Automatisierte Modelladaptionen durch Evolution - RELaX in the Garden of Eden

Thomas Nösinger, Meike Klettke, Andreas Heuer
Lehrstuhl Datenbank- und Informationssysteme
Institut für Informatik
Universität Rostock
(tn, meike, ah)@informatik.uni-rostock.de

Abstract: Die in diesem Artikel vorgestellte Methode zur *XML-Schemaevolution* automatisiert ebenenübergreifend Veränderungen ausgehend von einem konzeptionellen Modell über XML-Schemata bis hin zu XML-Dokumenten; Schwerpunkt dabei ist die Entwicklung der Evolutionssprache ELaX (Evolution Language for XML-Schema).

1 Motivation

XML ist nach wie vor das Format zum Austausch und zur Speicherung von strukturierten und semistrukturierten Daten und wird in unzähligen Wissenschaftsbereichen wie Biologie, Medizin und industriellen Anwendungen eingesetzt.

Die Einfachheit von XML ist der Hauptgrund
für die große Verbreitung des Formates.

Die meisten Anwendungen verwenden XML-Schemata als Mechanismus zur Beschreibung der Struktur und des Aufbaus von XML-Dokumenten.

XML-Schema ist keineswegs einfach, wird von den Anwendern
jedoch durch den Einsatz geeigneter Tools beherrscht.

Bereits aus den 70er Jahren ist aus der Softwaretechnik bekannt: “systems must be continually adapted else they become progressivelyless satisfactory [LRW+97]“. Alle Anwendungsgebiete von XML (außer temporären Austauschformaten) sind davon betroffen. Hauptgrund für Änderungen sind veränderte Anforderungen, die sich aus einer Änderung der Umgebung ergeben. Jede Weiterentwicklung führt über kurz oder lang zu Adaptionen, sowohl eines Schemas als auch aller bereits gespeicherten XML-Dokumente. Dieser Prozess wird auch *Evolution* genannt. Nach wie vor ist die Möglichkeit zur Adaption für XML Schemata nicht vorgesehen und kann nur über Umwege erreicht werden.

Die Evolution von XML-Anwendungen ist hochkomplex und
fehleranfällig - eine Toolunterstützung ist dafür zwingend notwendig.

Es besteht also die Notwendigkeit, eine Weiterentwicklung (Evolution) von XML-Anwendungen intuitiv und leicht bedienbar, mit so wenig Interaktion wie möglich, robust und fehlerfrei anbieten zu können. In diesem Artikel wird ein Verfahren gezeigt, das die Nutzerinteraktion dabei auf ein Minimum beschränkt. Es wird dargestellt, mittels welcher Mechanismen Änderungen übergreifend zwischen verschiedenen Informationen (Modell, Schema, Dokument) erfolgen können und welche formalen Schnittstellen benötigt werden.

Die nachfolgend thematisierte *XML-Schemaevolution* ist ein Spezialfall obiger Evolution, bei der XML-Dokumente bei Änderung des zugeordneten XML-Schemas automatisch angepasst werden. Das XML-Schema wird dabei mit Hilfe eines konzeptionellen Modells grafisch dargestellt. Ziel ist es, die Erstellung und Veränderung von XML-Schema für den Benutzer zu erleichtern. Eine Weiterentwicklung oder Evolution eines Schemas erfolgt durch Editieroperationen auf dem konzeptionellen Modell. Dafür stehen Änderungsmöglichkeiten bereit, die an die Mächtigkeit von XML-Schema angepasst sind. Die vom Benutzer ausgeführten Operationen auf dem konzeptionellen Modell werden geloggt und ausgewertet (siehe auch [NKH12]). Eine XML-Schemaevolution bedingt auch eine erforderliche Anpassung der XML-Dokumente, die dem XML-Schema zugeordnet sind. Um die Gültigkeit der XML-Dokumente auch nach Schemaänderung zu gewährleisten, müssen gegebenenfalls die XML-Dokumente ebenso adaptiert werden. Dazu werden die protokollierten Änderungsoperationen in Transformationsschritte übersetzt, die auf XML-Dokumente angewendet werden können. Diese Transformationen werden als XSLT (Extensible Stylesheet Language Transformation) Skripte gespeichert. Eingebettet ist die Evolution von XML-Dokumentkollektionen in den an der Universität Rostock entwickelten Forschungsprototypen *CodeX* (Conceptual Design und Evolution for XML Schema).

Der Artikel ist wie folgt gegliedert: Zunächst wird in **Abschnitt 2** der aktuelle Forschungsstand der *XML-Schemaevolution* dargestellt, der Fokus wird auf die kommerziellen XML-DBMS von IBM, Oracle, Microsoft und Software AG sowie Altova und Forschungsprototypen gelegt. **Abschnitt 3** dient der Definition der formalen Grundlagen, hier wird gezeigt, welche Modellebenen, Korrespondenzen und XML-Schema Organisationsformen existieren. Diese Grundlagen werden für den **Abschnitt 4** verwendet, in dem die *Evolutionssprache ELaX (Evolution Language for XML-Schema)* zur Anpassung von *XML-Schema* definiert wird. Die Anwendung der Evolutionssprache wird danach an einem umfassenden Beispiel in **Abschnitt 5** erläutert, bevor ein kurzer Überblick über die derzeitige Umsetzung des Forschungsprototypen *CodeX* in **Abschnitt 6** gegeben wird. Der Artikel schließt mit einer Zusammenfassung in **Abschnitt 7** und einem Ausblick in **Abschnitt 8**.

2 State of the Art

Die *XML-Schemaevolution* wird sowohl von den großen Datenbank- und Softwareherstellern (u.a. Microsoft [mic11], IBM [db212], Oracle [ora12], Altova [alt12], Software AG [tam12]) als auch in Forschungsprototypen (u.a. XCase [KKLM09], PRISM++ [CMDZ10], X-Evolution [GM08], EXup [Cav10] und GEA [DLP+11]) thematisch behandelt und auch teilweise umgesetzt.

Microsoft SQL Server unterstützt den XML-Datentyp, lässt den Benutzer in Collections XML-Schemata sammeln und danach mittels einer Typisierung die Spalten einer Relation einem Schema zuordnen. XML-Instanzen typisierter Spalten können bezüglich ihres XML-Schemas auf Gültigkeit geprüft werden, ändert sich allerdings ein XML-Schema, muss dieses unter einer neuen Version erneut eingefügt werden. Microsoft unterstützt die hier angestrebte *XML-Schemaevolution* nicht, die gegenwärtig verfügbare Lösung stellt eine Versionierung von XML-Schemata dar.

IBM DB2 ermöglicht die Registrierung von XML-Schemata in einem XML-Schema-Repository (XSR) und eine anschließende Erweiterung dieser unter Beachtung von zehn strengen Kompatibilitätsanforderungen. Es werden die Prozeduren XSR_REGISTER und XSR_UPDATE und/oder die Befehle REGISTER XMLSCHEMA und UPDATE XMLSCHEMA verwendet, sodass das alte XML-Schema gegen ein neues XML-Schema im XSR austauscht wird. Sind beide Schemata nicht zueinander kompatibel, wird eine Fehlermeldung generiert, "es findet keine Weiterentwicklung des Schemas statt [db212]". In Oracle können Schemata ebenfalls registriert und anschließend mittels der Methoden copyEvolve und inPlaceEvolve evolutioniert werden. Beim copyEvolve-Ansatz werden die gültigen XML-Dokumente kopiert und temporär zwischengespeichert, das alte XML-Schema wird gelöscht und das neue XML-Schema wird registriert. Danach werden alle XML-Dokumente dem neuen Schema zugeordnet, allerdings nur, wenn keine Fehler bei der Validierung auftreten. Im Fall eines Fehlers, u.a. wenn die vorhandenen, ehemals gültigen XML-Dokumente nicht länger valide sind, wird das alte XML-Schema reaktiviert (rollback). Der inPlaceEvolve-Ansatz benötigt ein diffXML-Dokument, in welchem alle Änderungen des XML-Schemas spezifiziert werden. Dabei muss allerdings eine Rückwärtskompatibilität gewährleistet werden, d.h. alle gültigen XML-Dokumente müssen ebenfalls valide bezüglich des evolutionierten Schemas sein. Dies ist nur unter Beachtung strenger Einschränkungen anwendbarer Evolutionsschritte möglich. Sowohl bei IBM als auch Oracle sind restriktive Anforderungen gestellt, die nur eine Generalisierung des alten XML-Schemas ermöglichen und somit nur teilweise die angestrebte *XML-Schemaevolution* unterstützen.

In der Dokumentation vom Tamino XML-Server der Software AG wird festgestellt, dass es unwahrscheinlich ist, dass einmal definierte Schemata immer den gleichen Zustand haben werden. Daher wird eine "sichere" Variante der *XML-Schemaevolution* vorgestellt, in der ähnlich zu IBM und Oracle nur Generalisierungen erfolgen sollten. Dies wird in allgemeinen Richtlinien definiert. In diesem Kontext wird bei Tamino darauf hingewiesen, dass falls XML-Dokumente nicht valide bezüglich des geänderten Schemas sind, demnach nicht generalisierende Schemaänderungen durchgeführt wurden, diese manuell vom Nutzer angepasst werden müssen.

Altova bietet mit Diffdog eine Erweiterung ihres Editors an, mit dem zwei Schemata miteinander verglichen werden können, um nachfolgend XSLT-Skripte zur Transformation der Instanzen zu erzeugen. Treten bei dem Vergleich Konflikte auf, d.h. es kann keine eindeutige Zuordnung zwischen den unterschiedlichen Strukturen der XML-Schemata hergeleitet werden, dann wird vom Anwender eine manuelle Zuordnung verlangt. Eine Automatisierung ist nicht möglich, eine Nutzerinteraktion und somit Expertenwissen bezüglich der XML-Schemata ist erforderlich.

Die Prototypen X-Evolution [GM08] und EXup [Cav10] nutzen eine grafische Oberfläche zur Spezifikation, Ausführung und Verifikation von Schemaänderungen. Diese werden durch Primitive beschrieben, wobei die Updatesprache XSchemaUpdate für die Beschreibung der Änderungen und die Dokumentadaption verwendet wird. Eine Grundlage dieser Systeme ist ein DBMS, welches XMLTYPE unterstützt. EXup und X-Evolution verwenden darüber hinaus XSUpdate als Evolutionssprache, welches XSPath nutzt (eine Teilmenge von XPath).

PRISM++ [CMDZ10] bietet einen fein-granularen Evolutionsmechanismus zur Evolution von Datenbankschemata, welcher Datenbankadministratoren unter Verwendung von SMO-ICMO (Evolutionssprache: Schema Modification Operators - Integrity Constraints Modification Operators) die Evolution erleichtern soll. PRISM++ ermöglicht keine XML-Schemaevolution, thematisiert allerdings die Notwendigkeit, auch Integritätsänderungen vollziehen zu können.

In [DLP+11] wird das GEA Framework (Generic Evolution Architecture) vorgestellt, in dem XML-Schemata als UML-Klassendiagramme mit Stereotypen beschrieben werden. Unter Verwendung von elementaren Transformationsregeln werden Änderungen in UML auf XML propagiert.

XCase [KKLM09] ist eine Implementierung von XSEM (konzeptionelles Modell: Xml SEmantics Modeling), welches unter Verwendung einer MDA (Model-Driven Architecture) die *XML-Schemaevolution* durchführt. Es existieren unterschiedliche Abstraktionslevel (u.a. PIM als Platform-Independent Model und PSM als Platform-Specific Model) die Änderungen auf abstrakterem Niveau ermöglichen und diese dann zwischen den verschiedenen Ebenen propagieren (ein XML-Schema ist ein PSM). Dieser Ansatz ist dem hier vorgestellten am ähnlichsten, unterscheidet sich aber grundlegend in der Herangehensweise der Erfassung von Änderungen, deren Auswertung, Kategorisierung und dem Umfang möglicher Änderungen bezüglich eines XML-Schemas.

Eine automatische Erzeugung von Transformationsschritten und die damit verbundene Anpassung von XML-Dokumenten sind in den hier vorgestellten Forschungsprototypen nicht möglich bzw. die umsetzbaren Änderungen sind nicht umfangreich genug. Des Weiteren wird bei keinem der Prototypen der hier vorgestellte Ansatz der *XML-Schemaevolution* verfolgt, es besteht somit weiterhin Forschungsbedarf.

3 Grundlagen

Grundlage und Ausgangspunkt für die *XML-Schemaevolution* sind die formalen Spezifikationen der zur Grunde liegenden Modelle, sowie die Korrespondenzen zwischen diesen. Des Weiteren sind die Operationen zur Adaption der Modelle ein wichtiger Aspekt in der Schemaevolution, denn aus diesen werden die zur Anpassung notwendigen Informationen abgeleitet. Die Zusammenhänge sind in Abbildung 1 dargestellt. Im Einzelnen existieren drei Ebenen, namentlich das an der Universität Rostock entwickelte konzeptionelle EMX (Entity Model for XML-Schema [Kle07]), das durch das W3C standardisierte XML-Schema XSD (XML Schema Definition), sowie die unterste Ebene, die XML-

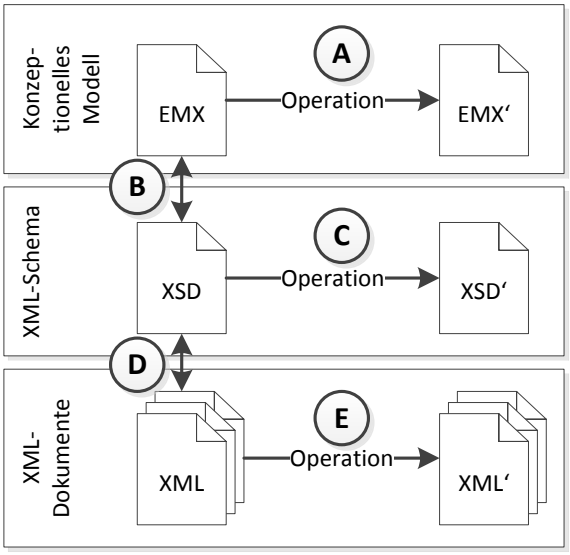


Abbildung 1: EMX, XML-Schema und XML-Instanzen

Dokumente. Auf dem konzeptionellen Modell (EMX) werden vordefinierte Änderungsoperationen vom Nutzer durchgeführt (*Punkt A*), die anwendbaren Operationen lassen sich gemäß der Kategorisierung von Änderungsoperationen herleiten und charakterisieren (siehe [Gru11]). Vom Nutzer anwendbare Operationen auf dem konzeptionellen Modell, d.h. von CodeX geloggte Entwurfsschritte, sind zum Beispiel: *changeElementType* (Umsortieren vom Inhaltsmodell), *changeElementRefMinOccur* (Erhöhen des minimalen Vorkommens einer Elementreferenz), *addElementToComplexElementType* (Einfügen nicht-optionaler Elemente), *renameElement* (Umbenennen eines Elementes) usw. Die Umsetzung der Operation renameElement wurde exemplarisch in [NKH12] dargestellt. Des Weiteren erfolgt eine formale Definition von renameElement auf den unterschiedlichen Ebenen (d.h. *Punkt A*, *Punkt C* und auch *Punkt E*), bevor anhand eines kurzen Beispiels die *XML-Schemaevolution* erläutert wird.

3.1 Formale Definitionen der Modelle

Das konzeptionelle Modell (M_M) ist ein Tripel, das aus Knoten (N_M), gerichteten Kanten zwischen Knoten (E_M) und zusätzlichen Eigenschaften (F_M) besteht. EMX ist die vereinfachte, grafische Repräsentation eines XML-Schemas und wurde direkt an die Möglichkeiten und Erfordernisse von XML-Schema angepasst. Unter Beachtung der formalen Definition von XSD wurden die folgende Knotentypen eingeführt: Elemente (*elements*$_M$), Attributgruppen (*attributegroups*$_M$), Inhaltsmodelle (*groups*$_M$), Typen (einfache (*simple-types*$_M$), komplexe (*complex-types*$_M$)), Module (*modules*$_M$, u.a. externe XML-Schemata), Annotationen (*annotations*$_M$) oder Integritätsbedingungen (*integrity-constraints*$_M$).

Die gerichteten Kanten werden jeweils zwischen Knoten definiert, wobei die Richtung die Zugehörigkeit (Enthalten-sein Beziehung) umsetzt und gemäß der Möglichkeiten von XML-Schema festgelegt wurden. Die zusätzlich zu definierenden Eigenschaften ermöglichen die Nutzer-spezifische Anpassung eines konzeptionellen Modells. Diese Eigenschaften und nachfolgend die entsprechenden zusätzlichen Eigenschaften von XSD und XML-Dokumenten sollen an dieser Stelle nicht näher beleuchtet werden, es sei an dieser Stelle auf [NKH12] verwiesen.

XML-Schema als strukturelle Beschreibung von XML-Instanzen ist formal durch das W3C definiert [xml12]. Ein XML-Schema (M_S) ist ein Tupel aus Knoten (N_S) und zusätzlichen Eigenschaften (F_S). Beziehungen zwischen den Knoten (z.B. gerichtete Kanten) sind implizit in den Knoten enthalten. Knoten sind laut *abstraktem Datenmodell* entweder Definitionen (*type-definitions*$_S$), Deklarationen (*declarations*$_S$), Modellgruppen (*model-group-components*$_S$), Gruppendefinitionen (*group-definitions*$_S$), Annotationen (*annotations*$_S$) oder Bedingungen (*constraints*$_S$). Neben der abstrakten Definition von Knoten existiert die *Element-Informationseinheit*, die für jeden Knoten die Realisierung innerhalb eines XML-Schemas definiert, d.h. welche Inhalte und Attribute können verwendet werden oder nicht.

XML-Instanzen (M_D) sind analog zum XML-Schema Tupel aus Knoten (N_D) und zusätzlichen Eigenschaften (F_D), wobei die Beziehungen zwischen den Knoten wieder implizit in diesen enthalten sind. Die Knoten einer XML-Instanz sind Dokumente (*documents*$_D$), Attribute (*attributes*$_D$), Prozessanweisungen (*processing-instructions*$_D$), Texte (*texts*$_D$), Namensräume (*namespaces*$_D$) oder Kommentare (*comments*$_D$) [xml10].

3.2 Korrespondenzen zwischen den Modellen

Zwischen den einzelnen Modellen existieren Korrespondenzen, die eine ebenenübergreifende Abbildung ermöglichen. Diese Korrespondenzen sind in Abbildung 1 dargestellt, dies sind zum einen die eindeutige Korrespondenz zwischen EMX und XML-Schema (*Punkt B*), zum anderen die mehrdeutigen Korrespondenz zwischen XML-Schema und XML-Dokumenten (*Punkt D*).

Die Mehrdeutigkeit der Korrespondenz zwischen einem Schema und einem Dokument entsteht durch die strukturellen Möglichkeiten, die XML-Schema bietet. Dazu zählen unter anderem Inhaltsmodelle wie Alternativen (*choice*), optionale Elemente (*minOccurs="0"*) und/oder Attribute (*use="optional"*), Wildcards (*any* bzw. *anyAttribute*), usw. Trotz der Mehrdeutigkeit kann für jedes wohlgeformte XML-Dokument beim Vorhandensein eines wohlgeformten, referenzierten XML-Schemas die Gültigkeit des Dokumentes überprüft werden.

Die Korrespondenzen zwischen EMX und XML-Schema sind in Tabelle 1 dargestellt. Es werden zu jedem *Knotentypen* von EMX (N_M) die entsprechenden Knoten des *abstrakten Datenmodells* von XSD (N_S) aufgezählt. Des Weiteren werden Informationen zur internen Speicherung von EMX in Form von Relationennamen, als auch die Realisierungen in der Element-Informationseinheit des XML-Schemas dargestellt. Elemente des Knoten-

EMX		XSD	
Knotentyp	**Interne Speicherung**	**Abstraktes Datenmodell**	**Element-Informationseinheit**
elements$_M$	element, element_ref, wildcard	*declarations*$_S$	<element>
attributegroups$_M$	attribute, attribute_ref, attribute_gr, attribute_gr_ref, wildcard	*declarations*$_S$, *group-definitions*$_S$	<attribute>, <attributeGroup>
groups$_M$	group	*model-group-components*$_S$	<all>, <choice>, <sequence>
simple-types$_M$	st, st_list, facet	*type-definitions*$_S$	<simpleType>
complex-types$_M$	ct	*type-definitions*$_S$	<complexType>
modules$_M$	module	-	<include>, <import>, <redefine>
annotations$_M$	annotation	*annotations*$_S$	<annotation>
integrity-constraints$_M$	constraint, path	*constraints*$_S$	<key>, <unique>, <keyref>
-	schema	-	<schema>

Tabelle 1: Korrespondenzen zwischen konzeptionellem Modell (EMX) und XSD

typs *elements*$_M$ werden zum Beispiel mit Hilfe der Relationen “element“, “element_ref“ und “wildcard“ gespeichert. Im Vergleich zum abstrakten Datenmodell werden Elemente den Deklarationen (*declarations*$_S$) zugeordnet und in der Element-Informationseinheit als <element> dargestellt.

3.3 Organisationsformen von XML-Schema

Bevor mit der Definition einer Evolutionssprache für XML-Schema begonnen werden kann, muss auf die unterschiedlichen Organisationsformen von XML-Schema eingegangen werden. In [Mal02] wird zwischen vier Modellierungsstilen unterschieden, bezeichnet als *Russian Doll*, *Salami Slice*, *Venetian Blind* und *Garden of Eden* Style. Der Style hat direkten Einfluss auf die Wiederverwendbarkeit und Anpassungsfähigkeit der Komponenten im XML-Schema, indem Element- und Attributdeklarationen sowie Typdefinitionen entweder lokal und/oder global spezifiziert werden (siehe Tabelle 2). Global bedeutet in diesem Zusammenhang, dass Deklarationen und Definitionen immer direkt unter (d.h. als Kindsknoten) der Element-Informationseinheit <schema> spezifiziert und dann referenziert werden. Alle Knoten, die nicht direkt unter <schema> aufgeführt werden, besitzen im XML-Schema einen lokalen Gültigkeitsbereich. Der *Garden of Eden* Style beinhaltet

	Gültigkeits-bereich	Russian Doll	Salami Slice	Venetian Blind	Garden of Eden
Element- und Attributdeklaration	lokal	x		x	
	global		x		x
Typdefinition	lokal	x	x		
	global			x	x

Tabelle 2: Organisationsformen von XSD im Überblick

zum Beispiel nur globale Element- und Attributdeklarationen sowie globale Typdefinitionen, sodass dadurch die Wiederverwendbarkeit und Anpassungsfähigkeit aller Komponenten erreicht werden. Wird ein XML-Schema angepasst, d.h. kommt es zur Evolution, dann sind Elemente und Typen einfacher zu identifizieren und lokalisieren, was für die Verwendung dieser Organisationsform spricht. Die Identifikation wird dadurch erleichtert, dass jeder Deklaration und Definition ein entsprechender qualifizierter Namen zugeordnet wird, im XML-Schema ist dies der Datentyp *QNAME* (qualified name). Ein QNAME ist eine Zeichenkette, die den Zielnamensraum (*targetNamespace*) als Präfix und dann durch Doppelpunkt getrennt den Namen der Deklaration bzw. Definition besitzt. Der Name der Deklaration bzw. Definition ist eine Zeichenkette des Datentyps *NCNAME* (non-colonized name), eine Zeichenkette ohne Doppelpunkt. Die Lokalisierung wird durch den globalen Gültigkeitsbereich erleichtert, es gibt keine anonymen oder lokalen, und somit nicht sichtbaren Element- oder Attributdeklarationen oder auch Typdefinitionen. Eine Transformation zwischen den unterschiedlichen Organisationsformen ist realisierbar, dennoch wird als Ausgangspunkt für die Definition einer Evolutionssprache der allgemeinste und für die *XML-Schemaevolution* geeignetste Style *Garden of Eden* gewählt.

4 Definition der Evolutionssprache ELaX

Die Evolutionssprache ELaX (Evolution Language for XML-Schema) ist aus der Notwendigkeit entstanden, Anpassungen von XML-Schemata vornehmen und formal darstellen zu können. Änderungen sollten dabei auf einfache, leicht verständliche und eindeutige Art und Weise beschrieben werden können. Die folgenden Kriterien beeinflussten die Entwicklung von ELaX maßgeblich:

1. Beachtung des zu Grunde liegenden Datenmodells (Abstraktes Datenmodell und Element-Informationseinheit von XSD) und des konzeptionellen Modells (EMX)
2. Adäquate und vollständige Realisierung der Operationen ADD, DELETE, UPDATE
3. Definition einer deskriptiven, lesbaren Schnittstelle zur Erzeugung, Änderung und Entfernung von XML-Schema
4. Intuitive Syntax zur Formulierung der Operationsschritte

Das abstrakte Datenmodell, die Element-Informationseinheit sowie das konzeptionelle Modell wurden in Abschnitt 3.1 vorgestellt und die Korrespondenzen in Abschnitt 3.2 definiert. Es existieren demnach unterschiedliche Knotentypen, welche im XML-Schema angepasst werden müssen: Elemente und Attribute (Deklarationen), Inhaltsmodelle (Modellgruppen), Datentypen (einfach und komplexe Typdefinitionen), Module (externe Schemata), Annotationen, Integritätsbedingungen und das Schema an sich. Auf diesen Bestandteilen sollen die Operationen ADD, DELETE und UPDATE ausgeführt werden, sodass sich folgende an die EBNF (Erweiterte Backus-Naur-Form) angelehnte Definition ergibt:

$$elax ::= (< add > | < delete > | < update >)+ ; \tag{1}$$

$$\begin{aligned} add ::= &"add" \ (< addannotation > | < addattributegroup > \\ &| < addgroup > | < addst > | < addct > | < addelement > \\ &| < addmodule > | < addconstraint >) ; \end{aligned} \tag{2}$$

$$\begin{aligned} delete ::= &"delete" \ (< delannotation > | < delattributegroup > \\ &| < delgroup > | < delst > | < delct > | < delelement > \\ &| < delmodule > | < delconstraint >) ; \end{aligned} \tag{3}$$

$$\begin{aligned} update ::= &"update" \ (< updannotation > | < updattributegroup > \\ &| < updgroup > | < updst > | < updct > | < updelement > \\ &| < updmodule > | < updconstraint > | < updschema >) ; \end{aligned} \tag{4}$$

Eine ELaX-Anweisung beginnt mit “add“, “delete“ oder “update“, gefolgt von einer der alternativen Komponenten zum Anpassen der unterschiedlichen Knotentypen. Jede Komponente der Regel (1) kann optional wiederholt werden, sodass die Kapselung bzw. geordnete Hintereinanderausführung ermöglicht wird. Ausgehend von den Regeln der Formeln (1), (2), (3) und (4) werden die unterschiedlichen Komponenten der Knotentypen näher charakterisiert. Die zur Anpassung von Elementen notwendigen Sprachbestandteile sollen nachfolgend erläutert werden.

4.1 Hinzufügen von Elementen

Elemente sind laut *Garden of Eden* Style entweder Elementdeklarationen im globalen Gültigkeitsbereich eines XML-Schemas oder Referenzen auf eben solche Deklarationen. Des Weiteren können Wildcards definiert werden, die “beispielsweise die Validierung von Attributen und Element-Informationseinheiten, je nach ihrem Namensraum-Namen, jedoch unabhängig von ihrem lokalen Namen“ bieten und einen Teil der hohen Erweiterbarkeit von XML begründen [xml12].

$$\begin{aligned} addelement ::= &< addelementdef > | < addelementref > \\ &| < addelementwildard > ; \end{aligned} \tag{5}$$

$$
\begin{aligned}
addelementdef ::= &\text{ "}element\text{" "}name\text{" ncname "}type\text{" qname} \\
&(\text{("}default\text{"|"}fixed\text{") string)?} \\
&(\text{"}final\text{"("}\#all\text{"|"}restriction\text{"|"}extension\text{"))?} \\
&(\text{"}nillable\text{"("}true\text{"|"}false\text{"))? ("}id\text{" id)? ;}
\end{aligned} \tag{6}
$$

$$
\begin{aligned}
addelementref ::= &\text{ "}element\text{" "}ref\text{" qname ("}minoccurs\text{" int)?} \\
&(\text{"}maxoccurs\text{" int)? ("}id\text{" id)? } < position > ;
\end{aligned} \tag{7}
$$

$$
\begin{aligned}
addelementwildard ::= &\text{ "}any\text{" ("}namespace\text{"("}\#\#any\text{"|"}\#\#other\text{"|} \\
&(\text{"}\#\#local\text{"|"}\#\#targetnamespace\text{")+))?} \\
&(\text{"}processcontent\text{"("}lax\text{"|"}skip\text{"|"}strict\text{"))?} \\
&(\text{"}minoccurs\text{" int)? ("}maxoccurs\text{" int)?} \\
&(\text{"}id\text{" id)? "}in\text{"} < locator > ;
\end{aligned} \tag{8}
$$

Für die Lokalisierung und Identifizierung von Elementen, sowie Knoten allgemein, werden weitere Komponenten benötigt. Dies ist einerseits die Positionsbestimmung in Inhaltsmodellen unter Beachtung des Knotenumfelds (9), andererseits die Identifizierung über die absolute Adressierung mittels einer Teilmenge von XPath (11).

$$
\begin{aligned}
position ::= &(\text{"}after\text{"|"}before\text{"| ("}as\text{"("}first\text{"|"}last\text{") "}into\text{") |"}in\text{")} \\
&< locator > ;
\end{aligned} \tag{9}
$$

$$
locator ::= < xpathexpr > \text{| emxid ;} \tag{10}
$$

$$
\begin{aligned}
xpathexpr ::= &(\text{"/" ("." | ("}node()\text{" | ("}node()[@name =' \text{" ncname "']"))} \\
&(\text{"[" int "]")?))+ ;}
\end{aligned} \tag{11}
$$

Eine Elementreferenz-Anweisung beginnt laut (7) mit “element ref“, gefolgt vom Namen der referenzierten Elementdeklaration (*qname*) und weiteren, optionalen Angaben über die Häufigkeit des Auftretens (“minoccurs“, “maxoccurs“) bzw. die Zuweisung einer XML-Schema ID (“id“). Die Position einer Elementreferenz kann wie in Formel (9) dargestellt, vor (“before“), nach (“after“), als erstes (“as first into“), als letztes (“as last into“) oder in (“in“) ein Inhaltsmodell unter Beachtung der Nachbarknoten erfolgen. Die Identifizierung von Knoten findet unter Verwendung von den eindeutigen Identifikatoren des konzeptionellen Modells statt (*emxid*), alternativ kann ein absoluter Pfad unter Verwendung einer Teilmenge von XPath (<xpathexpr>) angegeben werden. Von XPath werden die Navigationsschritte Kind (“child::node()“ bzw. “/“) und eigener Knoten (“self::node()“ bzw. “.“), sowie die allgemeine Navigation ohne Prädikat (“node()“), mit Angabe eines spezifizierten Namens innerhalb eines Prädikats (“node()[...]“) bzw. die genaue Angabe einer Position ("[int]") unterstützt. Die Angabe einer Position muss unter Kenntnis des XML-Schemas immer dann erfolgen, wenn ein XPath-Ausdruck statt eines Knotens eine Menge von Knoten liefert und somit nicht eindeutig ist. Die gewählte Teilmenge von XPath ist ausreichend, um auf einfache Art und Weise Knoten im XML-Schema zu identifizieren bzw. zu lokalisieren. Dies ist durch die Verwendung des *Garden of Eden* Styles möglich, da alle Deklarationen und Definitionen global gültig sind.

Elementdeklarationen (6) benötigen keine spezifische Lokalisierung, da diese generell unter der Element-Informationseinheit <schema> deklariert werden, wobei die Reihenfol-

ge in XML-Schema keinen Einfluss hat. Elementwildcards (8) werden nach der XML-Schema Spezifikation am Ende eines Inhaltsmodells hinzugefügt, ein gemäß Formel (11) definierter XPath-Ausdruck genügt demnach zur Identifizierung des entsprechenden Vaterknotens.

4.2 Entfernen von Elementen

Elementdeklarationen, Referenzen und Wildcards können durch die Regel der Formel (2) hinzugefügt werden. Der nächste zu realisierende Schritt ist das Entfernen dieser Knoten (3). Der entscheidende Unterschied im Vergleich zur Definition der Regeln zum Hinzufügen von Knoten ist, dass nur Informationen zur Identifikation benötigt werden. Dies sind auf der einen Seite der qualifizierte Namen, aber auch im Fall von Elementreferenzen und Wildcards die Postion innerhalb eines XML-Schemas, sodass die folgenden Regeln definiert wurden:

$$\begin{aligned} delelement ::= &< delelementdef > | < delelementref > \\ &| < delelementwildcard > ; \end{aligned} \tag{12}$$

$$delelementdef ::= "element"\ "name"\ \text{qname}\ ; \tag{13}$$

$$\begin{aligned} delelementref ::= &"element"\ "ref"\ \text{qname} \\ &("at" < locator > | < refposition >) ; \end{aligned} \tag{14}$$

$$delelementwildcard ::= "any"\ "at" < locator > ; \tag{15}$$

$$\begin{aligned} refposition ::= &(("first"|"last"|"all"| ("at"\ "position"\ \text{int}\ )) \\ &"in" < xpathexpr >) | \text{emxid}\ ; \end{aligned} \tag{16}$$

Eine Elementreferenz-Anweisung beginnt mit “element ref "(14). Anschließend wird der qualifizierte Namen (*qname*) der Referenz angegeben, bevor die Anweisung mit einer Positionsangabe schließt. Es kann entweder die in Abschnitt 4.1 vorgestellte Lokalisierung mittels <locator> erfolgen, oder, beim mehrfachen Vorhandensein der gleichen Referenz innerhalb eines Inhaltsmodells, die Lokalisierung unter zur Hilfenahme der Formel (16) stattfinden. Die Lokalisierung mittels (16) ermöglicht die Adressierung der ersten (“first“), der letzten (“last“), aller (“all“) oder die an einer bestimmten Position befindliche Elementreferenz (“at position“). Ist der eindeutige Identifikator (*emxid*) bekannt, kann dieser abkürzend ebenfalls verwendet werden.

4.3 Update von Elementen

Die Änderung von vorhandenen Knoten wird durch die Regel (4) realisiert. Grundlegend können alle vorher durch (2) hinzugefügten Elementdeklarationen nachträglich geändert werden. Es werden dazu die entsprechenden qualifizierten Namen, die zu ändernden Werte der entsprechenden Knoten, sowie Angaben zur Positionierung benötigt. Die nachfolgen-

den Regeln wurden bezüglich der Elementdeklarationen definiert:

$$\begin{aligned} updelement ::= & < updelementdef > | < updelementref > | \\ & < updelementwildcard > ; \end{aligned} \tag{17}$$

$$\begin{aligned} updelementdef ::= & \text{"}element\text{" "}name\text{" qname "}change\text{"} \\ & (\text{"}name\text{" ncname })? (\text{"}type\text{" qname })? \\ & ((\text{"}default\text{"}|\text{"}fixed\text{"}) \text{ string })? \\ & (\text{"}final\text{"}(\text{"}\#all\text{"}|\text{"}restriction\text{"}|\text{"}extension\text{"}))? \\ & (\text{"}nillable\text{"}(\text{"}true\text{"}|\text{"}false\text{"}))? (\text{"}id\text{" id })? ; \end{aligned} \tag{18}$$

$$\begin{aligned} updelementref ::= & \text{"}element\text{" "}ref\text{" qname} \\ & ((\text{"}at\text{"} < locator >) | < refposition >) \\ & (\text{"}change\text{" (}\text{"}ref\text{" qname })? \\ & (\text{"}minoccurs\text{" int })? (\text{"}maxoccurs\text{" int })? \\ & (\text{"}id\text{" id })?)? (\text{"}move\text{" "}to\text{" } < position >)? ; \end{aligned} \tag{19}$$

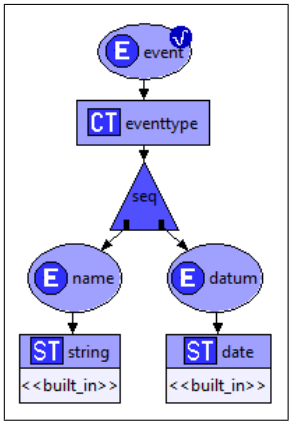
$$\begin{aligned} updelementwildcard ::= & \text{"}any\text{" "}at\text{" } < locator > \text{"}change\text{"} \\ & (\text{"}namespace\text{" (}\text{"}\#\#any\text{"}|\text{"}\#\#other\text{"}| \\ & (\text{"}\#\#local\text{"}|\text{"}\#\#targetnamespace\text{"})+))? \\ & (\text{"}processcontent\text{" (}\text{"}lax\text{"}|\text{"}skip\text{"}|\text{"}strict\text{"}))? \\ & (\text{"}minoccurs\text{" int })? (\text{"}maxoccurs\text{" int })? \\ & (\text{"}id\text{" id })? ; \end{aligned} \tag{20}$$

Elementreferenzen werden mit der Regel (19) angepasst. Beginnend mit “element ref“, dem qualifizierten Namen (*qname*), sowie der Positionierungsinformation kann eine Referenz geändert werden. Die entsprechende Änderungsoperation (19) wird danach durch “change“, gefolgt von dem entsprechenden Bezeichner eines Wertes und dessen zu ändernden Wert ergänzt, bevor die Möglichkeit zum Verschieben (“move to“) gegeben wird. Das Verschieben entspricht dem kompletten Entfernen und Hinzufügen einer Elementreferenz, kann aber durch die verkürzte Regel leichter vorgenommen und formal dargestellt werden. Die Positionsbestimmung wurde in den Abschnitten 4.1 (<locator> in Formel (10)) und 4.2 (<refposition> in Formel (16)) bereits beschrieben.

5 Beispielanwendung

Die in Abschnitt 4 vorgestellten Operationen ADD, DELETE und UPDATE sollen im nachfolgenden Abschnitt an einem umfassenden Beispiel erläutert werden.[1] Dabei werden die Regeln zum Hinzufügen und Verändern von Knoten des XML-Schemas angewendet. Ausgangspunkt ist das in Abbildung 2a dargestellte konzeptionelle Modell einer Veranstaltungsanwendung.

[1] Die Regeln nicht vorgestellter Knotentypen sind unter `www.ls-dbis.de/elax` aufgelistet.



(a) Konzeptionelles Modell

```
<?xml version = "1.0" encoding = "UTF − 8"? >
< xs : schema xmlns : xs = "http : //www.w3.org/2001/XMLSchema"
  xmlns : event = "event.xsd" targetNamespace = "event.xsd" >
   < xs : element name = "event" type = "event : eventtype"/ >
   < xs : element name = "name" type = "xs : string"/ >
   < xs : element name = "datum" type = "xs : date"/ >
   < xs : complexType name = "eventtype" >
     < xs : sequence >
       < xs : element ref = "event : name"/ >
       < xs : element ref = "event : datum"/ >
     < /xs : sequence >
   < /xs : complexType >
< /xs : schema >
```

(b) XML-Schema des konzeptionellen Modells in 2a

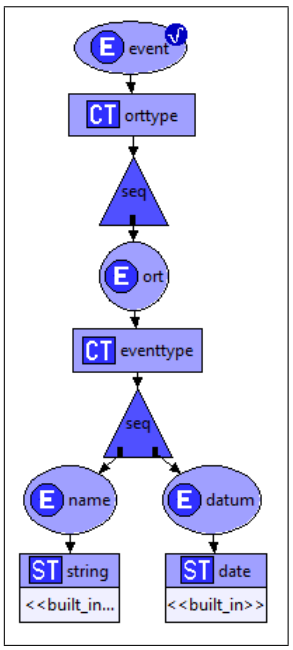
Abbildung 2: Beispielanwendung vor Anwendung von ELaX-Operationen

Eine Veranstaltung (“event“) besitzt einen komplexen Typen (“eventtype“), welcher als Inhaltsmodell eine Sequenz (“seq“) beinhaltet. Die Sequenz des Veranstaltungstyps ist die Folge zweier Elemente, einerseits der Name der Veranstaltung (“name“ vom einfachem Typ “string“), gefolgt vom einen Datum (“datum“ vom einfachem Typ “date“). Das zum konzeptionellen Modell zugeordnete XML-Schema ist in Abbildung 2b dargestellt. Aufgrund des *Garden of Eden* Styles existieren drei Elementdeklarationen (*“event“*, *“name“* und *“datum“*) und eine komplexe Typdefinition (*“eventtype“*), die zwei Elementreferenzen in deren Inhaltsmodell (*sequence*) besitzt. Die qualifizierten Namen dieser Elementreferenzen hängen vom definierten Zielnamensraum (*targetNamespace*) ab, dieser ist im dargestellten XML-Schema mit dem Präfix *event* adressierbar. Somit ist der qualifizierte Name der Elementdeklaration *“name“ “event:name“*. Der Zielnamensraum, Informationen zur Kodierung und Version, sowie der Namensraum der XML-Schemakomponenten (*xs*) sind im konzeptionellen Modell implizit definiert und nicht in Abbildung 2a aufgeführt. Die entsprechenden Werte werden bei der initialen Erzeugung von einem EMX automatisch gesetzt und sind im Nachhinein modifizierbar. Die angeführten Attribute können im XML-Schema durch die Regel *updschema* [2] verändert werden.

Die Modelle aus Abbildung 2 sollen nun verändert werden. Das Ziel der Anpassung ist es, eine neue Hierarchiestufe einzuführen, sodass Veranstaltungen pro Ort aufgeführt werden können. Zur Realisierung dieses Vorhabens müssen drei Schritte getätigt werden:

[2] Syntax der Komponente <updschema> der Formel (4):

$updschema$ ::= (”$targetnamespace$” anyuri)? ((”$targetnamespaceprefix$”|”$language$”| ”$version$”) string) ∗ (”$elementform$” (”$qualified$”|”$unqualified$”))? (”$attributeform$” (”$qualified$”|”$unqualified$”))? (”$finaldefault$” (”$\#all$”| (”$extension$”|”$list$”| ”$restriction$”|”$union$”)+))? (”id” id)? ;



(a) Konzeptionelles Modell

```
<?xml version = "1.0" encoding = " UTF − 8"? >
< xs : schema xmlns : xs = "http : //www.w3.org/2001/XMLSchema"
  xmlns : event = "event.xsd" targetNamespace = "event.xsd" >
   < xs : element name = "event" type="event:orttype"/ >
   < xs : element name = "name" type = "xs : string"/ >
   < xs : element name = "datum" type = "xs : date"/ >
   < xs : complexType name = "eventtype" >
     < xs : sequence >
       < xs : element ref = "event : name"/ >
       < xs : element ref = "event : datum"/ >
     < /xs : sequence >
   < /xs : complexType >
  <xs:element name="ort" type="event:eventtype"/>
  <xs:complexType name="orttype" mixed="false">
    <xs:sequence>
      <xs:element ref="event:ort"/>
    </xs:sequence>
  </xs:complexType>
< /xs : schema >
```

(b) XML-Schema des konzeptionellen Modells in 3a

Abbildung 3: Beispielanwendung nach Anwendung von ELaX-Operationen

1. Einführung einer neuen Elementdeklaration “ort“ mit Typ “event:eventtype“
2. Definition eines neuen komplexen Typs “orttype“ mit Elementreferenz “event:ort“
3. Änderungen des Typs der alten Elementdeklaration in “event:orttype“

Das Ergebnis dieser Anpassung ist in Abbildung 3 dargestellt. Im konzeptionellen Modell 3a wurde das Element “ort'", der komplexe Typ “orttype“, sowie das Inhaltsmodell “seq“ hinzugefügt, bevor die entsprechenden gerichteten Kanten des EMX angepasst bzw. neu definiert wurden. Somit wurden die obigen Schritte realisiert, die Anpassung des konzeptionellen Modells an die neuen Anforderungen ist somit abgeschlossen.

Das XML-Schema aus Abbildung 2b kann nun ebenfalls unter Verwendung der Evolutionssprache ELaX angepasst werden. Das Ergebnis der Anwendung der ELaX Operationen ist in Abbildung 3b dargestellt, die entsprechenden Änderungen sind hervorgehoben. Die Realisierung der oben definierten Schritte zur Einführung der neuen Hierarchiestufe werden nachfolgend erläutert.

1. Schritt: Es muss ein neues Element deklariert werden, welches den Namen “ort“ erhalten soll. Diese Elementdeklaration bekommt den Typ “event:eventtype“. Es ergibt sich somit die folgende ELaX Operation:

add element name **ort** *type* **event:eventtype**
Reihenfolge der Regeln: (1), (2), (5), (6) (21)

Die Reihenfolge der angewendeten Regeln ist unter der jeweiligen Operation aufgelistet und dient der besseren Nachvollziehbarkeit. Die hervorgehobenen Bestandteile sind einerseits die eingesetzten Werte der jeweiligen Datentypen, unter anderem die qualifizierten oder nicht-qualifizierten Namen, andererseits die entsprechenden XPath-Ausdrücke zur Identifikation bzw. Lokalisierung der Knoten im XML-Schema.

2. Schritt: Nachdem eine neue Elementdeklaration eingefügt wurde, muss ein neuer komplexer Typ definiert werden. Dieser Typ erhält den Namen “orttype“ und eine Elementreferenz auf “event:ort“. Zusätzlich soll der Inhaltstyp keine Textknoten als direkte Kindsknoten enthalten (mixed=“false“), dies ist der Standardwert von komplexen Typen gemäß deren Element-Informationseinheit. Die nachfolgend genutzten, allerdings in diesem Artikel nicht detaillierter vorgestellten ELaX Regeln sind *addct* [3] , zum Hinzufügen von komplexen Typen, und *addgroup* [4] , zum Hinzufügen von Inhaltsmodellen. Zur Realisierung des zweiten Schritts sind folgende Operationen notwendig:

add complextype name **orttype** *mixed false*
Reihenfolge der Regeln: (1), (2), addct[3] (22)

add group mode sequence in **/node()/node()[@name='orttype']**
Reihenfolge der Regeln: (1), (2), addgroup[4], (10), (11), (11) (23)

add element ref **event:ort** *in* **/node()/node()[@name='orttype']/node()**
Reihenfolge der Regeln: (1), (2), (5), (7), (9), (10), (11), (11), (11) (24)

Die Reihenfolge der Ausführung ist durch die Element-Informationseinheit des XML-Schemas vorgegeben und resultiert direkt aus den implizit in den Knotentypen enthaltenden Beziehungen. Somit wird erst der Knoten des komplexen Typs erzeugt, dem dann ein Inhaltsmodell zugewiesen wird, anschließend wird eine Elementreferenz zugefügt. Eine umgekehrte Ausführung ist unter Berücksichtigung der verwendeten Knotentypen nicht möglich, da zur Identifikation eines zu modifizierenden Knotens (d.h. des Vaterknotens) dieser erst vorhanden sein muss. Die verwendeten XPath-Ausdrücke können unter Kenntnis des *Garden of Eden* Styles wie in Tabelle 3 dargestellt interpretiert werden.

[3] Syntax der Komponente <addct> der Formel (2):

$addct$ ::= "$complextype$" "$name$" ncname ("$mixed$"("$true$"|"$false$"))?
("$final$" ("$\#all$"|"$restriction$"|"$extension$"))? ("$mode$" ("$extension\_cc$"|
"$extension\_sc$"|"$restriction\_cc$"|"$restriction\_sc$") "$with$" "$base$" qname)?("id" id)? ;

[4] Syntax der Komponente <addgroup> der Formel (2):

$addgroup$ ::= "$group$" "$mode$" ("$sequence$"|"$choice$"|"all") ("$with$" $< groupdefault >$)?
("$minoccurs$" int)? ("$maxoccurs$" int)? ("id" id)? "in" $< locator >$;
$groupdefault$ ::= "$first$"|"$last$"| int ;

XPath-Ausdruck	Erklärung
/node()	Im *Garden of Eden* Style sind alle Deklarationen und Definitionen aufgrund des globalen Gültigkeitsbereichs direkt unter <schema>, sodass jeder absolute XPath-Ausdruck entsprechend beginnt.
/node()[@name='orttype']	Es soll direkt in den komplexen Typen mit dem entsprechenden Namen eine Elementreferenz eingefügt werden. Der Name ist in diesem Fall eindeutig, somit auch der XPath-Ausdruck (keine Positionsangabe notwendig).
/node()	Elementreferenzen werden in das gegebene Inhaltsmodell des komplexen Typs einfügt (hier "sequence"), dabei existieren keine weiteren Nachbarknoten (keine Positionsangabe notwendig).

Tabelle 3: Lokalisierungsschritte des XPath-Ausdrucks der Operation (24) im Detail

3. Schritt: Der letzte Schritt der Anpassung des XML-Schemas der Veranstaltungsanwendung ist die Änderung des Typs der Elementdeklaration "event". Dieses Element soll den neuen Typen "event:orttype" erhalten. Zur Änderung der Elementdeklaration kann die Regel (18) verwendet werden, folgende Operation wird ausgeführt:

update element name **event:event** *change type* **event:orttype**
Reihenfolge der Regeln: (1), (4), (17), (18) (25)

Durch Anwendung der Operationen (21), (22), (23), (24) und (25) wird das XML-Schema der Abbildung 2b in das Schema in Abbildung 3b überführt. Es ist unter Anwendung der ELaX Operationen möglich, Anpassungen an einem vorhandenen XML-Schema vorzunehmen und formal darzustellen. Dabei wurde ebenso auf das zugrunde liegende Datenmodell geachtet, als auch auf die adäquate Umsetzung der Operationen ADD, DELETE und UPDATE. Die vorgestellte Evolutionssprache ermöglicht darüber hinaus die intuitive Formulierung von Operationsschritten und bietet zeitgleich eine deskriptive und lesbare Schnittstelle. Die in Abschnitt 4 formulierten Kriterien, die die Entwicklung von ELaX maßgeblich beeinflussten, sind erfüllt.

6 Umsetzung

Der CodeX-Editor bietet Funktionalitäten zur Durchführung und Unterstützung der *XML-Schemaevolution*, ein Komponentenmodell ist in Abbildung 4 dargestellt. Das konzeptionelle Modell bzw. dessen grafische Repräsentation kann von einem *Nutzer* verändert werden (siehe auch [Ste06]), ein entsprechendes *Logfile* wird dabei gespeichert. Welche Änderungen bzw. vordefinierten Operationen anwendbar sind, wird durch *Operationsspezifikationen* beschrieben (siehe Kategorisierung in [Gru11]). Das konzeptionelle Modell wird aus einem gegebenen XML-Schema unter Verwendung vom *Modell-Mapping*

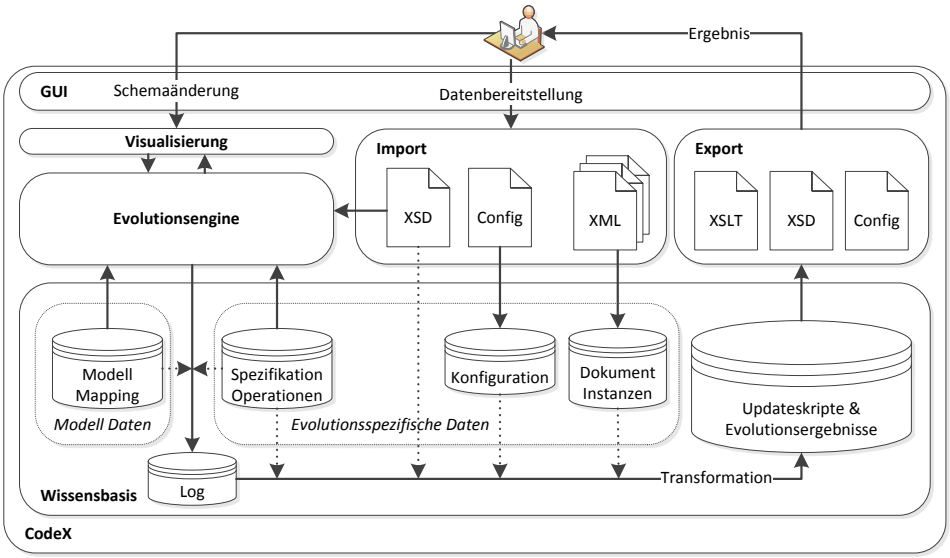


Abbildung 4: Komponentenmodell von CodeX

Informationen (Korrespondenzen) extrahiert oder neu angelegt. Dabei wird eine Umwandlung in den *Garden of Eden* Style vorgenommen, sodass nur globale Deklarationen und Definitionen im XML-Schema auftreten. Die Korrespondenzen zwischen XML-Schema und EMX wurden in Abschnitt 3.2 vorgestellt. Das *Log* kann nach einer entsprechenden Analyse zur Erzeugung von XSLT-Skripten zur Adaption von XML-Dokumenten verwendet werden, welche die aus den Nutzeraktionen hergeleiteten Transformationsschritte umsetzen. Diese *Transformation* nutzt die *evolutionsspezifischen Daten*, u.a. erneut die *Operationsspezifikation*, *XML-Schemainformationen*, gegebene *Konfigurationen* oder auch *Dokumentkollektionen*. Die Datenbereitstellung und Extraktion von Ergebnissen werden entsprechend durch *Import* und *Export*-Komponenten realisiert.

Die Evolutionssprache ELaX ist aktuell noch nicht im Forschungsprototypen integriert, wird aber wie folgt in die Architektur von CodeX eingefügt: die Sprachdefinition mit deren unterschiedlichen Operationen wird in die *Operationsspezifikationen*-Komponente integriert werden, und ist gleichzeitig eine Schnittstelle nach außen für die Anwendung und Formulierung von ELaX-Operationen. Durch die Integration in die *GUI* werden Editieroperationen am konzeptionellen Modell in ELaX-Operationen übersetzt. Diese können auf importierte oder neu erzeugte XML-Schemata angewendet werden.

7 Zusammenfassung

Der vorliegende Artikel definiert die Evolutionssprache ELaX, mit der auf einfache, intuitive Art und Weise Änderungen an einem XML-Schema formal beschrieben und formu-

liert werden können. Diese neue Sprache ist ein Bestandteil der *XML-Schemaevolution*, einem Prozess bei dem XML-Dokumente bei Änderung des zugeordneten XML-Schemas automatisiert angepasst werden.

Die *XML-Schemaevolution* wurde als ein ebenenübergreifender Prozess definiert, bei dem die Nutzeraktionen auf einem konzeptionellen Modell geloggt und ausgewertet werden. Dies ist ein Alleinstellungsmerkmal des vorgestellten Ansatzes, wie der aktuelle Forschungsstand bzw. die Umsetzung in kommerziellen Produkten zeigt. Das konzeptionelle Modell (EMX) kann als oberste Ebene verstanden werden, dieses wurde formal vorgestellt. Ein Nutzer kann auf einem EMX vordefinierten Operationen ausführen, indem die visualisierten Knotentypen des konzeptionellen Modells verändert, hinzugefügt oder gelöscht werden.

Diese Operationen haben direkten Einfluss auf die nächste Ebene, die der XML-Schemata. Jedes XML-Schema hat ein eindeutiges konzeptionelles Modell. Mit Hilfe von Korrespondenzen kann ein XML-Schema in ein EMX und anders herum jedes EMX in ein XML-Schema überführt werden. Eine Voraussetzung ist das Vorliegen des XML-Schemas in der Organisationsform *Garden of Eden*, in dem alle Element- und Attributdeklarationen sowie alle Typdefinitionen einen globalen Gültigkeitsbereich besitzen.

Die unterste Ebene ist die der XML-Dokumente, wobei Mehrdeutigkeiten aufgrund der strukturellen Möglichkeiten von XML-Schema bezüglich der Korrespondenzen zwischen Dokument und Schema auftreten können. Die formale Definition von XML-Dokumenten sowie die Korrespondenzen zwischen XML-Schema und zugeordneten Dokumenten wurden in diesem Artikel nicht detaillierter vorgestellt, hier sei auf [NKH12] verwiesen.

ELaX als Evolutionssprache ist auf Ebene der XML-Schemata einzuordnen, wie in Abbildung 5 dargestellt ist. Neben der Einordnung sind in der Abbildung ebenfalls alle im Ar-

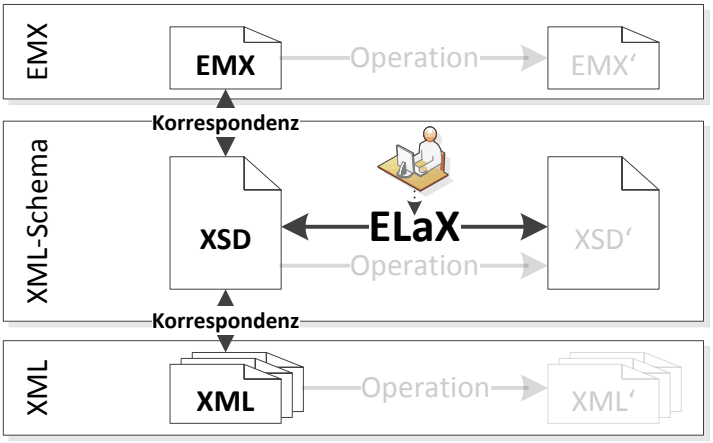


Abbildung 5: Einordnung von ELaX im Ebenenmodell der Abbildung 1

tikel vorgestellten Modelle und Korrespondenzen hervorgehoben, nicht behandelte Konzepte sind im Vergleich zur Abbildung 1 ausgegraut.

Die Unterstützung der *XML-Schemaevolution* wurde in den Forschungsprototypen CodeX integriert. Neben der Evolution ist dieser auch zum einfachen und intuitiven Entwurf von XML-Schemata geeignet. Für die Evolution werden formale Sprachen an allen Schnittstellen benötigt, eine davon ist die in diesem Artikel vorgestellte Sprache zur Schemaevolution ELaX.

8 Ausblick

CodeX ist zur Zeit eine RCP (Rich Client Platform) Anwendung, welche die Basisfunktionalität der *XML-Schemaevolution* anbietet. Um einerseits CodeX einem breitem Publikum zugänglich zu machen und andererseits neue vorgestellte Aspekte präsentieren zu können, wird CodeX aktuell als Webapplikation unter Nutzung von GWT (Google Web Toolkit) umgesetzt.

Im Zuge dieser Erneuerung bzw. Erweiterung soll auch die vorgestellte Evolutionssprache integriert werden, sodass eine Anpassung von XML-Schemata über eine Schnittstelle ermöglicht wird. Dabei soll der Nutzer weitestgehend unterstützt werden, indem fehlerhafte, nicht der Spezifikation entsprechende ELaX-Operationen schon im Vorfeld abgelehnt, ergänzt oder korrigiert werden. Um die Konsistenz der Schemata zu erhalten, sollen unter anderem, aber nicht ausschließlich, XPath-Ausdrücken sofort analysiert, verwendete qualifizierte und nicht-qualifizierte Namen kontrolliert und notwendige Trennwörter der Operationen ergänzt werden (zum Beispiel: “name“, “change“, etc.).

Neben der Integration von ELaX soll ebenfalls die Erzeugung der XSLT-Skripte zur Adaption der XML-Dokumente angepasst werden. Zur Zeit werden in einem mehrstufigen Prozess aus einem internen Format die notwendigen Transformationsschritte erzeugt. Dieses interne Format wird zukünftig durch ELaX ersetzt.

Literatur

[alt12] Altova Produkt diffdog: XML-Schemavergleich. `http://www.altova.com/de/diffdog/xml-schema-diff-tool.html`, 2012. Online; accessed 11-October-2012.

[Cav10] Federico Cavalieri. EXup: an engine for the evolution of XML schemas and associated documents. In *Proceedings of the 2010 EDBT/ICDT Workshops*, EDBT ’10, Seiten 21:1–21:10, New York, NY, USA, 2010. ACM.

[CMDZ10] Carlo Curino, Hyun Jin Moon, Alin Deutsch und Carlo Zaniolo. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *PVLDB*, 4(2):117–128, 2010.

[db212] IBM DB2 LUW V10R1 Online Documentation. `http://publib.boulder.ibm.com/infocenter/db2luw/v10r1/topic/com.ibm.db2.luw.xml.doc/doc/t0020917.html`, 2012. Online; accessed 19-September-2012.

[DLP+11] Eladio Domínguez, Jorge Lloret, Beatriz Pérez, Áurea Rodríguez, Angel Luis Rubio und María Antonia Zapata. Evolution of XML schemas and documents from stereotyped UML class models: A traceable approach. *Information & Software Technology*, 53(1):34–50, 2011.

[GM08] Giovanna Guerrini und Marco Mesiti. X-Evolution: A Comprehensive Approach for XML Schema Evolution. In *DEXA Workshops*, Seiten 251–255, 2008.

[Gru11] Hannes Grunert. XML-Schema Evolution: Kategorisierung und Bewertung. Bachelor Thesis, Universität Rostock, 2011.

[KKLM09] Jakub Klímek, Lukás Kopenec, Pavel Loupal und Jakub Malý. XCase - A Tool for Conceptual XML Data Modeling. In *ADBIS (Workshops)*, Seiten 96–103, 2009.

[Kle07] Meike Klettke. *Modellierung, Bewertung und Evolution von XML-Dokumentkollektionen*. Habilitation, Fakultät für Informatik und Elektrotechnik, Universität Rostock, 2007.

[LRW+97] Meir M. Lehman, Juan F. Ramil, Paul Wernick, Dewayne E. Perry und Wladyslaw M. Turski. Metrics and Laws of Software Evolution - The Nineties View. In *IEEE METRICS*, Seiten 20–, 1997.

[Mal02] Eve Maler. Schema Design Rules for UBL...and Maybe for You. In *XML 2002 Proceedings by deepX*, 2002.

[mic11] Microsoft TechNet: Verwenden von XML in SQL Server. `http://technet.microsoft.com/de-de/library/ms190936(SQL.90).aspx`, 2011. Online; accessed 15-December-2011.

[NKH12] Thomas Nösinger, Meike Klettke und Andreas Heuer. Evolution von XML-Schemata auf konzeptioneller Ebene - Übersicht: Der CodeX-Ansatz zur Lösung des Gültigkeitsproblems. In *Grundlagen von Datenbanken*, Seiten 29–34, 2012.

[ora12] Oracle XML DB Developer's Guide 11g Release 2 (11.2). `http://docs.oracle.com/cd/E11882_01/appdev.112/e23094/xdb07evo.htm`, 2012. Online; accessed 19-September-2012.

[Ste06] Robert Stephan. Entwicklung und Implementierung einer Methode zum konzeptuellen Entwurf von XML-Schemata. Diploma Thesis, Universität Rostock, 2006.

[tam12] Tamino XML-Server online documentation. `http://documentation.softwareag.com/webmethods/tamino/ins8/advconc/FromModeltoSchema.htm#Schemaevolution`, 2012. Online; accessed 19-September-2012.

[xml10] XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition). `http://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/`, December 2010. Online; accessed 01-March-2012.

[xml12] W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. `http://www.w3.org/TR/2012/PR-xmlschema11-1-20120119/`, January 2012. Online; accessed 01-March-2012.