Deklarative Spezifikation von Datentransformationen

Mathias Weber

15. Juni 2012

Technische Universität Kaiserslautern
Fachbereich Informatik
Interner Bericht 390/12

Inhaltsverzeichnis

1 Motivation **3**

2 Anforderungen aus der Praxis **4**
2.1 Einschränkungen durch das Schema 4
2.2 Mappings 5
2.2.1 Komplexe Muster 5
2.2.2 Anwendungsfälle 7

3 Logik-Ansatz **9**
3.1 XPathLog 9
3.2 Vorteile der Logik Programmierung für die Transformation 10
3.3 Probleme des Ansatzes 11

4 Deklarativer Ansatz **11**
4.1 Grundlagen der Transformation 11
4.2 Voraussetzungen 12
4.3 Schema 13
4.4 Grundideen des Mappings 14
4.4.1 get-Phase 15
4.4.2 filter-Phase 16
4.4.3 transform-Phase 17
4.4.4 put-Phase 17
4.5 Syntax 18
4.5.1 Problem der Adressierung 19
4.5.2 Umstrukturierung 25
4.5.3 Spezialfall XML Dokumentordnung 27
4.5.4 Mögliche Erweiterung für Aggregation 32
4.6 Syntaktische Erweiterungen (syntactic sugar) 34

5 Analyse **35**
5.1 Prüfung auf Vollständigkeit 36
5.2 Semantik 38

6 Fazit **39**
6.1 Gelöste Probleme 39
6.2 Ungelöste Probleme 40
6.3 Implementierung 41

1 Motivation

Im Business-to-Business Bereich findet eine weite Verschiebung von der papierbasierten auf die digitale Kommunikation statt. Daher existiert immer mehr der Bedarf, Daten über Unternehmensgrenzen hinaus zu kommunizieren. Dabei gibt es verschiedene Formate für den Austausch und die Datenhaltung. Oft ist es der Fall, dass Firmen, die Daten im Rahmen einer Business-to-Business Beziehung austauschen müssen, verschiedene Formate benutzen. Die Unterschiede können dabei relativ einfache Umbenennung, strukturell unterschiedliche Darstellung, oder sogar komplett andere Repräsentation von Daten sein. Somit stellt sich die Frage, mit Hilfe welcher Beschreibungsform die Transformation realisiert werden soll.

Grundsätzlich immer möglich ist es, die Transformation mit Hilfe einer universellen Programmiersprache zu beschreiben. In diesem Umfeld kommt es hauptsächlich darauf an, in wie weit bereits fertige Bibliotheken für standardisierte Formate, wie zum Beispiel XML oder EDIFACT, bestehen und in wie weit diese für die Transformation verwendet werden können und sollen. Jedoch ist ein Problem dieses Ansatzes, dass die Beschreibung in einer Form erfolgt, die nicht auf die Bedürfnisse des aktuellen Anwendungsfeldes, hier der Transformation zwischen zwei gegebenen Schemata, angepasst ist. Somit muss auf sehr generische Methoden zurückgegriffen werden, die nicht spezielle Vorbedingungen oder Eigenschaften, hier die gegebenen Schemata, mit verwenden, um die Integrität der Transformation zu unterstützen.

Neben den universellen Programmiersprachen, wie zum Beispiel Java, gibt es auch die Möglichkeit, domänenspezifische Sprachen für die Transformation einzusetzen. Die Vorteile hierfür sind darin zu finden, dass die Sprache in ihrer Mächtigkeit auf das aktuelle Anwendungsfeld angepasst werden und dadurch Wissen über die konkrete Anwendung in das Design und die Semantik einfließen kann. Durch genügende Einschränkung der Mächtigkeit kann so eventuell sogar erreicht werden, dass man bestimmte Eigenschaften statisch garantieren kann und dadurch der Programmierer, der eine Transformation beschreiben soll, unterstützt wird. Außerdem können offensichtliche Fehler reduziert oder sogar in bestimmten Fällen ausgeschlossen werden.

Eine Transformation zwischen den Daten muss zuverlässig sein, die Semantik der Daten vor und nach der Transformation muss die gleiche sein. Damit stellt sich die Frage, wie diese Semantik formal gefasst werden kann. In der Praxis werden für ein Datenformat Regeln aufgestellt, denen die Daten gehorchen müssen. Diese werden als Einschränkungen (Constraints) bezeichnet und bilden zum Beispiel Gültigkeitsbereiche eines Datums oder Zusammenhänge zwischen Daten ab. Die Gesamtheit aller Einschränkungen bilden ein Schema (Guideline), dem die Daten gehorchen müssen. Somit stellen sich bei der Transformation zwei Probleme. Zum einen müssen die Daten vor der Transformation auf deren Konformität mit den Ausgangsschema geprüft werden. Zum anderen müssen die Daten nach der Transformation dem Zielschema entsprechen. Beide Prüfungen können live während der Transformation erfolgen. Allerdings entsteht dabei zur Laufzeit ein erheblicher Overhead, welcher in der Praxis nicht tragbar ist. Somit soll die Prüfung der

Konformität mit dem Zielschema nicht auf den Daten erfolgen, sondern eine Eigenschaft der Transformation, also statisch garantiert, sein.

Die Voraussetzungen für die statische Prüfung der Einschränkungen sind

- Die Formalisierung der Einschränkungen der Daten in abstrakter Form mit Sprachmitteln
- Die Beschreibung der Transformation in formaler Form eines Programms

Will man aussagekräftige Ergebnisse aus einer solchen statischen Analyse erhalten, so muss man die dem Programmierer zur Verfügung stehenden Mittel entsprechend erweitern und die Freiheiten einschränken, sodass Aussagen zur Übersetzungszeit möglich sind. Die größte Hürde hierbei ist, dass viele der Eigenschaften, die man statisch garantieren möchte, im Allgemeinen erst zur Laufzeit mit konkreten Daten entscheidbar sind. So möchte man bei der Auswahl von Daten, die auf einen Pfad des Zieldokuments geschrieben werden sollen, allgemeine boolesche Ausdrücke über alle Daten des Eingabedokuments erlauben. Jedoch ist schon dadurch die Menge der selektierten Daten unentscheidbar.

In dieser Arbeit geht es darum, eine Sprache für die Beschreibung von Transformationen zwischen zwei Schemata, sowie entsprechend passende Beschreibungssprachen für Schema und Einschränkungen, zu finden, die möglichst weitgehende statische Prüfungen erlaubt und somit den Programmierer in vielen Punkten unterstützt. Dabei soll die Sprache den Programmierern beim Verständnis der Transformation unterstützen und das „Was" statt dem „Wie" beschreiben, weshalb ein deklarativen Ansatz sinnvoll erscheint. Es soll untersucht werden, welche Probleme gelöst werden müssen, um dabei sowohl die Anforderungen der Praxis zu bewältigen und gleichzeitig als Basis für eine statische Prüfung Informationen über die Transformation gewinnen zu können.

2 Anforderungen aus der Praxis

2.1 Einschränkungen durch das Schema

Wie schon in der Einleitung erwähnt gibt es in der Praxis Einschränkungen, was die Form und Darstellung von Daten betrifft. Diese Einschränkungen nehmen zwei verschiedenen Formen an. Zum einen können sie die Struktur der Darstellung beschreiben. In XML wäre dafür eine Beschreibung in XML Schema oder DTD üblich. Zum anderen können Einschränkungen der Werte gegeben sein, die unter anderem die Darstellung der konkreten Daten regeln. Ein Beispiel hierfür wäre die Darstellung einer Postleitzahl, die bekanntlich fünf Stellen zwischen 0 und 9 besitzt. Der Einfachheit halber kann man annehmen, dass jeder Wert als Typ eine Ganzzahl, eine Gleitkommazahl oder ein String hat. Alle elementaren Typen, die nicht explizit unterstützt werden, zum Beispiel ein Datum oder eine Postleitzahl, sind als String darstellbar. Als Einschränkung kann eine Liste von zulässigen fixen Werten, sowie für Ganz- und Gleitkommazahlen als häufigste Einschränkung Wertebereiche und für Strings aufgrund ihrer günstigen Eigenschaften

reguläre Ausdrücke angegeben werden. Dadurch lassen sind viele Fälle in der Praxis abdecken, unter anderem auch der Fall der Postleitzahl, die als regulärer Ausdruck mit „[0-9]{5}" beschrieben werden kann.

2.2 Mappings

Durch Zusammenarbeit mit der Firma Seeburger haben sich immer wiederkehrende Muster und Anwendungsfälle herausgebildet, die für eine Transformation im Geschäftssektor typisch sind. Um eine für die Praxis relevante Sprache zu erhalten, müssen diese Fälle damit realisierbar sein. Dazu sollte es relativ einfach möglich sein, diese Muster und Anwendungsfälle in der Sprache auszudrücken.

2.2.1 Komplexe Muster

Im Folgenden habe ich mich mit dem Anforderungsdokument von Seeburger beschäftigt. In Klammern habe ich dabei jeweils angegeben, auf welches Problem oder auch Muster ich mich damit beziehe.

Häufig kommt es vor, dass die Zuordnung auf der Zielseite durch einen oder mehrere Kennzeichner bestimmt wird (Combined Key Mapping). Dabei können Kennzeichner auch bestimmen, zu welchem Element auf der Zielseite abgebildet wird (Qualifier-to-field) oder welche Operation mit den Daten vorgenommen werden muss (Qualifier determines String-Operation und Key-based String-Operation). Eine wichtige Aufgabe eines Mappings ist es, für jedes Element einer Menge von gleichartigen Elementen ein Ziel anzugeben. Dabei wird in vielen Sprachen eine Art von Schleife verwendet (For-each). Eine besondere Herausforderung stellt dabei die Zuordnung der einzelnen Elemente auf der Zielseite dar (z. B. auch Determine Segment by Target-Lookup). Dazu in folgenden Kapiteln mehr. Jedoch kann eine Menge von Elementen der Quellseite auch aggregiert werden. Ein wichtiges Beispiel hierfür ist die Verarbeitung von mehreren Zahlen zu einer Summe oder das Bestimmen der Anzahl von Elementen (Counter).

Ein weiterer wichtiger Punkt bei der Definition von Mappings ist die String-Verarbeitung. Da nicht alle Daten mit einem entsprechenden und für die Bedürfnisse dieses speziellen Falls angepasste Datentyp versehen sind, ist die Repräsentation solcher Informationen als String geläufig. Dabei sind häufig verwendete Operationen das Einteilen eines Strings in Substrings, eventuell getrennt durch ein bestimmtes Trennzeichen, das Entfernen von Leerzeichen sowie das Zusammensetzen eines komplexeren Strings aus mehreren Substrings. Diese Operationen können entweder als Funktionen in die Sprache aufgenommen werden oder als Vorverarbeitung vorgenommen werden, soweit die Aufteilung einem festen Muster folgt. Es kommt in der Praxis häufig vor, dass ein String-Feld mehrere Teilinformationen enthält, wie ein Label, eine Nummer und eine Checksum. Diese Teilinformationen haben eine feste Struktur und sind somit in der Vorverarbeitung aufteilbar. Somit ist beim Schreiben der Mappings keine oder nur elementare Verarbeitung, wie Konkatenation, der Felder nötig.

Eine besondere Herausforderung stellen komplexe strukturelle Änderungen dar. Insbesondere in diese Kategorie fallen Änderungen, die Informationen im Schema in die Daten projizieren oder Daten zu Informationen im Schema machen (Sub-Schema I, II und III). So können anhand von Qualifieren verschiedene Schemata instantiiert werden, genau so können aber auch Daten aus verschiedenen Schemaelementen zu gleichförmigen Elementen im Ziel abgebildet werden. Ein besonderer Aspekt im letzten Fall ist dabei, dass erkannt werden muss, ob es sich um das selbe Datum handelt und somit diese Elemente mit gleicher Information zu nur einem Element zusammengefasst werden müssen (Merging Parts).

Bei der Analyse der Anforderungen wurde klar, dass es neben den oben genannten nachzuvollziehenden Mustern auch Anforderung als wichtig angesehen werden, die nicht Teil einer Mappingsprache sein sollten. Als eine dieser Anforderungen ist zu nennen, dass komplexe Strukturen aufgebaut und zwischengespeichert werden können sollen (Structured Buffer). Dies ist klar ein Teil der Umsetzung und sollte auf dieser Ebene nicht Teil der Beschreibung des Mappings sein. Eine weitere Anforderung, die in diese Richtung geht ist die Kontrolle darüber, ob ein so erzeugtes Dokument oder ein Teil des Dokuments überhaupt auf der Zielseite geschrieben wird (Control Output I). Durch die extreme Freiheit, die sich durch die Möglichkeit der manuellen Speicherverwaltung bietet entstehen oft ungeahnte Fehlerquellen. Teile von Ausgabedaten, die als temporär erstellt und später nicht aufgeräumt wurden, können zu erheblichen Speicherproblemen führen. Außerdem leidet so die Analysierbarkeit erheblich, da nicht zu jedem Zeitpunkt klar ist, ob und wo die soeben erzeugten Daten auf der Zielseite letzten Endes geschrieben werden. Eine bessere Lösung ist hier, Konstrukte via Aggregation direkt in die Sprache einzubauen und die temporär benötigten Daten automatisch zu behandeln.

Als weiteres Problem der Anforderungen stellte sich ein Antimuster heraus. So wurde von den Mitarbeitern, die sich mit Mappings bei Seeburger beschäftigen, gefordert, dass es möglich sein soll, einmal geschriebene Werte auf der Zielseite zu überschreiben. Genutzt wurde dieses Muster zum Beispiel, um Summen oder andere aggregierte Werte in einer Schleife zu berechnen (Overriding values in loop). Dies kann wesentlich besser deklarativ gelöst werden, indem man mit Summen und anderen aggregierenden Funktionen arbeitet, wie oben schon beschrieben.

Spezialfälle, wie Mapping-Tabellen zwischen Werten und Variablen/Elementen auf der Zielseite habe ich bei der Betrachtung außen vor gelassen, da diese für einen Sprachkern nicht relevant und für eine erste Evaluation nicht von Nöten sind (Variable Field-Mapping). Anforderungen, die nicht klar definiert waren, wie zum Beispiel das iterieren über nicht-leere Felder, werden zunächst nicht betrachtet, da die Semantik teilweise auch von Fall zu Fall und von Quellschema zu Quellschema unterschiedlich sein kann (Field-iteration for non-empty values). So ist es nicht immer klar, ob ein Feld, das bei der Transformation als leer erscheint, an dieser Stelle nicht existiert und somit der Zugriff zu einem Fehler führen müsste, oder ob das Feld existiert aber der Inhalt leer ist, wie zum Beispiel bei relationalen Datenbanken mit dem Wert NULL möglich. Gerade im letzten Fall ist die Intention nicht klar und hängt vom konkreten Schema und den konkreten Daten ab.

2.2.2 Anwendungsfälle

Im Folgenden gehe ich auf Anwendungsfälle ein, die von Seeburger als wichtig für die Praxis herausgestellt werden.

Der einfachste Anwendungsfall ist das eins-zu-eins Mapping von Elementen. Dabei wird keine Transformation des Inhaltes, sondern im komplexesten Fall eine strukturelle Transformation durchgeführt. Weitere Anwendungen sind Verarbeitung von Daten via Funktionen. Der Umfang der in der Praxis benötigten Funktionen ist jedoch nicht klar umrissen. Ein Beispiel hierfür ist die Zuweisung des aktuellen Datums oder Funktionen, die auf Daten der Quelle angewendet werden wie zum Beispiel mathematische Funktionen. Alle diese relativ einfachen Transformationen können bedingt sein, das heißt, abhängig von Werten in der Quelle werden andere Zuweisungen an das Ziel gemacht.

Wie im Kapitel 2.2.1 schon angedeutet ist ein relativ wichtiger Anwendungsfall das Verarbeiten von Strings. Dabei kommt es häufig vor, dass diese nach einem Trennzeichen aufgeteilt werden müssen. Allerdings stellt sich hierbei die Frage, ob dies nicht in einen Vorverarbeitungsschritt ausgelagert werden sollte, da in diesem Fall effektiv mehrere Daten zu einem komplexen String zusammengesetzt wurden und deren Verarbeitung durch aufteilen in mehrere Elemente wesentlich vereinfacht werden kann.

Wie schon oben erwähnt gibt es Anwendungsfälle, die recht häufig vorkommen und recht standardmäßige Implementierungen haben, wie zum Beispiel das Sortieren oder das Bilden einer Summe. Diese Funktionalität muss mit in die Sprache aufgenommen werden, da zur Aggregation die Verarbeitung temporäre Daten halten muss, um das Aggregat zu bilden. Ein Fall ist jedoch nicht ganz unproblematisch, nämlich der Zugriff auf ein Element nach dessen Position in der Quelle. Das Konzept der Dokumentordnung ist nicht in allen Quellschemata/-paradigmen verfügbar und ist somit nur bedingt umsetzbar. Mehr dazu im Kapitel 4.5.3 über den Spezialfall des Pallet Problems.

In der Praxis wurde bei Seeburger bisher mit einer modifizierten Variante von Basic gearbeitet. Dadurch ergeben sich Anwendungsfälle, die nicht verallgemeinerbar sind. Zu diesen zählt auch die Anforderung, den Ausgabe-Record anlegen zu können sowie mehrere Zuweisungen in einem Schritt machen zu können, den angelegten Record damit also zu füllen. Dies lässt sich zum Beispiel bei deklarativen Mappings wesentlich eleganter lösen und findet somit keine zu große Beachtung im Sinne der Anforderungen. Jedoch könnte man ohne weiteres die Sprache um eine Möglichkeit erweitern, Debugging-Ausgaben zu machen oder Auditing betreiben zu können. Jedoch muss auch diese Anforderung bei der initialen Modellierung nicht berücksichtigt werden.

In manchen Anwendungsfällen müssen Elemente erzeugt werden, deren Namen auf einem Datum basiert, in manchen ist es umgekehrt und etwas, was vorher ein Elementname war, wird zu einem Datum auf der Zielseite. Diese Konvertierungen von Meta- und Wertebene sind jedoch nur beschränkt möglich, was die Modellierung einfacher macht. Da das Quell- und Zielschema feststehen und beide Schemata endlich sind, gibt es auch nur endlich viele Möglichkeiten, wie ein solches Element erzeugt werden kann. Diese Möglichkeiten stehen zur Übersetzungszeit fest und ändern sich nicht mehr. Es ist somit

nicht nötig, dies als separate Funktionalität zu betrachten, da der Programmierer diese Konvertierung von Hand vornehmen kann, ohne spezielle Sprachmittel zu verwenden. So kann in einer Abfrage der Wert an der Quellstelle abgefragt werden und darauf basierend das richtige Zielelement erzeugt werden. Umgekehrt gibt es für verschiedene Elemente auf der Quellseite auch verschiedenen Mapping-Regeln. Dadurch ist der Name des Elements in der Regel bekannt und die Daten auf der Zielseite können entsprechend geschrieben werden. Es kann angedacht werden, für diesen speziellen Fall eine Erleichterung für den Programmierer als syntaktische Erweiterung zu implementieren. Jedoch ist dies in der aktuellen Fassung nicht vorgesehen.

Neben der Mapping-Table, welche, wie bei den Anforderungen schon beschrieben, im Moment nicht berücksichtigt wird, bleiben somit zwei Anwendungsfälle offen, die in mancherlei Beziehung zu erheblichen Problemen führen können, da diese sehr stark auf strukturelle Eigenschaften zurückgreifen. Dies ist einerseits das so genannte Restructure Problem und zum anderen das so genannte Pallet Problem.

Das Restructure Problem hat als Kern die Umstrukturierung des Baumes von der Quelle zum Ziel. Ein interessanter Grenzfall dieses Problems ist es, die Vater-Kind-Beziehung umzukehren.

```
sourceroot
  outer (1)
    ...
    inner (1)
    inner (2)
  outer (2)
    inner (1)
    inner (3)
  ...
```

```
targetroot
  inner (1)
    ....
    outer (1)
    outer (2)
  inner (2)
    ...
    outer (1)
  inner (3)
    ...
    outer (2)
  ...
```

Die Probleme hierbei sind einerseits, dass „inner”-Elemente identifiziert und entsprechend zusammengefasst werden müssen auf der Zielseite, andererseits müssen die „outer”-Elemente den entsprechenden „inner”-Elementen zugeordnet werden. Dies ist nicht in einem Streamingdurchlauf machbar. Man muss bei der Verarbeitung mit „look ahead” arbeiten, um alle „inner”- und „outer”-Elemente zu finden bevor man sie auf der Zielseite schreibt. Eine weitere Möglichkeit wäre das Problem umzukehren und zunächst

mit einem linearen Durchlauf durch die Quelle eine komplexe Datenstruktur im Speicher aufzubauen in der wahlfreier Zugriff besteht und diese erst bei Komplettierung raus zu schreiben. Dabei ist bei diesem Anwendungsfall das Hauptproblem, die Zuordnung der „inner"- und „outer"-Elemente deklarativ zu beschreiben, sodass die Verarbeitung der Elemente eindeutig ist.

Ein weiterer Anwendungsfall, der eine recht große Herausforderung für eine generelle Mappingsprache, die nicht auf Paradigmen mit Dokumentordnung spezialisiert ist, darstellt, ist das Pallet Problem. Die Aufgabenstellung hierbei ist es, auf der Zielseite eine Vater-Kind-Relation aus einer Vorgänger-Nachfolger-Relation auf der Quellseite aufzubauen. Dabei ist die Definition von Vorgänger und Nachfolger ohne Dokumentordnung weder einfach, noch eindeutig. Auf dieses Problem werde ich in Kapitel 4.5.3 noch einmal ausführlicher eingehen.

3 Logik-Ansatz

Steht man vor dem Problem der Verifikation einer Transformation, so liegt es nahe, über einen Ansatz nachzudenken, der eine formale Logik zur Verifikation verwendet. Bei der Transformation wird dies zum Beispiel auch dadurch unterstützt, dass mit Datalog eine Sprache zur Verfügung steht, die sehr viele Einflüsse von Prolog hat und speziell auf Datenbanken ausgelegt ist. Dabei werden die Daten als Regeln in das System eingetragen. Danach kann aus diesen Regeln ein Graph erstellt werden, womit Anfragen an das System in Findung von Pfaden auf dem Graphen umgewandelt werden.

Vorteil eines solchen Ansatzes ist es, dass schon ein komplexes Logik-System vorhanden ist und somit, sobald man die Transformation in das Regelwerk aufgenommen hat, die Eigenschaften, die für die Zielseite zu beweisen sind, direkt als Anfragen an das Logiksystem formalisiert werden können. Ansätze in diese Richtung gibt es schon.

3.1 XPathLog

Bei XPathLog handelt es sich um eine Erweiterung von XPath und Konstrukte der Logikprogrammierung. Es werden somit deklarative Spezifikation von Variablen und Regeln auf Basis von XPath (auf XML Dokumenten) möglich.

Ausdrücke ohne Variablen werden dabei als existentielle Anfragen interpretiert:

```
?- //country[@name = "Switzerland"]//languages[@name="German"]/text().
true
```

Das Ergebnis von Path-Ausdrücken kann einer Variable zugewiesen werden:

```
?- //country[@name = "Switzerland"]//languages[@name="German"]/text() -> P.
P/65
```

Dabei können auch mehrere Variablen gleichzeitig gebunden werden, was zur Rückgabe eines Tupels führt:

```
?- //country[@name->C and @population>1000000]/languages[@name->L]/text()->P.
C/'Austria' L/'German' P/100
C/'Belgium' L/'French' P/32
C/'Belgium' L/'German' P/1
...
```

Auf diese Art sind auch recht simpel Filter realisierbar:

```
?- //Type!X[@name="German"].
Type/languages X/lang-D-german
Type/ethnicgroups X/ethngrp-D-german
Type/languages X/lang-CH-german
...
```

Das Datenmodell ist dabei edge-labeled Graph-basiert. Es lässt wesentlich mehr Freiheit als XML selbst:

- Elemente können mehr als einen Vaterknoten haben
- Strukturen können zyklisch sein
- Es gibt keine globale Ordnung der Elemente, nur die *child* Relation ist sortiert für jedes individuelle Element

Dadurch lassen sich auch mehrere XML-Dokumente in der selben Datenstruktur verwalten. Mit Hilfe von Regeln lassen sich diese Bäume fusionieren. Auf ähnliche Art könnte man neue Knoten mit bestehenden Daten als Ergebnis erzeugen.

XPathLog ist hauptsächlich auf die Daten-Integration ausgelegt. Für komplette Transformation fehlen noch einige Möglichkeiten wie echte Berechnungen und Funktionen und Templates.

Quelle: [8]

3.2 Vorteile der Logik Programmierung für die Transformation

Fasst man das gesamte Transformationssystem als ein Logik System auf, so lassen sich Transformation und Verifikation evtl. in das Gesamtprogramm integrieren. Dabei werden die Regeln, die aus dem Source-Schema entstehen in den initialen Regelsatz eingefügt (quasi als Axiome), die Prüfung dieses Schemas gegen die Daten ist dabei nicht Teil der Transformation. Auch eingefügt werden die Regeln, die sich durch die eingehenden Daten ergeben. Nun werden aus den Axiomen neue Regeln abgeleitet und die Transformation, die ebenfalls mit logischen Regeln beschrieben ist, durchgeführt. Dabei entsteht aus dem Baum, den man eingelesen hat, eine Art von Forest, dessen gemeinsame und unveränderte Knoten vereinigt sind. Nun definiert man, welches der Target-Baum ist. Dies ließe sich evtl. auch schon auf Grundlage des Target-Schemas automatisch realisieren. Nun könnte man die Regeln, die sich aus dem Target-Schema ergeben auf diesem Endregelsatz prüfen und damit sicherstellen, dass dieses Schema erfüllt ist, nachdem die Transformation abgelaufen ist.

Diese Schritte ließen sich auch schon ohne Daten ausführen, da die Strukturregeln schon aus dem Source- bzw. Target-Schema klar sind. Die Schwierigkeit der Verifikation würde sich auf das Definieren der richtige Regeln/Anfragen reduzieren. Es wären (in syntaktisch richtiger Form) also auch solche Regeln möglich wie:

```
descendent(a, sourceroot) ∧ leaf(a) ∧
[b entsteht aus a durch Transformation] ?− descendent(b, targetroot)
```

Diese Regel könnte prüfen, ob alle Datensätze/Tupel, durch Transformation aus einem Blatt des Source-Baumes entstehen, im Target-Baum enthalten sind.

Die Frage, die sich an dieser Stelle stellt ist, ob ein System denkbar ist, das sowohl die Mächtigkeit solcher und noch viel komplexerer Anfragen erlaubt, gleichzeitig auch das gesamte Spektrum der Transformationen abdeckt und gleichzeitig mit den zur Verfügung stehenden Mitteln und in der Verfügbaren Zeit realisiert werden kann.

3.3 Probleme des Ansatzes

Diese Formalisierung in Logik Programmierung wurde schon mit XQuery versucht. Es werden zwar nicht alle Sprachkonstrukte unterstützt, jedoch sind weite Teile der Sprache formalisiert. Ein für eine Transformation wichtiges Element der Sprache ist dabei nicht implementiert worden: aggregate functions (sowie auch positional variables wie in „for $item at $count in ..."). Jedoch haben sich Probleme herausgestellt, wie sie schon aus anderen Logiksystemen bekannt sind: Überführt man ein XQuery-Programm in ein Logik-Regelsystem, so kann es sein, dass dieses System nicht stratifizierbar ist (der Abhängigkeitsgraph zwischen den Regeln enthält einen Kreis durch eine negative Kante). Dies kann man ähnlich betrachten wie Rekursion und dessen Fixpunkt-Berechnung. Somit ist nicht garantiert, dass es einen Fixpunkt für dieses Regelsystem gibt. [1],[5],[9]

Des Weiteren ist es sehr schwer, die Eigenschaften und Garantien so zu formalisieren, dass diese sich mit Hilfe eines automatischen Beweissystems lösen lassen. An dieser Stelle wäre man bei einem halbautomatischen Beweissystems nach Vorbild von Logiksystemen wie Isabelle/HOL, welche für einen Programmierer von Transformationsregeln unzumutbar komplex ist. Aus diesen Gründen muss man sich gegen einen solchen Logik-basierten Ansatz entscheiden, möchte man in annehmbarer Zeit eine Sprache entwickeln, die sowohl Transformationen beschreiben, als auch bestimmte Eigenschaften garantieren oder zumindest Hinweise geben kann, ob diese Eigenschaften erfüllt sein können.

4 Deklarativer Ansatz

4.1 Grundlagen der Transformation

Für die Transformation von Daten kommen grundsätzlich zwei Arten in Frage, die inplace Transformation sowie die Transformation durch Übertragung in ein neues Dokument.

Inplace Transformationen können zum Beispiel mit XUpdate für XML Dokumente durchgeführt werden. Dabei werden die Daten aus dem Dokument gelesen, verarbeitet und wieder in das Dokument, eventuell an eine andere Stelle, geschrieben. Ein Vorteil dabei ist es, nur auf einem Dokument zu iterieren und somit nur die Daten verarbeiten zu müssen, die tatsächlich verändert werden sollen. Die Daten, die unverändert übernommen werden sollen, können unberührt bleiben. Somit eignet sich diese Art der Transformation vor allem für Datenanpassung in der Evolution eines Schemas, bei dem weite Teile des ursprünglichen Dokuments erhalten bleiben und hauptsächlich neue Daten in das Dokument integriert werden sollen. Jedoch überwiegen bei der Transformation von an sich recht unterschiedlichen Schemata die Nachteile. So entsteht bei der Transformation ein Zwischenergebnis, das weder dem Quell- noch dem Zielschema entspricht. Die Quelldaten müssen aus dem Dokument explizit gelöscht werden, um zum fertigen Zieldokument zu kommen. Außerdem kann es unter Umständen zu Kollisionen kommen, wenn Pfade in beiden Schemata gültig sind, jedoch verschiedene Daten tragen sollen. [2, 7]

Die zweite Möglichkeit der Transformation, die Übertragung der verarbeiteten Daten in ein neues Dokument, besitzt andere Eigenschaften. Sie ist eher geeignet, wenn sich bei der Transformation in das Zieldokument die Daten oder die Struktur des Quelldokuments grundlegend ändern. Da so die Daten ohnehin komplett verarbeitet werden müssen, können diese auch gleichzeitig in ein Zieldokument übertragen werden, welches somit immer einer Untermenge des Zielschemas entspricht. Dieser Ansatz ist somit eher geeignet bei einer allgemeinen Transformation verwendet zu werden, wie sie in diesem Fall vorliegt.

Dabei sind zwei Unterarten der Verarbeitung möglich. Bei der, der Hierarchie der Daten folgenden, Transformation werden zunächst die Blätter des Baumes transformiert, danach wird die nächst höhere Ebenen verarbeitet und die zuvor verarbeiteten Daten als Kindelemente aufgenommen und so bis zur Wurzel alle Datensätze schichtweise verarbeitet. Ein Beispiel dieser Technik ist XQuery. Dabei folgt jedoch die Verarbeitung eher der Struktur des Quell- oder Zielschemas, was nicht unbedingt die Beziehungen zwischen den beiden Schemata widerspiegelt. Eine solche Transformationsbeschreibung kann schwer lesbar und zu verstehen sein. Für die Generierung des Ergebnisses einer Anfrage und somit zur Filterung von Daten ist diese Technik gut geeignet, jedoch zur Transformation von Daten in ein Zielschema gilt dies weniger. Bei der deklarativen Verarbeitung wird sich konzeptionell eher auf die Beziehung zwischen den beiden Schemata konzentriert. Es werden somit Beziehungen einzelner Elemente des Quell- und Zielschemas und die Überführung der Daten in die entsprechende Form in den Mittelpunkt gestellt und somit von der Verarbeitung zur Laufzeit abstrahiert.

4.2 Voraussetzungen

Bei der Transformation von hierarchischen Daten, wie sie hier betrachtet wird, gehe ich von bestimmten Voraussetzungen aus. Zum einen müssen bei Schemata, das Quell- sowie das Zielschema, die die Struktur der Daten einhalten muss, bekannt sein. Zudem müssen diese Schemata in der hier verwendeten und definierten Schemabeschreibungssprache vorliegen oder in diese transformiert werden können. Dies sehe ich, mit Vortransformation

der XML-, EDIFACT- und SQL-Schemata, als gegeben an. Die meisten dieser Transformationsschritte sind, dank der gleichen Struktur, nicht weiter interessant. Auf ein Problem, der Position eines Elements innerhalb eines Dokuments, werde ich in späteren Sektionen eingehen. Weiterhin müssen die Relationen der beiden Schemata bekannt sein. Dies ist eine nicht-triviale Aufgabe, die als Hauptaufgabe eines Programmierers einer solchen Transformation gesehen werden kann.

Hier soll es nur um die Sprache selbst gehen. Deshalb bleiben Implementierungsaspekte weitgehend außen vor. Insbesondere bleiben Probleme wie eine effiziente Laufzeitumgebung oder die Umsetzung in ein Operatornetz zur Realisation einer Streaming-Verarbeitung unberücksichtigt.

4.3 Schema

Als Schemabeschreibung soll eine reduzierte Schemasprache verwendet werden. So soll nicht, wie in XML üblich, zwischen Elementen und ihren Attributen unterschieden werden. Die Patterns, wie Subelemente angegeben werden können, sind beschränkt auf die Folgenden:

- Sequenz von Elementen (alle Elemente kommen als Kinder vor, die Reihenfolge ist hierbei nicht festgelegt)
- Choice über Elemente (genau eines der Elemente kommt als Kind vor)
- Kleene-Star eines Elements (beliebig viele Elemente dieser Form kommen als Kinder vor, bei '+' muss mindestens ein solches Element vorkommen)
- Optionale Elemente (höchstens ein Element dieser Form kommt als Kind vor)

Die Bindungsstärke nimmt bei der aufgeführten Reihenfolge zu. Somit bindet die Option am stärksten, die Sequenz am schwächsten. Elemente können drei Formen annehmen.

- Leere Elemente besitzen keine Kinder und keinen Wert. In XML entspräche dies zum Beispiel etwa dem folgenden: *
*
- Elemente, die nur einen Wert beinhalten, können keine Unterelemente haben. Der Wert hat einen elementaren Typ. Dies entspräche in XML zum Beispiel etwa dem folgenden: *<h1>Titel</h1>*
- Bei Elementen, die Unterelemente haben, werden die Kinder durch ein Pattern beschrieben. Somit hat der Inhalt einen komplexen Typ.

Es ist in einem Element somit nicht möglich, Daten und Unterelemente gleichzeitig zu haben. Dies bedeutet bei der Transformation eines speziellen Schemas, wie zum Beispiel eines XML-Schemas, müssen entsprechende Umformungen nicht-kompatibler Beschreibungen gemacht werden. Im Speziellen müssen Attribute zu Unterelementen umgeformt werden, womit sich die Frage stellt, wie der Inhalt eines solchen Elements aussieht, wenn es neben Attributen auch einen Wert haben kann. Dafür muss ein weiteres Unterelement

angelegt werden, was den eigentlichen Inhalt des XML-Elements repräsentiert. Nach dieser Umformung wird somit von Attributen komplett abstrahiert.

Die oben beschriebene Schemasprache hat den Vorteil, dass sie mit relativ wenigen Konstrukten (Eltern-Kind-Beziehung, Sequenz, Choice, Kleene-Star, optionales Element, Datentypen) eine sehr große Menge an möglichen Schemata abdeckt.

```
Element    ::= id ('*' | '+' | '?')? '{' Child '}'
Child      ::= Sequence
             | type
             | ϵ

Sequence   ::= Choice ',' Sequence | Choice
Choice     ::= Element '|' Choice | Element
```

Dabei muss folgende Regel gelten:

- Wenn S eine Sequenz ist und C_1 und C_2 Choices, $C_1 \in S$, $C_2 \in S$, $e_1 \in C_1$ und $e_2 \in C_2$, so müssen die Namen von e_1 und e_2 verschieden sein.

Somit ist sichergestellt, dass die Elemente eines solchen Schemas eindeutig mit Pfaden adressiert werden können.

4.4 Grundideen des Mappings

Pro Mapping können vier Phasen unterschieden werden:

1. get: Elemente, die durch einen Pfad adressiert werden, werden selektiert und einzeln der Verarbeitung zugeführt
2. filter: Die Elemente, die aus der get-Phase kommen und nicht bestimmte Eigenschaften besitzen, werden aussortiert und nicht verarbeitet. Die restlichen Elemente werden zur Transformation weitergegeben. Außerdem werden verschiedene Transformationen durch verschiedene Filter ausgewählt.
3. transform: Die Elemente, die aus den früheren Phasen übergeben wurden, werden transformiert und die Daten darin verarbeitet. Die Darstellung wird an das Zielschema angepasst.
4. put: Die nun dem Zielschema entsprechenden Elemente werden an einer Stelle des Zieldokuments eingehängt.

Dabei stellen sich verschiedene Herausforderungen in den verschiedenen Phasen. Für die get-Phase wird eine Art der Adressierung benötigt, die es erlaubt, auf alle Elemente des Dokuments zuzugreifen. Dabei sollen diese Adressen gegen ein gegebenes Schema validiert werden können sowie als Basis-Adresse für die Adressierung von Kind-Elementen dienen können. In der filter-Phase soll jeweils eine Bedingung definiert werden können, die die zu verarbeitenden Elemente hinreichend einschränken kann, jedoch soll möglichst auch statisch auf komplette Überdeckung gegenüber dem Schema geprüft werden können. In der Transformation soll es möglich sein, die Daten des Dokuments zu verarbeiten und

die nötigen Berechnungen durchzuführen, jedoch ohne dabei mit dem Durchlaufen der Elemente zu interferieren. Schlussendlich muss in der put-Phase sicher gestellt werden, dass die Adressierung von Zielelementen eindeutig ist und mächtig genug, die Elementrelationen der Zielseite aufzubauen. Gleichzeitig soll jedoch möglichst weitgehend sicher gestellt werden, dass die Informationen ausreichen, um die Konformität mit dem Zielschema statisch sicher zu stellen.

4.4.1 get-Phase

In der get-Phase werden zunächst die Elemente eines Eingabedokuments adressiert und die durch ein Mapping zu verarbeitenden Elemente bestimmt. Dazu ist es nötig zu definieren, welche Elemente auf diese Weise ausgewählt werden können und wie diese ausgewählten Elemente verarbeitet werden. Bei der deklarativen Verarbeitung hierarchisch geordneter Daten scheint es sinnvoll, eine Menge von Elementen durch einen Pfad zu adressieren und diese dann einzeln der Verarbeitung zuzuführen. Dadurch umgeht man die Einführung eines Schleifenkonstrukts, welches die Festlegung einer Ausführungsreihenfolge zur Folge hätte. Für die Adressierung mit Pfaden spricht die Tatsache, dass dieses System mit XPath[4] weite Verbreitung bei der Adressierung von hierarchisch geordneten Daten gefunden hat. Ein Pfad stellt somit einen Ast des Baumes dar, startend von der Wurzel und endend bei dem Element, welches zur Verarbeitung ausgewählt wird. Da die Verschachtelungstiefe der Elemente nur endlich groß ist, sind diese Pfade auch nur endlich lang.

Wichtig für die Transformation ist in diesem Zusammenhang auch, dass die Pfade eindeutig dem Schema zugeordnet werden können. Diese Anforderung ergibt sich aus der Tatsache, dass den so adressierten Elementen zur Compilezeit ein Typ zugeordnet werden soll, welcher im Schema definiert ist. Dazu wären zwei naheliegende Lösungsansätze möglich. Zum einen könnte man verlangen, dass alle Elemente im Schema einen eindeutigen Namen bekommen. Dadurch würde die Zuordnung von Elementname zu Typ eindeutig. Zum anderen könnte man auch mit absoluten Pfaden arbeiten. Dadurch, dass man den Pfad somit immer von der Wurzel an beschreibt, ist das Element eindeutig, soweit dies vom Schema her sicher gestellt ist.

In der Praxis ist die Verwendung von global eindeutigen Namen nicht immer gegeben und intuitiv. Jedoch ist die Verschachtelungstiefe meist relativ gering. Dadurch bietet sich die Verwendung von absoluten Pfaden zur Adressierung von Elementen an. Dies hat ebenso den Vorteil, dass die absolute Adresse von Kindelementen durch Konkatenation erzeugt werden kann und somit relativ einfach ist.

Pfade können zwei Arten von Elementen adressieren. Elemente, die nicht Kleene-Star Elemente sind, können nur einmal relativ zu ihrem Eltern-Element auftauchen. Elemente, die Kleene-Star Elemente sind, im Schema durch '*' oder '+' gekennzeichnet, können mehrfach auftreten. Somit wird mit einem solchen Pfad nicht ein einzelnes Element, sondern eine Menge von Elementen adressiert. Jedes Element aus dieser Menge wird

einzeln zur Transformation weiter gegeben. Die Adressierung der Kinder eines Kleene-Star Elements erfolgt auf Basis seines Eltern-Elements. Somit wären alle Kinder eines Kleene-Star Elements prinzipiell nicht eindeutig, jedoch können sie für ein beliebiges aber festes Kleene-Star Element als eindeutig adressierbar angesehen werden. Dabei wird davon ausgegangen, dass die Verarbeitung des Quelldokuments hierarchisch erfolgt und somit das Eltern-Element bei der Verarbeitung seines Kind-Elements bekannt und fest ist.

4.4.2 filter-Phase

In der filter-Phase werden Elemente, die in der get-Phase ausgewählt wurden, weiter gefiltert. So kann entschieden werden, dass bestimmte Elemente, die in der vorherigen Phase zwar selektiert wurden, nicht oder anders als andere Elemente verarbeitet werden sollen. Dies kann nötig werden, wenn die Elemente nicht homogen sind und somit semantisch nicht gleich verarbeitet werden können. Ein Filter stellt dabei grundsätzlich einen booleschen Ausdruck dar. Jedoch sind bei der Verwendung von booleschen Ausdrücken ohne Einschränkung bestimmte Eigenschaften, wie die Überdeckung des gesamten Wertebereichs eines Elements, nicht statisch entscheidbar. Spezielle interessante Ausdrücke sind somit solche, bei denen man statisch anhand der Informationen, die man aus dem Schema gewinnen kann, solche Eigenschaften entscheiden kann. Nur für solche Fälle können statische Garantien abgegeben werden. Einige solcher Klassen von Ausdrücken sollen in der Analyse der Sprache gefunden werden. Im weiteren gehe ich davon aus, dass beliebige boolesche Ausdrücke über den Eingabedaten verwendet werden können, jedoch nur für spezielle Klassen von Ausdrücken auch statische Garantien über bestimmte Eigenschaften gegeben werden können.

Zwar findet in der get-Phase eine Selektion der Elemente statt, die transformiert werden sollen. Die filter-Phase scheint noch einmal etwas ähnliches zu tun. Tatsächlich könnte die get-Phase komplett entfallen. Die Vorselektion, die in ihr erfolgt, könnte ebenso gut als Ausdruck in die Filterbedingung aufgenommen werden. Jedoch soll in der Sprache eine deklarative Beschreibung der Transformationsregeln erfolgen. Somit ist es natürlicher, die Regeln für einzelne Elemente zu trennen und in einer gesonderten Syntax zu beschreiben.

Die filter-Phase entscheidet nicht nur darüber ob, sonder auch welche Transformation auf dem Element ausgeführt werden soll. Dabei können mehrere Transformationen angegeben werden, die Selektion erfolgt anhand der Filter-Ausdrücke. Dabei muss entschieden werden, ob, sollten zwei oder mehr Filter-Ausdrücke zutreffen, alle oder nur die „erste" Transformation ausgeführt werden soll, wobei die Reihenfolge der Transformationen klar definiert sein muss. Ich habe mich für letztere Möglichkeit entschieden, da dadurch die Semantik der Filterausdrücke auch beinhaltet, dass entweder genau eine der Transformationen oder keine Transformation ausgeführt wird. Dadurch muss nicht noch geprüft werden, ob mehrere Transformationen durchgeführt werden, was zu einem Konflikt führen kann, falls auf das gleiche Element geschrieben werden soll. Was zu prüfen bleibt, ist, ob eine der Filterausdrücke auf das Element zutrifft und somit, ob überhaupt eine Transformation ausgeführt wird. In den meisten Fällen ist es wünschenswert, dass die Daten

aus der Quelle komplett in das Zieldokument übertragen werden, da ansonsten Informationen verloren gehen. Dies kann somit anhand der Mapping-Regeln zusammen mit den Filterregeln überprüft werden. Die Hoffnung ist, dass vieles davon tatsächlich statisch prüfbar ist, soweit die Möglichkeiten der Filterausdrücke weit genug eingeschränkt sind.

4.4.3 transform-Phase

In der tranform-Phase werden die Elemente, die von der filter-Phase dieser Transformation zugeordnet wurden, verarbeitet. Für jedes Element, das elementare Daten beinhaltet, kann eine Transformationsfunktion angegeben werden. Dabei werden die Daten des Elements dem Zielschema angepasst. Es besteht Zugriff auf die vom aktuellen Element eindeutig erreichbaren Unter- und Oberelemente. Es ist kein Zugriff auf Elemente innerhalb einer Menge von Elementen möglich, also keine Kleene-Star Elemente, außer sie befinden sich auf dem Pfad zur Wurzel. Diese Einschränkung stammt daher, dass die Transformation nicht mit der Iteration des Eingabedokuments interferieren soll. In dieser Phase sind jedoch auch weitere Hindernisse zu überwinden, wie Aggregation von Werten über einer Menge und eventuelle Konvertierung in den Typ, der im Zielschema vorgegeben ist. Initial soll das Thema Aggregation nicht zu tief behandelt werden, da eine komplette Unterstützung von Aggregation und die Weiterverarbeitung dieser Werte einen erheblichen Mehraufwand bedeutet. Deshalb konzentrieren wir uns zunächst auf die strukturelle und einfache Datentransformation ohne Aggregation. Später werden wir noch kurz anreißen, wie Funktionen, die auf einer Menge von Elementen operieren, in die Sprache eingebaut werden könnten.

4.4.4 put-Phase

In der put-Phase werden die Elemente, die in der transform-Phase erzeugt wurden, an die richtige Stelle im Zieldokument geschrieben. Dabei müssen viele Hindernisse angegangen werden. So muss die Zuordnung eindeutig sein. Dies ist vor allem dann problematisch, wenn das Elternelement ein Kleene-Star Element ist und somit mehrfach mit dem gleichen Pfad auftauchen kann. Hierfür muss eine Zuordnung geschaffen werden, die über einen einzigen Pfad hinaus geht und die Beziehung der Elternelemente angibt. Dazu muss jedoch unter Umständen auf die Quellseite zugegriffen werden, um diese Beziehung definieren zu können. Des Weiteren ist die Frage zu klären, ob das Mapping eindeutig ist in der Beziehung, dass nicht zwei Werte auf das selbe Elemente geschrieben werden. Dies würde einen Konflikt darstellen, da die Deklarationen die Beziehung zwischen Quell- und Zielschema darstellen, und somit an dieser Stelle die Beziehung nicht klar wäre.

4.5 Syntax

Ein Vorschlag für eine vorläufige Syntax wäre wie folgt:

```
transformation ::= 'transformation' 'from' schemaName 'to'
                                     schemaName '{' mapping* '}'
mapping ::= 'map' ('*' | '+' | '?')? path 'to' path '{' case* '}'
case    ::= 'case' boolExpression '{' put '}'
put     ::= 'put' (valueExpression)? ('where' boolExpression)? ';'
boolExpression       ::= equalsExpression | pathEquality |
                         boolValueExpression
boolValueExpression ::= 'true' | 'false'
pathEquality         ::= path '~' path
equalsExpression     ::= path '=' valueExpression
valueExpression      ::= floatValue | intValue |
                         stringValue | pathValue
path         ::= ( ('source' | 'target') ':' )? ('/')?
                 ((".." | "." | ident) '/')* ident
pathValue    ::= path
schemaName   ::= ident
floatValue   ::= "wie üblich"
intValue     ::= "wie üblich"
stringValue ::= "wie üblich"
ident        ::= "wie üblich"
```

Case-Ausdrücke werden von oben nach unten abgearbeitet und der erste Fall der zutrifft bestimmt die Transformation. So können beim Verarbeiten eines Elements in einer Transformation nur genau ein case abgearbeitet werden. Im Rumpf des case-Ausdrucks muss wiederum ein put-Ausdruck stehen. Soll für einen Fall kein Element erzeugt werden, so darf dieser Fall nicht als case-Ausdruck angegeben werden.

Innerhalb eines Transformationsrumpfes (und somit innerhalb eines case) kann der aktuelle Wert des mit dem Quellpfad adressierten Elements mit "." verwendet werden.

Eine Beispieltransformation könnte wie folgt aussehen:

```
schema exin{
  inventory{
    item *{
      weight{int}
    }
  }
}
schema exout{
  ship{
    containter{
      item *{
        type{string}
      }
    }
  }
}
transformation from exin to exout{
map /inventory to /ship{
```

```
  case true{
    put;
  }
}
map /inventory to /ship/container{
  case true{
    put;
  }
}
map /inventory/item to /ship/container/item{
  case true{
    put;
  }
}
map /inventory/item/weight to /ship/container/item/type{
  case .<500{
    put "light";
  }
  case .<1000{
    put "medium";
  }
  case true{
    put "large";
  }
}
}
```

Dabei werden items aus einem Lager in einen Container eines Schiffes "verladen" und dabei mit einem Label nach Gewicht versehen. Dabei tauchen jedoch Probleme bei der Adressierung des Ziel-Elements auf, die im nächsten Kapitel 4.5.1 erläutert werden sollen.

4.5.1 Problem der Adressierung

Im folgenden verwende ich folgende Konvention zur Beschreibung eines Pfades:

- /a/b/c
  Ein Pfad, der nur Elemente mit Kardinalität eins beinhalten, das heißt, der Pfad ist eindeutig.
- /a/b*/c*/d
  Ein Pfad, in dem die Elemente "b" und "c" mehrfach vorkommen können (siehe Kleene-Star Pattern).

In der get-Phase werden einzelne Elemente ausgewählt und der Verarbeitung zugeführt. Es stellt sich jedoch die Frage, wo genau dieses Element auf der Zielseite geschrieben werden soll. Dies ist vor allem der Fall bei Elementen aus einer Menge scheinbar gleichförmiger Elemente (Kleene-Star-Elemente). Dabei adressiert der Zielpfad nicht eindeutig ein Element, sondern eine Menge von Elementen. Im optimalen Fall unterscheiden sich diese Elemente nach den Daten ihrer Kindelemente. Diese Beziehung soll bei der Ausgabe berücksichtigt werden.

Ein möglicher Lösungsansatz zu diesem Problem ist, beim Schreiben des Ergebnisses mit *put* die Angabe von Bedingungen zuzulassen, die die Adressierung steuern und soweit einschränken, dass diese eindeutig wird. Somit können in der Berechnung erhaltene Ergebnisse aus den Quelldaten zur Adressierung verwendet werden. Ein Beispiel eines Mappings von */A/child*/i* nach */B/category*/value*:

```
schema exin{
  A{
    category *{
      type{string}
    }
    child *{
      i{int}
    }
  }
}
schema exout{
  B{
    category *{
      value *{int}
      type{string}
    }
  }
}
transformation from exin to exout{
map /A to /B{
  case true{
    put;
  }
}
map /A/category to /B/category{
  case true{
    put;
  }
}
map /A/category/type to /B/category/type{
  case true{
    put . where
      source:/A/category ~ target:/B/category;
  }
}
map /A/child/i to /B/category/value{
  case .<100{
    put . where
      target:/B/category/type = "small";
  }
  case .<1000{
    put . where
      target:/B/category/type = "medium";
  }
  case true{
    put . where
      target:/B/category/type = "large";
```

```
  }
}
}
```

In diesem Beispiel werden die Werte aus /A in verschiedene Kategorie eingeteilt: *small*, *medium* und *large*. Diese Einteilung steht im Element *category* im Unterelement *type*. Die Zuordnung erfolgt in der put-Phase durch die angegebenen Bedingung.

Es stellt sich jedoch ein weiteres Problem. Zwar erlaubt die Transformation innerhalb des case-Blocks Zugriffe auf die Kind- und Elternelemente des adressierten Quellpfades. Somit kann die Zuordnung durch Zugriff auf die Daten und Vergleich mit festen Werten erfolgen. Ein Beispiel ist eine Identitätstransformation von /a/b*/c* nach /a/b*/c*. Dabei sei eine ganz bestimmte, nicht vom Inhalt von c abhängige Zuordnung von c-Elementen zu b-Elementen gegeben. Man kann weiterhin annehmen, dass jedes Element eine eindeutige ID enthält. Um die Zuordnung bei der Transformation zu erhalten, benötigt man Zugriff auf das ID-Element von b und zwar innerhalb der put-Operation einer Transformation von c.

```
schema ex{
  a{
    b *{
      id{id},
      c *{string}
    }
  }
}
transformation from ex to ex{
map /a to /a{
 case true{ put;}
}
map /a/b to /a/b{
  case true{ put; } //Annahme: Dabei wird auch die id mit übertragen
}
map /a/b/c to /a/b/c {
  case true {
    put . where /a/b/id = ../id;
  }
}
}
```

Wie man anhand des letzten Mappings erkennt, ergibt sich dabei jedoch ein Problem: Die Eindeutigkeit eines Pfades ist nicht zwangsläufig gegeben. Es ist nicht klar, ob mit */a/b/id* der Pfad auf der Quell- oder der Zielseite gemeint ist. Hier stellt sich die Frage, wie zwischen Quell- und Zielpfad unterschieden werden kann. Dies ist bei der map-Operation kein Problem, da dort vor dem „to” ein Quellpfad und nach dem „to” ein Zielpfad stehen muss.

Ein Vorschlag an dieser Stelle ist, Quell- und Zielpfad anhand einer speziellen Syntax unterscheidbar zu machen:

```
...
map /a/b/c to /a/b/c {
  case true {
    put . where target:/a/b/id = ../id;
  }
}
...
```

Dadurch ist sowohl dem Programmierer als auch der Analyse vollkommen klar, welcher Pfad gemeint ist. Voraussetzung für die Eindeutigkeit des Pfades auf der Zielseite ist, dass der Subpfad von der Wurzel bis zu jedem mehrfach vorkommenden Element (hier /a/b) auch ein Subpfad des adressierten Pfades ist.

Um die Sprache an dieser Stelle analysierbar zu halten, darf man keine allgemeinen booleschen Ausdrücke erlauben, da man die Entscheidbarkeit an dieser Stelle nicht voraussetzen kann. Somit wäre nicht klar, ob das Ziel überhaupt existiert und somit, ob ein Mapping überhaupt vorgenommen werden kann. Doch diese Entscheidung stellt sich auch bei Einschränkung auf Wertvergleich. Hier ein Beispiel:

```
schema exin{
  a{
    b *{
      i{int},
      c *{string}
    }
  }
}
schema exout{
  d{
    e *{
      i{int},
      f *{string}
    }
  }
}
transformation from exin to exout{
map /a to /d {
  case true{
    put;
  }
}
map /a/b to /d/e {
  case true{
    put; // erstelle das Element /d/e
         // beim matchen des Pfades /a/b
  }
}
map /a/b/i to /d/e/i {
  case true{
    put .; // ordne das Element i dem entsprechenden
           // e-Element zu
  }
}
```

```
map /a/b/c to /d/e/f {
  case true{
    // Ordne das Element f dem Element e unter,
    // dessen Unterelement i den Wert von c hat
    put . where target:/d/e/i = .;
  }
}
}
```

Wenn man nun davon ausgeht, dass die Elemente /a/b/i und /a/b/c vom Typ Integer sind und keine Beschränkungen haben, so lässt sich statisch nichts über die Existenz des gesuchten Zielelements /d/e aussagen:

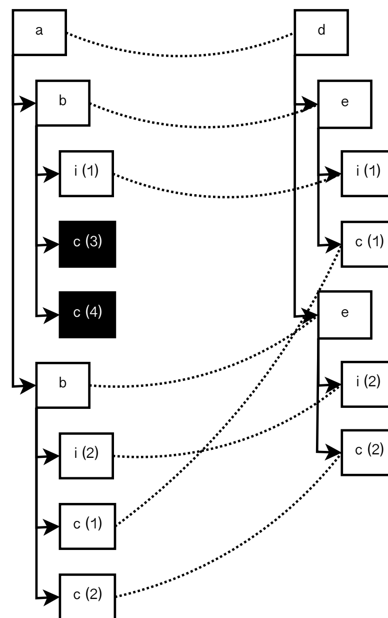

Abbildung 1: Transformation auf Basis der Werte der Elemente

In diesem Fall könnten die c-Elemente mit dem Wert 1 und 2 zugeordnet werden, die c-Elemente mit den Werten 3 und 4 jedoch nicht. Zu erkennen ist dies jedoch nur zur Laufzeit und bei konkreten Daten.

Ein weiteres Problem, das bei diesem Beispiel bei näherem Hinsehen existiert, ist das Mapping des Elements i. Die Intention ist, das i-Element dem „richtigen" e-Element zuzuordnen. Gemeint ist hierbei das /d/e, das aus dem entsprechenden /a/b entstanden ist. Um dies deklarieren zu können ist einer der beiden folgenden Ansätze nötig:

- Jedes Element bekommt eine explizite ID, welche automatisch wieder dem Element mit zugeordnet wird. Die ID für das Element muss vom Schema-Ersteller angegeben werden.

- Jedes Element hat implizit eine Identität, die über das Mapping hinaus erhalten bleibt.

Beim ersten Ansatz wäre es notwendig, die expliziten IDs direkt während des Mappings des Elements mit auf die Zielseite zu übertragen. Dabei kann es jedoch vorkommen, dass ein einzelnes Element keinen eindeutigen Schlüssel darstellt und somit ein komplexerer Schlüssel angegeben werden muss. Ein solcher zusammengesetzter Schlüssel lässt sich nicht ohne weiteres auf das Zieldokument übertragen, da pro Mapping nur ein einzelnes Element erzeugt wird. Ein Beispiel ist die Transformation eines Datensatzes, der durch einen zusammengesetzten Schlüssel identifiziert wird:

```
schema ex{
  a{
    b *{
      ida{int},
      idb{int}
    }
  }
}
transformation from ex to ex{
map /a to /a{
  case true{ put; }
}
map /a/b to /a/b{
  case true{ put; }
}
map /a/b/ida to /a/b/ida {
  case true {
    put . where ?;
  }
}
map /a/idb to /b/idb {
  case true {
    put . where ?;
  }
}
}
```

Da nur /a/b/ida und /a/b/idb zusammen eine eindeutige Identifikation des Elements /a/b zulassen ist die Zuordnung auf der Zielseite nicht möglich. Es wäre somit nötig, die Sprache um die Möglichkeit zu erweitern, mehrere Elemente pro Mapping zu transformieren (um *b*, *ida* und *idb* gleichzeitig zu mappen), was jedoch wiederum zu komplexen Problemen führen kann, oder zusammengesetzte Schlüssel zu verbieten. Letzteres jedoch würde bedeuten, dass ein künstlicher Schlüssel generiert werden müsste, der auf allen Teilen des zusammengesetzten Schlüssels beruht, was wiederum dem Konzept der Beibehaltung von im Quelldokument enthaltenen Schlüsseln widerspricht.

Ich habe mich deshalb für den zweiten Ansatz entschieden, da sich der Schema-Ersteller somit nicht um die ID kümmern muss, es aber trotzdem garantiert ist, dass immer eine Äquivalenzrelation zwischen den Elementen auf der Quell- und der Zielseite existiert. Ein Zugriff auf diese ID direkt ist somit allerdings nicht möglich, weil auch nicht nötig. Zwei

Elemente sind nach dieser Definition genau dann äquivalent, wenn sie die gleiche Dokumentposition im selben Dokument haben, somit also identisch sind, oder das Element auf der Zielseite aus dem Element auf der Quellseite durch Transformation entstanden ist.

Das Beispiel von oben wäre somit wie folgt zu erweitern:

```
...
map /a/b/ida to /a/b/ida {
  case true {
    put . where source:/a/b ~ target:/a/b;
  }
}
map /a/b/idb to /a/b/idb {
  case true {
    put . where source:/a/b ~ target:/a/b;
  }
}
...
```

Die Bedingung `source:/a/b ~ target:/a/b` gibt hierbei an, dass das Element auf der Zielseite, welches durch den Pfad `target:/a/b` adressiert wird mit dem Element auf der Quellseite, welches durch den Pfad `source:/a/b` adressiert wird, in Relation steht. Das heißt, das Element auf der Zielseite ist durch Verarbeitung eines Mappings aus dem Vater des Elementes `ida` beziehungsweise `idb` entstanden.

4.5.2 Umstrukturierung

Wie schon in Kapitel 2.2.2 angedeutet ist eines der schwierigen Probleme, die mit dieser Sprache gelöst werden können sollen, die Restrukturierung von Daten. Die Schwierigkeit liegt in diesem Fall bei der Umkehrung der Vater-Kind-Beziehung und deren Beschreibung in deklarativer Form. Eine Möglichkeit, die recht eingängig ist, möchte ich in einem kleinen Beispiel demonstrieren.

```
schema ex4in{
  lib{
    book * {
      bookid {string},
      author * {
        authorid {string}
      }
    }
  }
}
schema ex4out{
  lib{
    author * {
      authorid {string},
      book * {
```

```
        bookid {string}
      }
    }
  }
}
transformation from ex4in to ex4out {
  map /lib to /lib{
    put;
  }
  map /lib/book/author to /lib/author{
    put;
  }
  map /lib/book/author/authorid to /lib/author/authorid{
    put . where target:/lib/author ~ source:/lib/book/author;
  }
  map + /lib/book/bookid to /lib/author/book/bookid{
    put . where target:/lib/author/book ~ source:/lib/book;
  }

  map + /lib/book to /lib/author/book{
    put where source:/lib/book/author ~ target:/lib/author;
  }
}
```

Beschrieben ist hier eine klassische Transformation, bei der die Vater-Kind-Relation umgekehrt ist. Auf der Quellseite sind die Autoren als Kinder der Buch-Element enthalten, auf der Zielseite soll diese Beziehung umgekehrt werden, die Bücher sollen den Autoren zugeordnet werden. Die Attribute der Bücher und Autoren sollen dabei durch ein Element *bookid* beziehungsweise *authorid* symbolisiert werden. Interessant sind folgende beiden Phänomene:

- Die Regel `map /lib/book/author to /lib/author` erzeugt für jeden Autor, der im Quelldokument vorkommt einen neuen Autor-Eintrag im Zieldokument. Jedoch werden doppelte Elemente nicht zusammengelegt. Somit entstehen für den gleichen Autor, für eine hinreichend genaue Definition der Gleichheit zweier Autoren Elemente, mehrere Einträge im Zieldokument.
- Die letzte Regel `map + /lib/book to /lib/author/book` muss auf ein Element zugreifen, das mehrfach im Quelldokument vorkommen kann. Das Element welches durch den Pfad `/lib/book/author` adressiert wird ist nicht eindeutig in Relation zu `/lib/book`. Somit wird durch diese Regel ein 1-zu-n Mapping beschrieben. Zusammen mit der obigen Aussage über die erzeugten Autoren-Elemente lässt sich folgern, dass keines der äquivalenten Autorenelemente alle Bücher enthält, die von dem jeweiligen Autor geschrieben wurden.

Das Zusammenführen auf Basis gleicher Kind-Elemente kann als ein Spezialfall der Aggregation angesehen werden. Dabei wird eine Liste von Elementen auf eine Liste von Elementen reduziert, in der Elemente, die äquivalent modulo eines angegebenen Kindes sind, entfernt wurden. Eine Konsequenz daraus wäre, dass die bisher immer als 1-zu-1 oder 1-zu-n Beziehung angesehene Äquivalenzrelation über die transformierten Elemente

zu einer n-zu-1 Relation mutiert. Mehrere Elemente der Quellseite sind somit äquivalent mit nur einem einzelnen Element auf der Zielseite.

Dies ist jedoch in der Praxis nicht ganz einfach zu realisieren. Es kommen zwei grundlegend unterschiedliche Realisierungen in Frage: Die Feststellung der Äquivalenz zweier Elemente kann auf der Quell- oder der Zielseite erfolgen. Verschiebt man die Erkennung äquivalenter Elemente auf die Quellseite, so lässt sich, nach dem Übertragen des ersten Elements, die weiteren Mappings verhindern, um keine doppelten Elemente im Ausgabedokument zu erzeugen. Trotzdem muss jedoch registriert werden, dass das aktuelle Element des Quelldokuments mit dem schon vorhandenen Element des Zieldokuments in Relation steht. Eine weitere Möglichkeit ist es, die Erkennung von doppelten Elementen auf die Zielseite zu verschieben. Dabei werden zunächst alle Elemente erzeugt und danach die Liste der so entstandenen Elemente gefiltert und so die doppelten Elemente entfernt. Dabei muss jedoch bedacht werden, dass eventuell nicht der gesamte Inhalt des doppelten Elements auch tatsächlich doppelt ist. Bei oben beschriebenem Beispiel würden die *book*-Elemente nicht in jedem *author*-Element erzeugt, sondern jedes *book*-Element in seinem eigenen *author*-Element. Es bleibt also zu entscheiden, welche Information tatsächlich redundant sind und welche zusammengeführt werden müssen. Vorteil währe jedoch, dass man potentiell vorhandene Vorrichtungen zur Behandlung von generellen Aggregaten mitbenutzen könnte. Zu entscheiden bliebe an dieser Stelle, wie mit möglichen Konflikten, Kindelementen also, die zusammengelegt werden sollen, jedoch nicht den gleichen Wert besitzen, umgegangen werden soll.

Eine spezielle Eigenschaft von Elementen eines XML-Dokuments wurde bis jetzt nicht beachtet. Da mit der Sprache auch XML Dokumente verarbeitet werden können sollen muss geprüft werden, wie mit der Dokumentordnung umgegangen werden soll. Da in der vorgestellten Sprache keine Ordnung der Elemente vorgesehen ist, muss eine andere Lösung gefunden werden.

4.5.3 Spezialfall XML Dokumentordnung

Das Pallet Problem ist ein Problem aus der Verpackungslogik, das nach Angaben von Seeburger sehr häufig vorkommt. Jedoch beruht das Problem auf Dokumentpositionen, was es nicht direkt möglich macht, das Problem mit der in diesem Dokument vorgestellten Sprache zu beschreiben. Im Nachfolgenden werde ich erklären, was das Pallet Problem ist, weshalb die Transformation mit der vorgestellten Sprache nicht direkt möglich ist und dabei auf das Problem der Dokumentordnung an sich eingehen.

Der Kern des Pallet Problem beschäftigt sich mit der Zuordnung von Produkten zu Paletten. Dabei ist auf der Quellseite eine Liste gleichartiger Elemente vorhanden, die sich nur anhand eines Labels unterscheiden lassen. Je nach Wert dieses Labels ist das Element entweder Start einer neuen Palette oder Teil einer vorher begonnen Palette. Ein weiterer Teil ist die Übertragung von Informationen aus Elternelementen, was jedoch verhältnismäßig einfach ist und daher nicht zur Komplexität des Problems beiträgt. Die Hauptaufgabe einer Transformation liegt hierbei darin, die strukturellen Informationen,

die auf der Quellseite in der Dokumentenordnung kodiert sind, auf die Ebene der Vater-Kind-Relation zu übertragen.

Grundsätzlich ist dazu zu sagen, dass es eher schlechter Stil ist, strukturelle Information in die Dokumentenordnung zu kodieren. Dies ist in manchen Formaten nicht möglich, oder nur mit erheblichem Aufwand realisierbar und erschwert die Transformation von Quell- in Zieldokument erheblich. Im Datenbank Bereich zum Beispiel ist es nur sehr schwer vorstellbar, den Index einer Tabelle als dessen Ordnung aufzufassen, da es nicht möglich ist, zwischen zwei Tupeln einer Tabelle mit fortlaufendem Index ein neues Tupel hinzuzufügen. Als Ergebnis einer Datenbankabfrage erhält man im Allgemeinen eine ungeordnete Menge an Tupeln, welche sich durch eine zusätzliche ORDER BY Klausel in die gewünschte Ordnung bringen lässt ([6], Abschnitt 13.1, General Rules 2 und 3). In XML lässt sich der Index der Datenbanktabelle mit der Dokumentenordnung von Elementen vergleichen. Hier ist es jedoch möglich, neue Elemente an beliebiger Stelle einzutragen. Dies veranlasst manche Entwickler dazu, Informationen auf diese Art zu kodieren. Dies ist dadurch möglich, dass eine Anfrage an ein XML-Dokument, zum Beispiel durch XPath, eine Node Liste zurückgibt, die automatisch die Dokumentenordnung widerspiegelt. Dadurch kann jedoch die Verarbeitung des XML-Dokuments nicht mehr zustandslos erfolgen. So muss sich gemerkt werden, welches das zuletzt gelesene Element in einer Reihe gleichförmiger Elemente war, um die Zuordnung zu erhalten.

Bei diesem in diesem Dokument gewählten Ansatz handelt es sich um eine rein deklarative Transformationssprache. Diese hat grundsätzlich keinen Zustand, was sich positiv auf die Modularität der Analyse auswirkt. Jedoch lässt sich keine Transformation beschreiben, die auf der Dokumentenordnung aufbaut. Ein Ansatz ist es, die Ordnung der Elemente als zusätzliches Attribut jedes Elements hinzuzufügen. Dies ist nicht ungewöhnlich und wird auch von Standard XML-verarbeitenden Sprachen verwendet, wie zum Beispiel auch XSLT [3], bei der es möglich ist, mit der eingebauten Funktion position() die Position des aktuellen Elements innerhalb einer Liste zu bestimmen.

Jedoch reicht dies nicht aus, um das Pallet Problem zu lösen. Hierbei ist der Vater des aktuell gelesenen Produktes die zuletzt gelesene Palette. Dies lässt sich nicht ohne das explizite oder implizite Speichern des Zustandes des Quelldokuments ausdrücken. Eine Möglichkeit wäre es, neue Operationen zur Adressierung auf der Zielseite hinzuzufügen, die einen Zugriff auf den zuletzt geschriebenen Datensatz eines bestimmten Typs zulässt. Ein Beispiel ist im Folgenden beschrieben.

Pfade: /a/b*/c*/d*; /pallets/pallet*/product*

```
schema palin{
  a{
    b *{
      i{int},
      c *{
        i{int},
        d *{
          i{int},
          n{int},
          l{string}
```

```
        }
      }
    }
  }
}
schema palout{
  pallets{
    pallet *{
      i{int},
      product *{
        i{int},
        bi{int},
        ci{int},
        n{int}
      }
    }
  }
}
transformation from palin to palout{
map /a/b/c/d to /pallets/pallet{
  case ./l = "M" {
    put where target:/pallets/pallet = new;
  }
}
map /a/b/c/d to /pallets/pallet/product {
  case ./l = "S" {
    put where target:/pallets/pallet = last
          && target:/pallets/pallet/product = new;
  }
}
map /a/b/c/d/i to /pallets/pallet/i {
  case ../l = "M" {
    put . where source:/a/b/c/d ~ target:/pallets/pallet;
  }
}
map /a/b/c/d/i to /pallets/pallet/product/i {
  case ../l = "S" {
    put . where source:/a/b/c/d ~ target:/pallets/pallet/product;
  }
}
map /a/b/c/d/n to /pallets/pallet/product/n {
  case ../l = "S" {
    put . where source:/a/b/c/d ~ target:/pallets/pallet/product;
  }
}
map /a/b/i to /pallets/pallet/product/bi {
  case true {
    put . where source:/a/b/c/d ~
                target:/pallets/pallet/product;
  }
}
map /a/b/c/i to /pallets/pallet/product/ci {
  case true {
    put . where source:/a/b/c/d ~
```

```
                    target:/pallets/pallet/product;
  }
}
}
```

Die Operation „new” erstellt hierbei einen neuen Datensatz auf dem angegebenen Pfad, die Operation „last” adressiert den letzten geschriebenen Datensatz, der durch den angegebenen Pfad adressiert werden kann. Der Pfad `/a/b/c/d` wird in diesem Beispiel entweder auf ein neues `/pallets/pallet` Element gemappet, wenn das Label „M” ist oder auf ein neues `/pallets/pallet/product` Element in der letzten Palette, wenn das Label „S” ist. Veranschaulicht ist dies in Abb. 2.

Eine weitere Möglichkeit ist es, eine Art globale Variablen in die Sprache einzubauen die es erlauben, bestimmte Daten für folgende Transformationen zwischenzuspeichern. Dies führt einen expliziten Zustand in der Sprache selbst ein.

Die dritte Möglichkeit wäre es, die Informationen über das zuletzt verarbeitete Elemente in das Zieldokument zu kodieren. Man könnte so versuchen, die ehemalige Position der Palette in das Zieldokument zu übernehmen und danach mit Hilfe dieser Position die richtige Palette zu finden.

```
...
map /a/b/c/d to /pallets/pallet {
  case ./l = "M" {
    put;
  }
}
map /a/b/c/d to /pallets/pallet/product {
  case ./l = "S" {
    put where target:/pallets/pallet/pos() =
      max(filter(/pallets/pallet/pos(), p => p < ./pos()));
  }
}
...
```

Dabei ergeben sich jedoch das Problem, dass definiert werden muss, welche Daten tatsächlich geschrieben werden sollen und welche nur zur temporären Zwischenspeicherung genutzt wurden. Dies widerspricht wiederum dem deklarativen Konzept der Sprache. Zum anderen müssen zur Adressierung Quantoren oder Funktionen eingeführt werden, um die richtige Palette adressieren zu können. Diese Palette hat eine Position, die kleiner ist als das aktuelle Produkt und *alle anderen* Paletten haben eine kleinere Position. Diese Quantoren oder Funktionen erschweren die Analyse, da deren Ergebnis im Allgemeinen nicht statisch beschrieben werden kann.

Unsere vorgeschlagene Lösung ist es, diesen Teil der Transformation in einen Vorverarbeitungsschritt zu verschieben. Dabei werden die eingehenden Daten nach einfachen matching-Regeln (sobald das Element mit dem Label erkannt wird, welches den Start einer neuen Palette markiert beende die alte Palette und erzeuge ein neues Paletten-Element, in das die kommenden Elemente geschrieben werden) vorverarbeitet. Dabei

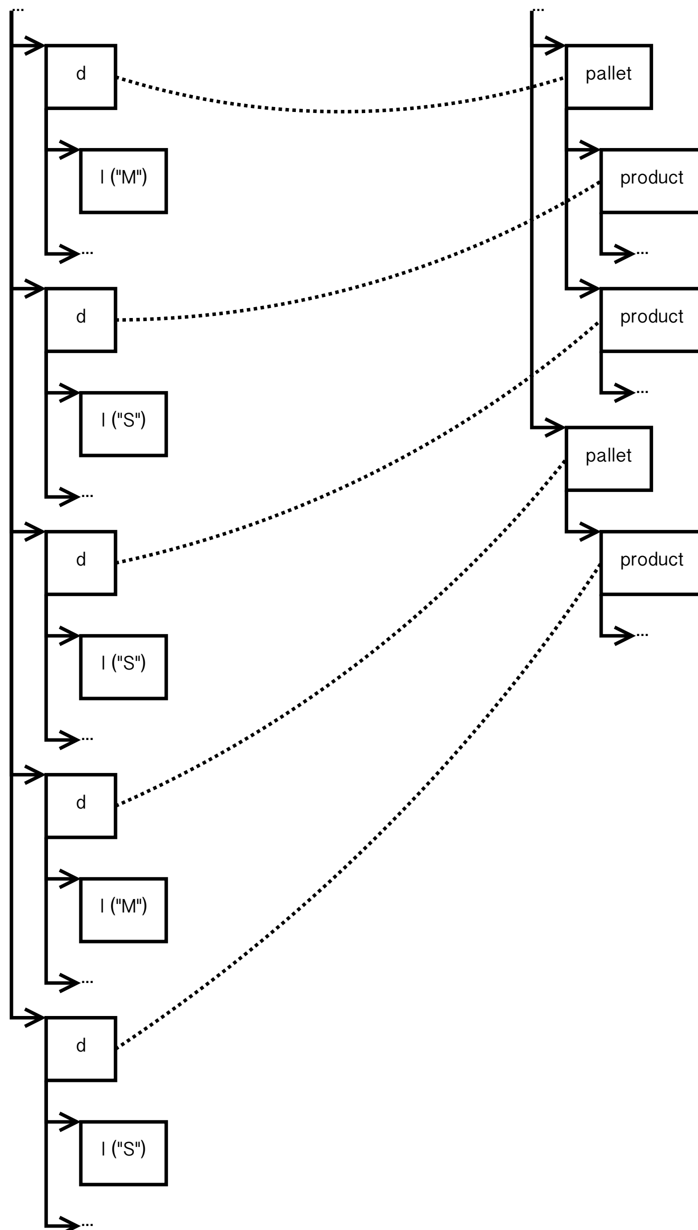

Abbildung 2: Transformation auf Basis der Dokumentordnung

entsteht eine hierarchische Struktur der Daten, die mit unserer Mappingsprache transformiert werden kann.

Fazit an dieser Stelle ist, dass Probleme, die auf der Dokumentordnung basieren, mit der vorgestellten Transformationssprache nicht ohne Vorverarbeitung lösbar sind.

4.5.4 Mögliche Erweiterung für Aggregation

Ein weiterer Use-Case ist Aggregation. Dabei werden mehrere Werte der Quellseite zu einem Wert der Zielseite zusammengefügt. Die vorgestellte Sprache kann um diese Möglichkeit erweitert werden, wie im folgenden kurz skizziert.

Um aggregierte Werte aus einer Menge von Elementen zu berechnen und diesen einem Element auf der Zielseite zuzuordnen ist eine Sonderform nötig, die eine Funktion auf das Ergebnis der Transformation anwendet und dabei das gewünschte Aggregat erzeugt.

```
schema inv {
  inventory{
    item * {
      weight{int},
      ...
    }
  }
}
schema ship [
  ship{
    container{
      weight{int},
      ...
    }
  }
}
transformation from inv to ship {
...
map /inventory/item/weight to /ship/container/weight by sum() {
  case true {
    put .;
  }
}
...
}
```

Vorteil des oben beschriebenen Ansatzes ist, dass auch für aggregierte Werte der direkte Mapping-Ansatz verwendet werden kann und somit die Übersicht über die Relationen zwischen Quell- und Zielschema im Toplevel erhalten bleibt. Ein Nachteil dieses Ansatzes ist jedoch, dass der aggregierte Wert nicht weiter verarbeitet werden kann, da er direkt einem Element auf der Zielseite zugeordnet wird und eine Transformation eines Zielelements nicht vorgesehen ist.

Um diese Beschränkung aufzuheben könnte man die Sprache um ein weiteres Konstrukt erweitern: Virtuelle Elemente. Diese Elemente existieren nicht im Quelldokument, son-

dern stellen Werte, die aus anderen Stellen des Dokuments oder aus Berechnungen stammen, unter einem Namen zur Verfügung. So könnte eine Summe als ein solches virtuelles Element definiert werden, welches somit in weiteren Mappings oder zur Berechnung weiterer Werte verwendet werden kann. Eine solche Definition könnte wie folgt aussehen:

```
schema inv {
  inventory{
    item * {
      weight{int},
      ...
    }
  }
}
schema ship [
  ship{
    container{
      weight{int},
      ...
    }
  }
}
transformation from inv to ship {
...
virtual /inventory/totalweight as /inventory/item/weight by sum(){
  case true {
    put .;
  }
}
map /inventory/totalweight to /ship/container/weight {
  case true {
    put .;
  }
}
...
}
```

Wie man in dem Beispiel sieht wird die Summe als neues virtuelles Element zur Verfügung gestellt, welches mit den normalen Sprachmittel weiterverarbeitet, zum Beispiel gemappt, werden kann. Dieses Konstrukt kann als mächtiges Werkzeug dienen, da die so erstellte Summe wiederum weiter verarbeitet werden kann. Auf ihrer Basis können weitere virtuelle Elemente erstellt werden, sie kann weiter aggregiert und in beliebig vielen Mappings verwendet werden.

Jedoch ergeben sich bei diesem Ansatz neue Probleme. So können dabei Abhängigkeiten zwischen den einzelnen virtuellen Elementen entstehen. Diese dürfen nicht zirkulär sein, da diese ansonsten nicht berechnet werden können. Dies hat unter anderem eine erhebliche Komplexität bei der Analyse zur Folge, weshalb dieses mögliche Konstrukt nur am Rande als mögliche Erweiterung erwähnt werden soll.

4.6 Syntaktische Erweiterungen (syntactic sugar)

Einige der hier vorgestellten Konstrukte sind für den Kern einer Mappingsprache durchaus interessant und ausreichend. Da der Kern so klein wie möglich gehalten werden soll wurde dabei kein großer Wert auf Benutzerfreundlichkeit gelegt. Jedoch können einige syntaktische Erweiterungen der Sprache dabei helfen, diese für zukünftige Benutzer interessant zu machen.

Bis jetzt wurde auf die möglichen Ausdrücke, die in einer `put` Anweisung verwendet werden können, nicht weiter eingegangen. Geplant ist jedoch, dass dies auch komplexere Berechnungen beinhalten können. Dies hat zur Folge, dass dabei der `put` Ausdruck erhebliche Ausmaße annehmen kann und dadurch nur schwer zu überschauen ist. Dies kann durch die Einführung von Variablen als unveränderbare Namen für einen nebeneffektfreien Teilausdruck verbessert werden.

Bei einem Ausdruck der Form

```
case B {
  v1 = A1
  v2 = A2
  ...
  vn = An
  put vn where PB;
}
```

mit v1 bis vn sind Variablenbezeichner und A1 bis An sind Ausdrücke werden in allen nachfolgenden Ausdrücken alle Vorkommen der bis zu diesem Punkt des Programms im case-Rumpf deklarierten Variablen durch deren Definition ersetzt. Da die Ausdrücke nebeneffektfrei sind bleibt die Bedeutung erhalten.

Case-Ausdrücke können nur eine Ebene tief sein. Das heißt, sie können nicht verschachtelt sein. Diese Verschachtelung kann jedoch auf recht einfache Weise syntaktisch hinzugefügt werden.

Ausdrücke der Form

```
case B1 {
  case B2 {
    ...
      case Bn{
        put ...;
      }
    ...
  }
}
```

können „ausmultipliziert“ werden. Die Verschachtelung kann durch Verundung der boolschen Ausdrücke aufgelöst werden. Die daraus entstehenden case-Ausdrücke haben die Form

```
case B1 && B2 && ... && Bn{
  put ...;
}
```

Da in vielen Transformationen gleichartige Elemente die durch den selben Pfad adressiert werden können auch gleichartig transformiert werden ist ein häufig vorkommendes Muster das Folgende:

```
...
map p1 to p2 {
  case true{
    put ...;
  }
}
...
```

Für diesen Spezialfall von `case true` kann eine Abkürzung eingeführt werden beziehungsweise der case-Ausdruck komplett entfallen. Damit wird obiges Beispiel zu:

```
...
map p1 to p2{
  put ...;
}
...
```

was automatisch wieder in obige Form transformiert wird.

Weitere syntaktische Erweiterungen für häufig verwendete Muster sind vorstellbar und müssten sich größten Teils aus der praktischen Verwendung der Sprache ergeben.

5 Analyse

Die hier beschriebene Mapping-Sprache bindet sowohl das Quell-, als auch das Zielschema in die Beschreibung der Transformation ein. Durch diesen Ansatz können zwei Dinge geprüft werden, die bei vielen anderen Sprachen zur Transformation, wie XQuery[4] und XSLT [3], nicht geprüft werden können. Einerseits stehen Information zu den gültigen Pfaden eines Schemas zur Verfügung, was die Plausibilitätsprüfung aller Pfade innerhalb der Transformation erlaubt. Somit können Tippfehler in Pfaden statisch geprüft und vor der Ausführung erkannt werden. Zum anderen kann auf Grund der Typ-Information in den Schemata eine Typprüfung der Mapping-Regeln durchgeführt werden. Dies ist für den aktuellen Stand der Sprache sehr einfach, da als Wertausdrücke des Inhalts der Elemente auf der Zielseite nur Konstanten und Zugriffe auf die Elemente des Quelldokuments erlaubt sind. Eine Erweiterung dieser Prüfung auf zusätzlich erlaubte Funktionen ist jedoch ohne Weiteres möglich.

Sowohl im Quell- als auch im Zielschema ist für jedes Element angegeben, ob es optional ist (* oder ?) oder ob es benötigt wird (+ oder normales Mapping). Ein interessanter Punkt ist, ob auf der Zielseite alle notwendigen Element erzeugt werden und ob von der Quellseite alle Elemente verarbeitet werden. Die Frage ist nun, in wie weit diese Eigenschaften statisch garantiert werden können. Des Weiteren stellt sich die Frage, wann es ein Fehler ist, wenn Daten, die im Eingabedokument enthalten sind, nicht verarbeitet werden.

Zuletzt stellt sich die Frage, was die Semantik eines Eingabedokuments sein soll, wie diese Semantik der Analyse bekannt gemacht werden kann und wie weit die Analyse die Erhaltung der Semantik garantieren kann.

5.1 Prüfung auf Vollständigkeit

Als eine der wichtigen Eigenschaften einer Transformation soll geprüft werden, ob alle Daten aus dem Eingabedokument auch transformiert wurden. Dazu muss als notwendige Bedingung, für jedes Element, das im Eingabedokument auftauchen kann, auch eine Mappingregel in der Transformation vorhanden sein. Dabei ist jedoch zu beachten, dass eine Verwendung eines Elementes auch indirekt durch Abhängigkeit eines Mappings vom Wert des Elements erfolgen kann.

Die (potentielle) Überdeckung der möglichen Eingabeelemete kann geprüft werden, indem für jedes Mapping die Menge der verwendeten Pfade berechnet wird und anschließend für jedes Element des Eingabeschemas geprüft wird, ob es in der Transformation Verwendung findet. Dabei erreicht man somit allerdings nur eine notwendige Bedingung und somit eine Annäherung. Die tatsächliche Abdeckung aller möglichen Eingaben ist statisch nur schwer zu prüfen, da diese unter anderem von der Vollständigkeit der Bedingungen in den case-Konstrukten abhängt. Diese wäre für einfache Fälle unkompliziert zu prüfen. Hat man für ein Element mit Typ „string“ die Information vorliegen, dass es nur die Werte `s1`, `s2`, ..., `sn` annehmen kann, so muss geprüft werden, ob für jeden dieser Fälle ein Mapping zur Verarbeitung vorliegt. Ohne solche Zusatzinformationen ist es jedoch statisch kaum möglich, die Vollständigkeit der Transformation zu garantieren. Für allgemeine boolsche Ausdrücke in den case-Bedingungen ist es rechnerisch unmöglich, die Vollständigkeit der Transformation zu prüfen.

Ein weiterer Punkt bei der Prüfung der Erhaltung aller Daten des Eingabedokuments ist auch, dass in den put-Ausdrücken auch Konstanten verwendet werden können. Es ist somit nicht notwendiger Weise der Fall, dass die gelesenen Daten der Eingabe auch tatsächlich in der Ausgabe erscheinen. Für eine vollständige Prüfung müsste man somit auch prüfen, dass die gelesenen Daten der Eingabe auch alle in die put-Ausdrücken auf die eine oder andere Weise eingehen. Jedoch muss auch hier darauf geachtet werden, dass die Information, nicht jedoch die rohen Daten erhalten bleiben müssen. So kann in verschiedenen System ein und die selbe Information durch verschiedenen Label symbolisiert werden. Die Prüfung dieser vollständigen Informationserhaltung ist somit nicht ohne weitere Hilfestellung durch den Programmierer möglich.

Eine Transformation muss nicht zwangsläufig fehlerhaft sein, wenn ein Element der Eingabe zwar durch ein Mapping verarbeitet wurde, dieser errechnete Wert jedoch nicht in das Ausgabedokument aufgenommen wird. Ein Beispiel hierfür ist die Auflösung von Referenzen.

```
schema refin {
  root {
    bestellungen {
```

```
      bestellung * {
        id{int},
        prodRef * {int}
      }
    }
    produkte {
      produkt * {
        id {int}
      }
    }
  }
}
schema refout {
  root {
    bestellung * {
      id {int},
      produkt * {
        id {int}
      }
    }
  }
}
transformation from refin to refout {
map /root to /root {
  put;
}
map /root/bestellungen/bestellung to /root/bestellung {
  put;
}
map /root/bestellungen/bestellung/id to /root/bestellung/id {
  put where source:/root/bestellungen/bestellung ~
            target:/root/bestellung;
}
map /root/bestellungen/bestellung/prodRef to /root/bestellung/produkt {
  put where source:/root/bestellungen/bestellung ~
            target:/root/bestellung;
}
map * /root/produkte/produkt/id to /root/bestellung/produkt/id {
  put . where source:/root/bestellungen/bestellung/prodRef = ../id &&
                  //join über prodRef-Element
              source:/root/bestellungen/bestellung/prodRef ~
              target:/root/bestellung/produkt;
}
}
```

Die Produkte sind auf der Quellseite alle aufgelistet, die Bestellungen sind mit entsprechenden Referenzen auf die Produkte versehen. Beim Mapping werden diese beiden Komponenten zusammengesetzt, sodass die Produkte sich anschließend unterhalb der Bestellungen befinden. Dabei kann es jedoch vorkommen, dass Produkte nicht referenziert wurden, da sie in keiner Bestellung vorkommen und dadurch auf der Zielseite nicht auftauchen. Die Annahme, dass alle Daten auch tatsächlich geschrieben werden müssen, hat sich somit als falsch herausgestellt. Auch muss zur weiteren Prüfung zur Laufzeit

der Programmierer angeben, ob der Datensatz nur einmal oder potentiell mehrmals im Ausgabedokument auftauchen soll. Deshalb besitzt das map-Konstrukt zusätzlich einen Operator, der die Multiplizität der Ausgabe angibt. `map * ...` signalisiert, dass das Ergebnis des Mappings potentiell an mehreren Stellen im Ausgabedokument auftauchen soll. Jedoch kann es auch sein, dass das Ergebnis nicht in der Ausgabe auftaucht. `map + ...` dagegen forciert die Ausgabe, womit dynamisch am Ende der Erzeugung des Ausgabedokuments ein Fehler ausgegeben wird, sobald das Element nicht in der Ausgabe geschrieben wurde. Ähnlich verhält es sich mit dem Operatoren `map ? ...`, welcher angibt, dass das Ergebnis höchstens einmal in der Ausgabe auftauchen kann. Ist kein Operator angegeben, so muss das Ergebnis genau einmal in der Ausgabe auftauchen. Somit kann zur Laufzeit geprüft werden, ob die Ausgabe vollständig ist.

5.2 Semantik

Die Semantik eines Eingabedokuments ist in der Praxis komplex. Jedoch kann diese Semantik nicht in der Verarbeitung der Daten in automatischer Form verwendet werden. Was ein Name eines Kunden ist und was nicht, liegt nicht im Rahmen dessen, was ein Computer entscheiden kann. Die realweltlichen Entitäten hinter den Werten bleiben außen vor.

Die Abstraktion, die an dieser Stelle verwendet wird, ist eine Menge von Beschränkungen, die als Teil des Schemas angesehen werden können. So kann der Wertebereich eines Elements auf einen Teil des möglichen Wertebereichs eingeschränkt werden. Ein Beispiel, das schon in den Anforderungen aus der Praxis erwähnt wurde, ist die Postleitzahl, die als regulärer Ausdruck „[0-9]{5}“ dargestellt werden kann. Die Semantik der Daten würde nach dieser Annahme erhalten bleiben, wenn im Eingabedokument die Beschränkungen des Quellschemas und nach der Transformation die Beschränkungen des Zielschemas gelten. Diese Erhaltung der Beschränkungen statisch zu prüfen ist im Allgemeinen nicht einfach, wenn beliebige Ausdrücke als Ergebnis eines Mappings erlaubt werden, da dabei die gültigen Beschränkungen des Ergebnisses neben den Beschränkungen der Eingabe auch in komplexer Weise von der Semantik der verwendeten Funktionen abhängt. Hier müsste man mit Methoden der Vor- und Nachbedingungen arbeiten, um zum Beispiel aus den Bedingungen, die für das Element auf der Zielseite gelten, auf die schwächste Vorbedingung zu schließen, die für das entsprechende Element auf der Quellseite gelten muss.

Eine dynamische Prüfung zur Laufzeit könnte jedoch eingebaut werden, um bei der Transformation entstehende Inkonsistenzen zu vermeiden. Dies ist möglich, da für jedes erzeugte Element aus den Mappings bekannt ist, auf welche Stelle der Zielseite dieser Wert geschrieben werden wird. Außerdem ist das Zielschema und damit die Bedingungen, die für dieses Zielelement gelten müssen, bekannt, und diese können anhand der konkreten erzeugten Daten geprüft werden.

6 Fazit

Die Probleme einer deklarativen Beschreibungssprache für Transformationen reduziert sich im Wesentlichen auf das Problem der Adressierung auf der Zielseite (oder bei umgekehrter Herangehensweise auf die Adressierung der Elemente der Quellseite). Dabei spielt die Identität eines solchen Elements und wie zwischen diesen Elementen unterschieden werden kann, eine wichtige Rolle. Diese Identität und wie die Zuordnung der Vater-Kind-Beziehung von der Quell- auf die Zielseite übertragen werden kann, kann in unterschiedlicher Weise erfolgen. Dabei ergeben sich bei unterschiedlichen Ansätzen unterschiedliche Probleme.

Verlangt man, dass in den Daten schon ein explizites Element existiert, welches das Vaterelement eindeutig identifiziert, so reduziert sich die Zuordnung der restlichen Kinder auf die Überprüfung des Wertes diese ID-Elements. Jedoch ist in der Praxis nicht immer ein solches Element gegeben, das alleine eine eindeutige Identifikation des Vaterelements ermöglicht. So muss bei zusammengesetzten Schlüsseln ein neues Element künstlich hinzugefügt werden, welches als neue ID verwendet werden kann. Auch stellt sich die Frage weiterhin, wie die Unterscheidung auf der Zielseite passieren soll, wenn ein 1-zu-n Mapping aus einem dieser Elemente auf der Quellseite mehrere Elemente auf der Zielseite erzeugt. Die ID dieser Elemente ist ohne Anpassung in diesem Fall nicht mehr eindeutig.

Der hier gewählte Ansatz, eine Äquivalenzrelation zu verwenden, die Elemente der Quellseite mit den Elementen in Relation setzt, die aus diesen Elementen auf der Zielseite erzeugt werden, hat den Vorteil, dass die daran anschließende Übertragung der Schlüssel immer ein eindeutiges Ziel besitzt, das Element, welches in Relation mit dem Vater des Schlüssels ist. Damit können auch aus mehreren Teilen bestehende Schlüssel behandelt werden. Zudem müssen keine neuen Schlüssel in die Daten eingefügt werden, um die Adressierung zu erhalten. Jedoch hat somit jedes Element auf der Quellseite seine eigene Identität. Es ist nicht möglich, mehrere Elemente mit der gleichen Identität bzw. auf das exakt gleiche Zielelement zu mappen, wie dies beim Umstrukturierungen (siehe Kapitel 4.5.2) geschehen müsste. Außerdem besteht das oben angesprochene Problem weiterhin. Bei einem 1-zu-n Mapping werden mehrere nicht-identische Elemente erzeugt, die in Relation mit dem gleichen Quellelement stehen und somit äquivalent und nicht mehr unterscheidbar sind.

6.1 Gelöste Probleme

Für den Transformationsentwickler ist es durch die spezielle Syntax der Sprache sehr einfach, sich einen Überblick über die verschiedenen Zusammenhänge zwischen Quell- und Zielschema zu machen. Dabei entspricht jedes Mapping einer Verbindung zwischen den Schemata. Für jeden Pfad kann geprüft werden, ob dieser im Quell- bzw. Zielschema existiert. Dadurch werden Tippfehler schnell erkannt. Die Transformation kann auf richtige Typisierung geprüft werden. Dies betrifft sowohl das Quell- als auch das Zielschema und die Verbindung der beiden Schemata in Form von Mappings.

Es kann für jedes Element im Quellschema geprüft werden, ob es ein dieses Element verarbeitendes Mapping gibt. Dies ist eine notwendige Bedingung für die Richtigkeit des Mappings und die Erhaltung aller Daten des Eingabedokuments. Es kann jedoch statisch nicht auf einfache Weise geprüft werden, ob die Daten auch in jedem Fall in das Ausgabedokument eingehen. Gleichfalls kann für jedes Element des Zielschemas geprüft werden, ob es ein dieses Element erzeugendes Mapping gibt. Dies ist eine notwendige Bedingung für die Richtigkeit des Ausgabedokuments und dessen Einhaltung des Schemas. Es kann jedoch statisch nicht auf einfache Weise geprüft werden, ob dieses Element auch für jedes beliebige Eingabedokument mit der richtige Kardinalität erzeugt wird. Dies würde weitere Untersuchungen in Richtung SAT/SMT Solver benötigen.

6.2 Ungelöste Probleme

Weiter ungelöst bleibt die Frage, wie mit Aggregation in der Sprache umgegangen werden soll. Schritte in eine mögliche Richtung wurden bereits in Kapitel 4.5.4 gegeben. Jedoch müssten diese Punkte weiter untersucht und deren Machbarkeit gerade auch in Bezug auf deren Einbeziehung in die Analyse geprüft werden. Die Verarbeitung von Strings wurde bis jetzt komplett ignoriert. Es wird angenommen, dass diese Verarbeitung komplett in eine ohnehin notwendige Vortransformation der Eingabedaten verschoben werden kann. Dies müsste weiter untersucht werden und eventuell verbleibende Transformationsschritte in Form von weiteren Sprachkonstrukten zur Verfügung gestellt werden. Die Sprache bietet keine explizite Unterstützung für die Konvertierung zwischen Daten der Wert- und der Metaebene. Es kann also nicht direkt ein Element erzeugt werden, dessen Namen auf einem Wert im Quelldokument basiert. Dies ist jedoch unproblematisch, da die Schemata statisch und endlich sind. Somit können diese Konvertierungen mit Hilfe von Filter-Ausdrücken und explizite Angabe des Wertes und Ziels gelöst werden.

Probleme, die in der aktuellen Fassung durch die Sprache nicht abgedeckt sind, sind das Zusammenführen von Elementen anhand einer festgelegten Definition von Äquivalenz, wie es bei der Umkehrung der Vater-Kind-Relation vorkommt, sowie Probleme, die sich nur mit Hilfe der Dokumentordnung lösen lassen.

Bei der statischen Prüfung stellte sich als noch nicht gelöst heraus zu prüfen, ob die Kardinalitäten der Element auf der Zielseite eingehalten wird. Für ein Element der Zielseite können alle Regeln ermittelt werden, die dieses Element erzeugen. Danach hängt die Kardinalität dieses Zielelements von der Kardinalität der Quellelemente und ob sie optional sind, sowie von den Filterregeln ab. Die Veroderung der Filterregeln muss true sein, die paarweise Verundung muss false sein, damit genau eine der Regeln angewendet wird und somit das Element nur genau einmal erzeugt wird. Problem sind dabei die zusätzlichen Bedingungen der Zuordnung auf der Zielseite und wie diese dem SAT/SMT-Solver mitgeteilt werden müssen, sodass dieser keine unmöglichen Fälle erzeugen kann.

6.3 Implementierung

Neben der schriftlichen Ausarbeitung gibt es zu der vorgestellten Sprache eine Implementierung in Scala. Die Umsetzung hat noch einige Einschränkungen. So können als Ausdrücke des Wertes, der auf auf der Zielseite geschrieben werden soll, nur eine einzelner Pfadausdruck verwendet werden, ohne Anwendung von Funktionenen. Außerdem sind die Filterausdrücke auf Gleichheit, kleiner-als und Negation beschränkt. Jedoch können schon interessante Beispieltransformationen anhand einer Umformung von einem Dokument in reduziertem XML zu einem XML-Dokument durchgeführt werden. Dabei wird davon ausgegangen, dass in dem XML-Dokument keine Attribute verwendet werden, um die Konformität mit dem in der Transformationsbeschreibung definierten Schema zu vereinfachen. Einige vorhandene Beispiele zeigen die Relation anhand von ID-Werten, ein 1-zu-n Mapping, die Restrukturierung einer Referenz sowie ein Beispiel, welches die Problematik des Zusammenführen von Elementen verdeutlicht und ein Beispiel für das Zusammenführen einer Referenz zu einer hierarchischen Struktur.

Literatur

[1] David Bednárek. Extending datalog to cover XQuery*.

[2] Don Chamberlin et al. XQuery Update Facility 1.0. *W3C Candidate Recommendation*, 2011.

[3] J. Clark et al. XSL transformations (XSLT) version 1.0. *W3C recommendation*, 16(11), 1999.

[4] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics. W3C Working Draft, 2005.

[5] Tim Hinrich. Herbrand logic. `http://people.cs.uchicago.edu/~thinrich/herbrand/html/applications.html`, 2006.

[6] American National Standards Institute. Database language - SQL, 1992.

[7] A Laux and L Martin. XUpdate working draft, 2000. In *Proceedings of the International Conference on Very Large Databases*, 2000. last accessed on July 29, 2011; `http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html`.

[8] Wolfgang May. Integration of XML Data in XPathLog, 2001.

[9] Martin Suda. Logic programming, datalog, and negation. `http://www.mpi-inf.mpg.de/~suda/datalogandnegation_short.pdf`, 2010.