das Hinzufügen des Trennsymbols und seine Behandlung in den verschiedenen Operationen.

Pragmatische Alternativen wären, die Verallgemeinerung mit einer Permutation oder einer Entfaltung zu koppeln:

$$E*\,A\,E?\ <\ E*\,A$$

[1] Eine „sichere“ Verschmelzung ist $E\,E* \Longrightarrow E+$ – dies ist jedoch keine Verallgemeinerung, sondern eine Äquivalenz-Transformation

$$\begin{array}{l} A \to B\,C{+} \\ B \to D\,C{*} \end{array} \quad \overset{\text{unfold(B)}_{\text{A}}}{\Longrightarrow} \quad \begin{array}{l} A \to D\,C{+} \\ B \to D\,C{*} \end{array}$$

Bei der Entfaltung würden dabei gegebenenfalls Ausdrücke am Rande des entfalteten Elements mit Ausdrücken der Umgebung verschmolzen werden. Beide Varianten widersprechen jedoch der Orthogonalitätsforderung.

Hinzu kommt, dass eine Verschmelzung grundsätzlich nicht auf eindeutige Art und Weise umkehrbar ist. Aus diesen Gründen wurde auf eine solche Verschmelzungs-Operation verzichtet.

Konjunktion Die Verallgemeinerung einer Sequenz in eine Konjunktion ist in jedem Fall problemlos und ohne Modifikation des Dokuments möglich. Der umgekehrte Fall jedoch erfordert die entsprechende Vorbereitung des Dokuments, da Restriktionen voraussetzen, dass das Dokument bereits dem Ziel-Typ entspricht. Konkret würde diese Restriktion dann wie folgt ablaufen: Zunächst erfolgt die „Explosion“ der Konjunktion, das heißt die Umwandlung in eine äquivalente Auswahl, beispielsweise[2]

$$(\,E_1 \,\&\, E_2 \,\&\, E_3\,) \equiv (E_1E_2E_3 \mid E_1E_3E_2 \mid E_2E_1E_3 \mid E_2E_3E_1 \mid E_3E_1E_2 \mid E_3E_2E_1)$$

Dann müssen die Zweige (nach dem Einfügen entsprechender Dummy-Elemente D_i, die die Wohlgeformtheit sicherstellen) durch entsprechende Permutationen angeglichen werden:

$$\Rightarrow \cdots \Rightarrow (D_1E_1E_2E_3 \mid D_2E_1E_2E_3 \mid D_3E_1E_2E_3 \mid D_4E_1E_2E_3 \mid D_5E_1E_2E_3 \mid D_6E_1E_2E_3)$$

Nach dem Ausklammern von $E_1E_2E_3$ und dem Entfernen der Dummy-Auswahl ist die Spezialisierung dann beendet:

$$\Rightarrow (D_1|D_2|D_3|D_4|D_5|D_6)E_1E_2E_3 \Rightarrow (D_1|D_2|D_3|D_4|D_5|D_6)?E_1E_2E_3 \Rightarrow E_1E_2E_3$$

Aus Gründen der Benutzbarkeit wäre es vorteilhaft, diese Modifikationen automatisch zu erzeugen.

3.3.3 Äquivalenz

Äquivalenz von Ausdrücken („$\equiv$“) muss hier von deren Identität („$=$“) unterschieden werden: Äquivalenz bedeutet, dass beide Ausdrücke genau dieselben Termsequenzen erzeugen, während Identität zusätzlich fordert, dass die Ausdrücke auch strukturell (das bedeutet in ihrer Notation) gleich sind. Im Folgenden werden einige Regeln für die Äquivalenz-Relation $\equiv$ für reguläre Ausdrücke (für XMLSchema-Notation) dargestellt:

[2] Das Beispiel ist so gewählt, dass die „Explosion“ des Ausdrucks und damit die pragmatische Problematik dieses Vorgehens deutlich wird.

Identität

$$E_1 = E_2 \Rightarrow E_1 \equiv E_2$$

Symmetrie

$$E_1 \equiv E_2 \Leftrightarrow E_2 \equiv E_1$$

Transitivität

$$(E_1 \equiv E_2) \ \wedge \ (E_2 \equiv E_3) \Rightarrow E_1 \equiv E_3$$

Quantoren

$$\begin{aligned} E_1? &\equiv (E_1|\epsilon) \\ E_1+ &\equiv E_1\, E_1* \end{aligned}$$

Konjunktion

$$\begin{aligned} (k = j\,!) \wedge \\ (a, l \in \{1, \dots, j\}) \wedge \\ (g, h \in \{1, \dots, k\}) \wedge \\ (\nexists\, g, h : g \neq h \ \wedge \ \forall\, l : i_{g,l} = i_{h,l}) \wedge \\ (\forall\, g : \forall\, a : \exists\, l : i_{g,l} = a) \Rightarrow \\ (E_1 \& \cdots \& E_j) &\equiv (E_{i_{1,1}} \cdots E_{i_{1,j}} \mid \cdots \mid E_{i_{k,1}} \cdots E_{i_{k,j}}) \end{aligned}$$

Permutation

$$\begin{aligned} \forall\, a \in \{1, \dots, j\} : \exists\, l \in \{1, \dots, j\} : i_l = a \Rightarrow \\ (E_1 | \cdots | E_j) &\equiv (E_{i_1} | \cdots | E_{i_j}), \\ (E_1 \& \cdots \& E_j) &\equiv (E_{i_1} \& \cdots \& E_{i_j}) \end{aligned}$$

Lifting

$$\begin{aligned} \forall i \in \{1, \dots, j\} : E_i \equiv E'_i \Rightarrow \\ E_1 \cdots E_j &\equiv E'_1 \cdots E'_j, \\ (E_1 | \cdots | E_j) &\equiv (E'_1 | \cdots | E'_j), \\ (E_1 \& \cdots \& E_j) &\equiv (E'_1 \& \cdots \& E'_j) \end{aligned}$$

3.4 Formale Definition der Operationen

In diesem Abschnitt sollen die eingeführten Operationen formalisiert werden. Hierbei werden Identitäts-Constraints nicht berücksichtigt. Zunächst werden die verwendeten Datentypen definiert.

3.4.1 Datentypen

Definition 3.1 (Format) *Ein Tupel* $\mathcal{S} = (N, T, P, n_0)$ *heißt* Format*, mit*

- N *ist die Menge der Nichtterminale (Element-Identifikatoren),*
- T *ist die Menge der Terminalsymbole (*Tag*-Namen) des Formats,*
- $P : N \to \mathcal{A} \times E \times B_F \times V$ *ist eine Abbildung der Nichtterminale auf Elementdeklarationen.*
  $\mathcal{A}$ *sind Attributdeklarationen,*
  E *ist der Unterstruktur-Typ, ein regulärer Ausdruck über* $N \cup \{\tau\}$ *(*τ *bezeichnet Vorkommen von Text im Dokument),*
  B_F *speichert, ob der Elementwert fixiert ist, und*
  V *ist der Default- bzw. fixierte Wert des Elements,*
- n_0 *ist das Startsymbol des Formats.*

Definition 3.2 (Attribut-Deklaration in Elementen) *Ein Tupel* $\mathcal{A} = (A, D)$ *heißt* Attribut-Deklaration in einem Element*, mit*

- A *ist die Menge der Attribut-Namen, die in dem betreffenden Element deklariert sind,*

- *$D : A \to B_R \times B_F \times V$ ist eine Abbildung von Attributnamen auf Attributeigenschaften.*
  B_R speichert, ob das Attribut obligatorisch ist,
  B_F, ob sein Wert fixiert ist, und
  V enthält den Default- bzw. fixierten Wert des Attributs.

In XMLSchema ist es möglich, Deklarationen zu verschachteln. Die Element-Namen müssen dann nur bezüglich der Elter-Deklaration eindeutig einen Typ identifizieren (das heißt, in einer Element-Deklaration können mehrere Unterelemente mit demselben Namen deklariert werden, wenn sie denselben Typ haben). Daher ist es nötig, den „Pfad" der Elter-Deklarationen in Element-Identifikatoren einzuschließen. Die Definition erfolgt ähnlich zu [BFRW01].

Definition 3.3 (Element-Identifikator, *local***)** *$N = i\#sn_1/\ldots/sn_n$ ist ein* Element-Identifikator *mit*

- *i bezeichnet den XML-Namensraum, in dem das betreffende Element deklariert ist,*

- *sn_i sind lokale Namen (falls das Element im Format $\mathcal{S} = (N, T, P, n_0)$ deklariert ist, gilt $sn_i \in T$).*

*$local : i\#sn_1/\ldots/sn_n \to sn_n$ ist eine Abbildung von Element-Identifikatoren auf den lokalen (*Tag-*) Namen des betreffenden Elements.*

XMLSchema erlaubt es im Gegensatz zu DTDs, mehrere mögliche Startsymbole zu spezifizieren: jedes Element, das auf oberster Ebene (also nicht verschachelt innerhalb eines anderen) deklariert wird, kann den Wurzelknoten eines Dokumentes bilden. Die Einführung eines einzigen (künstlichen) Startsymbols vereinfacht die Modellierung dieser Eigenschaft. Das Startsymbol eines Formats ist also immer wie folgt deklariert:

$$P(n_0) = (\,(\emptyset, D),\ (n_1|\ldots|n_j),\ false,\ \epsilon_V\,)$$

($\{n_1, \ldots, n_j\}$ ist die Menge der Identifikatoren der obersten Element-Deklarationen, $false$ steht für „nicht fixiert" und ϵ_V ist der leere String).

Als Zwischenformat bei der Durchführung der Transformationen werden Ausdrücke verwendet, die auf der Notation für reguläre Ausdrücke basieren, wie sie von XMLSchema verwendet wird:

Definition 3.4 (Trafo-Ausdruck) *Ein* Trafo-Ausdruck *bezüglich eines Formats $\mathcal{S} = (N, T, P, n_0)$ wird als Erweiterung eines regulären Ausdruck in XMLSchema-Notation definiert:*

- *Jeder reguläre Ausdruck über $N \cup \{\tau\}$ in XMLSchema-Notation ist ein Trafo-Ausdruck.*

- *$\overset{\mathrm{Op}}{\hookrightarrow}(E)$ (Transformation) ist ein Trafo-Ausdruck.*

- *$\pi^{i_1,\ldots,i_n}(E_1, \ldots, E_n)$ (Permutation) ist ein Trafo-Ausdruck.*

Als ein Beispiel für einen solchen Trafo-Ausdruck soll nun in dem Ausdruck

$$A\ B\ C+$$

das Element C nach D umbenannt und die Elemente A und B vertauscht werden. Dies lässt sich in dem Trafo-Ausdruck

$$\pi^{2,1}(A, B) \overset{\mathrm{rename(D)}}{\hookrightarrow} (C)+$$

spezifizieren. rename ist in diesem Beispiel eine spezielle Prozedur, die mittels $\overset{\text{Op}}{\hookrightarrow}(\ldots)$ in den Ausdruck „implantiert" wird. Einen Überblick über die verwendeten Prozeduren gibt Abschnitt 3.4.2, S. 33.

Trafo-Ausdrücke werden von der „Compiler-Funktion" erzeugt und dann zur Format- und Dokumenttransformation interpretiert. Die Idee ist dabei, dass dieser Ausdruck an seinen Blättern dem Ursprungsausdruck gleicht (sowohl in $A\,B\,C{+}$ als auch der Trafo-Ausdruck akzeptieren (in dieser Reihenfolge) die Elemente A, B und eine Sequenz von Cs). Beim Hinaufreichen der akzeptierten Werte zur Wurzel der Ausdrucks werden diese nun entsprechend transformiert: $\pi^{2,1}(A,B)$ setzt die von A und B akzeptierten Elemente in neuer Reihenfolge zusammen (erst die von B akzeptierten Elemente, dann die von A), und $\overset{\text{rename(D)}}{\hookrightarrow}(C)$ wendet rename auf das von C akzeptierte Element an und liefert das entsprechende Ergebnis zurück. Abschnitt 3.5, S. 37 beschreibt diesen Vorgang detaillierter.

Mehrere Trafo-Ausdrücke, die jeweils mit einem zugehörigen Element-Identifikator als Schlüssel abgelegt sind, speichert die folgende Datenstruktur. Sie wird im Weiteren „Trafo-Grammatik" genannt.

Definition 3.5 (Trafo-Grammatik) *Ein Trafo-Grammatik $\mathcal{S}_T = (N, T, P_T, n_0)$ bezüglich eines Formats $\mathcal{S} = (N, T, P, n_0)$ basiert auf $\mathcal{S}$, verwendet jedoch die Abbildung*

$$P_T : N \to E_T$$

von einem Element-Identifikator N auf E_T, einen Trafo-Ausdruck bezüglich $\mathcal{S}$.

3.4.2 Semantische Aktionen

Hilfsfunktionen

- find E_1 in E_2: Testet, ob die Phrase E_1 in E_2 enthalten ist. Schlägt fehl wenn nicht.

- makeTrafoGrammar(N_F, Action): Iteriert über die Nichtterminale des Formats und erzeugt entsprechende Produktionen für das Trafo-Grammatik. Für Nichtterminale, die in N_F enthalten ist, wird Action auf den Unterstruktur-Typ des entsprechenden Elements angewendet und der resultierende Trafo-Ausdruck in der Trafo-Grammatik verwendet. Für die anderen Elemente wird der ursprüngliche Unterstruktur-Typ übernommen.

Aktionen für makeTrafoGrammar

- replace E_1 with E_2 in E: Ersetzt in E an allen Stellen, die E_1 entsprechen, durch E_2. Wenn E_1 nicht gefunden wurde, wird der übergebene Ausdruck nicht verändert. Ein spezieller Wert für E_1 ist ϵ. Es wählt in diesem Fall den Anfang von E zur Ersetzung aus, E_2 wird also E vorangestellt. Eine weitere Suche in E erfolgt in diesem Fall nicht.

- replace!: wie replace, nur dass die Aktion fehlschlägt, wenn nicht mindestens eine Ersetzung vorgenommen wurde.

Aktionen in $\overset{\text{Op}}{\hookrightarrow}(\ldots)$

- fold(n): Ersetzt im Format den Unterbaum von $\overset{\text{fold(n)}}{\hookrightarrow}(\ldots)$ durch n, und im Dokument die vom Unterbaum akzeptierten Elemente $e_1, \ldots, e_k$ durch ein n entsprechendes, attributloses Element, dessen Inhalt $e_1, \ldots, e_k$ ist.

- $\mathsf{unfold}(E)$: Ersetzt im Format das Unterelement von $\overset{\text{unfold(E)}}{\hookrightarrow}(\ldots)$ durch E, und im Dokument das Unterelement durch die Liste seiner Unterelemente seines Inhalts.

- $\mathsf{idreplace}(E)$: Ersetzt im Format den Unterausdruck von $\overset{\text{idreplace(E)}}{\hookrightarrow}(\ldots)$ durch E. Am Dokument wird keine Modifikation vorgenommen, allerdings wird geprüft, ob die Elemente dem Ausdruck E entsprechen. Die Aktion schlägt fehl, wenn das nicht der Fall ist.

- $\mathsf{edit}(Op, E)$: Hat keine Auswirkungen auf das Format (es wird vorausgesetzt, dass der Unterausdruck mit E identisch ist). Im Dokument wird die modifizierende Aktion Op auf die Elemente des Unterbaums von $\overset{\text{edit(Op,E)}}{\hookrightarrow}(\ldots)$ angewendet und dann geprüft, ob die Resultate vom Typ E sind. Schlägt fehl wenn nicht. Op ist häufig anwendungsspezifisch (beispielsweise beim Generieren von Elementen), das heißt, wird vom Nutzer spezifiziert. Folgende Aktionen werden jedoch schon definiert, da sie zur Implementierung der Operationen-Menge benötigt werden oder sehr gebräuchlich sind:

  - set(e*): Ersetzt den Unterbaum durch die Elementliste e*. Diese Operation kann auch zum Löschen des Unterbaums verwendet werden, indem für e* die leere Liste verwendet wird.
  - $\mathsf{changeValue}(V)$: Ersetzt den Inhalt des betreffenden Elements durch V, wenn das Element nicht leer ist (ansonsten bleibt es leer).
  - $\mathsf{isEmptyOrValue}(V)$: Prüft, ob der Inhalt des betreffenden Elements entweder leer oder identisch mit V ist. Wenn nicht, schlägt die Aktion fehl.

- $\mathsf{el2attr}(l, B_R, B_F, V)$: Im Format wird der Unterausdruck E von $\overset{\text{el2attr}(\mathrm{l},\mathrm{B_R},\mathrm{B_F},\mathrm{V})}{\hookrightarrow}$(E) durch ϵ ersetzt und dem aktuellen Element ein Attribut l mit der Deklaration (B_R, B_F, V) hinzugefügt. Im Dokument wird der Attributmenge des aktuellen Elements ein Attribut l hinzugefügt, dessen Wert dem Inhalt des E zugeordneten Unterelements entspricht.

- $\mathsf{attr2el}(l, B_R, m)$: Im Format wird der Unterausdruck E von $\overset{\text{attr2el}(\mathrm{l},\mathrm{B_R},\mathrm{m})}{\hookrightarrow}$(E) durch m (wenn $B_R = true$) bzw. $m?$ (wenn $B_R = false$) ersetzt und das Attribut l aus den Attributdeklaration des aktuellen Elements entfernt. Im Dokument wird das Attribut l aus der Attributliste entfernt und sein Wert in einem m entsprechenden Element eingesetzt. Existieren Elemente, die dem Unterausdruck E von $\overset{\text{attr2el}(\mathrm{l},\mathrm{B_R},\mathrm{m})}{\hookrightarrow}$(E) zugeordnet sind, so gehen diese hierbei verloren.

- $\mathsf{rename}(m)$: Benennt sowohl im Format als auch im Dokument das Unterelement von $\overset{\text{rename(m)}}{\hookrightarrow}(\ldots)$ in m um.

Format-modifizierende Aktionen

- $\mathsf{setFixed}(n, B_F)$: Setzt den Fixierungsstatus der Element-Deklaration von n auf B_F.

- $\mathsf{setDefault}(n, V)$: Setzt den Default-Wert der Element-Deklaration von n auf V.

- $\mathsf{introduce}(n, E, B_F, V)$: Fügt ein neues Nichtterminal $n \to (\ (\emptyset, D), E, B_F, V)$ in das Format ein.

- $\mathsf{delete}(n)$: Löscht das Nichtterminal n und seine zugehörige Element-Deklaration aus dem Format.

- $\mathsf{rename}(n, m)$: Benennt das Nichtterminal n in m um.

3.4.3 Operationen

Die hier beschriebenen Aktionen stellen eine Art „Compilierung" der entsprechenden Operation dar: Zunächst wird anhand des Ausgangsschemas abgesichert, dass die Voraussetzungen für die betreffende Operation erfüllt sind (so weit das allein durch Betrachtung des Formats möglich ist). Ist das der Fall, wird ein Trafo-Grammatik sowie eine Liste von Format-modifizierenden Aktionen erzeugt, die dann als Grundlage für die Transformation von Format und Dokumenten dienen. Die Signatur der Compiler-Funktion ist

$$(M^*, \mathcal{S}_T^*) = [\![O]\!]$$

Dabei ist O die jeweilige Operation. Die anderen Symbole sind die folgenden, die in den Definitionen verwendet werden:

- $\mathcal{S} = (N, T, P, n_0)$ ist das Ursprungsschema.
- $\mathcal{S}_T$ ist ein Trafo-Grammatik, $\mathcal{S}_T^*$ eine Liste davon.
- E und F sind reguläre Ausdrücke (E_F wird zur Fokussierung verwendet).
- n und m sind Element-Identifikatoren.
- N_F ist die Menge der fokussierten Element-Identifikatoren.
- M^* enthält eine Liste von Format-modifizierenden Aktionen.

Die Notation von Listen erfolgt ähnlich zu der in Prolog:

- $[]$ ist die leere Liste.
- $[X]$ ist eine Liste, die nur das Element X enthält.
- $[X|X^*]$ ist eine Liste mit dem ersten Element X und der Restliste X^*.

fold: $(M^*, [\mathcal{S}_T|\mathcal{S}_T^*]) = [\![\mathsf{fold}\ E\ \mathsf{to}\ n\ \mathsf{in}\ N_F]\!]$, mit
$(M^*, \mathcal{S}_T^*) = [\![\mathsf{introduce}\ n\ \mathsf{as}\ (E, false, \epsilon_V)]\!]$,
$\mathcal{S}_T = \mathsf{makeTrafoGrammar}(\ N_F,\ \mathsf{replace!}\ E\ \mathsf{with}\ \overset{\mathrm{fold(n)}}{\hookrightarrow}(E)\ )$.

unfold: $([], [\mathcal{S}_T]) = [\![\mathsf{unfold}\ n\ \mathsf{in}\ N_F]\!]$, mit
$P(n) = ((\emptyset, \_), E, false, \epsilon_V)$,
$\mathcal{S}_T = \mathsf{makeTrafoGrammar}(\ N_F,\ \mathsf{replace!}\ n\ \mathsf{with}\ \overset{\mathrm{unfold(E)}}{\hookrightarrow}(n)\ )$.

permute: $([], [\mathcal{S}_T]) = [\![\mathsf{permute}\ E_1 \cdots E_k\ \mathsf{to}\ i_1, \ldots, i_k\ \mathsf{in}\ N_F]\!]$, mit
$\forall\, j : \exists\, l : (j = i_l),\ j, l \in \{1, \ldots, k\}$,
$\mathcal{S}_T = \mathsf{makeTrafoGrammar}(\ N_F,\ \mathsf{replace!}\ E_1 \cdots E_k\ \mathsf{with}\ \pi^{i_1,\ldots,i_k}(E_1, \ldots, E_k)\ )$.

distribute: $([], [\mathcal{S}_T]) = [\![\mathsf{distribute}\ E\ \mathsf{in}\ N_F]\!]$, mit
$E = (E_1| \cdots |E_j)E_x$,
$E_y = (E_1E_x| \cdots |E_jE_x)$,
$\mathcal{S}_T = \mathsf{makeTrafoGrammar}(\ N_F,\ \mathsf{replace!}\ E\ \mathsf{with}\ \overset{\mathrm{idreplace(E_y)}}{\hookrightarrow}(E)\ )$.

extract: $([],[\mathcal{S}_T]) = [\![\mathsf{extract}\ E\ \mathsf{in}\ N_F]\!]$, mit
$E = (E_1|\cdots|E_j)E_x$,
$E_y = (E_1E_x|\cdots|E_jE_x)$,
$\mathcal{S}_T = \mathsf{makeTrafoGrammar}(\ N_F,\ \mathsf{replace!}\ E_y\ \mathsf{with}\ \overset{\mathrm{idreplace(E)}}{\hookrightarrow}(E_y)\ )$.

preserve: $([],[\mathcal{S}_T]) = [\![\mathsf{preserve}\ E_1\ \mathsf{as}\ E_2\ \mathsf{in}\ N_F]\!]$, mit
$E_1 \equiv E_2$,
$\mathcal{S}_T = \mathsf{makeTrafoGrammar}(\ N_F,\ \mathsf{replace!}\ E_1\ \mathsf{with}\ \overset{\mathrm{idreplace(E_2)}}{\hookrightarrow}(E_1)\ )$.

generalize: $([],[\mathcal{S}_T]) = [\![\mathsf{generalize}\ E_s\ \mathsf{to}\ E_g\ \mathsf{in}\ N_F]\!]$, mit
$E_g > E_s$,
$\mathcal{S}_T = \mathsf{makeTrafoGrammar}(\ N_F,\ \mathsf{replace!}\ E_s\ \mathsf{with}\ \overset{\mathrm{idreplace(E_g)}}{\hookrightarrow}(E_s)\ )$.

restrict: $([],[\mathcal{S}_T]) = [\![\mathsf{restrict}\ E_g\ \mathsf{to}\ E_s\ \mathsf{in}\ N_F]\!]$, mit
$E_g > E_s$,
$\mathcal{S}_T = \mathsf{makeTrafoGrammar}(\ N_F,\ \mathsf{replace!}\ E_g\ \mathsf{with}\ \overset{\mathrm{idreplace(E_s)}}{\hookrightarrow}(E_g)\ )$.

cast: $([],[\mathcal{S}_T]) = [\![\mathsf{cast}\ n\ \mathsf{to}\ m\ \mathsf{in}\ N_F]\!]$, mit
$m, n \in N$,
$P(n) = P(m)$,
$\mathcal{S}_T = \mathsf{makeTrafoGrammar}(\ N_F,\ \mathsf{replace!}\ n\ \mathsf{with}\ \overset{\mathrm{rename(m)}}{\hookrightarrow}(n)\ )$.

editData: $([],[\mathcal{S}_T]) = [\![\mathsf{editData}\ E\ \mathsf{by}\ Op\ \mathsf{in}\ N_F]\!]$, mit
$\mathcal{S}_T = \mathsf{makeTrafoGrammar}(\ N_F,\ \mathsf{replace!}\ E\ \mathsf{with}\ \overset{\mathrm{edit(Op,E)}}{\hookrightarrow}(E)\ )$.

element2attr: $([],[\mathcal{S}_T]) = [\![\mathsf{element2attr\ in}\ n]\!]$, mit
$P(n) = (\ (A, \_), E, \_, \_)$,
$E_l = \begin{cases} E_1 & \text{wenn } E = E_1 \cdots E_j \\ E & \text{sonst} \end{cases}$,
$(m, B_R) = \begin{cases} B_R = true & \text{wenn } E_l = m \\ B_R = false & \text{wenn } E_l = m? \end{cases}$,
$P(m) = (\ (\emptyset, \_), \tau, B_F, V)$,
$l = local(m)\ \wedge\ l \notin A$,
$\mathcal{S}_T = \mathsf{makeTrafoGrammar}(\ \{n\},\ \mathsf{replace!}\ E_l\ \mathsf{with}\ \overset{\mathrm{el2attr(l,B_R,B_F,V)}}{\hookrightarrow}(E_l)\ \mathsf{in}\ E\ )$.

attr2element: $(M^*,[\mathcal{S}_T|\mathcal{S}_T^*]) = [\![\mathsf{attr2element}\ l\ \mathsf{in}\ n]\!]$, mit
$n = i\#sn_1/\cdots/sn_j$,
$m = i\#sn_1/\cdots/sn_j/l$,
$P(n) = (\ (A, D), E, \_, \_)$,
$l \in A$,
$D(l) = (B_R, B_F, V)$,
$(M^*, \mathcal{S}_T^*) = [\![\mathsf{introduce}\ m\ \mathsf{as}\ (\tau, B_F, V)]\!]$,
$\mathcal{S}_T = \mathsf{makeTrafoGrammar}(\ \{n\},\ \mathsf{replace!}\ E\ \mathsf{with}\ \overset{\mathrm{attr2el(l,B_R,m)}}{\hookrightarrow}(\epsilon)E\ \mathsf{in}\ E\ )$.

setFixed: $([M],[\mathcal{S}_T]) = [\![\mathsf{setFixed\ to}\ B_F\ \mathsf{in}\ n]\!]$, mit
$P(n) = (\_, E, \_, V)\ \wedge\ (B_F = true\ \Rightarrow\ V \neq \epsilon_V)$,
$E_c = \begin{cases} E & \text{wenn } B_F = false \\ \overset{\mathrm{isEmptyOrValue(V)}}{\hookrightarrow}(E) & \text{sonst} \end{cases}$,
$M = setFixed(n, B_F)$,
$\mathcal{S}_T = \mathsf{makeTrafoGrammar}(\ \{n\},\ \mathsf{replace}\ E\ \mathsf{with}\ E_c\ \mathsf{in}\ E\ )$.

setDefault: $([M],[\mathcal{S}_T]) = [\![\mathsf{setDefault\ to}\ V\ \mathsf{in}\ n]\!]$, mit
$P(n) = (\_, E, B_F, \_)\ \wedge\ (B_F = true\ \Rightarrow\ V \neq \epsilon_V)$,

$$E_c = \begin{cases} E & \textsf{wenn } B_F = false \\ \overset{\text{changeValue(V)}}{\hookrightarrow}(E) & \textsf{sonst} \end{cases}$$
$M = setDefault(n, V)$,
$\mathcal{S}_T = \textsf{makeTrafoGrammar(}\ n,\ \textsf{replace}\ E\ \textsf{with}\ E_c\ \textsf{in}\ E\ \textsf{)}$.

introduce: $([M], []) = [\![\textsf{introduce}\ n\ \textsf{as}\ (E, B_F, V)]\!]$, mit
$n \notin N$,
$M = introduce(n, E, B_F, V)$.

eliminate: $([M], []) = [\![\textsf{eliminate}\ n]\!]$, mit
$n \in N$,
$\forall\, m \in N \setminus \{n\} : P(m) = (\_, E, \_, \_)\ \wedge\ \neg\ \textsf{find}\ n\ \textsf{in}\ E$,
$M = delete(n)$.

rename: $([M], [\mathcal{S}_T]) = [\![\textsf{rename}\ n\ \textsf{by}\ m]\!]$, mit
$n \in N$,
$m \notin N$,
$M = rename(n, m)$,
$\mathcal{S}_T = \textsf{makeTrafoGrammar(}\ N,\ \textsf{replace}\ n\ \textsf{with}\ \overset{\text{rename(m)}}{\hookrightarrow}(n)\ \textsf{in}\ n\ \textsf{)}$.

3.5 Dokument-Transformation

Die Dokument-Transformation ist mit dem Test auf Validität gegenüber dem Format (*membership checking*, also dem Test darauf, ob das Dokument in der vom Format spezifizierten Sprache Mitglied ist) und dem Zuordnen eines Typs zu den Elementen (*type assignment*) verknüpft. Bei XMLSchema ist diese Typzuordnung im Gegensatz zu DTDs nicht trivial, da der *Tag-Name* eines Elements je nach Kontext verschiedenen Typen entsprechen kann und nicht, wie bei DTDs, im gesamten Format eindeutig ist.[3]

Die Dokument-Transformation basiert auf einem Trafo-Ausdruck (siehe Abschnitt 3.4, S. 32), der aus dem Ausgangs-Format unter Berücksichtigung der vorzunehmenden Transformationen erstellt wird (siehe Abschnitt 3.4.3, S. 35). Beispielsweise werden aus den regulären Ausdrücken

$$A{*}\ B \text{ und } A\ X$$

bei einer Permutations- bzw. Element-nach-Attribut-Konvertierung die folgenden Trafo-Ausdrücke erzeugt:

$$\pi^{2,1}(A{*}, B) \text{ bzw. } A \overset{\text{el2attr}}{\hookrightarrow}(X)$$

erzeugt.

Bei der Dokument-Transformation wird dieser Trafo-Ausdruck interpretiert. Die Interpretation umfasst das Zuordnen der Elemente des Dokuments zu den einzelnen Bestandteilen des regulären Ausdrucks („Parsen“) sowie die Modifikation der Elemente entsprechend den $\overset{\text{Op}}{\hookrightarrow}(E)$ und $\pi^{i_1,\ldots,i_j}(E_1, \ldots, E_j)$-Ausdrücken.

Das Parsen bedeutet auch die Typzuordnung zu den einzelnen Elementen. Dieser Typ entspricht einem Element-Identifikator in der Format-Spezifikation, über den nun der Trafo-Ausdruck dieses Elements ermittelt werden kann. Mit diesem Trafo-Ausdruck werden dann die Unterelemente des betreffenden Elements transformiert. Auf diese Weise erfolgt also der Abstieg in den Dokument-Baum.

[3]Theoretische Erläuterungen zu den Grammatikklassen, denen DTD und XMLSchema angehören, finden sich in [LMM00]

Nachdem auf diese Weise der Unterbaum eines Elementes transformiert wurde, wird es an seinen Oberausdruck weitergereicht. Dieser sammelt gegebenenfalls (beispielsweise $*$- oder $+$-Ausdrücke) mehrere solcher Elemente an. Sind alle Unterausdrücke eines solchen Ausdrucks erfolgreich geparst, reicht dieser die Liste der akzeptierten Elemente wiederum an seinen Oberausdruck weiter. Lediglich Permutations- und Transformationsausdruck führen noch Modifikationen auf den von ihren Unter-Ausdrücken heraufgereichten Elementlisten durch. Der Permutationsausdruck konkateniert die Elementlisten seiner Unterausdrücke in der angegebenen Reihenfolge ($e(E)$ ist dabei die dem jeweiligen Ausdruck zugeordnete Liste von Elementen):

$$e(\pi^{i_1,\ldots,i_j}(E_1,\ldots,E_j)) = e(E_{i_1})\cdots e(E_{i_j})$$

Der Transformationsausdruck modifiziert nun nicht nur die Elementliste seines Unterausdrucks, sondern potenziell auch die Attributliste A_V des Elements, dessen Inhalt momentan transformiert wird (Op ist dabei eine der in Abschnitt 3.4.2, S. 33 definierten Prozeduren):

$$(e(\overset{\mathrm{Op}}{\hookrightarrow}(E)), A'_V) = Op(e(E), A_V)$$

Die Attributmenge wird dabei von links nach rechts und von unten nach oben durch den Ausdruck gereicht. Dies entspricht der Besuchsreihenfolge beim Analysieren der Elementliste. In der vorliegenden Formalisierung ist die Reihenfolge allerdings nicht von Bedeutung, da die Attributmenge nur höchstens einmal je Durchlauf durch den Ausdruck geändert wird.

Das Ergebnis der Transformation ist dann die dem gesamten Trafo-Ausdruck zugeordnete Elementliste sowie die modifizierte Attributmenge.

Abbildung 3.1 zeigt den Ablauf einer Permutations-Transformation: Man sieht zunächst den Trafo-Ausdruck

$$\pi^{2,1}(A*, B)$$

in Baum-Repräsentation. Der Abstiegspfad des Interpreters ist darin durch die gepunktete Linie eingezeichnet. Das gepunktete Kästchen rahmt den momentan interpretierten Ausdruck ein.

Der Interpreter steigt im Trafo-Ausdruck bis zum linken Blatt ab und ordnet das erste Element des Dokuments diesem Ausdruck zu (Schritt 1). Im nächsten Schritt (2) wird dieses Element zum „$*$"-Ausdruck hinaufgereicht und dort zunächst gespeichert. Nun wird das nächste Element eingelesen (Schritt 3) und dem „$*$"-Ausdruck zugeordnet (Schritt 4). Da das folgende Element nicht dem Typ A entspricht, wird nun der rechte Nachbarknoten des „$*$"-Ausdrucks interpretiert (Schritt 5). Danach ist der Unterbaum des Permutationsausdrucks eingelesen und die Permutation kann ausgeführt werden (Schritt 6). Das Ergebnis der Permutation wird schließlich zum Wurzel-Ausdruck hinaufgereicht (Schritt 7).

Die Transformation eines Elements in ein Attribut läuft analog ab (Abbildung 3.2). Hier wird die Umwandlung des Elementes x in ein Attribut innerhalb des Transformationsausdrucks durch die Prozedur el2attr durchgeführt.

Für jede Transformations-Operation muss der gesamte Dokument-Baum jeweils neu analysiert und transformiert werden. Daher ist es für die Verarbeitung einer Serie von Transformationen erforderlich, dass die Transformationen nur wohlgeformte Formate/Dokumente erzeugen, um die Analysierbarkeit des Dokuments sicherzustellen.

Eine Optimierung wäre allerdings leicht möglich, indem aufeinanderfolgende Transformationen, die auf Element-Mengen wirken, die sich nicht überlappen, in einer Trafo-Grammatik zusammengefasst werden. Hierbei könnten dann auch Identitäts-Transformationen (generalize) und Wohlgeformtheitstests (restrict) gegebenenfalls ignoriert werden.

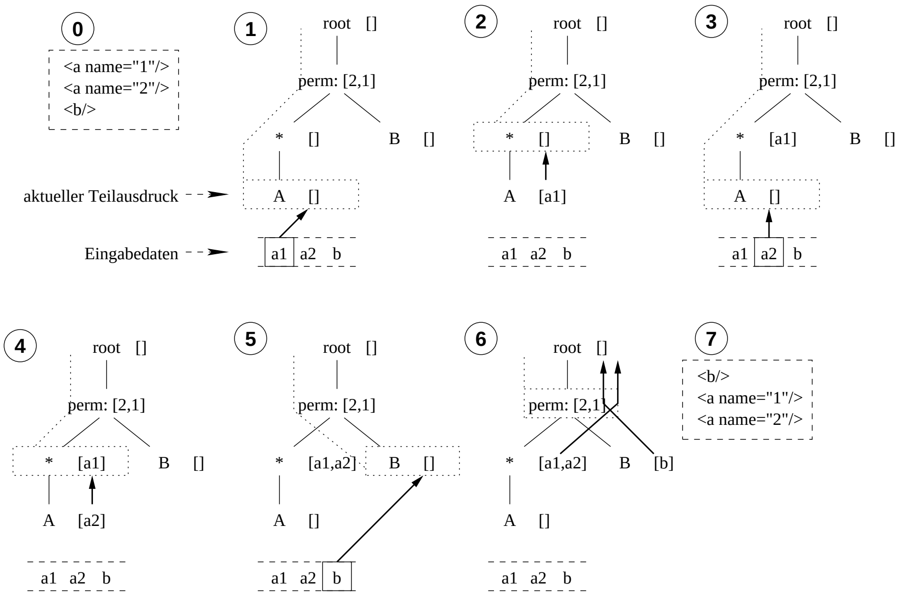

Abbildung 3.1: Ausführung einer Element-Permutation durch Interpretation von $\pi^{2,1}(A{*}, B)$

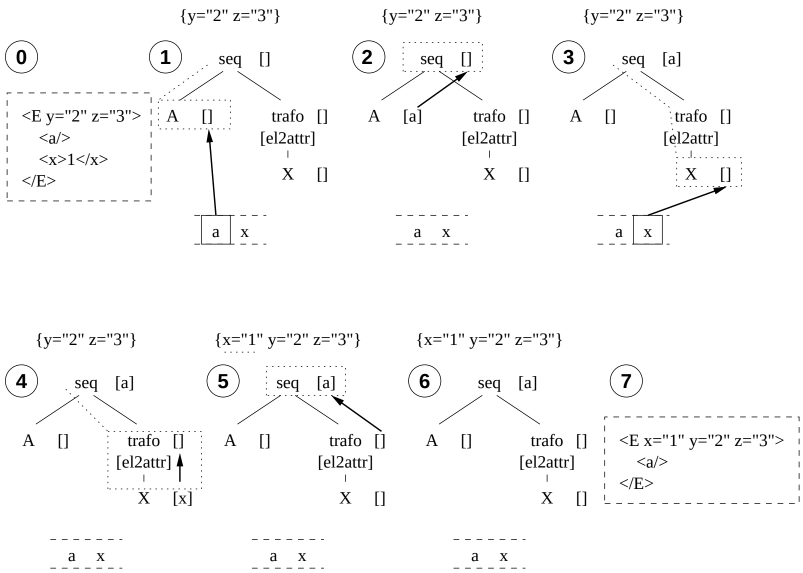

Abbildung 3.2: Ausführung einer Element-nach-Attribut-Transformation durch Interpretation von $A \overset{\text{el2attr}}{\hookrightarrow} (X)$

3.6 Format-Transformation

Dokument- und Format-Transformation werden unabhängig voneinander auf der Basis von Trafo-Ausdrücken durchgeführt, die in einer Menge von Trafo-Grammatiken gespeichert sind.

Bei der Format-Transformation werden die Trafo-Ausdrücke so interpretiert, dass mit der Eingabe der Menge von Attributdefinitionen eine gegebenenfalls geänderte Menge von Attributdefinitionen und ein regulärer Ausdruck für den Unterstruktur-Typ des Elements berechnet werden. Zusätzlich wird jedoch noch eine Menge von Format-modifizierenden Aktionen benötigt, die weitere Parameter einer Element-Deklaration wie Fixierungsflag, Default-Wert und Element-Identifikator ändern können sowie Deklarationen zum Format hinzufügen oder daraus entfernen.

Die Format-Transformation (Abbildung 3.3) beginnt mit der Interpretation der Trafo-Grammatiken (1). Die Ausführung der Format-modifizierenden Aktionen (2) erfolgt erst zum Schluss, da diese beispielsweise Element-Identifikatoren umbenennen könnten, so dass dann bei umgekehrter Reihenfolge die Korrespondenz zwischen dem Format und der Trafo-Grammatik verloren ginge.

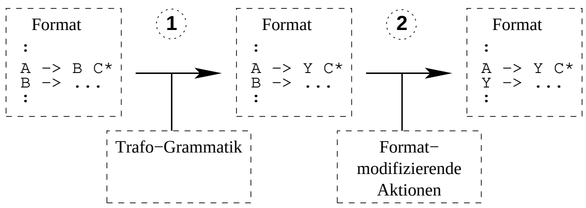

Abbildung 3.3: Überblick Format-Transformation am Beispiel der Umbenennung von B nach Y

Bei der Interpretation einer Trafo-Grammatik werden ihre Einträge nacheinander durchlaufen. Für jeden Eintrag werden aus der Ausgangs-Formatspezifikation die Attributdeklarationen ermittelt. Nun erfolgt die Interpretation des Trafo-Ausdrucks. Auch hier wird der Ausdruck von links nach rechts und von unten nach oben traversiert. Im Unterschied zur Dokument-Transformation wird aber jeder Knoten des Ausdrucks genau einmal besucht. Auch die semantischen Aktionen werden anders interpretiert als bei der Dokument-Transformation. Die Standard-Operation ist hier, dass jeder Ausdruck sich selbst als Ergebnis in Richtung der Wurzel weiterreicht ($x(E)$ steht dabei für den Ergebnisausdruck des Ausdrucks E):

$$x(E) = E$$

Der Permutationsausdruck ergibt eine entsprechend permutierte Sequenz seiner Unterausdrücke:

$$x(\pi^{i_1,\ldots,i_j}(E_1,\ldots,E_j)) = x(E_{i_1})\cdots x(E_{i_j})$$

Transformationsausdrücke schließlich transformieren ihren Unterausdruck und die Attribut-Deklarationen $\mathcal{A}$ der aktuellen Elementdeklaration:

$$(x(\overset{\text{Op}}{\hookrightarrow}(E)), \mathcal{A}') = Op(x(E), \mathcal{A})$$

Auch hier ist die Semantik der Operationen Op anders definiert als bei der Dokument-Transformation (siehe Abschnitt 3.4.2, S. 33).

Nach Abschluss der Transformation entsteht am Wurzelknoten des Trafo-Ausdrucks ein regulärer Ausdruck in XMLSchema-Notation,[4] der dann zusammen mit den

[4] Transformations- und Permutationsausdruck werden während der Transformation ersetzt. Auch die semantischen Aktionen erzeugen keinen dieser Ausdrücke. Daher liegt am Ende tatsächlich ein regulärer Ausdruck in XMLSchema-Notation vor.

modifizierten Attributdeklarationen als neue Deklaration des aktuellen Elements gesetzt wird.

Abbildung 3.4 zeigt die Transformation des Ausdrucks $\pi^{2,1}(A*, B)$. Die Ausdrücke werden hier im Standard-Fall durch sich selbst interpretiert (1). Eine Ausnahme bildet hierbei der Permutationsausdruck (2), der die Sequenz seiner permutierten Unterausdrücke ergibt (3).

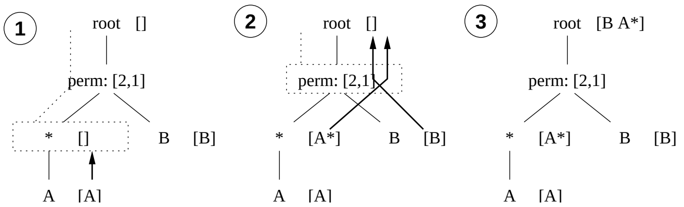

Abbildung 3.4: Ausführung einer Ausdrucks-Permutation durch Interpretation von $\pi^{2,1}(A*, B)$

Nun wird die Transformation mit der nächsten Trafo-Grammatik fortgesetzt. Bei einer Menge von mehreren Trafo-Grammatiken muss berücksichtigt werden, dass die Operationen auch die weiteren Trafo-Grammatiken auf der Grundlage des ursprünglichen Formats erzeugen würden, also nichts von den bereits vorgenommenen Änderungen wissen. Das bedeutet, dass die Änderungen, die die einzelnen Trafo-Grammatiken vornehmen, sich nicht mit den Änderungen durch die anderen Trafo-Grammatiken der Menge überschneiden dürfen. Daher ist dann auch die Reihenfolge, in der die Trafo-Grammatiken interpretiert werden, unabhängig von der, in der sie erzeugt werden. In der vorliegenden Formalisierung ist dies zunächst nicht relevant, da für jede Operation höchstens eine Trafo-Grammatik erzeugt wird.

Ist die Menge der Trafo-Grammatiken leer, wird das ursprüngliche Format unverändert übernommen.

Schließlich werden die Format-modifizierenden Aktionen ausgeführt. Auch diese müssen aus denselben Gründen wie bei den Trafo-Grammatiken voneinander unabhängig sein. Die Semantik dieser Aktionen ist in Abschnitt 3.4.2, S. 33 definiert.

3.7 Diskussion von Alternativen

Dieses Kapitel soll Alternativen zu dem dann schließlich gewählten Design diskutieren und die getroffenen Designentscheidungen erläutern. Die hier diskutierten Aspekte sind sehr technischer Natur und zum Verständnis dieser Arbeit nicht unbedingt erforderlich.

3.7.1 Einschränkung der Distribution

Die Distribution unterliegt in der vorliegenden Definition der Einschränkung, dass sie nur zulässig ist, wenn sich die entsprechenden Dokumentterme durch Identität transformieren lassen. Ursprünglich wurde eine freiere Distribution favorisiert, die beispielsweise folgende Transformation zugelassen hätte:

$$A\,(B|C) \overset{\text{distribute}'}{\Longrightarrow} (\,A\,B\,|\,C\,A\,)$$

Das Problem hierbei ist, dass bei der Durchführung der Dokument-Transformation implizit eine Permutation durchgeführt werden müsste – und zwar abhängig davon, welcher Zweig der Disjunktion die geparsten Elemente akzeptiert hat. Wenn diese Permutation innerhalb der Disjunktion passieren soll, ist es erforderlich, die Elemente, die A entsprechen, beim Parsen in die Disjunktion hineinzureichen. Wenn das zu distributierende Element hinter einer solchen Disjunktion steht, ist diese Variante allerdings nicht möglich, da die A entsprechenden Elemente noch nicht vorliegen wenn die Disjunktion geparst wird. Soll die Permutation von dem gemeinsamen Oberausdruck von A und der Disjunktion durchgeführt werden, muss die Disjunktion Informationen darüber bereitstellen, welche Alternative die betreffenden Elemente akzeptiert hat.

In beiden Fällen werden Parsing und die Transformation komplexer: Die Disjunktion benötigt entweder einen zusätzlichen Eingabe- oder Ausgabeparameter.

Hinzu kommt, dass die Operationen damit nicht mehr orthogonal sind. Das ist auch der Grund, warum auch `extract` entsprechend eingeschränkt ist (hier ließe sich die benötigte Permutation problemlos lokal in dem entsprechenden Zweig der Distribution durchführen).

Das oben gewählte Beispiel lässt sich auch mit den jetzt vorhandenen Operationen (mit etwas mehr Aufwand) umsetzen:

1. $A\,(B|C) \overset{\text{permute}}{\Longrightarrow} (B|C)\,A$: Das in die `choice` hinein zu bewegende Element muss immer hinter den bestehenden Elementen der Zweige eingefügt werden, um die Wohlgeformtheit der Disjunktion zu erhalten.

2. $\overset{\text{distribute}}{\Longrightarrow} (\,B\,A\,|\,C\,A\,)$: Die Distribution erfordert keine Umordnung der Elemente.

3. $\overset{\text{permute}}{\Longrightarrow} (\,A\,B\,|\,C\,A\,)$: Der Ausdruck ist in der gewünschten Form.

3.7.2 Elemente und Attribute in XMLSchema

Dass auf spezielle Operationen zur Behandlung von Attributen verzichtet werden konnte, liegt daran, dass Elemente in XMLSchema (im Gegensatz zu DTD) alle Eigenschaften von Attributen besitzen können. Es lässt sich also jedes Attribut ohne Informationsverlust in ein Element umwandeln, dann modifizieren und schließlich zurück in ein Attribut konvertieren.

Im Einzelnen besitzt XMLSchema folgende Eigenschaften, die in diesem Kontext relevant sind und in DTD fehlen:

- Default-Werte lassen sich auch für Elemente deklarieren. Dieser Default-Wert ist auf einfachen Text beschränkt (was im Hinblick auf die Äquivalenz zu Attributen keine Einschränkung bedeutet).

- Auch Elemente lassen sich als fixiert (`FIXED`) deklarieren, so dass sich diese Eigenschaft von Attributen bei der Konvertierung in Elemente und der nachfolgenden Modifikation dieser Elemente ohne Weiteres erhalten lässt.

3.7.3 Operatoren-Syntax

In der momentanen Formalisierung besteht die Fokus-Spezifikation lediglich aus einem Element-Identifikator bzw. einer Menge von Element-Identifikatoren. Ursprünglich bestand der Fokus noch zusätzlich aus einer Kontext-Phrase, das heißt, es war möglich, eine Umgebung für die Ersetzungsstelle anzugeben:

$$\textsf{permute}\ B\,C\ \textsf{to 2,1 in}\ A\,B\,C\,D$$

In diesem Beispiel wird innerhalb der fokussierten Elemente die Permutation nur für solche Vorkommen von $B\,C$ durchgeführt, die zwischen einem A und einem D stehen.

Das folgende Beispiel demonstriert diese Vorgehensweise für die cast-Operation:

```
family → {(marriage: implied)} human human human*
human → {(name: required), (sex: required)} petName*
child → {(name: required), (sex: required)} petName*
petName → {} τ

▷ cast human to child in human* of {family}
family → {(marriage: implied)} human human child*
```

Auf diese Weise wurde die Spezifikation einfacher, falls nur bestimmte Phrasen eines Ausdrucks transformiert werden sollen. Mit den jetzt vorhandenen Operationen ist dieses immer noch möglich. Es ist dazu jedoch erforderlich, den Kontext mittels fold in ein separates Element zu falten und dieses dann zu transformieren (siehe Abschnitt 3.3.1, S. 27).

Eine andere Möglichkeit wäre, Operationen durch eine Ersetzung einer Phrase durch eine andere zu spezifizieren, wobei an diese Phrasen dann je nach Operation bestimmte Anforderungen gestellt würden. Das obige Beispiel sähe mit dieser Methode wie folgt aus:

```
family → {(marriage: implied)} human human human*
human → {(name: required), (sex: required)} petName*
child → {(name: required), (sex: required)} petName*
petName → {} τ

▷ cast by replacing (A B human*) with (A B child*) in {family}
family → {(marriage: implied)} human human child*
```

A und B sind hierbei Variablen für Teilausdrücke. Durch diese Variablen lässt sich die Spezifikation verkürzen (in der weiter oben vorgestellten Variante musste die zu transformierende Phrase in der Fokusspezifikation wiederholt werden).

Diese Spezifikationsmethode hat allerdings zahlreiche Nachteile:

- Für bestimmte Operationen (beispielsweise attr2element, siehe unten) müssen spezielle Ausdrücke definiert werden.
- Die vorgenommene Transformation kann in einem komplizierten Fokus visuell „untergehen“, was es erschwert, Fehler zu finden.
- Mit den Phrasen lassen sich Ersetzungen spezifizieren, die dann jedoch gegen die spezifischen Einschränkungen der Operation verstoßen. Beispielsweise wäre es problemlos möglich gewesen, in obigem Beispiel Teilausdrücke zu löschen oder zu duplizieren sowie die Folge zu permutieren:

  ```
  ▷ cast by replacing (A B human*) with (B child* B) in {family}
  ```

  Diese Spezifikation würde dann erst bei der Überprüfung durch die Compiler-Funktion zurückgewiesen werden.

Weitere Beispiele für diese Spezifikationsvariante wären

- attr2element by replacing (A B) with (A attr(L) B) in X: An der Stelle, an der ein dem Attribut L entsprechendes, neues Element eingefügt werden soll, wird der spezielle Ausdruck attr() angegeben. Das Problem ist hier, dass jeweils nur genau eine Ersetzung durchgeführt werden darf (da jedes Attribut nur einmal vorkommt) und dass das dem Attribut entsprechende Element auf der obersten Ebene (nicht innerhalb von beispielsweise Auswahloperatoren) vorkommen muss. Diese Einschränkung lässt sich mit dem jetzt gewählten Ansatz, das aus dem Attribut erzeugte Element als erste Silbe einzufügen, besser erzwingen.

- permute by replacing (A B C D) with (A C B D) in X
- generalize by replacing (A B C) with (A* B C D?): Hier lassen sich in einem Durchgang mehrere Verallgemeinerungen durchführen.
- element2attr by replacing AY with A in X: Der Ausdruck Y muss dabei ein Element sein, dass in ein Attribut umgewandelt werden kann. Es wird dadurch ausgewählt, dass es im Zielausdruck gelöscht ist. – Im Gegensatz dazu ist die jetzt gewählte Vorgehensweise deutlich expliziter.

3.7.4 Attributmodifikation

Die Modifikation von Attributen wird in den verschiedenen Quellen unterschiedlich gehandhabt. Meist werden keine speziellen Operationen für Attribute verwendet. Die Unterscheidung zwischen Elementen und Attributen geschieht dann analog zu Spezifikation in XPath [W3C99b] dadurch, dass Attribute durch ein spezielles Präfix („@") gekennzeichnet werden. In [BFRW01] werden Attribute zudem (in Form einer Konjunktion) mit dem Unterstruktur-Typ der Element-Definition zusammengefasst:

```
a[ ( t/@b & t/@c ), ( u/d | u/d/*/e ) ]
```

(Dieses Beispiel definiert den Typ von `a`. Attribute werden durch ein vorangestelltes `@` ausgezeichnet, die Sequenz wird durch ein Komma, die Konjunktion durch `( ... & ... )` und eine Auswahl mittels `( ... | ... )` notiert.)

Diese Methode ist zur vereinheitlichten Notation von Attribut-Spezifikation und Unterstruktur-Typ sicherlich sehr gut geeignet, bereitet aber Probleme, sobald dieser Ausdruck umgeformt werden soll, da die Attributspezifikation gegenüber dem Unterstruktur-Typ speziellen Einschränkungen unterliegt (konkret: jedes Attribut darf nur genau einmal, und zwar in einer Konjunktion vorkommen). In den vorliegenden Operationen wurde daher auf eine vereinheitlichte Modellierung verzichtet.

Zu erwähnen ist auch, dass in [LC00a] Format-Spezifikations-Formalismen diskutiert werden, die weiter gehende Constraints für Attributdeklarationen erlauben. Beispielsweise lassen sich in bestimmten Formalismen der gegenseitige Ausschluss von Attributen spezifizieren, das heißt „Das Element muss entweder Attribut A oder Attribut B enthalten".

3.7.5 Weitere Operationen

Es wären weitere Operationen denkbar, die hier jedoch aufgrund des hohen Aufwands bei Formalisierung und Implementation nicht berücksichtigt wurden. Beispiele sind:

- A+ B+ → (A B)+: Hier muss mit geeigneten Mitteln spezifiziert werden, welche Elemente jeweils gruppiert werden sollen. Außerdem müssten gewisse Randbedingungen geprüft werden, hier beispielsweise, ob A genauso häufig auftritt wie B.
- Generisches datenerhaltendes Umordnen: (A | B A) (D | F) → (D A | A) F? B? . Diese Ersetzung zu spezifizieren, ist mit den vorliegenden Operatoren aufwändiger:

extract (ϵ | B) A	$\Rightarrow$	(ϵ| B) A (D | F)
preserve (ϵ | B) as B?	$\Rightarrow$	B? A (D | F)
permute B? A (D | F) to 3,2,1	$\Rightarrow$	(D | F) A B?
distribute (D | F) A	$\Rightarrow$	(D A | F A) B?
permute F A to 2,1	$\Rightarrow$	(D A | A F) B?
generalize F to F?	$\Rightarrow$	(D A | A F?) B?
fold D A to Z	$\Rightarrow$	(Z | A F?) B?; Z $\rightarrow$ D A
generalize ϵ to F? in {Z}	$\Rightarrow$	(Z | A F?) B?; Z $\rightarrow$ F? D A
unfold Z; eliminate Z	$\Rightarrow$	(F? D A | A F?) B?
permute F? D A to 2,3,1	$\Rightarrow$	(D A F? | A F?) B?
extract (D A | A) F?	$\Rightarrow$	(D A | A) F? B?

Vorteil dieser Operation wäre die Kompaktheit der Spezifikation. Der entscheidende Nachteil der generischen Ersetzung ist der Aufwand bei Implementation und Formalisierung, beispielsweise müsste bei der Dokumenttransformation automatisch entschieden werden, welcher Zweig einer Auswahl im Zielausdruck zur Generierung der Zielelemente benutzt wird. Außerdem müsste eine Möglichkeit gefunden werden, zu prüfen, ob der Such-Ausdruck dieselben Basiselemente und Gültigkeitsbedingungen (beispielsweise spezifizieren Auswahl-Operatoren einen gegenseitigen Ausschluss der Ausdrücke in ihren Zweigen) enthält wie der Zielausdruck, was nicht trivial ist, da wegen der großen Anzahl der impliziten Transformationen zahlreiche Ausdrücke stark mutieren können (beispielsweise (A|ϵ) $\rightarrow$ A?).

- Verschmelzung von Ausdrücken (siehe Abschnitt 3.3.2, S. 29).
- Operationen auf den Datentypen von XMLSchema (da momentan außer einfachem Text keine weiteren Datentypen unterstützt werden, existieren auch keine entsprechenden Operationen).

3.8 XMLSchema Identitäts-Constraints

XMLSchema ermöglicht es durch die sogenannten Identitäts-Constraints, Eindeutigkeit und Schlüssel-Eigenschaften von bestimmten Werten innerhalb eines Dokumentes zu sichern. Da der Aufwand für die Formalisierung und Implementierung dieser Identitäts-Constraints nicht unbeträchtlich ist, wurden sie in dieser Arbeit nicht weiter berücksichtigt. Da sie aber in bestimmten Anwendungen eine gewichtige Rolle spielen, soll in diesem Abschnitt zumindest eine Diskussion der Auswirkungen der vorgestellten Operationen auf die Identitäts-Constraints erfolgen.

Zunächst werden Identitäts-Constraints kurz charakterisiert. Dann folgt die allgemeine Erläuterung bestimmter auftretender Probleme. Der Abschnitt schließt mit einer je-Operation-Diskussion der Wechselwirkung mit Identitäts-Constraints.

3.8.1 Identitäts-Constraints in XML Schema

Die Identitäts-Constraints (*identity constraints*) von XML Schema modellieren gewisse Abhängigkeiten zwischen Element- und/oder Attributwerten und stellen eine Verallgemeinerung des `ID`/`IDREF`-Konzeptes aus XML dar. Gegenüber `ID`/`IDREF` können nicht nur Attribute verwendet werden, deren Typ `ID` bzw. `IDREF` ist, sondern Tupel aus beliebigen Attributen und Elementen (diese Elemente müssen jedoch von einfachem Typ sein, das bedeutet, dass sie keine Unterelemente besitzen können). Hinzu kommt, dass die Werte aller `ID`-Attribute innerhalb des gesamten Dokumentes eindeutig sein müssen, während es in XMLSchema möglich ist, mehrere solcher Constraints zu deklarieren und deren jeweiligen Gültigkeitsbereich auf den Unterbaum eines bestimmten Elementes zu beschränken. Es existieren drei Arten von Identitäts-Constraints:

- `unique`: Fordert, dass das der Wert des Tupels aus den beteiligten Attributen/ Elementen (wenn für diese ein Wert, ggf. nach dem Einsetzen des Defaults, existiert) im Gültigkeitsbereich der Bedingung eindeutig sein muss.
- `key`: Wie `unique`, allerdings wird hier zusätzlich die Existenz eines solchen Tupels in jedem Fall gefordert.
- `keyref`: Für jedes `keyref`-Tupel muss genau ein korrespondierendes `key`-Tupel existieren.

Das folgende Beispiel (aus [W3C01a]) definiert das Format einer nach Regionen aufgeschlüsselten Verkaufsbilanz. Diese Bilanz enthält eine Liste von Beschreibungen der bilanzierten Waren `parts` sowie eine Aufstellung `regions`, wieviele dieser Waren in den einzelnen Regionen vertrieben wurden. Das Beispiel wurde der Übersichtlichkeit wegen stark gekürzt.

In dieser Aufstellung existieren nun gewisse Schlüsselbeziehungen und Eindeutigkeits-Bedingungen. Diese wurden durch entsprechende Identitäts-Constraint-Definitionen ausgedrückt:

- Jede Waren-Nummer, die in der Bilanz aufgeführt wird, muss einen entsprechenden Eintrag in den Waren-Beschreibungen haben (Element `<keyref name="kr1" refer="partNumberKey">`).
- Keine der Regionen (die über ihre Postleitzahlen identifiziert werden) darf in der Bilanz mehrmals auftreten. Diese Bedingung wird mit dem Element `<unique name="u1">` spezifiziert.
- Die Waren-Nummer ist ein Schlüssel für die Waren-Beschreibung, das bedeutet, dass sie eindeutig sein und in jedem Beschreibungs-Element auftreten muss. Dieses wird in dem `<key name="partNumberKey">`-Element definiert.

Der Eindeutigkeits-Raum wird durch das Elter-Element der `unique`- und `key`-Definitionen definiert. Hier müssen die betreffenden Werte also jeweils innerhalb eines `purchaseReport`-Elements eindeutig sein. Kommen in einem Dokument also mehrere `purchaseReport`-Einträge vor, können dieselben Schlüsselwerte im Dokument mehrfach vorkommen - höchstens einmal je `purchaseReport`.

```
<element name="purchaseReport">
:
      <element name="regions" type="RegionsType">

        <keyref name="kr1" refer="partNumberKey">
          <selector xpath="zip/part"/>
          <field xpath="@number"/>
        </keyref>
      </element>

      <element name="parts" type="PartsType"/>
:
  <unique name="u1">
    <selector xpath="regions/zip"/>
    <field xpath="@code"/>
  </unique>

  <key name="partNumberKey">
    <selector xpath="parts/part"/>
    <field xpath="@number"/>
  </key>
</element>
```

```
<complexType name="RegionsType">
:
    <element name="zip" maxOccurs="unbounded">
    :
          <element name="part" maxOccurs="unbounded">
          :
                  <attribute name="number" type="xipo:SKU"/>
          :
          </element>
    :
        <attribute name="code" type="positiveInteger"/>
    :
    </element>
:
</complexType>

<complexType name="PartsType">
:
    <element name="part" maxOccurs="unbounded">
    :
          <extension base="string">
            <attribute name="number" type="xipo:SKU"/>
          </extension>
    :
    </element>
:
</complexType>
```

3.8.2 Die Fokussierungs-Problematik

Grundsätzlich entsteht durch die Fokussierung der Transformation ein Problem: Die Identitäts-Constraint muss über die (syntaktisch allerdings unter Umständen verschiedenen) Stellen hinweg gültig sein. Da für verschiedene Syntax aber auch verschiedene Deklarationen nötig sind, und die betreffende Identitäts-Constraint (Schlüssel/Eindeutigkeit) nur innerhalb der von dieser Deklaration erfassten Elemente, Attribute bzw. ihren Kombinationen gilt, sind fokussierte Transformation und die Identitäts-Constraints von XML Schema prinzipiell unverträglich. Allerdings ist die Transformation unter folgenden Bedingungen trotzdem zulässig:

- die transformierten Elemente/Attribute sind an keiner Identitäts-Constraint beteiligt, oder
- wenn Elemente/Attribute an einer Identitäts-Constraint beteiligt sind, dann müssen *alle* Auftreten transformiert werden, und die Transformation darf den Typ nicht einer solchen Art und Weise modifizieren, dass die Werte dieses Typs dann die Identitäts-Constraint verletzen können (oder gar müssen).

3.8.3 Auswirkungen von fixierten und Default-Werten

Defaultwerte sind für Elemente/Attribute, die `unique`- und `key`-Bedingungen beteiligt sind, nicht nützlich:

- Falls für mehrere `key`-Tupel im Dokument *alle* beteiligten Elemente/Attribute auf ihren Defaultwert zurückgreifen, ist die Eindeutigkeitseigenschaft des Schlüssels verletzt – da der Defaultwert in diesem Fall immer derselbe ist, sind es damit auch die Schlüsselwerte.

- Wenn einzelne Elemente/Attribute solcher Tupel im gesamten Dokument ihren Defaultwert *niemals* überschreiben, sind sie für die Schlüsseleigenschaft bedeutungslos und könnten weggelassen werden: Falls die Schlüssel in solchem Fall trotzdem eindeutig sind, so liegt das an den anderen Bestandteilen der Tupel.

Aus dem letztgenannten Grund sind fixierte Elemente/Attribute in `unique`- und `key`-Deklarationen überflüssig, weil das `FIXED`-Flag ja gerade erfordert, dass sie immer nur mit demselben Wert auftreten.

3.8.4 Anpassung der Pfade

Durch die Transformationen wird es nötig, die in den Identitäts-Constraint-Deklarationen vorhandenen Bezüge auf die transformierten Elemente zu ändern. Zum Beispiel müssen bei unfold und fold Pfade und bei rename Namen angepasst werden. Im Dokument selbst sind keine Änderungen nötig.

Bei unfold kann es zu Problemen kommen, wenn der Gültigkeitsbereich der entsprechenden Identity-Constraint eben das entfaltete Element war. Eine implizite Ausdehnung des Gültigkeitsbereiches auf das neue Elter-Element wäre nicht semantikerhaltend: Falls beispielsweise das später entfaltete Element zweimal im übergeordneten Element verwendet wurde, konnten seine Schlüsselfelder in beiden Auftreten gleich sein, ohne dass eine `unique`-Constraint verletzt war. Würde der Gültigkeitsbereich auf das übergeordnete Element ausgedehnt werden, würde das die Verletzung der Constraint bedeuten. Daher sollte diese Transformation nicht möglich sein, wenn das zu entfaltende Element den Gültigkeitsbereich einer Identitäts-Constraint bildet.

3.8.5 Generalisierung und Restriktion

Da Schlüssel die Existenz der an ihnen beteiligten Elements/Attribute erfordern, muss das betreffende Element/Attribut obligatorisch sein. In jedem Fall fordert die Spezifikation von XMLSchema, dass die in Identitäts-Constraints verwendeten Felder nur höchstens einmal auftreten dürfen – Listen von Elementen (`star`- und `plus`-Quantor) können in diesem Fall also nicht verwendet werden.

Ein Element, das an Identitäts-Constraints beteiligt ist, wird also in den meisten Fällen genau einmal auftreten müssen. Für `unique` und sich auf eine `unique`-Constraint beziehende `keyref` kommen zusätzlich noch `optional`e Elemente in Frage.

Diese Einschränkung ist bei Generalisierungs- und Restriktions-Transformationen, die Elemente betreffen, die an Identitäts-Constraints beteiligt sind, in solchen Fällen zu beachten (siehe Abschnitt 3.8.6, S. 48).

3.8.6 Diskussion für die einzelnen Operationen

fold, unfold In beiden Fällen ist das Ändern der Pfade zu den in Identitäts-Constraint-Definitionen nötig. Probleme können entstehen, wenn durch die Fokussierung der Operation Eindeutigkeits-Bereiche zerteilt werden. unfold erfordert es zudem, dass das zu entfaltende Element nicht Gültigkeitsbereich eines Identitäts-Constraints ist.

permute, distribute, extract, preserve Diese Operationen berühren Integritäts-Constraints nicht.

generalize Hier muss die Existenz und Eindeutigkeit der in Identitäts-Constraints benutzten Felder gesichert werden. Das bedeutet, dass die obere Grenze der

Auftretens-Anzahl solcher betroffenen Elemente nicht über Eins erhöht werden. Für `key`-Elemente ist es zusätzlich erforderlich, die untere Grenze nicht auf Null zu senken.

restrict Da diese Operation voraussetzt, dass das Dokument bereits dem neuen Typ entspricht, ist hier keine direkte Relevanz für die Identitäts-Constraints vorhanden. Zu beachten wäre ansonsten der Fall des Löschens von Elementen. Dieses wird für `editData` diskutiert.

cast Bei dieser Operation ist es wiederum nötig, Bezüge auf die geänderten Elemente in den Identitäts-Constraint-Definitionen anzupassen. Hinzu kommt die Fokus-Problematik. Ein weiteres Problem ist, dass durch diese Operation unter Umständen zwei verschiedene Schlüssel-Räume verschmolzen werden, so dass bisher gültige Schlüsselwerte nun doppelt vorkommen. In diesem Fall muss die Transformation scheitern.

editData Hier ist es nötig, dass das transformierte Dokument noch sämtliche Constraints erfüllt. Speziell tritt dieses Problem im Zusammenhang mit dem Löschen von Einträgen auf, da dadurch Eindeutigkeit unerfüllbar werden kann, falls das gelöschte Element bei ansonsten gleichen Schlüssel-Werten für den Unterschied zweier Schlüssel sorgte. Ebenfalls nicht zulässig ist es, Schlüsselwerte zu löschen, die anderer Stelle aus referenziert werden (`keyref`).

setFixed Das Löschen berührt die Identitäts-Constraints nicht. Ein Element oder Attribut zu fixieren, darf jedoch nur gestattet werden, wenn dieses weder in `key`- noch `unique`-Definitionen verwendet wird, das heißt die Eindeutigkeit der Werte keine Rolle spielt.

setDefault Die Änderung des Default-Wertes eines Attributes/Elementes, das Teil einer `keyref` ist, ist nur zulässig, wenn ein dem neuen Wert entsprechender `key`-Wert existiert.

element2attr, attr2element Diese beiden Operationen erfordern gegebenfalls die Anpassung der Bezüge aus Identitäts-Constraints auf das umgewandelte Element/Attribut. Auch hier besteht die Gefahr, dass Eindeutigkeits-Räume zerteilt werden, da die Operationen fokussiert sind.

introduce Da das eingefügte Element neu im Format ist, hat diese Operation keine Auswirkungen auf Identitäts-Constraints.

eliminate Diese Operation kann nur Elementdefinitionen aus dem Format entfernen, die im Format (und somit auch in den Dokumenten) nicht verwendet werden. Daher berührt diese Operation keine Identitäts-Constraints.

rename Da diese Operation global abläuft, besteht keine Gefahr, dass Eindeutigkeits-Räume zerteilt werden. Die Element-Umbenennung muss jedoch auch in den Identitäts-Constraints durchgeführt werden.

Kapitel 4

Implementation

4.1 Überblick

$$\begin{array}{l}\textsf{XMLSchema}\\ \textsf{(XML)}\end{array} \stackrel{\text{XML}-\text{Parser}}{\Longrightarrow} \begin{array}{l}\textsf{XMLSchema}\\ \textsf{(Terme)}\end{array} \stackrel{\text{Normalisierung}}{\Longrightarrow} \textsf{Schema} \stackrel{\text{Trafo}}{\Longrightarrow} \begin{array}{l}\textsf{Schema}\\ \textsf{(transformiert)}\end{array}$$

$$\begin{array}{l}\textsf{XML-Datei}\\ \textsf{(Text)}\end{array} \stackrel{\text{XML}-\text{Parser}}{\Longrightarrow} \textsf{Dokument} \stackrel{\text{Trafo}}{\Longrightarrow} \begin{array}{l}\textsf{Dokument}\\ \textsf{(transformiert)}\end{array}$$

Abbildung 4.1: Transformationsschritte für Format- bzw. Dokumenttransformation

Nach dem Parsen mit dem in SWI-Prolog 4.0.3 enthaltenen XML-Parser wird bei den Format-Dokumenten eine Normalisierung vorgenommen, um sie in das dann im folgenden verwendete Format zu bringen. Der nächste Abschnitt beschreibt diese Normalisierung.

Weder die Gültigkeit (das bedeutet, die Konformität mit den entsprechenden Spezifikationen) des Formats noch der Dokumente wird geprüft. Es wird vorausgesetzt, dass diese korrekt sind. Der Test, ob die Dokumente dem Format entsprechen, wird bei der Transformation der Dokumente durchgeführt.

Der Interpreter des Trafo-Ausdrucks zur Dokument-Transformation ist als Parser mit prozeduralem Abstieg implementiert. Hierbei werden die Backtracking-Möglichkeiten von Prolog genutzt.

Eine Ausgabe der transformierten Ergebnisse nach XML ist zunächst nicht vorgesehen. Ebenso existiert keine nutzerfreundliche Möglichkeit, die Transformationen zu spezifizieren. Hier ist die manuelle Erstellung entsprechender Strukturen in Prolog nötig.

4.2 Schema-Normalisierung

Die Implementierung basiert auf den den hier formalisierten Operationen. Diese benutzen allerdings nur eine Teilmenge von XMLSchema. Daher ist es zunächst nötig, die XMLSchema-Spezifikation auf die interne Format-Repräsentation abzubilden, sofern das möglich ist (Abbildung 4.1). Der entsprechende Abbildungsalgorithmus führt dabei unter anderem folgende Aktionen durch:

- Normalisierung der Element-Identifikatoren (das bedeutet, lokal definierte Elemente durch Voranstellen des Identifikators des Elter-Elements eindeutig zu machen).

- Erzeugung des Startsymbols und einer entsprechenden Deklaration, die eine Auswahl über die eigentlichen Startsymbole des Formats ist.
- Prüfen, ob nicht unterstützte Definitionen verwendet werden. Werden solche gefunden, erfolgt der Abbruch der Verarbeitung.

Der folgende Tabelle gibt einen Überblick über nicht unterstützte Konstrukte von XMLSchema:

- `<annotation>`
- globale benannte Typ-Definitionen
- globale Modelgroup-Definitionen
- globale Attribut- und Attributgruppen-Definitionen
- Definitionen, die mit Typ-Ableitung zusammenhängen
- Identitäts-Constraint-Definitionen
- Attribut-Wildcards (`any` and `anyAttribute`)

Die folgenden Konstrukte werden ignoriert, ohne dass die Verarbeitung abbricht:

- „`prohibited`“-Wert des „`use`“-Parameters in Attribut-Deklarationen wird als „`optional`“ betrachtet.
- `simpleType`-Definitionen werden ignoriert. Der Typ wird als Text angenommen.
- Kardinalitäten werden auf eine der folgenden „gerundet“: `0..1`, `0..unbounded`, `1..1` und `1..unbounded`, so dass sie sich mit den Quantoren `?`, `*` und + darstellen lassen: jede von Null verschiedene untere Grenze wird als Eins betrachtet, und jede obere Grenze größer als Eins als `unbounded`.
- Das `nillable`-Attribut von Element-Deklarationen wird ignoriert (als „`false`“ betrachtet).
- MixedContent wird nicht unterstützt, der „`mixed`“-Parameter von Typdefinitionen wird als „`false`“ betrachtet.

Literaturverzeichnis

[Alt00] Murray Altheim. Review of XML Schema specification, 2000. `http://www.doctypes.org/spec/schema-review-1.html`.

[B$^+$00] M. G. Burke et al. XML transformation: Matching and reconciliation, 2000. `http://www.research.ibm.com/hyperspace/mr/`.

[Bal98] Helmut Balzert. *Lehrbuch der Softwaretechnik, Teil 2: Softwaremanagement, Software-Qualitätssicherung, Unternehmensmodellierung.* Spektrum Akademischer Verlag, Heidelberg, Berlin, 1998.

[BEGO71] Ronald Book, Shimon Even, Sheila Greibach, and Gene Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20(2):149–154, Feb 1971.

[BFRW01] Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. MSL: A model for W3C XML Schema. In *International World Wide Web Conference (WWW)*, Hong Kong, 2001. `http://www.research.avayalabs.com/user/wadler//topics/xml.html`.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry 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.

[BKW91] Anne Brüggemann-Klein and Derick Wood. Deterministic regular languages. Technical Report 38, 1991.

[BKW98] Anne Brüggemann-Klein and Derick Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998. 140(2): 229–253.

[BMN00] Geert Jan Bex, Sebastian Maneth, and Frank Neven. A formal model for an expressive fragment of XSLT. In J. Lloyd et al., editors, *Computational Logic – CL 2000, First International Conference*, number 1861 in LNCS, pages 1137–1151. Springer-Verlag, 2000.

[Boe81] Barry W. Boehm. *Software Engineering Economics.* Prentice Hall, Englewood Cliff, New Jersey, 1981.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(2):61–72, May 1988.

[CS00] S. Cluet and J. Siméon. YATL: a functional and declarative language for XML. Technical report, 2000.

[Dec00] DecisionSoft, Ltd. *XML Script tutorial*, 2000. `http://www.xmlscript.org/docs/Tutorial.Contents.html`.

[Fow99] Martin Fowler. *Refactoring—Improving the Design of Existing Code.* Addison Wesley, 1999.

[Fre67] Gottlob Frege. Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought (1879). In *From Frege to Godel: A Source Book in Mathematical Logic, 1879-1931*, Cambridge, Massachusetts, 1967. Harvard University Press, Jan van Heijenoort.

[GKM95] H. Gall, R. Klösch, and R. Mittermaier. Object-oriented re-architecturing. In *5th European Software Engineering Conference (ESEC '95)*, number 989 in LNCS, pages 499–519, Barcelona, Spain, 1995. Springer-Verlag.

[HP00] Haruo Hosoya and Benjamin Pierce. XDuce: A typed XML processing language (preliminary report). In *WebDB Workshop*, 2000.

[HP01] Haruo Hosoya and Benjamin Pierce. Tree automata and pattern matching. In *Principles of Programming Languages (POPL)*, 2001.

[HTJ+93] 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.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *ACM SIGPLAN Notices*, volume 35, pages 11–22, Montreal, Canada, 2000. ACM.

[ISO86] International Organization for Standardization. *ISO 8879:1986 Information processing – Text and office systems – Standard Generalized Markup Language (SGML)*, 1986.

[Läm01] Ralf Lämmel. Typed Generic Traversals in S'_γ. Technical Report SEN-R0122, CWI, August 2001.

[LC00a] Dongwon Lee and Wesley W. Chu. Comparative analysis of six XML schema languages. *ACM SIGMOD Record*, 29(3), 2000.

[LC00b] Dongwon Lee and Wesley W. Chu. Constraints-preserving transformation from XML document type definition to relational schema. In *Proc. 19th International Conference on Conceptual Modeling (ER)*, number 1920 in LNCS, Salt Lake City, Utah, 2000. Springer-Verlag.

[LL01] R. Lämmel and W. Lohmann. Format evolution. In *Proc. 7th International Conference on Reverse Engineering for Information Systems (ReTIS 2001)*, Lyon, France, 2001. Austrian Computer Society, OCG.

[Läm01a] R. Lämmel. Grammar adaption. In *POPL'01. Proc. Formal Methods Europe (FME) 2001*, number 2021 in LNCS. Springer-Verlag, 2001.

[Läm01b] Ralf Lämmel. Generic sort-preserving traversal strategies. In *International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, Utrecht, The Netherlands, 2001. `http://www.logic.at/wrs01/main-contents.html`.

[LMM00] Dongwon Lee, Murali Mani, and Makoto Murata. Reasoning about XML schema languages using formal language theory. Technical Report RJ#10197, Log#95071, IBM Almaden Research Center, 2000. `http://www.cs.ucla.edu/~dongwon/paper/by-year.html`.

[LR01] Ralf Lämmel and Günter Riedewald. Prological Language Processing. In Mark van den Brand and Didier Parigot, editors, *Proc. Workshop on Language Descriptions, Tools and Applications – LDTA'2001*, volume 44 of *Electronic Notes in Theoretical Computer Science ENTCS*. Elsevier Science, April 2001.

[MB00] David McKelvie and Ken Beesley. *XMLPERL 1.7.2 manual*, 2000. `http://www.cogsci.ed.ac.uk/~dmck/xmlperl/xmlstuff/xmlperl/index.html`.

[MN00] Sebastian Maneth and Frank Neven. Structured document transformations based on XSL. In R. Conner and A. Mendelzon, editors, *Research Issues in Structured and Semistructured Database Programming - Revised Papers DBPL'99*, number 1949 in LNCS, pages 80–98. Springer-Verlag, 2000.

[Mur00] Makoto Murata. How to RELAX. Technical report, 2000. `http://www.xml.gr.jp/relax/html3/howToRELAX\_full\_en.html`.

[OAS01] OASIS. *DocBook 4.1*, 2001. `http://www.oasis-open.org/specs/docbook.shtml`.

[Pre01] Paul Prescod. A proposal for the creation of W3C-recommended transformation language, 2001. `http://www.prescod.net/xsl/petition/`.

[vNG01] Gertjan van Noord and Dale Gerdemann. Finite state transducers with predicates and identities, 2001. `http://grid.let.rug.nl/~vannoord/papers/preds.pdf`.

[W3C99a] World Wide Web Consortium (W3C). *HTML 4.01 Specification*, 1999. `http://www.w3.org/TR/1999/REC-html401-19991224`.

[W3C99b] World Wide Web Consortium (W3C). *XML Path Language (XPath) Version 1.0*, 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[W3C99c] World Wide Web Consortium (W3C). *XML Transformations (XSLT) Version 1.0*, 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`.

[W3C00] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0 (Second Edition)*, 2000. `http://www.w3.org/TR/2000/REC-xml-20001006`.

[W3C01a] World Wide Web Consortium (W3C). *XML Schema Part 0: Primer*, 2001. `http://www.w3.org/TR/2001/PR-xmlschema-0-20010330/`.

[W3C01b] World Wide Web Consortium (W3C). *XML Schema Part 1: Structures*, 2001. `http://www.w3.org/TR/2001/PR-xmlschema-1-20010330/`.

[W3C01c] World Wide Web Consortium (W3C). *XML Schema Part 2: Datatypes*, 2001. `http://www.w3.org/TR/2001/PR-xmlschema-2-20010330/`.

[WR99] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *International Conference on Functional Programming*, pages 148–159, 1999.

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 5. September 2001

Thesen

Format-Evolution lässt sich durch Software unterstützen Format-Evolution, also die Weiterentwicklung eines Dateiformats, lässt sich für wichtige Fälle formalisieren und durch automatische Methoden unterstützen. In der vorliegenden Arbeit wurde dazu eine Menge von Operationen entwickelt, diese formal definiert und in PROLOG implementiert.

Induzierte Dokumenttransformation Die vorgestellten Operationen spezifizieren Änderungen auf dem Dateiformat. Existierende Dokumente lassen sich automatisch an das geänderte Format anpassen. Der Konvertierungsalgorithmus ergibt sich dabei direkt aus der angewendeten Operation.

Allgemeingültigkeit Die Implementierung verwendet XML als Rahmen für die Dokumente und XMLSchema als Spezifikationsformalismus für das Dateiformat. Die entwickelten Operationen sind jedoch hinreichend allgemein, dass sie auch außerhalb dieses Kontexts gültig sind.