Lehrstuhl für Integrierte Schaltungen
Technische Universität München

Algorithmus und Softwarearchitektur für die binäre Codierung von strukturierten Dokumenten

Ulrich Niedermeier

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr.-Ing. Klaus Diepold

Prüfer der Dissertation:

1. Univ.-Prof. Dr.-Ing. Ingolf Ruge, em.
2. Univ.-Prof. Dr.-Ing. André Kaup,
   Friedrich-Alexander-Universität Erlangen-Nürnberg

Die Dissertation wurde am 26. Juni 2003 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 17. November 2003 angenommen.

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Angestellter am Lehrstuhl für Integrierte Schaltungen der Technischen Universität München.

Mein erster Dank gilt meinem Doktorvater, Herrn Prof. Dr.-Ing. Ingolf Ruge, der mir die Durchführung dieser Arbeit ermöglicht hat. Durch die Kontakte seines Instituts, vor allem auch zur MPEG-Gruppe, habe ich viele Anregungen erhalten.

Herrn Prof. Dr.-Ing. André Kaup danke ich für das Interesse an meiner Arbeit und die Übernahme des Zweitberichts.

Herrn Dr.-Ing. Walter Stechele, dem Leiter meiner Arbeitsgruppe, danke ich für die freundliche fachliche und administrative Unterstützung.

Außerdem gilt mein Dank Herrn Dr.-Ing. Eckart Hundt für die Kooperation mit seiner Abteilung bei Siemens CT, aus der ich wichtige Impulse für meine Arbeit erhalten habe.

Der Siemens AG danke ich für die finanzielle Unterstützung meiner Arbeit im Rahmen des Ernst von Siemens Stipendiums.

Besonders danke ich meinen Kollegen Jörg Heuer und Dr.-Ing. Andreas Hutter für die interessanten fachlichen Diskussionen und Anregungen, die wesentlich zum Gelingen dieser Arbeit beigetragen haben.

Meinen aktuellen und ehemaligen Kollegen Michael Eiermann, Nuria Pazos Escudero, Jürgen Foag, Stephan Herrmann, Kilian Jacob, Torsten Mahnke, Hubert Mooshofer, Thomas Wild, Armin Windschiegl und Paul Zuber danke ich für die fachliche Hilfe und das angenehme Arbeitsklima am Lehrstuhl für Integrierte Schaltungen.

Verena Draga, Alwine Jakobs, Gabriele Spöhrle und Doris Zeller danke ich für die schnelle Bearbeitung aller administrativen Angelegenheiten. Wolfgang Kohtz und die Admingruppe sorgte stets für ein stabil laufendes Rechnernetzwerk.

München im Juni 2003

Inhaltsangabe

Abbildungsverzeichnis

Abkürzungen

API	Application Programming Interface
XSD	XML Schema Definition
W3C	World Wide Web Consortium
$\lceil X \rceil$	Aufrundung von *X* zur nächsten ganzen Zahl
AU	Access Unit
AV Material	Audio Video Material
BiM	Binary format for multimedia description streams
BVC	Baumverzweigungscode
D	Deskriptor
DDL	Description Definition Language
DOM	Document Object Model
DS	Description Schemes
DTD	Dokument Type Definition
DVB	Digital Video Broadcasting
EPG	Electronic Program Guide
FUU	Fragment Update Unit
HTML	Hypertext Markup Language
MPC	Multiple Element Position Code
MSB	Most Signifikant Bit
SAX	Simple API for XML
SBC	Schema Branch Code
SGML	Standard Generalized Markup Language
SPC	Single Element Position Code
TBC	Tree Branch Code
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VLC	Variable Length Code
vluimsbf	Variable length unsigned integer, most significant bit first
XML	Extensible Markup Language

Kapitel 1 - Einleitung

1.1 Entwicklungen im Umgang mit Information

Der Zugang zu und Umgang mit Information ist für den Menschen von großer Bedeutung - er ist ein wesentlicher Faktor für den beruflichen Erfolg, die Lösung von Aufgaben und Problemen im privaten Bereich und nicht zuletzt eine Bereicherung des Lebens. Über die Zeit sind immer ausgefeiltere Methoden für den Umgang mit Information entwickelt worden.

Nicht nur die Methoden der Informationsverarbeitung haben sich entwickelt, auch die Datenmenge, die in den verschiedensten Bereichen von den unterschiedlichsten Quellen neu hervorgebracht wird, und für das Individuum mit seinen Interessen und Problemen relevant ist, hat dramatisch zugenommen. Es gibt kaum ein Themengebiet, sei es Gesundheit /Medizin, Finanzwesen oder kulturelle Ereignisse, zu denen nicht qualitativ hochwertige Information dem Privatmann oder professionellen Anwender zur Verfügung steht. - So gibt es kaum eine Rundfunksendung oder ein TV-Magazin, das nicht auf einer eigenen Internetseite zusätzliche Information oder Referenzen zu den behandelten Themen anbietet. Des weiteren ist die Umstellung auf digitale Verbreitung des Fernsehens im Gange, und interaktive Dienste werden entwickelt, die beispielsweise auf dem „Multimedia Home Platform“ Standard aufbauen [MHP].

Vorangetrieben wird dieser Trend durch die rasante technologische Entwicklung, welche preiswerte und leistungsfähige Endgeräte ermöglicht hat, sowie durch den Ausbau der informationstechnologischen Infrastruktur. Unter den vielen Entwicklungen seien einige hervorgehoben:

Die Speichermedien haben eine so große Kapazität und ein so niedriges Preisniveau erreicht, dass dies den Umgang v.a. mit multimedialen Daten erst sinnvoll macht. Die Kapazität moderner Festplatten reicht aus, um mehrere Stunden digitales Video in hoher Qualität zu speichern (80GB fassen bei 6Mbit/s knapp 30 Stunden). Dies liefert die Grundlage für die Schaltzentralen der digitalen Infrastruktur wie Server und Datenbanken, sowie für leistungsfähige Endgeräte wie PC oder digitale Videorecorder mit fortschrittlichen Funktionen. Mobile Speichermedien (z.B. Compactflash) oder mobile DVD Brenner erlauben es überall aufzuzeichnen. Digitale Kameras finden immer mehr Verbreitung für private und professionelle Anwendungen. Drahtgebundene (FireWire, USB) oder drahtlose Schnittstellen (Bluetooth) ermöglichen den Datenaustausch zwischen den unterschiedlichsten Geräten. Für den Zugang zu Datennetzen stehen dem Endanwender DSL Anschlüsse zur Verfügung. Auch die Leistungsfähigkeit mobiler Geräte steigt rasant. Sowohl bei der Datenübertragung (Mobilfunknetze der dritten Generation erlauben mit einer Datenrate von 64 Kbit/s bis 2Mbit/s die Übertragung von Video) als auch bei der Ausstattung mit Rechenleistung und Speicher sowie bei der Anzeige (z.B. Flachdisplays mit zunehmend höherer Auflösung) sind erhebliche Fortschritte erkennbar.

Während also die Erzeugung und Verbreitung von (multimedialer) Information durch die technologische Entwicklung beschleunigt wird ergibt sich das Problem, dass es zunehmend schwieriger wird, die relevante und interessante Information aus dem riesigen Angebot herauszufiltern. Die manuelle Auswahl ist aufgrund des gewaltigen Angebots oft nicht praktikabel, die Unterstützung durch automatische Systeme ebenfalls häufig unbefriedigend. Die Probleme sind mannigfaltig: zum einen gibt es keine einheitliche Schnittstelle für die Suche nach Information. Zum anderen fehlt es auch an einem Format für

die Darstellung der Information, das es automatischen Suchhilfen erlaubt, den Inhalt eines Dokuments richtig zu bewerten. Die bisher von Suchmaschinen praktizierte Stichwortsuche leidet unter dem Problem, dass der Kontext in dem ein Schlüsselwort auftritt für das automatische System nicht bestimmbar ist. Es lässt sich im wesentlichen nur nach Kombinationen von Schlüsselwörtern suchen, oder bestimmte Schlüsselwörter ausschließen etc. Zum dritten liegt Information in zunehmendem Masse multimedial statt textbasiert vor. Hier versagen konventionelle Techniken völlig, da die Bedeutung von Audio- oder Videodaten sich nur dem menschlichen Betrachter erschließt und einem automatischen System verborgen bleibt.

In diesem Kontext laufen Bemühungen das Internet so weiterzuentwickeln, dass die Daten in einer Form repräsentiert werden, die von Maschinen “verstanden” werden kann. Erst wenn automatisierte Werkzeuge ebenso wie Menschen in der Lage sind Information auszutauschen und zu verarbeiten wird das Internet sein volles Potential erreichen.

Tim Berners Lee, der Erfinder des Internet, schreibt dazu in [SCI01]:

“The Semantic Web is an extension of the currend web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.”

Beim “Semantic Web” werden als Inhalte zunächst nur textuelle Informationen betrachtet. Auch diese werden gegenwärtig in einer bezüglich des Inhalts unstrukturierten Form gespeichert, die es einem automatisierten System fast unmöglich macht, die Bedeutung der Daten richtig zu interpretieren. Beim semantischen Internet wird dagegen die Information schon bei der Darstellung so aufbereitet, dass später die Arbeit von Suchmaschinen oder intelligenten Agenten erleichtert wird. Ohne diese Aufbereitung wären letztlich Suchmaschinen oder Softwareagenten mit menschlicher Intelligenz und Hintergrundwissen erforderlich. Ein Teil der Aufgabe verlagert sich so zum Autor, welcher der Information schon bei der Erzeugung eine aus der Bedeutung der Daten abgeleitete Struktur gibt.

Um multimediabasierte Information in den Kontext des semantischen Netzes einzubinden ist ein weiterer technologischer Sprung erforderlich: durch standardisierte Beschreibungen, den „Metadaten“, welche den eigentlichen Video-, Bild- oder Audiodaten beigefügt werden, sollen sich die Inhalte und die Bedeutung dieser Daten auch automatischen Systemen erschließen. Dabei werden verschiedene Abstraktionsstufen betrachtet: von elementaren Eigenschaften der Video- oder Audiodaten wie Farbe, Form oder Spektrum bis hin zu Szenenbeschreibungen oder Inhaltsangaben und vieles mehr.

Strukturierte Dokumente bilden einen wesentlichen Bestandteil für das semantische Internet. Der bekannteste Vertreter dieser Art ist XML. Sowohl das textbasierte semantische Netz [SCI01] als auch die metadatenbasierte Darstellung von Multimediainformation [MPEG02] bauen auf XML auf. Diese strukturierten Dokumente haben die Eigenschaft, dass sie die Datenelemente in einen hierarchischen Kontext einbinden, der Rückschlüsse auf die semantische Bedeutung der Datenelemente zulässt. Allerdings gibt es auch einen Nachteil – die Strukturierung lässt die Information oft auf ein Mehrfaches der eigentlichen Nutzdaten anwachsen, weshalb eine kompaktere Darstellung wünschenswert ist. Des weiteren ergeben sich durch diverse Anwendungen spezielle Anforderungen, die sich durch eine rein textuelle Darstellung der Information nicht erfüllen lassen. In diesem Zusammenhang steht die Entwicklung eines neuen binären Formats für strukturierte Dokumente - das Thema dieser Arbeit.

1.2 Ziele der Arbeit

Den Vorteilen, welche man durch die strukturierte Darstellung von Information erzielt steht auch ein bedeutender Nachteil gegenüber: das erforderliche Datenvolumen ist häufig ein mehrfaches der Nutzinformation. Deshalb wäre eine kompaktere Darstellung, also eine binäre Codierung wünschenswert, wobei jedoch die entscheidende Eigenschaft, nämlich die Strukturierung erhalten bleiben soll. Zudem ergeben sich durch neue Anwendungen neben der Kompression spezifische Anforderungen an eine solche Codierung, welche sich durch das Rohdatenformat in Textform nicht oder nur schwer erfüllen lassen.

Durch diese Arbeit sollen die in 1.1 skizzierten technologischen Entwicklungen durch eine neues binäres Formats für strukturierte Dokumente unterstützt werden. Ausgangspunkt und Motivation ist der MPEG-7 Standard, welcher sich zum Ziel gesetzt hat im oben beschriebenen Sinne die Bedeutung von Multimediadaten automatischen Systemen zugänglich zu machen.

Neben der Leistungsfähigkeit des Codieralgorithmus an sich ist auch die effiziente Implementierung ein wichtiges Kriterium für die Anwendbarkeit eines Verfahrens. Hierfür sollen mit dem Modell einer optimierten Softwarearchitektur Hinweise gegeben werden.

1.3 Aufbau der Arbeit

Das **Kapitel 2** enthält eine kurze Einführung in den MPEG-7 Standard und erläutert wie die in dieser Arbeit entwickelten Verfahren in ein Multimedia-Informationssystem eingebettet sind. Eine Schlüsselkomponente ist ein Datenformat, welches für die Beschreibung multimedialer Inhalte verwendet wird. Dieses Datenformat beruht auf einer Syntaxdefinition und in seiner Grundversion auf einer textuellen Darstellung der Information, welche jedoch diverse Nachteile aufweist. Eine binäre Codierung soll diese umgehen und spezifische Funktionalitäten zur Verfügung stellen, welche die Anforderungen, die sich aus neuen Anwendungen ergeben abdecken.

Kapitel 3 gibt einen Überblick über binäre Formate für textbasierte Information und beleuchtet die Eigenschaften dieser Formate im Hinblick auf die in Kapitel zwei identifizierten Anforderungen.

Kapitel 4 beschreibt ein neu entwickeltes syntaxbasiertes Binärformat für strukturierte Dokumente, welches speziell auf die spezifischen Anforderungen optimiert wurde.

In **Kapitel 5** wird der neue Algorithmus mit den bereits bekannten Verfahren verglichen und die Möglichkeiten weiterer (statistischer) Optimierung diskutiert. Außerdem wird eine informationstheoretische Betrachtung zur Abschätzung der Grenze der Kompression syntaxbasierter Codierverfahren durchgeführt.

In **Kapitel 6** wird für den Algorithmus eine sowohl im Hinblick auf Speicher-anforderungen als auch Rechenleistung optimierte Softwarearchitektur entwickelt. Dies wird durch eine Aufteilung in zwei Schritte, in einen Kompilationsprozess und den eigentlichen Codier-/Decodiervorgang erreicht. Der Kompilationsprozess erzeugt aus der Syntaxdefinition ein Zwischenformat, welches den Codiervorgang vorbereitet. Darüber hinaus ist die Softwarearchitektur so angelegt, dass die verschiedenen algorithmischen Teile des Codierverfahrens durch ein einheitliches Grundsystem implementiert werden.

In **Kapitel 7** wird das im vorhergehenden Kapitel entwickelte Modell verwendet, um eine Komplexitätsabschätzung des neuen Codierverfahrens durchzuführen. Dabei werden

sowohl durch Simulation und Messung gewonnene Daten verwendet, als auch die Komplexität des entscheidenden Arbeitsschrittes theoretisch analysiert. Dieses Kapitel soll Hinweise auf die Implementierbarkeit und mögliche Optimierungen beim Einsatz des Codierverfahrens geben.

Kapitel 8 gibt einen kurzen Ausblick auf die weiteren technologischen Entwicklungsmöglichkeiten des Codierformats. Außerdem wird das Potential der Codierung bei den praktischen Anwendungsfeldern abgeschätzt.

Kapitel 2 – MPEG-7

Dieses Kapitel gibt einen Überblick über den MPEG-7 Standard. Ausgehend von neuen Anforderungen an Informationssysteme werden Anwendungsszenarien des neuen Standards vorgestellt und das Funktionsgrundprinzip erläutert. Einer groben Beschreibung des Aufbaus von MPEG-7 folgt eine detailliertere Darstellung der MPEG-7 Deskriptoren. Im abschließenden Teil des Kapitels wird erläutert, wie sich der eigentliche Gegenstand dieser Arbeit, ein binäres Format für die Codierung strukturierter Dokumente, in den Standard einfügt und welche speziellen Anforderungen sich aus den Anwendungen an dieses Format ableiten.

2.1 Ziele von MPEG-7

In der Einleitung wurden bereits einige neuere Tendenzen im Umgang mit Information erläutert. Das grundlegende Problem ist, dass die Menge an multimedialer Information, welche von den verschiedensten Quellen zu den unterschiedlichsten Zwecken erzeugt wird in vielen Bereichen für eine manuelle Verarbeitung zu umfangreich ist, und eine Unterstützung durch automatische Systeme bisher (bis auf spezialisierte Systeme) nicht möglich war, da die Bedeutung von multimedialem Material einem solchen System verborgen bleibt.

Der MPEG-7 Standard, auch bezeichnet als “Multimedia Content Description Interface” hat sich das Ziel gesetzt für diesen Problemkreis Lösungen anzubieten. Es gibt schon Beispiele für Werkzeuge, welche für spezielle Aufgaben entworfen wurden, und den Menschen beim Umgang mit multimedialer Information unterstützen: die Suche eines gegebenen Fingerabdrucks in einer Bilddatenbank mit Fingerabdrücken beispielsweise kann durch ein automatisches System entscheidend vereinfacht werden: dazu werden spezifische Merkmale eines Fingerabdrucks gewonnen, etwa wo sich Linien gabeln oder enden. Mit Hilfe dieser Merkmale kann der gegebene Fingerabdruck mit denen in der Datenbank verglichen werden. Auch andere Systeme zur Authentifizierung anhand biometrischer Merkmale, z.B. des Gesichts, werden schon in der Praxis verwendet [Zie03].

Solche spezialisierten Systeme sind deshalb sinnvoll einzusetzen, weil nur der Datenbestand von abgegrenzten Datenbanken bearbeitet wird, weshalb kein universelles Format für die Merkmale verwendet werden muss. Bei der Entwicklung von MPEG-7 war jedoch das Ziel ein einheitliches System für die Verarbeitung von multimedialer Information zu schaffen, bei dem die Daten verschiedenster Quellen vergleichbar und damit beispielsweise einer Recherche zugänglich sind. Ein weiteres Ziel ist es nicht nur die verschiedensten Informationsquellen zu berücksichtigen, sondern auch die Anforderungen der unterschiedlichsten Anwendungen abzudecken.

2.1.1 Anwendungen

Die Anwendungsszenarios von MPEG-7 lassen sich im Prinzip in zwei Klassen einteilen. In der ersten Klasse wird nach spezifischer Information in einem Informationspool (z.B. einer Datenbank, dem Internet etc.) gesucht (“pull-application”). In der zweiten Klasse wird in einem kontinuierlichen Medienstrom spezifische Information herausgefiltert („push-application“).

Die Anwendungsgebiete umfassen beispielsweise:

- Den kulturellen Bereich, z.B. Erziehung, Digitale Bibliotheken, Museen oder Kunstgalerien
- Den Bereich der Freizeitgestaltung, z.B. automatische Auswahl von Rundfunk- oder Fernsehsendungen nach einem Benutzerprofil, Tourismus-Information
- Professionelle Anwendungen wie Journalismus (Suche nach Bild-/Tonmaterial über spezielle Personen), Überwachung, Architektur/Design, Fernerkundung oder Medizin (z.B. medizinische Bilddatenbanken mit Krankheitsbildern)
- Den Bereich des Handels wie E-Commerce, Einkaufen (z.B. Suche nach der persönlich bevorzugten Kleidung etc.)

... und viele weitere.

2.2 Aufbau und Funktion von MPEG-7

2.2.1 Das Grundprinzip von MPEG-7

Die Grundlage für die gezielte inhaltsbezogene Verarbeitung multimedialer Information ist es Eigenschaften des Materials zu erfassen, die eine semantische Aussage über das Material beinhalten. Im Gegensatz dazu sind die Formate der Rohdaten dieser Information für andere Ziele ausgelegt: die digitale Darstellung von Videosequenzen, beispielsweise im MPEG-1 oder MPEG-2 Format, ist auf eine möglichst hohe Kompression der Daten optimiert. Ein MPEG Datenstrom besteht im wesentlichen aus den Koeffizienten einer diskreten Cosinus Transformation (DCT) zur Beschreibung der Textur und Bewegungsvektoren für kleine Blöcke zwischen aufeinanderfolgenden Einzelbildern eines Videos, mit deren Hilfe die zeitliche Redundanz in der Videosequenz eliminiert wird. Aus diesen Daten lassen sich jedoch über den Inhalt der Sequenz fast keine Aussagen machen. Lediglich, ob viel oder wenig Bewegung im Bild ist, und ob das Bild eher gleichförmig oder fein strukturiert ist, sowie über vorherrschende Farben kann man einen groben Eindruck gewinnen.

Um semantisch verwertbare Information aus den multimedialen Rohdaten zu gewinnen definiert MPEG-7 eine Reihe von sogenannten Deskriptoren, mit deren Hilfe eine detailliertere Beschreibung der Daten möglich ist. Die einem Medienstrom zugeordneten Deskriptoren werden auch Metadaten genannt, was in diesem Zusammenhang “Daten über Daten” bedeutet.

Bei den Deskriptoren unterscheidet man die “low-level” von den “high-level” Deskriptoren. Die low-level Deskriptoren beschreiben elementare Eigenschaften der Mediadaten, also etwa Farbe und Form von Objekten in einer Videosequenz. Diese Deskriptoren können zum großen Teil automatisch gewonnen werden. Die “high-level” Deskriptoren beschreiben abstraktere Aspekte der Medien, beispielsweise eine Szenenbeschreibung oder Hintergrundinformationen zu den Urhebern eines Mediums und erfordern bei der Erzeugung oft mehr manuelle Arbeit.

Eine MPEG-7 Beschreibung ist keine einfache Aneinanderreihung solcher Deskriptoren, sondern bettet sie in eine hierarchische Struktur ein, welche den Kontext enthält in dem ein Deskriptor für die gesamte Beschreibung steht. Die Deskriptoren erfüllen zwei Funktionen: zum einen werden sie aus den Mediadaten gewonnen und zusammen mit diesen

in einer Datenbank gespeichert oder über eine Datenverbindung übertragen. Zum anderen liefern sie auch das Mittel, mit dem Suchanfragen oder Informationsfilter spezifiziert werden.

Eine „pull-Anwendung“ wird mit Hilfe der Deskriptoren folgendermaßen umgesetzt: die Suche wird mit den MPEG-7 Deskriptoren formuliert, und über eine geeignete Schnittstelle dem Informationspool übergeben. In dessen Datenbestand wird nach passendem Material gesucht, indem die Deskriptoren der Suchanfrage mit denen des gespeicherten Materials verglichen werden (für das Material im Informationspool müssen die selben Deskriptoren extrahiert worden sein). Die besten Treffer werden zurückgegeben.

Bei einer „push-Anwendung“ wird der kontinuierliche Medienstrom mit einem zusätzlichen Beschreibungsstrom aus Deskriptoren hinterlegt. Der Nutzer hat vorher mit Hilfe der Deskriptoren Ereignisse oder Inhalte spezifiziert, an denen er interessiert ist. Es können dann, wenn ein solches Ereignis eintritt die entsprechenden Mediadaten, welche für die Deskriptoren referenziert werden, aufgezeichnet, oder eine andere Aktion ausgelöst werden. Der Nutzer wird von der Aufgabe entlastet das Material vollständig von Hand zu sichten.

Ein konkretes Beispiel für das erste Szenario ist die Suche nach einem Musikstück, von dem man nur die Melodie, nicht aber den Titel, den Interpreten oder Komponisten kennt. Man spezifiziert das gesuchte Musikstück durch Summen der Melodie. Daraus werden die relevanten Audio-Deskriptoren extrahiert, und mit den Deskriptoren der Musikstücke in einer Datenbank verglichen. Unter den besten Treffern sollte sich das gewünschte Musikstück finden lassen. Die Funktionsfähigkeit solcher Systeme ist bereits demonstriert worden.

Ein Beispiel für das zweite Szenario ist ein “Persönlicher Videorekorder”. Der Nutzer spezifiziert die Inhalte an denen er interessiert ist mit Hilfe der Deskriptoren. Der Rekorder scannt die zur Verfügung stehenden Kanäle, und zeichnet Sendungen auf, die dem Nutzerprofil entsprechen.

An diesen Beispielen wird auch die Bedeutung eines Standards für die Deskriptoren deutlich: die Such- bzw. Filterfunktion funktioniert nur, wenn die Deskriptoren, welche die Medien beschreiben auch für die Spezifikation der Suche oder der Informationsfilter verwendet werden können, also einer verbreiteten Norm entsprechen.

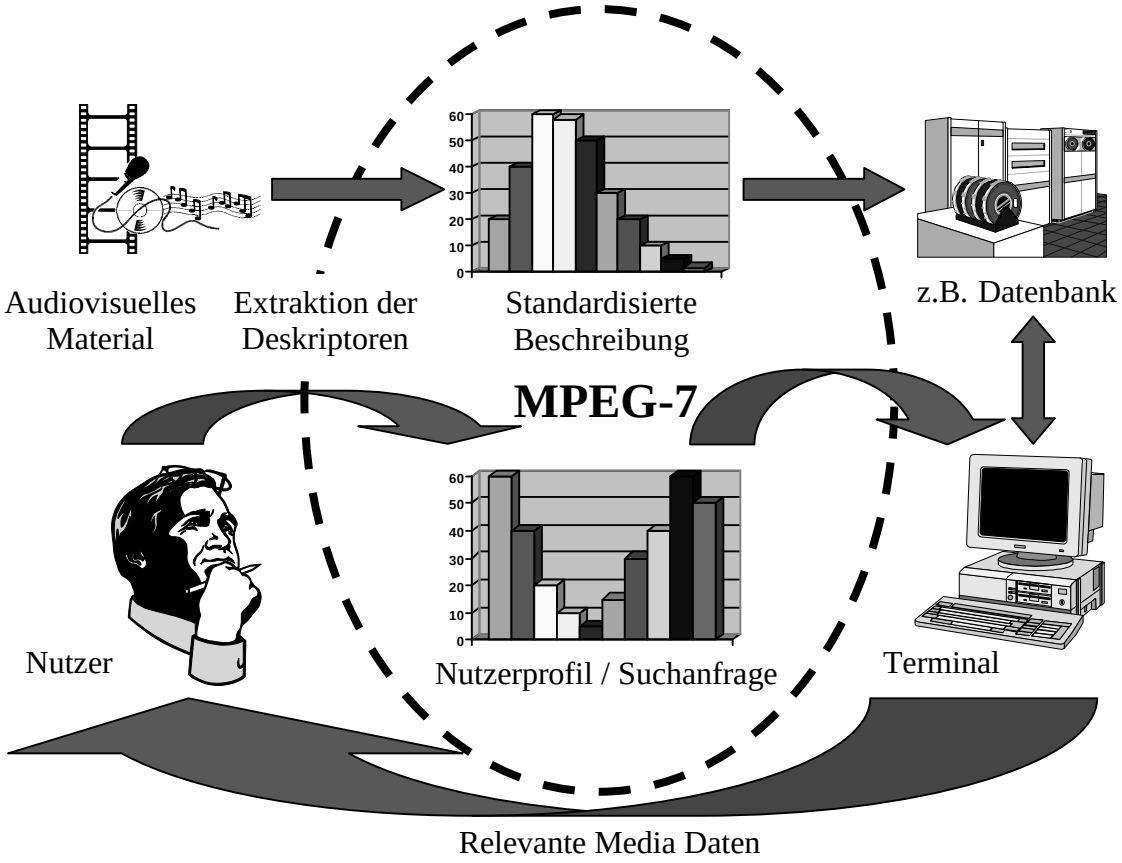


Abb. 2.1: MPEG-7 Anwendungsszenario

Die Beschreibungen erfassen ein breites Spektrum an Eigenschaften des multimedialen Materials. Sie sind grob gegliedert in drei Bereiche, nämlich in Beschreibungsschemata für Video, Audio und Multimedia.

- Videodeskriptoren erfassen elementare Bildeigenschaften wie Farbe, Textur, Form, oder Ortsangaben.
- Audiodeskriptoren beschreiben elementare Eigenschaften des Tonsignals wie Spektrum, Melodie, Klangfarbe etc.
- Die “Multimedia Description Schemes” (MDS) erfassen vorwiegend allgemeinere Eigenschaften des Quellenmaterials, z.B. Information über die Erzeugung des Materials (Autor, Studio, Rechte etc.), Möglichkeiten der Benutzerinteraktion (wie Nutzerprofile oder –gewohnheiten), Orientierung im Material (Inhaltsangaben, Zusammenfassungen) oder die Organisation des Inhalts.

2.2.2 Aufbau von MPEG-7

Der MPEG-7 Standard ist in mehrere Teile untergliedert, die jeweils einen Aspekt der Gesamtfunktionalität abdecken. Die Teile sind:

- MPEG-7 Systems: dieser Teil spezifiziert die Terminalarchitektur und ein binäres sowie ein textuelle Format für den Datenaustausch
- MPEG-7 DDL: spezifiziert eine Sprache, mit der die Syntax von Beschreibungsschemata definiert werden kann
- MPEG-7 Visual: enthält Werkzeuge zur Beschreibung von Bild- oder Videomaterial

- MPEG-7 Audio: enthält Werkzeuge zur Beschreibung von Tondaten
- MPEG-7 Multimedia Description Schemes: enthält Werkzeuge, die allgemeinere Eigenschaften der Beschreibungen abdecken und diese strukturieren
- MPEG-7 Reference Software: eine Software-Implementierung wesentlicher Teile des Standards
- MPEG-7 Conformance Testing: definiert Richtlinien und Vorschriften, wie die Konformität von Implementierungen mit dem Standard getestet werden kann
- MPEG-7 Extraktion and use of descriptions: enthält informatives (also nicht normatives) Material über die Extraktion und Benuzung einiger Deskriptoren

2.3 MPEG-7 Deskriptoren

Ohne zu sehr ins Detail zu gehen werden in diesem Abschnitt einige Deskriptoren der drei Bereiche Audio, Video und MDS vorgestellt, um einen Eindruck zu vermitteln, was mit diesen Deskriptoren beschrieben werden kann. Eine detailliertere und doch allgemein gehaltene Einführung findet sich beispielsweise in [MPEG02].

2.3.1 Aufbau der Beschreibungen

Die Beschreibungen bestehen aus „Descriptors“ (D) und „Description Schemes“ (DS), die wiederum mit der „Description Definition Language“ (DDL) definiert werden. Die Deskriptoren enthalten die Nutzdaten der Beschreibung. Im Falle der elementaren Video- oder Audiodeskriptoren wurde eine Berechnungsvorschrift standardisiert, wie der betreffende Deskriptor aus den Mediadaten zu gewinnen ist. Die Beschreibungsschemata dienen im wesentlichen dazu, die Beschreibungen zu strukturieren, und die einzelnen Deskriptoren in den richtigen Kontext zu stellen (die Eigenschaft welches Objekts beschreibt der betreffende Deskriptor). Die „Description Definition Language“ ist auf dem vom W3C standardisierten XML Schema aufgebaut. Sie definiert die Regeln, nach denen die Beschreibungsschemata aufgebaut sind, und die auch bei der Spezifikation benutzerdefinierter DS verwendet werden müssen. Die Gesamtheit aller DS ergibt das XML Schema, nach dessen Syntaxdefinition ein Dokument aufgebaut ist. Man spricht auch davon, dass das XML Dokument das Schema instanziiert.

2.3.2 MDS Deskriptoren

Die MDS Deskriptoren sind in mehrere Bereiche unterteilt:

1. Basic Elements
Hierunter fallen Werkzeuge zur Gliederung und Kommentierung des Inhalts. Eine MPEG-7 Beschreibung beginnt mit einem Wurzelelement, welches signalisiert, ob es sich um eine vollständige oder partielle Beschreibung handelt. Eine vollständige Beschreibung beinhaltet eine in sich abgeschlossene, eigenständige Beschreibung der audiovisuellen Inhalte. Im Gegensatz dazu enthält eine Beschreibungseinheit nur den Teil oder einen Baustein einer vollständigen Beschreibung, der möglicherweise eine vorhandene Beschreibung ergänzt. Bei einer vollständigen Beschreibung folgt das Grundelement (“Top-Level Element”), das die Beschreibung auf einen speziellen Zweck hin ausrichtet. Dieses Grundelement stellt alle wesentlichen Werkzeuge für die vorgesehene Aufgabe zur Verfügung. Im Falle der Beschreibungseinheit kann dem Wurzelelement ein beliebiges Beschreibungsschema nachfolgen. Außerdem werden grundlegende Datentypen definiert, z.B. im mathematischen

Bereich wie Zahlen und Matrizen, oder zur Beschreibung von Zeitpunkten oder -intervallen. Zu den grundlegenden Elementen zählen auch Konstrukte für die Verknüpfung von Mediadateien, Information um Inhalte zu lokalisieren, oder um (strukturierte) Kommentare zu erfassen.

2. Content Managment
Hierunter fallen zum einen Deskriptoren, die Information über die Herstellung der AV Inhalte enthalten. Dazu zählen etwa Titel, Genre, Autor oder Datum. Des weiteren können Hinweise enthalten sein, welche anderen Quellen zu den präsentierten Daten in Beziehung stehen. Informationen über das Medium geben Aufschluss über das Datei- oder Codierformat. Hinweise zur Nutzung klären beispielsweise darüber auf, wo die Rechte an dem Material liegen.

3. Beschreibung der Struktur des Inhalts
Das grundlegende Beschreibungsschema für diesen Bereich ist das Segment DS. Es ist eine abstrakte Klasse, die nicht selbst instanziiert wird. Vielmehr wird eine der abgeleiteten Klassen, die auf den jeweiligen Anwendungsfall zugeschnitten ist verwendet, z.B. Multimedia Segment DS, Video Segment DS, AudioVisual Region DS, Still Region DS, Moving Region DS etc. Diese Segmentierung kann sich auf die Zeit oder den Raum beziehen. Eine Videosequenz mit zeitlicher Untergliederung kann ein Satz an Einzelbildern eines Videostroms sein, eine räumliche Untergliederung dagegen kann z.B. aus einer unbewegten Region bestehen, ein raum-zeitliches Segment wäre eine bewegte Region. Die Segmentierung beschreibt also die räumliche und/oder zeitliche Partitionierung des Mediums. Sie kann auch hierarchisch erfolgen, mit fortschreitend feinerer Untergliederung der Segmente. Zusätzlich können auch zeitliche oder räumliche Relationen zwischen verschiedenen Segmenten beschrieben werden.

Neben dem Segment Deskriptor, der physikalische Eigenschaften des AV Materials erfasst gibt es auch noch einen Deskriptor für semantische Eigenschaften, den Semantic DS. Hier werden Ereignisse, Konzepte und Objekte in einer “erzählenden” Form erfasst. Beispiele für semantische Deskriptoren sind Object DS, Event DS, Concept DS, SemanticState DS, SemanticPlace DS oder SemanticTime DS.

4. Navigation und Access
Dieser Bereich von Deskriptoren ist dafür gedacht sich im AV-Material besser zurecht zu finden, und den Zugriff darauf zu erleichtern. Spezifiziert sind Inhaltsangaben oder Zusammenfassungen, Ansichten, Partitionen und Variationen. Die Inhaltsangaben sollen ein schnelles Durchsuchen des AV Materials ermöglichen. Die Deskriptoren für verschiedene Ansichten sind hilfreich, wenn das AV Material in verschiedenen räumlichen (Pixel) oder zeitlichen (Bilder/s) Auflösungen zur Verfügung steht. Wenn das Material in mehreren Variationen für verschiedene Terminals, Übertragungskanäle (z.B. fest/mobil) oder Vorlieben des Benutzers codiert wurde, so kann dies ebenfalls beschrieben werden.

5. Organisation des Inhalts
Hierunter fallen Deskriptoren, mit denen Material, das gemeinsame Eigenschaften hat gruppiert werden kann (Collection DS). Die Art der Gemeinsamkeiten wird ebenfalls erfasst. Außerdem werden verschiedene Gruppen von abstrakten Modellen zur Verfügung gestellt, mit denen gewisse Attribute und Eigenschaften abgebildet werden können, die sich z.B. durch Markov-Modelle darstellen lassen. Darüber hinaus gibt es Deskriptoren für Wahrscheinlich-keitsaussagen.

6. User Interaction
Hier werden Eigenschaften des Benutzers beschrieben, so etwa die Vorlieben des Nutzers (UserPreferenceDS) oder die Nutzungsstatistik. Diese ist beispielsweise für die Definition von Filtern hilfreich. Die Vorgeschichte der Nutzung (UsageHistory) kann ebenfalls dazu verwendet werden, um auf die Interessen des Nutzers zu schließen, und personalisierte Informationsangebote zu machen.

2.3.3 Video Deskriptoren

Die MPEG-7 Visual Deskriptoren wurden entworfen, um grundlegenden Bild- oder Videosequenzeigenschaften zu beschreiben wie Farbe, Textur, Form, Bewegung oder Ort. In jedem Bereich gibt es Basisdeskriptoren und Spezialdeskriptoren.

1. Grundlegende Strukturen
In diesem Bereich sind Deskriptoren definiert, die räumliche (Spatial2DCoordinates) oder zeitliche Referenzen in die Mediadaten erlauben, sowie die Beschreibung von dreidimensionalen Objekten durch einen Satz von zweidimensionalen Ansichten.

2. Farbdeskriptoren
Hier kann der Farbraum (z.B. R,G,B; Y,Cr,Cb; Monochrom etc.) festgelegt werden, sowie die Auflösung von Farben in Bits. Mit den verschiednen Deskriptoren können dominante Farben ebenso erfasst werden, wie die Verteilung der Farben in Bildern und Kenngrößen (Mittel-/Medianwert etc.) der Farben in Videosequenzen.

3. Texturdeskriptoren
Mit diesen Deskriptoren wird die Struktur der Textur nach räumlicher Orientierung und räumlicher Frequenz differenziert. Außerdem werden Kanten in verschiedenen Richtungen beschrieben.

4. Formdeskriptoren
Hier ist ein Deskriptor definiert, der aus allen Pixeln einer Form berechnet wird, und robust gegenüber geringfügigen Deformationen ist, sowie ein Deskriptor, der aus der Kontur gewonnen wird, und sich bei partiellen Verdeckungen oder perspektivischen Verzerrungen nur wenig ändert. Außerdem gibt es einen Deskriptor für dreidimensionale Gitternetze.

5. Bewegungsdeskriptoren
Hier findet sich ein Deskriptor, der die Bewegung der Kamera beschreibt (er kann aus dem Bildmaterial gewonnen werden), sowie ein Deskriptor für die Trajektorie eines Objekts in Form von (x,y,z,t) Koordinaten. Darüber hinaus gibt es einen Deskriptor, der Bewegungen mit einem parametrisierten Modell beschreibt und einen Deskriptor für die allgemeine Intensität von Bewegung in einer Videosequenz.

6. Ortsdeskriptoren
Hier können Bereiche im Bild durch ein Rechteck oder ein Polygon identifiziert werden. Außerdem kann die Bewegung eines Objekts über der Zeit beschrieben werden.

7. Sonstige
Hier ist ein Deskriptor definiert, der Gesichter beschreibt, indem ein normalisiertes Bild des Gesichts mit einem Referenzgesicht verglichen wird, und die Abweichungen bestimmter Eigenschaften in einem Vektor zusammengefasst werden.

2.3.4 Audio Deskriptoren

1. Basic

Ein Deskriptor beschreibt die Einhüllende des Audiosignals (für die Anzeige), ein anderer die momentane Leistung.

2. Basic Spectral

Hier wird durch einen Deskriptor grob das Leistungsspektrum, also die Leistung in verschiedenen Frequenzbändern beschrieben. Weitere Deskriptoren geben den Mittelwert und die Breite der Verteilung der Leistung über der Frequenz an.

3. Signal Parameters

Ein Deskriptor beschreibt die Grundfrequenz eines Audiosignals, ein anderer erlaubt die Unterscheidung zwischen harmonischen Tönen (Musik, Sprache) unharmonischen Tönen (metallische Töne) oder nicht harmonischen Tönen (Geräusche).

4. Timbral Temporal

Ein Deskriptor erfasst die Zeit zwischen Stille und maximaler Amplitude, ein anderer zeigt an, wo sich die Signalenergie über der Zeit konzentriert.

5. Timbral Spectral

In diesem Bereich beschreiben mehrere Deskriptoren mit Hilfe des linearen Leistungs-spektrums die Klangfarbe eines Audiosignals, wo sich die Signalenergie konzentriert, und wie stark sie über das Spektrum verteilt ist.

6. Spectral Basis

Zwei Deskriptoren beschreiben eine zweidimensionale Projektion des hochdimensionalen Spektralraums. Anwendungen sind im Bereich Klassifikation und Indexierung von Klängen möglich.

7. High Level Audio Descripors

Fünf Bereiche von Deskriptoren, die in etwa den Anwendungsszenarien entsprechen sind in diesem Bereich enthalten:

Der Audio Signatur Deskriptor ist eine statistische Zusammenfassung des Spectral Flatness Deskriptors, die insbesondere für den Vergleich von Audiosignalen geeignet ist.

Eine weitere Klasse von Deskriptoren beschreibt die Klangfarbe von Audiosignalen durch eine Kombination verschiedener low-level Deskriptoren. Für den Vergleich zwischen Deskriptoren dieser Klasse wurde eine spezielle experimentell abgeleitete Distanzmetrik entwickelt.

Für die Beschreibung von Melodien gibt es eine weitere Klasse von Deskriptoren, die mit unterschiedlichem Aufwand und unterschiedlicher Präzision eine Folge von Tönen codiert. Der einfachste beschreibt in einem groben Raster die Intervalle aufeinanderfolgender Töne, sowie einfache Rhythmik. Mehr Präzision bietet ein anderer Deskriptor, der die exakten Intervalle codiert, sowie eine genaue Beschreibung der zeitlichen Intervalle zwischen dem Anschlag der einzelnen Töne zur Verfügung stellt.

Für die allgemeine Klassifikation und Indexierung von Klängen gibt es eine weitere Klasse von Deskriptoren. Diese nutzen low-level Spektraldeskriptoren als Grundlage, und binden diese in ein statistisches Modell ein. Die dadurch erreichte Klassifikation kann verschiedene Bereiche wie Sprache und Musik unterscheiden, sowie auf die Unterscheidung

speziellerer Eigenschaften, wie männliche oder weibliche Stimmen oder Instrumentklassen ausgelegt werden.

Für die Beschreibung von Sprache gibt es weitere spezialisierte Deskriptoren. Als Anwendung soll das wiederfinden von gesprochenem Material möglich sein, indem man einen Teil der Worte, der im Gedächtnis geblieben ist für die Suche verwendet. Als Ergebnis erhält man eine Referenz zum Zeitpunkt dieser Worte, beispielsweise in einer Rede. Ein anderer Anwendungsfall wäre das wiederfinden ganzer Audiodateien mit gesprochenem Inhalt (z.B. einer Rede) in einer Datenbank. Dafür wird ein Netz aus Knoten benutzt, das aus wort- oder phonemindizierten Verbindungen aufgebaut ist. Diese Worte oder Phoneme stammen aus einem Wort/Phonem-Lexikon, das auch ein Teil des Beschreibungssystems ist. Durch diese Netzstruktur soll die Wiedererkennungsrate erhöht werden, wenn Worte auftreten, die nicht im Lexikon stehen oder falsch erkannt wurden.

2.3.5 Gewinnung der Deskriptoren

Für den praktischen Einsatz von MPEG-7 basierten Systemen ist es wichtig, dass die Beschreibungen mit vertretbarem Zusatzaufwand erzeugt werden können. Auch wenn die low-level Deskriptoren zum großen Teil automatisch aus den Mediadaten gewonnen werden können, und die dafür notwendige Rechenleistung heute keine technischen (oder finanziellen) Probleme bereiten dürfte, so ist es doch mit erheblicher redaktioneller Arbeit verbunden beispielsweise für eine Fernsehsendung einen Metadatenstrom zu erzeugen, der für den Anwender auch einen Gewinn darstellt. Bei älterem Material, für das noch keine Hintergrundinformationen systematisch erfasst wurde wird das ein Problem sein. Bei neurem Material müsste diese Information zumindest neu formatiert werden, um von einem MPEG-7 System verarbeitet zu werden. Das Ziel sollte jedoch eine integrierte Lösung sein, bei der schon für den Erzeugungsprozess im Studio eine MPEG-7 basierte Authoring-Software verwendet wird. Die Authoring-Software unterstützt schon im Studio bei der Produktion von multimedialem Material die redaktionelle Arbeit. Der zusätzliche Aufwand für die Erzeugung einer Beschreibung für den Konsumenten des Materials würde deutlich geringer ausfallen. Der Entwicklung von Systemen, die eine effiziente Generierung der Metadaten ermöglichen wird eine Schlüsselrolle für die Akzeptanz der MPEG-7 Technologie zukommen.

2.3.6 Anwendung von Deskriptoren

Für die Anwendbarkeit der Deskriptoren sind einige Eigenschaften relevant, die beim Entwurf der Deskriptoren berücksichtigt werden mussten. Bei der Gewinnung eines Deskriptors ist ein wichtiges Kriterium, welcher Rechenaufwand erforderlich ist, um ihn aus den Mediadaten zu ermitteln. Die Deskriptoren sollten darüber hinaus möglichst kompakt sein, um den zusätzlichen Speicherbedarf in einer Datenbank oder die zusätzliche Bandbreite auf einem Übertragungsmedium gering zu halten. Der Vergleich, der notwendig ist um die Ähnlichkeit zwischen Deskriptoren festzustellen sollte möglichst einfach durchgeführt werden können. Und natürlich müssen die Deskriptoren bezüglich der Eigenschaften, die sie beschreiben sollen auch aussagekräftig sein, ohne sich von Störgrößen wie beispielsweise Rauschen allzu sehr beeinflussen zu lassen.

Berechnung der Übereinstimmung zwischen Deskriptoren

In vielen Fällen ist es notwendig sowohl bei Datenbankabfragen als auch bei der Filterung von Beschreibungen ein Maß für die Übereinstimmung zwischen den Deskriptoren der Suchanfrage oder des Filters und den Deskriptoren des geprüften Materials in der Datenbank oder dem Datenstrom zu berechnen (“matching”). Dadurch wird es beispielsweise möglich

eine geordnete Liste der Treffer zu erzeugen, oder ausgefeilte Kriterien festzulegen, welche Daten eine Filterbedingung erfüllen und welche nicht. Dabei können mehrere Deskriptoren in beliebiger Kombination verwendet werden. Dann entsteht allerdings das Problem für diese Kombination ein einziges Distanzmaß zu gewinnen. Für die Berechnung dieses Maßes sind viele Möglichkeiten denkbar, z.B. eine Summe der absoluten Differenzen zwischen Deskriptorwerten, die euklidsche Distanz oder noch komplexere Vorschriften. MPEG hat für diesen Prozess keine Verfahren spezifiziert, da das Problem der Gewinnung vernünftiger Distanzmaße von der Anwendung abhängig ist.

Es wurden bereits Experimente vorgenommen, welche Strategien zu brauchbaren Ergebnissen führen, z.B. bei der Recherche in Bilddatenbanken [Smi02].

Referenzierung der Mediadaten

Wenn nach einem Such- oder Filtervorgang auf die relevanten Mediadaten zugegriffen werden soll muss in einer Metadatenbeschreibung die Information zur Verfügung stehen, auf welches Medium und welche Stelle innerhalb des Mediums sich ein Deskriptor bezieht. Dafür gibt es eine spezielle Klasse von Deskriptoren (MediaLocator und davon abgeleitete Deskriptoren), mit denen Referenzen in die Mediadaten formuliert werden können, beispielsweise durch eine Media URL, eine Zeitangabe innerhalb eines Videostroms etc. Die Transportschicht ist dafür verantwortlich, dass diese Referenzen auch aufgelöst werden können (z.B. durch Time-Stamps in einem Video).

2.4 Darstellung der Metadaten

Neben der Frage was die Metadaten beschreiben und wie sie aus den Mediadaten gewonnen werden ist auch entscheidend, wie sie dargestellt werden. Der elementare Datentyp der Deskriptoren ist durch die Art der beschriebenen Eigenschaft festgelegt, beispielsweise eine ganze Zahl, eine Fließkommazahl, ein Vektor, eine Matrix oder einfach eine Zeichenkette. Für semantisch aussagekräftige Beschreibungen müssen diese einzelnen Datenelemente aber auch in eine Beziehung gesetzt werden, aus der die Bedeutung jedes Datenelements für die gesamte Beschreibung hervorgeht. Dafür muss eine geeignete Datenstruktur eingesetzt werden.

2.4.1 Textuelle Darstellung

Um diese Aufgabe zu erfüllen werden die Metadaten in Form hierarchisch strukturierter Dokumente dargestellt. Diese Darstellung ermöglicht es die einzelnen Elemente einer Beschreibung in eine definierte Beziehung zu setzen: die Eigenschaften eines Datenelements werden durch die hierarchisch tiefer liegenden Datenelemente weiter aufgeschlüsselt. In der Rohform haben die Metadaten ein textuelles Format, das sowohl von Menschen gelesen werden, als auch von automatischen Systemen verarbeitet werden kann.

Für dieses Datenformat hat man sich bei MPEG auf den Standard XML verständigt. In XML wird den Datenelementen eines Dokuments eine strikte Syntaxdefinition zugrundegelegt. Diese Syntaxdefinitionen können mittels einer speziellen Sprache frei und den Erfordernissen entsprechend entworfen werden. Dadurch wird eine hierarchische Strukturierung der XML Dokumente erreicht, allerdings um den Preis, dass das Datenvolumen auf ein Mehrfaches (typisch ist etwa der Faktor drei bis fünf) der Nutzinformation ansteigt. Die Sprache, mit der die XML Syntaxdefinitionen spezifiziert werden heißt bei MPEG Description Definition Language (DDL). Diese Sprache baut auf der XML Schema Sprache auf, welche vom W3C entworfen worden ist, und erweitert sie um einige wenige

Datentypen. Die Syntaxdefinitionen für die MPEG-7 Deskriptoren sind in Form eines solchen XML Schemas spezifiziert, das einen nicht unerheblichen Umfang hat (es sind insgesamt fast 800 Typdefinitionen). Weitere Details zu XML finden sich am Anfang von Kapitel vier.

2.4.2 Binäre Darstellung

Die textuelle Darstellung hat eine Reihe von Nachteilen, die durch ein binäres Format kompensiert werden sollen. Die Hauptvorteile eines binären Formats sind eine kompaktere Darstellung der Information und die Möglichkeit eine Reihe von zusätzlichen Funktionalitäten zu unterstützen (s.u.). Ein binär codiertes Format ist allerdings nicht mehr von Menschen lesbar.

2.5 Anforderungen von MPEG-7 an das Binärformat

Das breite Spektrum an Anwendungen, für das MPEG-7 Lösungen anbieten will, hat einen Katalog von Anforderungen an eine Repräsentation der Metadaten zur Folge, die durch eine binäre Darstellung zu erfüllen sind. Folgende Aufstellung leitet diese Anforderungen aus den Anwendungsszenarios ab.

Die offensichtlichste erforderliche Eigenschaft ist eine möglichst kompakte Darstellung der Metadaten. Dies ist wichtig bei der Übertragung über mobile Kanäle mit geringer verfügbarer Datenrate. Selbst wenn der Kanal eine größere Kapazität hat, wie beispielsweise ein Kabelkanal für das digitale Fernsehen, können Konstellationen auftreten die eine kompakte Darstellung relevant werden lassen: bei einem Beschreibungsstrom für eine TV-Sendung kann es erforderlich sein die Metadaten wiederholt zu übertragen (in einem ‘Datenkarussell’), um Nutzern, die sich später eingeschaltet haben trotzdem einen Dienst anbieten zu können. Dadurch erhöht sich das zu übertragende Datenvolumen und macht eine Kompression interessant, auch wenn die Metadaten zunächst im Vergleich zu den Mediadaten einen eher geringen Anteil der Datenrate beanspruchen. Auch für die Archivierung von Metadaten ist eine kompakte Darstellung interessant. Sowohl die Verbraucher mit kostengünstigen Endgeräten, als auch die Betreiber von großen Archiven oder Datenbanken profitieren davon.

Eine hohe Kompression sollte allerdings nicht um den Preis einer hohen Komplexität bei Encodierung oder Decodierung erreicht werden. Vor allem die Decodierung sollte im Hinblick auf mobile Endgeräte auch mit geringer Rechenleistung durchgeführt werden können. Da oftmals neben den Metadaten auch ein Medienstrom zu decodieren ist, und die Metadaten synchron zu den Mediadaten zur Verfügung stehen sollen ergeben sich dadurch auch Echtzeitanforderungen an die Decodierung.

Für eine paketorientierten Übertragung ist es vorteilhaft, wenn eine Beschreibung in mehrere Fragmente unterteilt und inkrementell übertragen werden kann. Zwar ist dies auch mit Text möglich, indem eine Datei an beliebigen Stellen aufgebrochen und zerteilt wird, aber es ist dann schwierig den Inhalt der einzelnen Teile zu kontrollieren. Die Anzeige oder Weiterverarbeitung kann sich verzögern, weil ein wichtiges Datenfragment noch nicht beim Empfänger angekommen ist. Deshalb wäre eine Fragmentierung wünschenswert, bei der man den Inhalt der Fragmente und die Reihenfolge der Übertragung beliebig bestimmen kann.

Es kann Situationen geben, beispielsweise bei der Beschreibung von Vorgängen in Echtzeit (live-Ereignisse), wo man keine Kontrolle darüber hat, wann und in welcher Reihenfolge die Metadaten anfallen. Es kann auch sein, dass Dateneinträge, die schon zum Empfänger übertragen worden sind ihre Gültigkeit verlieren und durch neue ersetzt werden

sollen. In diesem Falle ist es vorteilhaft Teile der Beschreibung dynamisch verändern zu können, ohne nochmals alles neu übertragen zu müssen. Dadurch spart man Bandbreite auf dem Übertragungsmedium oder Speicher im Terminal, da man Teile der Beschreibung, die nicht mehr gültig sind, auch löschen kann. Es ist dabei umso besser, je feiner man die Daten identifizieren kann die verändert werden sollen.

Beim Einsatz von MPEG-7 in Informationsfiltern kann der Fall auftreten, dass eine große Menge an Daten parallel gefiltert werden soll. Wenn beispielsweise ein digitaler Videorecorder für hundert Kanäle die einem Benutzerprofil entsprechenden Sendungen herausfiltern soll, so ist es wichtig, dass diese Filteraufgabe mit möglichst moderatem Rechenaufwand durchgeführt werden kann. Dazu sollte sich die binäre Darstellung der Metadaten einfach parsen lassen.

Die Anforderung eine Beschreibung in Teilen zu verändern oder zu aktualisieren kann auch durch eine Benutzerinteraktion ausgelöst werden. In Fällen, wo ein Rückkanal vom Empfänger zum Sender besteht kann der Nutzer Information anfordern. Auch in diesem Fall ist es vorteilhaft, wenn nur die angeforderte Information neu übertragen werden muss, das Dokument also dynamisch aktualisiert werden kann, und ein Zugriff auf möglichst jedes einzelne Datenelement einer Beschreibung möglich ist.

Kurz zusammengefasst sind die Anforderungen an eine binäre Codierung der Metadaten also (siehe auch [MPEG00a]):

- Hohe Kompression – v.a. für Übertragung auf schmalbandigen Kanälen und Archivierung großer Datenmengen
- Geringe Komplexität des Encodier- und vor allem Decodiervorgangs
- Vorkehrungen für inkrementelle Übertragung – für paketorientierten Informationsaustausch
- Dynamische Aktualisierung eines Dokuments, möglichst feingranular – für die Beschreibung von Live-Ereignissen und Reaktion auf Benutzerinteraktion
- Beliebige Übertragungsreihenfolge der Elemente
- Einfaches Parsen des Bitstroms – z.B. für Filterung von spezifischer Information

Kapitel 3 – Bekannte Codierverfahren

In diesem Kapitel werden bekannte Verfahren vorgestellt, welche für die Codierung von XML oder verwandten Formaten geeignet sind. Dabei werden sowohl allgemeine als auch auf XML optimierte Algorithmen behandelt[1]. Die Verfahren werden hinsichtlich der in Kapitel 2 aus den Anwendungen abgeleiteten Anforderungen an eine Codierung bewertet.

3.1 Universelle Codierverfahren

3.1.1 ZIP

Dieses klassische Kompressionsverfahren lässt sich auf alle möglichen Quellendatenformate anwenden, ist aber in erster Linie für Text gedacht. Der ursprüngliche Algorithmus geht auf eine Veröffentlichung von J. Ziv und A. Lempel aus dem Jahr 1977 zurück („LZ-Algorithmus“) [ZIV77]. Er zielt darauf ab, die Redundanz innerhalb eines Datensatzes zu eliminieren: während der Kompressor die zu komprimierende Zeichenkette parst, sucht er vorwärts bei jedem neuen Zeichen die längste Zeichenkette, die bis dahin schon einmal aufgetreten ist, und codiert eine Referenz auf diese Zeichenkette.

Soll beispielsweise die folgende Zeichenkette komprimiert werden:

a b c d e f g c d e c d e h i j

so stellt der Kompressor beim zweiten c fest, dass die Zeichenkette “c d e” fünf Positionen zuvor schon einmal aufgetreten ist, und codiert eine Referenz der Form (Relative Position, Länge) dementsprechend wird für das Beispiel codiert:

a b c d e f g (-5,6) h i j

Besonders effizient ist der Algorithmus bei periodischen Wiederholungen. Um die Komplexität bei der Detektion der Redundanzen in vertretbarem Rahmen zu halten, wird ein Suchfenster begrenzter Größe verwendet, so dass maximal bis zu einer vorgegebenen Position rückwärts gesucht wird.

Diese Grundidee wurde in neueren Verfahren weiterentwickelt, welche beispielsweise während der Kompression ein Codebuch mit bereits aufgetretenen Zeichenketten aufbauen [Wel84].

Prinzipiell steigt für LZ-verwandte Algorithmen die Kompressionsrate mit der Größe der komprimierten Datei bis zu eine Grenze an, da die Redundanz zwangsläufig zunimmt, wenn dem Dokument ein begrenzter Wortschatz zugrunde liegt, wie dies bei natürlicher Sprache aber auch bei XML-Dokumenten der Fall ist.

[1] Teilweise wird in der Behandlung der Codierungsalgorithmen auf Eigenheiten von XML verwiesen. Ein kurze Einführung in XML findet sich am Anfang von Kapitel vier.

Bewertung

Die hier beschriebene Klasse von Kompressionsalgorithmen erlaubt keinen wahlfreien Zugriff auf Elemente in der komprimierten Datei, da diese nur inkrementell decodiert werden kann: um einen Teil zu decodieren, muss die gesamte Datei bis dahin decodiert worden sein.

Die spezifischen Eigenheiten von XML Dateien werden bei der Kompression nicht ausgenutzt: beispielsweise kann durch die Strukturierung der Information das Starttag eines Elements so weit vom Endtag entfernt sein, dass dieses wegen der Begrenzung des Suchfensters nicht mehr als großteils redundant zum Starttag erkannt wird. Die weiter unten gelisteten Verfahren bauen teilweise auf dem LZ-Algorithmus auf, und zielen darauf ab ihn durch eine Vorverarbeitung auf die Besonderheiten von XML anzupassen.

Um den LZ-Algorithmus bei einer inkrementellen Übertragung einzusetzen könnte man die ursprüngliche Datei in mehrere Fragmente aufteilen. Allerdings würde dies die Redundanz verringern wenn als Ziel einer Referenz nur das jeweils dekodierte Fragment in Frage kommt. Bei der Referenzierung über Fragmentgrenzen hinweg würde die Fehlerrobustheit leiden, da dann Datenabhängigkeiten zwischen den Fragmenten bestehen.

Der Kontext in dem ein Element in der XML Beschreibung instanziiert wird ist in einer LZ-komprimierten Datei nicht transparent, da die ursprüngliche Strukturierung einer XML Datei verloren geht. Deshalb ist eine einfache Filterung nach speziellen Elementen ohne vorherige Dekompression nicht zu realisieren. Jedoch wurden Verfahren entwickelt, mit denen nach Zeichenketten im komprimierten Dokument gesucht werden kann [Klein02].

Der Hauptvorteil des LZ-Algorithmus ist seine universelle und unkomplizierte Einsetzbarkeit, ohne dass auf die Verfügbarkeit oder die Version des XML Schemas Rücksicht genommen werden muss.

3.2 Auf XML optimierte Codierverfahren

3.2.1 XMLZIP

XMLZIP, entwickelt von *XML Solutions* [XMLZ] war eines der ersten speziell für XML optimierten Kompressionsverfahren. Es unterteilt den XML-Beschreibungsbaum in ein Fragment, das an der Wurzel beginnt, und sich bis zu einer vorgegebenen Tiefe *T* erstreckt. Die verbleibenden Teilbäume werden in andere Fragmente verpackt, und von dem Fragment, welches die Wurzel enthält referenziert. Die Fragmente werden mit einer Standard ZIP Bibliothek komprimiert. Da das Aufteilen in Fragmente die Redundanzen verringert ist die Kompressionsleistung normalerweise schlechter als der ursprüngliche LZ-Algorithmus. Der Hauptvorteil des Verfahrens liegt darin, dass man auf Teile der Beschreibung zugreifen kann ohne den gesamten Baum zu dekomprimieren. Die Bedeutung des Verfahrens für die Zukunft erscheint zweifelhaft.

3.2.2 XMILL

Das Grundprinzip von XMILL [Lie00] ist es die ursprüngliche XML Datei umzuformen um die Redundanz zu erhöhen, und dann eine auf einem Lempel-Ziv Verfahren basierende Kompression anzuwenden. Bei der Umformung werden drei Maßnamen durchgeführt:

- Die Struktur (d.h. die XML-Tags) wird vom Inhalt getrennt und separat komprimiert.
- Daten mit verwandter Bedeutung werden gruppiert. Dabei werden Datencontainer gebildet und getrennt komprimiert. Beispielsweise können alle Datenelemente vom Typ <name> in einen Container gepackt werden, alle Datenelemente vom Typ <phone> bilden einen anderen.
- Verschiedene Container können mit verschiedenen, speziell optimierten Kompressoren codiert werden, die z.B. auf Text, Zahlen oder DNA-Sequenzen ausgelegt sind.

Folgendes Beispiel illustriert die Arbeitsweise von XMILL:

```
<Bestellung>
   <Artikel Preis=”15.95”>
     <ProduktName>Mozartkugeln</ProduktName>
   </Arikel>
   <Artikel Preis=”1555.95”>
     <ProduktName>Zauberflöte</ProduktName>
   </Arikel>
</Bestellung>
```

Abb. 3.1: XML Beispielbeschreibung zur Illustration von XMILL

Trennung von Struktur und Inhalt

Die Tags werden mit einem Codebuch codiert, d.h. ihnen wird eine ganze Zahl zugewiesen, z.B. Bestellung=#1, Artikel=#2, Preis=#3, ProduktName=#4,

Die Struktur wird dann codiert zu: #1 #2 #3 C1 / #4 C2 / / #2 #3 C1 / #4 C2.

Das Zeichen “/” beendet das entsprechende Element oder Attribut. Im Container C1 werden alle Preise abgelegt, im Container C2 alle Produktnamen etc. Eine Referenz “C1” in der Strukturinformation bedeutet einen Zugriff auf den nächsten noch nicht verwendeten Eintrag in dem entsprechenden Container. Diese Strukturinformation wird in den Strukturcontainer verpackt, und kann mit ZIP komprimiert werden, weshalb auch reguläre Strukturen effizient codiert werden. Diese Redundanz der Struktur wäre in der ursprünglichen XML Datei wegen der eingestreuten Nutzdaten durch ZIP evtl. nicht zu erkennen.

Gruppierung der Daten in Container

Die im Vergleich zu Standard-ZIP verbesserte Kompression beruht darauf, dass semantisch verwandte Daten zusammen gruppiert werden. Um semantische Ähnlichkeit festzustellen wird der Pfad von der Wurzel des Dokuments zu dem betreffenden Datenelement verwendet (dieser kann aus der Strukturinformation gewonnen werden). Pfade, welche den letzten Abschnitt oder mehrere Abschnitte am Ende des Pfades gemeinsam haben enthalten üblicherweise ähnliche Daten, welche zusammen in einen Container gepackt werden. Der Nutzer hat auch die Möglichkeit von Hand eigene Container zu spezifizieren.

Optimierte Kompressoren

Da XML Dateien oft spezielle Daten enthalten, z.B. IP Adressen, Postleitzahlen, Matrizen etc. bietet es sich an für den entsprechenden Datentyp optimierte Codecs zu entwerfen. In der Orginalversion von XMill sind Kompressoren für Text, Zahlen verschiedener Art, Lauflängen oder Aufzählungen integriert, der Nutzer kann auch selbst für spezifische Datentypen optimierte Codecs einbinden. Allerdings kann die Datei dann nicht mehr universell decodiert werden, da zum Decodieren wieder dieser Codec benötigt wird.

Bewertung

XMill ist nicht für inkrementelle Übertragung oder dynamische Aktualisierung von Dokumenten entworfen. Die Reihenfolge der Übertragung ist nicht beliebig. Die Gruppierung der Daten in Containern setzt eine Grenze für die Granularität mit der auf die Information zugegriffen werden kann. Der wesentliche Vorteil von XMill ist wie bei ZIP die Unabhängigkeit vom Schema, zumindest auf der Decoderseite. Für die Definition sinnvoller Container und die Zuweisung von Nutzinformation zu geeigneten Containern ist ein Schema oder eine Document Type Definition (DTD) zumindest hilfreich.

3.2.3 WBXML

WBXML ist ein binäres Codierverfahren, das vom Wireless Application Protocol Forum vorgeschlagen wurde [W3C99]. Es ist vor allem für die Übertragung über schmalbandige Funkkanäle gedacht.

Bei WBXML werden für häufig verwendete Symbole Codebücher, sogenannte Token Tabellen generiert. Diese Codebücher sind in mehrere Seiten mit 256 Einträgen organisiert (ein Token hat eine Länge von einem Byte). Es wird unterschieden zwischen globalen Token, die fest eingebaut sind und anwendungsspezifischen Token. Die globalen Token enthalten Codes welche die Decodierung steuern, z.B. Codeseitenwechsel, Delimiter, Anzeige von Inhalt in Form einer Zeichenkette etc. Die anwendungsspezifischen Token sind eine Kurzform für die vorkommenden Attribut- oder Tagnamen, welche aus der DTD oder dem XML Schema gewonnen werden können. Auch häufig vorkommende Zeichenketten in Attributwerten (“http://”) können einem Token zugewiesen werden. Ansonsten werden Zeichenketten als “Inline String” unkomprimiert codiert.

Die Codetabellen, welche die anwendungsspezifischen Token zu den Zeichenketten in Beziehung setzten werden vor dem eigentlichen Dokument codiert. Die Übersetzung eines Dokuments in eine Tokenfolge oder zurück wird durch einen einfachen Zustandsautomaten beschrieben, der zwischen dem Lesen von Tags oder Attributen unterscheidet.

Bewertung

Dynamische Aktualisierung von Dokumenten sowie beliebige Übertragungsreihenfolge der Elemente ist bei WBXML nicht möglich, da die Struktur des Dokuments implizit durch die Reihenfolge, in der die Elemente übertragen werden aufgebaut wird. Deshalb muss diese der Anordnung der Elemente im textuellen Dokument entsprechen.

3.2.4 Millau

Der im vorhergehenden Abschnitt beschriebene Kompressor nutzt zwar das Vorwissen über die Struktur für eine Kompression, der Inhalt wird aber nicht speziell codiert. Millau [Gir00] erweitert daher die Tabelle für globale Token, um z.B. ZIP komprimierte Zeichenketten oder Zahlen verschiedener Formate (Bool, Byte, Integer, Fließkommazahl) im Tokenstrom anzuzeigen.

Ansonsten werden statistische Abhängigkeiten nicht ausgenutzt, was vor allem bei größeren Dokumenten Vorteile bringen würde. Millau wurde für den Anwendungsbereich elektronischer Handel („eBusiness“) konzipiert, in dem üblicherweise kleine Dateien (<5KB) ausgetauscht werden.

Die Kompression mit Millau reduziert die Dateigröße auf etwa ein Fünftel, unabhängig von der Größe der ursprünglichen Datei. Für Dateien kleiner als 5KB ist das Verfahren besser als gzip. Bei größeren Dateien macht sich die Redundanz der Tags bemerkbar, da diese aus einem durch die Syntaxdefinition begrenzten Satz an Möglichkeiten stammen. Hier hat gzip Vorteile: für eine 3MB große Beispieldatei mit 65% Strukturinformation (Markup) ist gzip um 58% besser als Millau.

Die Entwickler von Millau hatten auch explizit webbasierte Anwendungen im Auge: so schlagen sie vor die codierte Datei aufzuteilen in einen Tokenstrom, welcher die Struktur codiert und einen Strom, welcher die Zeichenketten der Nutzdaten enthält. In diesem Fall braucht nur dann auf ein spezielles Datum in den Nutzdaten zugegriffen werden, wenn die Anwendung dies explizit verlangt. Da der Tokenstrom für die Struktur sehr kompakt ist, geht nicht nur die Übertragung über schmalbandige Kanäle sondern auch das Parsen dieses Stroms sehr schnell.

Bewertung

Da Millau auf WBXML aufbaut gilt im Prinzip dasselbe wie in der dortigen Bewertung. Die Trennung von Struktur und Inhalt ist ähnlich wie bei XMILL, sowie die Idee spezielle Daten in dafür angepassten Formaten zu codieren. Trotz der separaten Codierung der Struktur muss diese in der Reihenfolge, in der sie im textuellen Dokument erscheint übertragen werden.

3.2.5 XMLPPM

Das Kürzel “PPM” steht für “Prediction by Partial Match”. Diese Methode stellt eine Statistik auf, die Informationen darüber enthält welche Symbole im Kontext vorangegangener Symbole auftreten. Das statistisches Modell wird verwendet, um die Wahrscheinlichkeit abzuschätzen, mit der ein Symbol in einem bestimmten Kontext auftreten kann. Es wird dann eine Arithmetische Codierung verwendet um das aktuell codierte Symbol auszuwählen. Das statistische Modell wird laufend aktualisiert. Für jedes Symbol wird zunächst der längste Kontext geprüft, für den das Modell eine Aussage macht. Wird das Symbol in diesem Kontext nicht gefunden, so wird eine Escape-Symbol codiert und der nächst längste Kontext überprüft.

Bei der in XMLPPM [Cha01] verwendeten Methode bilden nicht die vorangegangenen Symbole den Kontext, sondern der Pfad zur Wurzel. Beispielsweise ist für

<a>XY der Kontext für X <a>, aber der Kontext für Y ist nicht X, sondern <a>.

Da sich Elemente, Attribute und Zeichenketten syntaktisch unterscheiden, werden verschiedene statistische Modelle für die Prädiktion verwendet, für Element- und Attributnamen, für die Elementstruktur, für Attribute und für Zeichenketten.

Die Kompression mit XMLPPM ist um durchschnittlich 12% besser als BZIP (eine im Vergleich zu GZIP optimierte, aber langsamere ZIP Variante).

Bewertung

Da die statistischen Modelle während der Codierung aus dem XML Dokument gewonnen und laufend aktualisiert werden, muss das Dokument von Beginn an vollständig decodiert werden, ein wahlfreier Zugriff auf Elemente ist nicht möglich. Für kleine Dateien ist die Leistungsfähigkeit des Verfahrens begrenzt, da keine aussagekräftigen Modelle generiert werden können. Fortschrittliche Codieroptionen wie dynamische Aktualisierung und beliebige Übertragungsreihenfolge sind nicht vorgesehen.

Bei Übertragungsfehlern sind die statistischen Modelle (die am Encoder und Decoder unabhängig voneinander nach dem gleichen Algorithmus aufgebaut werden) nicht mehr konsistent, weshalb die Fehlerrobustheit nicht gewährleistet ist.

3.3 Syntaxbasierte Codierverfahren

3.3.1 SoC

Das Kürzel SoC steht für “Syntax oriented Coding” [Eck98]. Es ist kein auf XML spezialisiertes Verfahren, sondern eher ein allgemeines Konzept, das sich auf Dokumente anwenden lässt, die nach einer wohldefinierten Syntax erzeugt wurden. Die Autoren hatten beispielsweise die Kompression von Java-Applets im Sinn, die über das Netz geladen werden. Sonstige Anwendungen wären: Codierung und Kompression des Quelltextes von Programmen in Programmiersprachen wie C oder Pascal oder komplexe Kommunikations-protokolle. Sie beschreiben das Grundprinzip folgendermaßen: “Es werden nicht die Symbole des Dokuments selbst übertragen, sondern eine Referenz auf die Regel, nach der sie erzeugt wurden.”

Die statistische Verteilung der Erzeugungsregeln wird ausgenutzt, um die Kompression zu erhöhen: häufig angewandte Regeln erhalten kurze Codeworte (z.B. durch eine Arithmetische Codierung). Für die Kompression von Java Bytecode erreicht das Verfahren eine Reduktion auf ein fünftel der ursprünglichen Größe.

Bewertung

Die Signalisierung einer Erzeugungsregel lässt sich auch für XML Dokumente anwenden, da das XML Schema als Satz von Erzeugungsregeln gelten kann. Statistische Optimierungen bergen aber im Kontext der Anforderungen Probleme (siehe Bewertung von XML Xpress). Da es sich nur um ein Grundkonzept handelt, das auf die eigentliche Anwendung erst ausgelegt werden muss kann man bezüglich der speziellen Anforderungen noch keine Aussage treffen.

3.3.2 XML Xpress

XML Xpress [XMLX00] ist ein schemabasiertes Verfahren, das die Information in einer DTD oder einem XML Schema ausnutzt um hohe Kompressionsraten zu erzielen. Wie bei der SoC kann optional neben dem Schema auch eine Statistik für die jeweilige Klasse der zu codierenden Dokumente verwendet werden um die Kompressionsrate zu erhöhen. Die Statistik wird aber nicht während der Codierung aus dem Dokument generiert, sondern es ist vorgesehen, dass man das XML Schema und aussagekräftige Beispiele für die erwarteten XML Dokumente der Firma, welche XML Xpress entwickelt hat zur Verfügung stellt. Daraus wird dann ein "Schema Model File" generiert, das für die Codierung verwendet wird, und für die Statistik der Beispieldokumente optimiert ist. Die Kompressionsrate schwankt zwischen 6 für kleine (<1K) und 35 für große Dateien (>10M).

Ebenfalls vorgesehen ist eine Optimierung für das Streaming von XML Dokumenten: Dateien können in Pakete aufgeteilt werden, die allerdings in der richtigen Reihenfolge übertragen werden müssen. Der durch die Paketisierung entstehende Overhead beträgt 5% bis 15%.

Bewertung

XML Xpress war Mitte 2000, als bei MPEG an einem Binärformat für XML gearbeitet wurde noch unbekannt. Es fehlt die dynamische Aktualisierung von Dokumenten, außerdem sind die Übertragungseigenschaften noch nicht optimal, da die Reihenfolge der Elemente bei der Übertragung nicht beliebig gewählt werden kann.

Problematisch bei der statistisch optimierten Codevergabe ist, dass ein und dasselbe Dokument, wenn es mit Schema Model Files codiert wird, die mit unterschiedlichen Beispielen aber dem selben Schema erzeugt wurden, in eine unterschiedliche binäre Repräsentation überführt wird. Für die Kompatibilität von Encoder und Decoder ist deshalb nicht nur dasselbe Schema, sondern auch derselbe Satz an Beispieldokumenten erforderlich. Für ein Verfahren, das in einem Standard eingesetzt werden soll ist diese Abhängigkeit von Beispieldokumenten unerwünscht.

3.3.3 ASN.1

Der Name ASN.1 steht für "Abstract Syntax Notation One". Es beschreibt den Datenaustausch zwischen zwei Kommunikationsanwendungen, indem es ein System zur Definition der Syntax dieser Botschaften zur Verfügung stellt. Diese Definitionen bestehen aus einfachen oder komplexen Datentypen. Eine ASN.1 Definition könnte beispielsweise wie folgt aussehen:

```
Age ::= INTEGER(0..7)
User::= SEQUENCE{
           Name    IA5String(SIZE(1..128)),
           Age     AGE,
           Address IA5String OPTIONAL,
           ...
        }
```

Ein erster ASN.1 Standard wurde bereits 1984 verabschiedet (eine neuere Version ist [ASN02]). ASN.1 wird mittlerweile in vielen Bereichen verwendet, z.B. in der

Telekommunikationsindustrie (Protokoll für GSM Mobiltelefone), für intelligente Transportsysteme, Logistik usw.. Neben der Definition der Syntax für den Datenaustausch gibt es auch noch Standards, welche die Codierung für die effiziente Übertragung der Botschaften beschreiben, nämlich die Basic Encoding Rules (BER) und die Packet Encoding Rules (PER).

Bei den BER [BER02] werden die Datenelemente durch ein Triple “Tag, Length, Value” repräsentiert. Das Tag identifiziert den Datentyp, die Länge den Umfang des Datenfeldes, und der Wert enthält die Nutzinformation. Bei BER werden alle Datenfelder mit Vielfachen ganzer Bytes codiert. Bei PER [PER02] dagegen wird das Tag nur codiert, wenn es eine Wahlmöglichkeit gibt, und die Länge und der Wert nur dann, wenn sie nicht auch durch die Syntaxdefinition festgelegt sind. Die Datenfelder sind bei PER nicht auf Bytegrenzen ausgerichtet. PER ist also auf eine besonders kompakte Darstellung optimiert. PER ist eine neuere Entwicklung die zum Einsatz kommt, wenn die Bandbreite oder die CPU-Kapazität eingeschränkt sind.

Seit neuerem gibt es Bestrebungen eine XML Erweiterung für ASN.1 zu entwickeln, beispielsweise um ASN.1 Werte in einem Browser anzuzeigen. Die XML Encoding Rules (XER) transformieren die ASN.1 Daten in eine XML Notation die unmittelbar lesbar ist, allerdings nicht so kompakt wie BER oder PER.

Bewertung

ASN.1 wäre als Datenformat für die Beschreibung der MPEG-7 Metadaten wohl auch in Frage gekommen. Das Grundsystem mit Syntaxdefinitionen für die Datentypen und Regeln für eine effiziente Codierung und Übertragung enthält wesentliche Bestandteile wie sie für ein MEPG-7 basiertes Informationssystem erforderlich sind, jedoch hat man sich bei MPEG für das modernere XML als Datenformat entschieden. Die XML Erweiterungen für ASN.1 waren zum Zeitpunkt der Arbeit von MPEG an einem Binärformat noch unbekannt. Da ASN.1 und die Codierschemata natürlich nicht für die speziellen Anwendungsfälle entworfen wurden, die man bei MPEG unterstützen wollte fehlen Eigenschaften wie die dynamische Codierung und Optimierungen für die Filterung.

Kapitel 4 – Das Binärformat für strukturierte Dokumente

In diesem Kapitel wird ausgehend von einer kurzen Darstellung der XML Sprache ein Binärformat für strukturierte Dokumente entwickelt, welches als Randbedingungen die Anforderungen aus den breitgefächerten Anwendungen der multimediabasierten Kommunikation aus Kapitel zwei berücksichtigt.

XML ist für das Binarformat deshalb von Bedeutung, weil es die Grundlage für die Darstellung der MPEG-7 Metadaten ist, und deshalb den Rahmen für die Details des Binärformats absteckt. Das Grundprinzip dieser binären Codierung lässt sich aber auf eine allgemeinere Klasse von Dokumenten anwenden.

4.1 Die XML Sprache

Für die Datenübertragung im World Wide Web ist die Hypertext Markup Language [HTML] das Standardformat. Es handelt sich dabei um eine Auszeichnungssprache (dieser Begriff leitet sich ursprünglich von der Auszeichnung textbasierter Daten für den Satz ab). Bei einer Auszeichnungssprache wird der Nutztext in Markierungen (sogenannte *Tags*) eingebettet. Diese Markierungen kennzeichnen als Paar in Form eines Anfangs- und Endtags eine ausgezeichnete Textpassage oder als Einzeltag z.B. einen Zeilenumbruch. Jedes Tag teilt durch Namen oder Attribut mit, welche Information die Markierung transportiert. HTML definiert nun einen festen Satz an Tags, die beispielsweise in einem Browser fest eingebaut sind und nicht verändert werden können. Allerdings wurden von konkurrierenden Unternehmen der Informationstechnologie laufend proprietäre Erweiterungen der HTML Syntax entwickelt, was auch die Gefahr von Inkompatibilitäten heraufbeschwor. Die Tags dienen bei HTML dazu das Dokument zu strukturieren und zu formatieren. Eine klare Trennung zwischen der durch den Inhalt festgelegten Struktur und der Formatierung bzw. dem Layout gibt es hier nicht. Außerdem sind Zusatzfunktionalitäten wie der Hyperlink-Mechanismus vorgesehen. Über den Inhalt eines Dokuments wird dagegen in der HTML Struktur wenig ausgesagt.

4.1.1 Entwicklung von XML

Der Hauptnachteil von HTML ist seine starre Syntax, die keine Anpassungen erlaubt. Für die Zukunft des WWW erschien die Sprache deshalb zu unflexibel. XML [XML00], [XML02] behebt diesen Nachteil. Schon im Namen e<u>X</u>tensible <u>M</u>arkup <u>L</u>anguage wird die Erweiterbarkeit als zentrale Eigenschaft betont. In XML können eigene Tags und Attribute definiert werden, wodurch die Daten eine individuelle Struktur erhalten, die mit Zusatzinformationen angereichert werden kann.

XML erlaubt die Darstellung beliebig tief verschachtelter Strukturen zur Nachbildung von komplexen Hierarchien aller Art. Diese Eigenschaft ist deshalb wertvoll, weil die Organisation in Hierarchien für den Menschen oft die einzige Möglichkeit ist komplexe Aufgaben zu bewältigen.

Entstanden ist XML 1996 aus der Standard Generalized Markup Language [SGML86]. Diese Sprache wurde schon zuvor von Industrie und staatlichen Institutionen beispielsweise für Dokumentationen verwendet, erschien aber für eine im WWW praktisch einsetzbare Sprache als zu umfangreich und sowohl bei der Anwendung als auch beim Erlernen als zu kompliziert. Beim Entwurf von XML wurde jedoch darauf geachtet, dass XML zu SGML kompatibel ist. Tatsächlich ist XML eine eingeschränkte Variante von SGML. Außerdem

sollte ein breites Spektrum an Anwendungen unterstützt werden können, und es sollte einfach sein Programme zu schreiben, die XML verarbeiten. Die knappe Darstellung von XML Markierungen war dagegen von untergeordneter Bedeutung. In der Praxis bedeutet dies, dass die Größe eines Dokument auf das (grob geschätzt) drei- bis fünffache ansteigt, wenn die Nutzinformation in eine XML Struktur überführt wird.

Bei XML wird klar zwischen der Struktur eines Dokuments, die im wesentlichen vom Inhalt bestimmt wird und dem Layout bzw. der Formatierung für die Darstellung unterschieden. Dies soll von sogenannten XSL/XSLT Dateien übernommen werden, die Anweisungen enthalten, welche Datenelemente wie darzustellen sind.

Die Entwicklung von XML wird vom World Wide Web Konsortium (W3C) vorangetrieben. Verschiedene Teilstandards behandeln weitere Aspekte der Anwendung von XML, beispielsweise XQL für das Content Management, XSLT für die Inhaltsgestaltung, XPOINTER, XPATH XLINK für Hypermedia Funktionalitäten, DOM und SAX für Verarbeitungsfunktionen. Beim W3C finden sich auch die Originalstandards sowie viele zusätzliche Informationen [W3C].

4.1.2 XML Überblick

XML hat im Vergleich zu HTML deshalb mehr Möglichkeiten, weil es eine Metasprache ist: HTML ist eine Strukturbeschreibung für Web-Dokumente, XML ist dagegen eine Sprache, in der sich beliebig viele Sprachen wie HTML definieren lassen.

Die Definition der Struktur von XML Dokumenten wird mit dem sogenannten XML Schema durchgeführt. In einem XML Schema können Datenelemente, Datentypen und Attribute spezifiziert werden. Der Name jedes dieser Konstrukte kann dabei frei gewählt werden. Nach der Strukturdefinition im XML Schema werden dann die eigentlichen XML Dokumente, welche die Nutzinformation enthalten aufgebaut, man spricht auch davon, dass das XML Dokument das XML Schema *instanziiert*. Mit Hilfe eines solchen Schemas ist es auch möglich Dokumente zu validieren, das heißt zu überprüfen, ob ein Dokument tatsächlich der Strukturdefinition des entsprechenden Schemas genügt.

Neben der Möglichkeit eine Strukturdefinition mit einem XML Schema durchzuführen kann man dies auch mit einer “Document Type Definition” DTD tun. Allerdings ist diese älter und weniger mächtig als XML Schema. Mit einer DTD kann nur die Struktur von Elementen eines Datensatzes festgelegt werden, Einschränkungen für den Inhalt sind aber nicht vorgesehen. Es fehlt die Möglichkeit Datentypen für XML zu definieren. Ein XML Schema hat außerdem die Eigenschaft selbst ein XML Dokument zu sein. Die zugrundeliegende Strukturdefinition wurde vom W3C als “XML Schema for XML Schemas” spezifiziert. Auf einen praktischen Nutzen dieser Tatsache wird in Kapitel acht eingegangen.

XML Schema besteht im wesentlichen aus zwei Teilen, genannt “Structures” und “Datatypes”. In „Structures“ werden die wesentlichen Konstrukte von XML Schemas spezifiziert, d.h. wie Datentypen definiert werden, wie Elemente oder Attribute deklariert werden, und wie der Inhalt eines Datentypen durch Gruppierungen (z.B. Auswahl oder Sequenz von Datenelementen) strukturiert wird. In „Datatypes“ werden einige elementare Datentypen wie Zeichenketten, Datum, Integer, oder Fließkommazahl definiert.

Die weiteren Details von XML, welche für die Erläuterung des Binärformats für strukturierte Dokumente wichtig sind, lassen sich am besten mit einem Beispiel veranschaulichen. Im folgenden werden anhand eines hypothetischen XML Schemas (das nicht etwa aus der MPEG-7 Spezifikation für die Metadaten stammt) und einem XML Dokument, das dieses Schema instanziiert die wesentlichen Bausteine von XML erklärt.

```
<xsd:schema targetNamespace=”http://www.UlrichN.de/KVV”
  xmlns=”http://www.UlrichnN.de/KVV” xmlns:std=“http://www.standardXMLTypes.org“
  xmlns:xsd=”http://www.w3.org/2001/XMLSchema”/>

  <xsd:element name=”Kartenvorverkauf” type=”KvvRevTyp”/>
  <xsd:element name=”Reservierung” type=”KvvRevTyp”/>

  <xsd:complexType name=”KvvRevTyp”/>
    <xsd:element name=”Veranstaltung” type=”VeranstaltungsTyp”/>
    <xsd:element name=”Karten” type=”KartenTyp” maxOccurs=”unbounded”/>
    <xsd:element name=”Kunde” type=”KundenTyp”/>
  </xsd:complexType>

  <xsd:complexType name=”KartenTyp”>
    <xsd:sequence>
      <xsd:element name=”Kategorie” type=”std:KategorieType”/>
      <xsd:element name=”Preis” type=”xsd:decimal”/>
      <xsd:element name=”Anzahl” type=”xsd:decimal”/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name=”VeranstaltungsTyp”/>
    <xsd:sequence>
      <xsd:element name=”Titel” type=”xsd:string”/>
      <xsd:element name=”Ort” type=”xsd:string”/>
      <xsd:element name=”Datum” type=”std:DatumType”/>
    </xsd:sequence>
    <xsd:attribute name=”Art” type=”xsd:string”/>
  </xsd:complexType>

  <xsd:complexType name=”KundenTyp”>
    <xsd:sequence>
      <element name=”Vorname” type=”xsd:string”/>
      <element name=”Nachname” type=”xsd:string”/>
      <element name=”Zahlungsart” type=”ZahlungsartTyp”/>
      <element name=”Rechnungsadresse” type=”std:AdressenTyp” minOccurs=”0”/>
    </xsd:sequence>
    <attribute name=“Stammkunde“ type=“xsd:boolean“/>
  </xsd:complexType>

  <xsd:complexType name=ZahlungsartTyp” abstract=”true”/>

  <xsd:complexType name=”BankeinzugTyp”>
    <xsd:extension base=”ZahlungsartTyp”>
      <xsd:element name=Bankeinzug” type=”Bankverbindung”/>
    </xsd:extension>
  </xsd:complexType>

  <xsd:complexType name=”KreditkartenTyp”>
    <xsd:extension base=”ZahlungsartTyp”>
      <xsd:element name=Kreditkarte”>
        <xsd:complexType>
          <attribute name=”Kreditkartennummer” type=”xsd:decimal”/>
        </xsd:complexType>
      </xsd:element>
    </xsd:extension>
  </xsd:complexType>

  <xsd:complexType name=”Bankverbindung”>
    <attribute name=”KtoNr.” type=” xsd:decimal”/>
    <attribute name=”BLZ” type=”d xsd:decimal”/>
  </xsd:complexType>

</xsd:schema>
```

Abb. 4.1: Syntaxdefinitionen eines XML Schemas

Das in Abb. 4.1 spezifizierte XML Schema definiert die Syntax eines Formulars für den Kartenvorverkauf. Die oberste Hierarchiestufe dieser Definition ist das *schema* - Element, in dessen Starttag der Namensraum für dieses Schema angegeben ist: “targetNamespace=...”. Der Namensraum bildet praktisch den Definitionsbereich in dem das Schema gültig ist. Im Inhalt des *schema* – Elements, der durch das Starttag <xsd:schema ...> und das Endtag </xsd:schema> abgegrenzt ist, werden zwei globale Elemente “Kartenvorverkauf” und “Reservierung” deklariert, sowie komplexe und einfache Datentypen definiert.

Diese Datentypdefinitionen enthalten Element- und Attributdeklarationen, sowie Gruppen, mit denen der Inhalt strukturiert werden kann. Möglich sind hier sequence-Gruppen, die festlegen, dass der Inhalt in der angegebenen Reihefolge im XML Dokument instanziiert werden muss, choice-Gruppen, welche die Auswahl eines Element aus seinem Inhalt, oder all-Gruppen, welche eine Instanziierung der Elemente des Inhalts in beliebiger Reihenfolge erlauben. Durch die Attribute “maxOccurs” und “minOccurs” wird die Häufigkeit, mit der das betreffende Element oder die betreffende Gruppe auftreten kann festgelegt. Wenn “minOccurs” den Wert “0” hat ist das Element optional.

Eine Elementdeklaration enthält im wesentlichen den Elementnamen, der auch im XML Dokument erscheint, und einen Verweis auf den Datentyp des Elements. Globale Elemente sind auf der obersten Hierarchiestufe des Schemas deklariert, lokale Elemente innerhalb von complexType- Definitionen. Der Datentyp legt dann wiederum den Inhalt des betreffenden Elementes fest. Alternativ besteht auch die Möglichkeit globale Elemente zu referenzieren. In diesem Fall ist der Name und der Typ durch das referenzierte globale Element festgelegt.

Bei den Typdefinitionen wird zwischen einfachen Typen “simpleType” und komplexen Typen “complexType” unterschieden. Einfache Typen können keinen keine weiteren Elemente enthalten. Der mögliche Inhalt eines einfachen Typen bildet die Nutzdaten des XML Dokuments. Ein komplexer Datentyp erlaubt dagegen die Deklaration weiterer Elemente, und dient dazu das XML Dokument zu strukturieren. Wenn eine Typdefinition ohne Namen direkt im Inhalt einer Elementdeklaration erfolgt, wie beim Kreditkartenelement des *Kreditkarten*-Typ so spricht man von einem anonymen Typen (“anonymous type”).

Global definierte Datentypen können im Gegensatz zu anonymen Typen vererbt werden, wie der *BankeinzugTyp*, der vom *ZahlungsartTyp* abgeleitet ist. Möglich ist eine Vererbung durch Erweiterung (“extension”) oder Einschränkung (“restriction”). Im ersten Fall wird der Inhalt des Basistypen sequenziell vor dem Inhalt des vererbten Typen angelegt. Im zweiten Fall kann die Häufigkeit von Elementen oder Gruppen eingeschränkt werden.

Im Zusammenhang mit der Vererbung von Datentypen bietet XML auch die Technik des Polymorphismus. Dabei besteht die Möglichkeit bei der Instanziierung eines Schemas durch ein XML Dokument den Datentyp eines Elements zu verändern (sogenannter Typecast). In Frage kommen dafür Elemente von Datentypen, von denen andere Typen abgeleitet sind. Durch das Attribut xsi:type=”...” kann dann festgelegt werden, dass das Element nicht vom Basistyp ist, sondern von einem abgeleiteten Typ. Ein Beispiel hierzu ist der *ZahlungsartTyp* von dem die beiden Typen *BankeinzugTyp* und *KreditkartenTyp* abgeleitet sind. In der Instanziierung dieses Beispiels findet ein Typecast des Elements *Zahlungsart* zum *BankeinzugTyp* statt. Näheres hierzu in Abschnitt 4.3.2.

Ein XML Dokument, welches das Schema aus Abbildung 4.1 instanziiert könnte beispielsweise wie in Abbildung 4.2 aufgebaut sein.

```
<Kartenvorverkauf>
   <Veranstaltung Art=”Oper”>
     <Titel>Zauberflöte</Titel>
     <Ort>Tollwood Festival</Ort>
     <Datum>1.12.2001</Datum>
   </Veranstaltung>
   <Karten>
     <Kategorie>A</Kategorie>
     <Preis>69</Preis>
     <Anzahl>3</Anzahl>
   </Karten>
   <Karten>
     <Kategorie>B</Kategorie>
     <Preis>59</Preis>
     <Anzahl>4</Anzahl>
   </Karten>
   <Kunde Stammkunde=“true“>
     <Vorname>Ulrich</Vorname>
     <Nachname>Niedermeier</Nachname>
     <Zahlungsart xsi:Type=”BankeinzugTyp”>
       <Bankeinzug KtoNr.=”12345678” BLZ=”70150101”/>
     </Zahlungsart>
   </Kunde>
</Kartenvorverkauf>
```

Abb. 4.2: XML Dokument mit Syntax nach Abb. 4.1

Am Aufbau des Dokuments wird die hierarchische Struktur deutlich, wie sie vom Schema vorgegeben ist. In der textuellen Repräsentation wird diese durch die Verschachtelung von Start- und Endtags der Elemente realisiert. Diese Struktur lässt sich anschaulich durch einen Datenbaum darstellen, in dem jeder Koten ein Element des Dokuments repräsentiert. Die Wurzel dieses Baums ist das global definierte Element *Kartenvorverkauf*. Hier findet bereits eine Auswahl statt: alternativ könnte auch das andere globale Element *Reservierung* instanziiert werden. Die von der Wurzel abgehenden Zweige sind Elemente die in der Typdefinition des Wurzelelements angelegt sind. Jede Elementdeklaration enthält die Information, von welchem Typ das Element ist. Die Definition dieses Typs legt wiederum die möglichen Kinder der nächsten Hierarchiestufe fest. Die Blattknoten des Baumes, von denen keine weiteren Zweige abgehen sind Elemente von einfachem Datentyp oder Attribute.

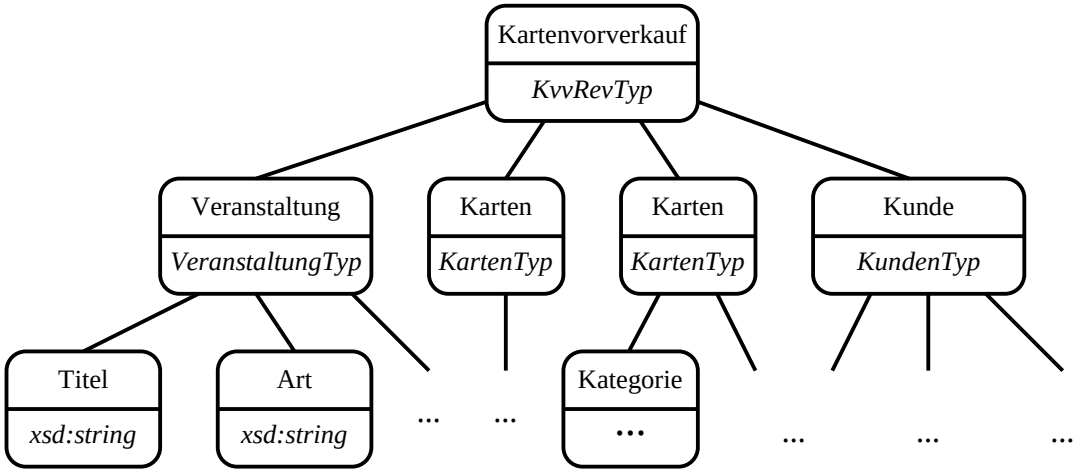


Abb. 4.3: Baumstruktur eines XML Dokuments

4.2 Ein neues Binärformat für XML

4.2.1 Das Grundprinzip des Algorithmus

Bei der Betrachtung des Beispiels aus Abbildung 4.2 fällt auf, dass XML Dokumente eine sehr aufwändige Repräsentation der Dokumentstruktur erfordern. Ein Grund dafür ist, dass XML so entworfen wurde, dass es sowohl von Menschen lesbar, als auch maschinell zu verarbeiten ist. Für viele Anwendungen ist eine effizientere Darstellung der Strukturinformation gefordert.

In diesem Kapitel wird ein Verfahren entwickelt, das eine effiziente Codierung der Struktur von XML Dokumenten erlaubt und darüber hinaus eine Reihe von speziellen Funktionalitäten unterstützt, welche im Zusammenhang der in Kapitel zwei identifizierten Anforderungen interessant sind. Der Algorithmus wird im folgenden “BiM Algorithmus” genannt, nach der Arbeitsgruppe bei MPEG in der er entwickelt wurde. Das Grundprinzip dieses Verfahrens ist es das XML Schema zu verwenden, um aus ihm die Codeworte des Binärformats abzuleiten, in das ein textuelles XML Dokument bei der Codierung überführt wird. Das Vorwissen über die Syntax des XML Dokuments, welches durch das Schema zur Verfügung steht wird benutzt, um eine hohe Kompression der Struktur des XML Dokuments zu erzielen. Die Nutzdaten werden zunächst nicht komprimiert.

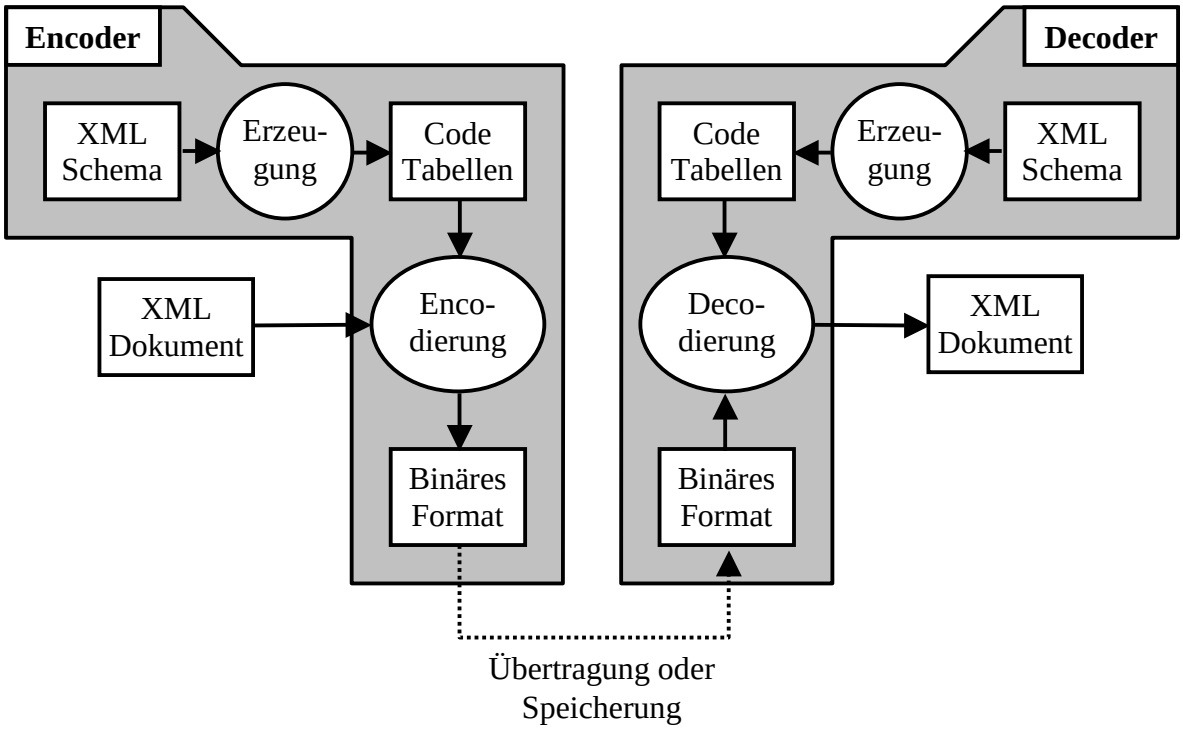


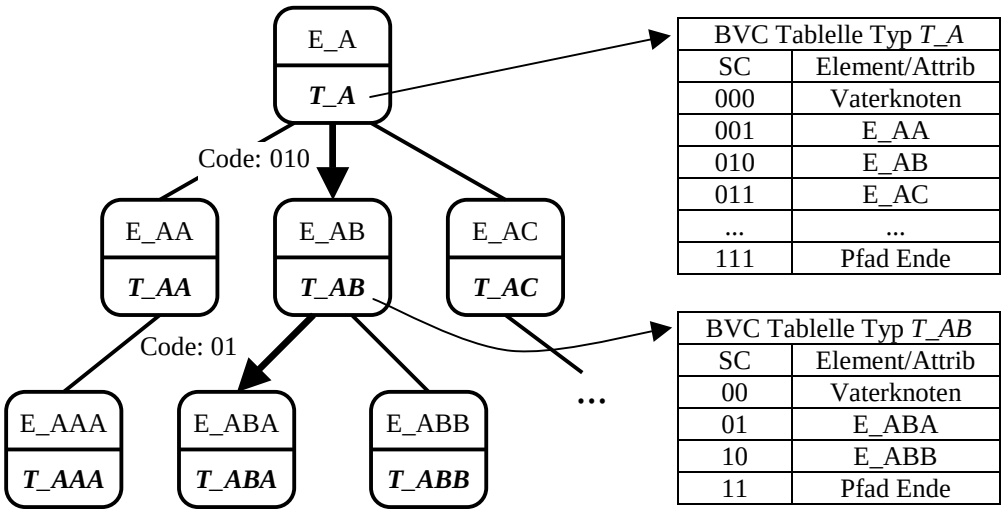
Abb. 4.4: Grundprinzip des Binärformats

Abbildung 4.4 verdeutlicht das Prinzip. Die Darstellung legt bereits nahe, dass dieses Verfahren nur funktioniert, wenn Encoder und Decoder die gleichen Codetabellen verwenden, weshalb das XML Schema, aus dem sie erzeugt werden, sowohl am Encoder als auch am Decoder bekannt sein muss. Dies kann erfüllt werden, indem entweder das Schema (welches z.B. im Falle des MPEG-7 Schemas standardisiert ist) am En- und Decoder fest eingebaut wird, oder weil das Schema vor dem eigentlichen Dokument zum Decoder übermittelt wird, oder weil der Decoder sich das Schema über eine Datenverbindung beschafft.

Das Binärformat besteht im Grunde aus zwei Teilen, welche dasselbe Grundprinzip verwenden, aber unterschiedliche Eigenschaften bezüglich Kompression und Flexibilität aufweisen. Einer der beiden Teile, nämlich die Pfadcodierung, wurde im Rahmen dieser Arbeit wesentlich mitentwickelt [MPEG00b]. Der andere Teil, die Payloadcodierung, wurde von einer französischen Gruppe bei MPEG vorgeschlagen [MPEG00c]. Beide Teile wurden in einer Arbeitsgruppe bei MPEG kombiniert, um die spezifischen Vorteile jedes dieser Ansätze zu verschmelzen.

4.3 Die Pfadcodierung

Die Aufgabenstellung, die es für eine Repräsentation der Dokumentenstruktur zu lösen gilt ist es den hierarchischen Aufbau eines XML Dokuments, wie er in Form des Datenbaums in Abbildung 4.3 symbolisiert wird, nachzubilden. Jedem Knoten dieses Baumes, der ein Element oder Attribut der XML Beschreibung repräsentiert, ist durch das XML Schema ein Datentyp zugeordnet. Die Definition dieses Datentyps gibt vor, welche weiteren Elemente oder Attribute möglich sind. Man kann nun jeder dieser begrenzen Anzahl an Möglichkeiten ein Codewort zuweisen, um einen Schritt in dem Datenbaum zu spezifizieren. Diese Codeworte werden im folgenden Strukturcode genannt. Der gesamte Satz an Codeworten ergibt eine für jeden Datentyp des Schemas spezifische Codetabelle, die Baumverzweigungscodetabelle, *BVC-Tabelle* (im Englischen *Tree Branch Code*, „TBC“). Durch eine Verkettung der Codeworte kann immer das nächste Kind eines Knotens ausgewählt werden, welches in dem XML Dokument instanziiert wird. Der Datentyp dieses Kindes gibt einen neuen Satz an Codeworten, mit denen der folgende Schritt spezifiziert wird, usf. Auf diese Weise wird ein Pfad in dem Datenbaum durchlaufen. Wenn ein Blattknoten erreicht wird, kann es notwendig sein sich in der Baumstruktur wieder zurück in Richtung Wurzel zu bewegen, um den nächsten Zweig einzusetzen (wenn nicht ein neuer Pfad von der Wurzel aufgebaut wird). Deshalb muss in die Tabellen noch ein zusätzliches Codewort aufgenommen werden, das nicht aus der Typdefinition abgeleitet ist, um einen Schritt in Richtung des Vaters eines Knotens zu spezifizieren. Um das Ende eines Pfades anzuzeigen gibt es einen weiteren Code, der immer der letzte Code einer Tabelle ist, dieser Code wird im Englischen *Termination Code* genannt.



BVC Tablelle Typ *T_A*	
SC	Element/Attrib
000	Vaterknoten
001	E_AA
010	E_AB
011	E_AC
...	...
111	Pfad Ende

BVC Tablelle Typ *T_AB*	
SC	Element/Attrib
00	Vaterknoten
01	E_ABA
10	E_ABB
11	Pfad Ende

Abb. 4.5: Identifikation im Datenbaum mittels Codetabellen

Die Gesamtzahl der Elemente und Attribute, die in einem Typ deklariert sind bestimmt die Länge der Tabellen und damit auch die Länge des Strukturcodes. Wenn die Zahl der Kinder keine Potenz von zwei ist bleiben Einträge in den Tabellen frei.

Mit diesem Ansatz ist es möglich den Datenbaum des XML Dokuments aufzubauen, indem zuerst in die Tiefe gegangen wird, und dann sukzessive von links nach rechts die einzelnen Zweige eingesetzt werden. Wenn ein Element mehrfach auftritt, wie das Element *Karten* in Abbildung 4.2, so könnte ausgehend von der ersten Instanziierung zum Vaterknoten (im Beispiel vom Typ *KvvRevTyp*) gegangen, und wiederum dasselbe Kindelement adressiert werden. Allerdings kann mit diesem Aufbau die Adressierungsreihenfolge nicht wahlfrei festgelegt werden (beispielsweise kann man in Abbildung. 4.2 nicht die Karten der Kategorie B vor denen der Kategorie A adressieren), da bei Elementen, die mehr als einmal auftreten können, nur die jeweils nächste Instanziierung im XML Dokument spezifiziert werden kann. Ein weiterer Code muss für solche Elemente eingeführt werden, um ihre Übertragungsreihenfolge frei wählen zu können. Dieser Code wird im folgenden *Positionscode* genannt. Zwar erhöht er das Datenvolumen für einen Pfad (etwa um 20% - 60%, siehe [Heu03]), und reduziert damit die Kompression, jedoch ist die zusätzliche Funktionalität im Kontext des gesamten Codieralgorithmus sinnvoll. Der genaue Aufbau der Positionscodes wird in 4.3.3 näher erläutert.

4.3.1 Aufbau der Codetabellen

Die Reihenfolge der Zuweisung der Codeworte zu den einzelnen Elementen und Attributen muss genau festgelegt werden, da sonst die Kompatibilität verschiedener Codecs oder mit verschiedenen aber äquivalenten Schemas erzeugter Codetabellen verloren geht.

In der vollständigen Codetabelle eines Typs müssen auch alle Elemente und Attribute der Basistypen enthalten sein. Die Vererbung von Typen kann auch mehrstufig sein: der Basistyp eines Typs kann seinerseits von einem anderen Typen abgeleitet sein. Die Elemente und Attribute der Basistypen werden vor dem Inhalt des Typs in die Codetabelle einsortiert. Die Elemente eines Typs werden vor den Attributen angelegt.

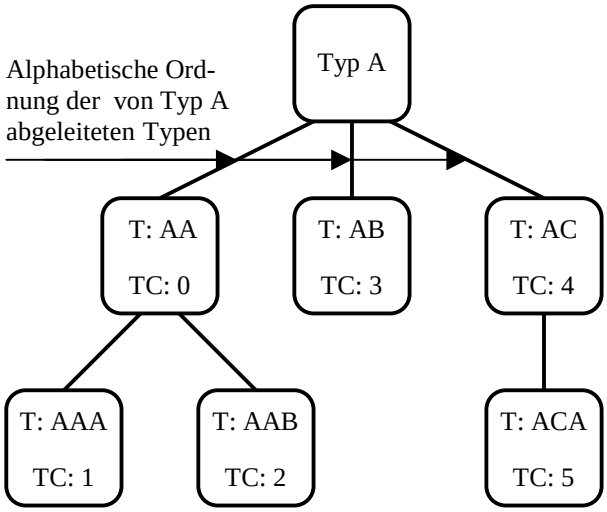
Wenn Elemente oder Gruppen in einer choice- oder all- Gruppe deklariert werden, so muss für diese Gruppe eine verbindliche Ordnung festgelegt werden. Datentypen, die sich nur bezüglich der Anordnung der Elemente innerhalb dieser Gruppe unterscheiden sind funktional äquivalent. Wenn aber die Codes für die Elemente z.B. in der Reihenfolge der Deklaration in der Gruppe vergeben würden, so wären die damit erzeugten Codetabellen nicht kompatibel. Funktional äquivalente Schemas würden zu unterschiedlichen Codes führen. Deshalb werden die Elemente alphabetisch bezüglich ihres vollen Namens (also inkl. Namensraum) geordnet. Der Name von evtl. enthaltenen sequence-Gruppen wird dabei durch seine “Signatur” festgelegt (s. Anhang A2). Die Reihenfolge der Elemente innerhalb von sequence–Gruppen bleibt unverändert.

4.3.2 Der Typecode

Wie bereits in Abschnitt 4.1.2 erwähnt besteht in XML die Möglichkeit den Typ eines Elements durch einen Typecast zu verändern. Da der Typ eines Elements für die Codierung des nächsten Schritts entscheidend ist, muss der veränderte Typ im Pfad signalisiert werden. Die Anzahl an möglichen neuen Typen für das Element ist begrenzt: es sind alle Typen, die direkt oder indirekt von dem ursprünglichen Typ des Elements abgeleitet sind. Jedem dieser Möglichkeiten wird nun eine Code, im folgenden *Typecode* genannt zugewiesen. Bei allen Elementen deren Typ durch einen Typecast geändert werden kann wird mit einem Flag

signalisiert, ob der Typecast auftritt. Falls ja (Flag=1) wird anschließend der *Typecode* übertragen.

Da mehrere Typen von einem Typ abgeleitet sein können, jeder Typ aber nur von genau einem Basistyp vererbt sein kann ergibt sich wiederum eine Baumstruktur, der Vererbungsbaum.



T: Name des Typen
TC: Typecodes, Vergabe depth-first

Abb. 4.6: Vererbungsbaum und Typecodes

Um den Vererbungsbaums eindeutig aufzubauen werden die Typen, welche den selben Basistyp haben beim Einsetzen in den Baum alphabetisch bzgl. ihres Namens geordnet. Die Vergabe der Typecodes erfolgt dann der Tiefe nach zuerst, beginnend bei Null. Die Länge eines Typecodes ergibt sich aus der Mächtigkeit des unter dem Typen aufgebauten Baums zu:

$$L_{TC} = \lceil \log_2(Zahl\_der\_abgeleiteten\_Typen) \rceil$$

Die Vergabe der Typcodes nach dem Prinzip “Tiefe zuerst” (im Gegensatz zu “Breite zuerst”) hat einen entscheidenden Vorteil: wenn für ein Schema nur ein Vererbungsbaum angelegt wird, dessen Wurzel der Urtyp “anyType” ist, und von dem alle Typen implizit abgeleitet sind (die Kinder dieses Urtyps sind alle Typen die nicht explizit mittels “base” Attributs von einem anderen Typ abgeleitet sind), so vereinfacht sich die Berechnung der Typecodes: aus dem Typecode eines Typs “X” und eines Typen „A“ bezüglich des Urtyps “U”, (TC_{X_U} und TC_{A_U}) lässt sich der Typcode bezüglich jedes andern Typs “A” (TC_{X_U}) berechnen zu:

$$TC_{X\_A} = TC_{X\_U} - TC_{A\_U} - 1$$

4.3.3 Der Positionscode

Wenn man die Reihenfolge, in der man die Elemente mittels Pfaden überträgt frei bestimmen will, so muss man bei Elementen die mehrfach instanziiert werden können angeben, welche Instanz adressiert wird. Ein Element kann mehrfach instanziiert werden, wenn es selbst das Attribute "maxOccurs" mit einem Wert größer als eins aufweist, oder wenn es in einer Gruppe enthalten ist, die diese Eigenschaft besitzt.

Entsprechend dieser unterschiedlichen Ursachen gibt es zwei Modi, in denen Positionscodes angegeben werden können. Welcher Modus anzuwenden ist wird durch die complexType Definition festgelegt: wenn das Element selbst mehrfach auftreten kann, und in dem komplexen Typen keine Gruppe spezifiziert ist, die ein Attribut *maxOccurs* größer als eins hat, so wird ein *lokaler Positionscode* verwendet. Dieser spezifiziert die Position einer Instanziierung des Elements unter seinen Geschwistern desselben Strukturcodes. Die Länge des Positionscodes ist festgelegt durch den Logarithmus zur Basis 2 des Wertes von *maxOccurs*, wenn *maxOccurs* < 2^{16}, ansonsten wird eine variable Repräsentation für ganze Zahlen verwendet, deren Länge dem Wert der Zahl angepasst ist, vgl. Anhang A1 (ansonsten würden auch kleine Zahlen, wie sie in der Praxis häufig auftreten, mit einer unverhältnismäßig hohen Zahl an Bits codiert).

Falls die complexType Definition Gruppen mit dem Attribut *maxOccurs* > 1 hat, so wird ein *globaler Positionscode* verwendet. In diesem Falle wird ermittelt, wie viele Elemente der Typ maximal haben kann (Rechenvorschrift im Anhang A3). Die Position eines Elements wird dann unter seinen Geschwistern gleich welchen Typs vergeben.

Diese gemischte Art der Positionscodierung ist ein Kompromiss zwischen kompakter Darstellung der Positionsinformation und Einfachheit. Die lokale Positionscodierung alleine könnte nicht alle Fälle die auftreten können behandeln. Wenn nur eine globale Positionscodierung verwendet wird, so würde dies beispielsweise dazu führen, dass ein Element mit unbegrenzter oberer Schranke die Positionscodes aller anderen Elemente verlängert.

4.3.4 Der Ersetzungscode

In XML gibt es auch die Möglichkeit für globale Elemente eine Ersetzungsgruppe „*Substitution Group*" zu spezifizieren. Wenn ein globales Element irgendwo in einer Elementdeklaration referenziert wird, und dieses globale Element ist das Headelement einer *Substitution Group*, so kann das referenzierte Element durch ein anderes Element der *Substitution Group* ersetzt werden. Da eine *Substitution Group* eine durch das Schema wohldefinierte Anzahl an Mitgliedern hat kann man wieder jedes dieser Mitglieder durch einen einfachen Code identifizieren. Im BiM Algorithmus werden die *Substitution Groups* folgendermaßen behandelt: wenn ein Element das Headelement einer *Substitution Group* ist, so wird durch ein Flag signalisiert, ob eine Ersetzung auftritt. In diesem Falle folgt der Ersetzungscode, der ein Element in der alphabetisch geordneten Liste aller Elemente in der Ersetzungsgruppe adressiert. Im MPEG-7 Schema treten allerdings keine *Substitution Groups* auf. Trotzdem ist es wichtig diese Option zu unterstützen, um die allgemeine Anwendbarkeit des Verfahrens zu gewährleisten.

4.3.5 Aufbau der Pfade

Durch Verkettung der Baumverzweigungcodes lassen sich Pfade im Datenbaum spezifizieren. Für die Übertragung eines Elements gibt es dabei im Prinzip die Möglichkeit, bei der Adressierung eines Elements von der Wurzel des Baums auszugehen "absoluter Pfad", oder von dem Knoten der als letztes adressiert wurde "relativer Pfad". Ein Pfad ist aufgebaut aus Pfadfragmenten, die einen sogenannten Kontext-Koten spezifizieren, einem Schluss-Code *"Termination-Code"*, der das Ende eines Pfades anzeigt, und genau einem weiteren Pfadfragment, das den *Operand-Knoten* spezifiziert.

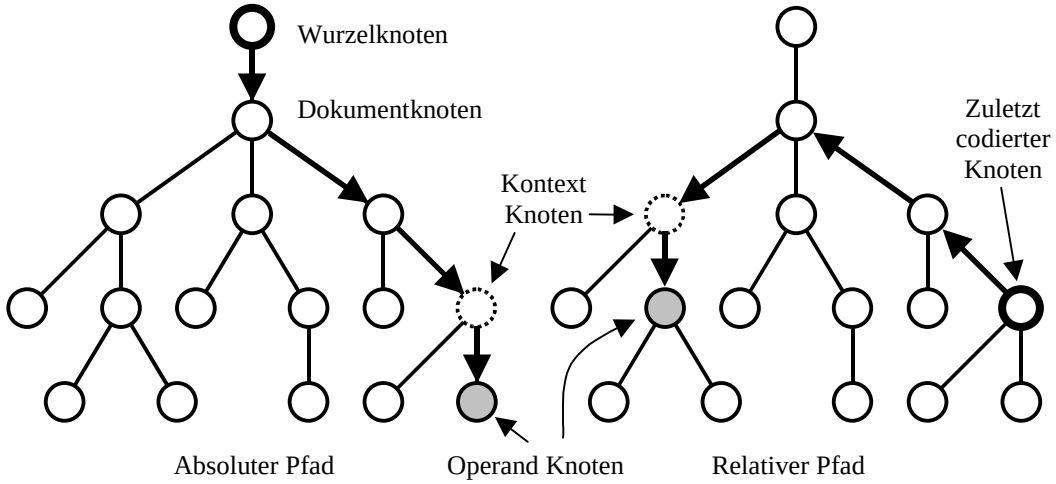


Abb. 4.7: Absoluter und Relativer Pfad, Kontext- und Operand-Knoten

Die Aufteilung in *Kontext-Knoten* und *Operand-Knoten* ermöglicht eine weitere Optimierung der Codierung: da bei der Verarbeitung eines Pfades vor dem Schluss-Code sichergestellt ist, dass noch mindestens ein weiteres Pfadfragment folgt, kann es sich bei den adressierten Knoten nicht um Elemente von einfachem Typ oder Attribute handeln (diese haben keine weiteren Kinder). Deshalb muss in den Codetabellen für solche Elemente oder Attribute keine Adresse reserviert werden. Vom Beginn eines Pfades bis zum *Termination Code* werden die Knoten mit diesen reduzierten Codetabellen adressiert, die Elemente von komplexem Typ und den *Termination Code* enthalten. Nach dem *Termination Code* werden die Operand Tabellen verwendet, die sowohl Elemente von komplexen Typ enthalten, als auch Elemente von einfachem Typ und Attribute. Dies verkürzt unter Umständen die Tabellen und damit den Strukturcode. Die Pfade werden dadurch kompakter.

Context BVC Tabelle für KundenTyp		
SBC	**Typecode**	**Element/Attribut**
00	Nein	Vaterknoten
01	Nein	E: Rechnungsadresse
10	Ja	E: Zahlungsart
11	Nein	Termination

Tab. 4.1: Context BVC-Tabelle

Operand BVC Tabelle für KundenTyp		
SBC	**Typecode**	**Element/Attribut**
000	Nein	Erweiterung
001	Nein	E: Nachname
010	Nein	E: Rechnungsadresse
011	Nein	E: Vorname
100	Ja	E: Zahlungsart
101	Nein	A: Stammkunde
110-111	--	-- Leer --

Tab. 4.2: Operand BVC-Tabelle

Tabellen 4.1und 4.2 zeigen ein Beispiel für den Aufbau der Codetabellen anhand des Typs *KundenTyp* aus Abb. 4.1, und geben einen Überblick der relevanten Eigenschaften: Aufteilung in *Context*- und *Operand-Tabelle*, alphabetische Ordnung der Elemente und Attribute, *Termination Code* und Referenz zum Vaterknoten. Bisher noch unerwähnt ist der Erweiterungscode in der Operand Tabelle, der als Reservecodewort für anwenderspezifische Anforderungen gedacht ist (beispielsweise könnte man Kommentare, die an beliebigen Stellen eingefügt sind, und denen keine Syntaxdefinition zugrunde liegt damit übertragen, wenn der Encoder und Decoder gemeinsam dies unterstützen).

Abbildung 4.8 zeigt den vollständigen Aufbau des Pfades der Länge n. Er beginnt mit n-1 Pfadfragmenten, die jeweils aus einem Strukturcode, und wenn erforderlich aus einem Ersetzungscode und einem Typecode bestehen. Die Strukturcodes sind dabei aus den *Context BVC-Tabellen* entnommen. Der *Termination Code* trennt das letzt Pfadframgent ab. Dessen Strukturcode ist aus der *Operand BVC-Tabelle* entnommen. Am Ende des Pfades sind die Positionscodes aller Pfadfragmente angeordnet, für die ein solcher erforderlich ist.

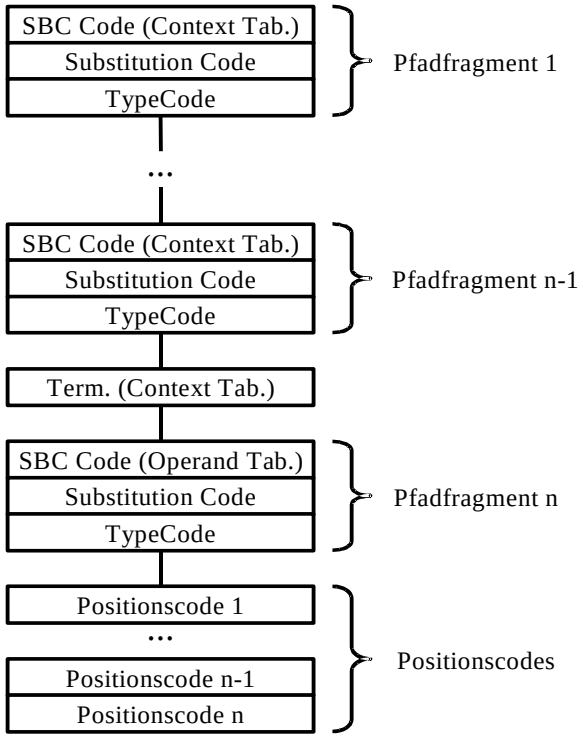


Abb. 4.8: Aufbau eines Pfades

4.4 Die Payloadcodierung

Während die Pfadcodierung für die Identifikation von einzelner Knoten im Dokumentenbaum entworfen wurde, dient die Payloadcodierung zur Codierung ganzer Baumfragmente einschließlich der Nutzdaten. Auch die Payloadcodierung beruht wie die Pfadcodierung auf dem Prinzip die Codeworte aus dem XML Schema abzuleiten. Jedoch ist sie aufgrund der unterschiedlichen Zielsetzung auf andere Eigenschaften hin optimiert. Im Vergleich zur Pfadcodierung erzielt die Payloadcodierung eine höhere Kompression der Dokumentstruktur, allerdings muss der Nachteil einer geringeren Flexibilität in Kauf genommen werden: die Elemente eines Typs können bzgl. der Reihenfolge nicht mehr wahlfrei adressiert werden, diese ist durch das Schema festgelegt.

Da das Kapitel 5 in der informationstheoretischen Analyse syntaxbasierter Codierverfahren, das Kapitel 6 über eine optimierte Implementierung des BiM Algorithmus sowie Kapitel 7 über die Komplexitätsanalyse des Algorithmus auch die Payloadcodierung umfasst, wird das Prinzip hier ausführlicher beschrieben. Für weitere Details sei auf [MPEG01a] sowie [MPEG00c] verwiesen.

4.4.1 Das Automatenmodell

Anders als bei der Pfadcodierung, bei der die Schemainformation in die Form von Codetabellen gebracht wurde, basiert die Payloadcodierung auf einem Zustandsautomaten-modell: die Syntaxkonstrukte von XML entsprechen vordefinierten und evtl. konfigurierbaren Mustern von miteinander durch gerichtete Übergänge verbundenen Zuständen. diese Muster werden entsprechend der Typdefinitionen verschaltet und ergeben so für jeden komplexen Typen einen Automaten, der den Codier- oder Decodiervorgang kontrolliert: die Modell-vorstellung ist, dass eine Marke (engl.: “token”) den augenblicklich aktiven Zustand des Automaten markiert, und im Zuge der Codierung durch den Automaten wandert. Tritt in einem Zustand eine Verzweigung in mehrere mögliche Folgezustände auf, so wird der Bitstrom gelesen, um festzustellen, welcher Folgezustand ausgewählt wird.

Abb 4.9 zeigt ein Beispiel für die Umsetzung einer complexType Definition in einen Zustandsautomaten: der Einstiegspunkt für die Decodierung eines Typen ist der Startzustand. Im Typen TypeX ist das erste Konstrukt eine Auswahl aus drei Alternativen. Entsprechend gehen vom Startzustand der Auswahl drei Übergänge ab. Jedem Übergang ist ein Code zugeordnet, der den ausgewählten Zweig identifiziert. Im ersten Zweig, der mit dem Codewort “00” signalisiert wird, tritt ein Element “a” vom Typ “Ta” auf. Wenn dieser Zweig ausgewählt wird, so wird im Typzustand “Ta” ein anderer Automat aufgerufen der aus der Typdefinition von “Ta” abgeleitet wurde. Ist die Decodierung von Ta beendet, so wird wieder mit der Decodierung von “TypeX” fortgefahren. Dort kann im Endzustand der Auswahl wieder in den Startzustand verzweigt werden, da die Auswahl unbegrenzt oft aufgerufen werden kann. Ist ein Element optional, wie das Element “e”, so werden die entsprechenden Zustände durch eine “Shunt-Transition” überbrückt. Man erkennt, dass überall, wo das Schema eine Wahlmöglichkeit lässt, der Decoder ein Codewort liest, um festzustellen, welche Alternative ausgewählt wird. Die Fälle in denen dies erforderlich ist sind Auswahlgruppen, Häufigkeiten (“minOccurs” oder “maxOccurs” ungleich 1), Polymorphismus und Ersetzungsgruppen. Die Länge dieses Codewortes richtet sich nach der Gesamtzahl der möglichen Alternativen. Entsprechend muss für die Codierung von Sequenzen von Elementen nichts signalisiert werden, wie im zweiten Zweig der Auswahl: ist dieser ausgewählt werden die Elemente “b” und “c” decodiert, ohne dass Information aus dem Bitstrom gelesen werden muss.

```
<complexType name="TypeX">
  <choice maxOccurs="unbounded">
    <element name="a" type="Ta"/>
    <sequence>
      <element name="b" type="Tb"/>
      <element name="c" type="Tc"/>
    </sequence>
    <sequence>
      <element name="d" type="Td"/>
      <element name="e" type="Te" minOccurs="0"/>
    </sequence>
  </choice>
</complexType>
```

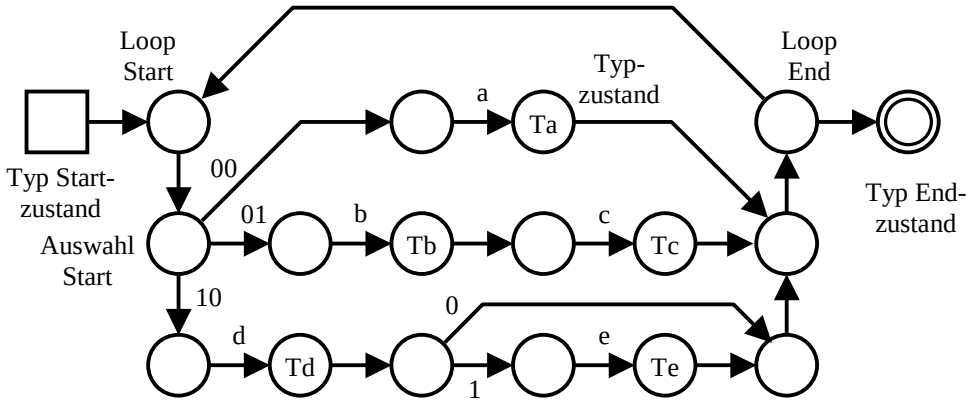


Abb. 4.9: XML Syntaxdefinition und daraus erzeugter Zustandsautomat

Beispiel für einen Bitstrom:
für den Fall, dass im Zustandsautomaten der unterste Zweig einmal ausgewählt und das Element “e” codiert wird ist der Bitstrom:

00001 10 <Codes für Typ “Td”> 1 <Codes für Typ “Te”>

Die ersten fünf Bits geben an, wie oft die Auswahlgruppe codiert wird (vgl. 4.4.3, Codierung von Häufigkeiten), die nächsten zwei Bit wählen den letzten Zweig der Auswahl aus, es folgt der Bitstrom für Typ “Td”, das nächste Bit signalisiert, dass Element “e” codiert ist, und es folgt der Bitstrom für Typ “Te”.

4.4.2 Erzeugung der Automaten

Die Erzeugung der Automaten aus dem XML Schema gliedert sich in vier Phasen:

1. Generieren des effektiven Inhalts eines Typs. Hier werden Vererbung von Typen realisiert (der Basistyp wird in einer Sequenz vor dem vererbten Typ angelegt), es werden Element- und Gruppenreferenzen aufgelöst.
2. Erzeugung eines Syntaxbaums aus der Definition des Komplexen Typen. Ein Syntaxbaum gibt die Struktur der Syntaxdefinition wieder. Er enthält Information über die Häufigkeiten und über die Art der Partikel. Die Syntaxbäume werden transformiert, um die Kompression zu verbessern.

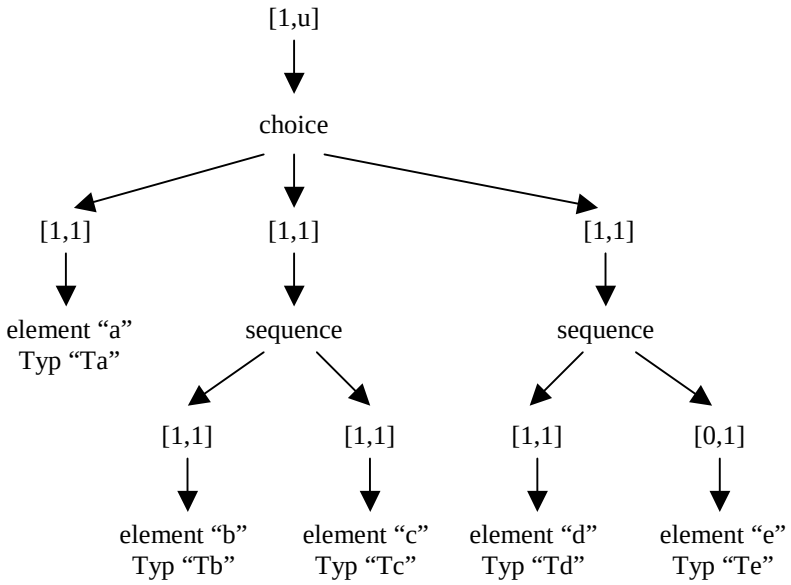


Abb. 4.10: Syntaxbaum des komplexen Typen “TypeX”

Für Transformationen der Syntaxbäume gibt es drei Möglichkeiten:

- Eine Gruppe (choice, sequence), die nur einen Partikel enthält. In diesem Fall wird diese Gruppe aufgelöst, und die Häufigkeit neu berechnet: das neue minimale Häufigkeit ist das Produkt aus den entsprechenden Zahlen des Partikels und der Gruppe, dasselbe gilt für die maximale Häufigkeit.
- Enthält eine Auswahl ein Partikel mit minimaler Häufigkeit Null, so bekommt es die minimale Häufigkeit 1, und die Auswahl die minimale Häufigkeit Null (der Vorteil ist, dass beim Codieren nicht zuerst ein Zeig ausgewählt werden muss, bevor spezifiziert wird, dass er keinen Inhalt hat).
- Enthält eine Auswahl eine andere Auswahl, die exakt einmal auftreten kann, so wird die innere Auswahl aufgelöst, und der Inhalt in der äußeren Auswahl eingruppiert.

3. Normalisierung der Syntaxbäume. Um eindeutige Codes vergeben zu können, muss festgelegt werden, welche Reihenfolge die einzelnen Zweige der Bäume erhalten. Dazu wird jedem Partikel eine Signatur zugeordnet, die aus den Namen der enthaltenen Elemente und Gruppen gebildet wird und die Partikel entsprechend dieser Signatur geordnet (s. Anhang A2).
4. Generierung der Zustandsautomaten. Hier werden aus den Syntaxbäumen die Automaten erzeugt, welche den Decodiervorgang steuern. Begonnen wird bei den Blättern des Syntaxbaums. Jeder Knoten des Syntaxbaums erzeugt ein Automatenfragment, indem es die in den Kindknoten erzeugten Fragmente verknüpft. Die Art der Verknüpfung hängt von der Art des Knotens im Syntaxbaum ab. Die folgende Übersicht erläutert, wie die einzelnen Knoten des Syntaxbaums in Automatenfragmente übersetzt werden.

Automatenfragment für eine Elementdeklaration

Ein Automatenfragment für eine Elementdeklaration Besteht aus einem Startzustand, einem Übergang, der den Elementnamen spezifiziert, und dem Endzustand. Dieser ist gleichzeitig ein Typzustand, der einen Automaten des entsprechenden Typs aufruft.

Automatenfragment für einen Häufigkeitsknoten

Fall 1: minOccurs=1 & maxOccurs=1
Der vom Kindknoten erzeugte Automat wird nicht verändert.

Fall 2: minOccurs=0, maxOccurs=1
Ein neuer Start und ein neuer Endzustand wird erzeugt, zwischen beiden wird ein Überbrückungsübergang angelegt. Ein Codeübergang wird zwischen dem neuen und dem alten Startzustand angelegt, ein einfacher Übergang wird zwischen dem alten und dem neuen Endzustand angelegt.

Fall 3: maxOccurs>1, minOccurs>0
Ein neuer Start- und Endzustand wird angelegt, ebenso ein Zwischenzustand. Ein Codeübergang wird zwischem dem neuen Startzustand und dem Zwischenzustand angelegt, ein anderer Übergang verbindet den Zwischenzustand mit dem alten Startzustand. Ebenso wird ein Übergang zwischen dem alten und dem neuen Endzustand installiert. Ein weiterer Übergang verbindet den alten Endzustand mit dem alten Startzustand, dadurch kann der Kindautomat des Occurrenceautomaten mehrfach durchlaufen werden.

Fall 4: maxOccurs>1, minOccurs=0
Aufbau wie in Fall 3, nur wird zusätzlich zwischen dem neuen Start- und Endzustand ein Überbrückungsübergang angelegt.

Automatenfragment für eine Auswahl

Ein neuer Start- und Endzustand wird angelegt. Dieser Startzustand wird mit einem Codeübergang mit jedem Startzustand der Kindautomaten, jeder Endzustand wird mit dem Endzustand der Auswahl durch einen einfachen Übergang verbunden.

Automatenfragment für eine Sequenz

Die Kindautomaten werden in der Reihenfolge, in der sie im Syntaxbaum angelegt sind miteinander verkettet. Der Endzustand eines Automaten wird mit dem Startzustand des folgenden Automaten verbunden. Der Startzustand der Sequenz ist der Startzustand des ersten Automaten, der Endzustand ist der Endzustand des letzten Automaten.

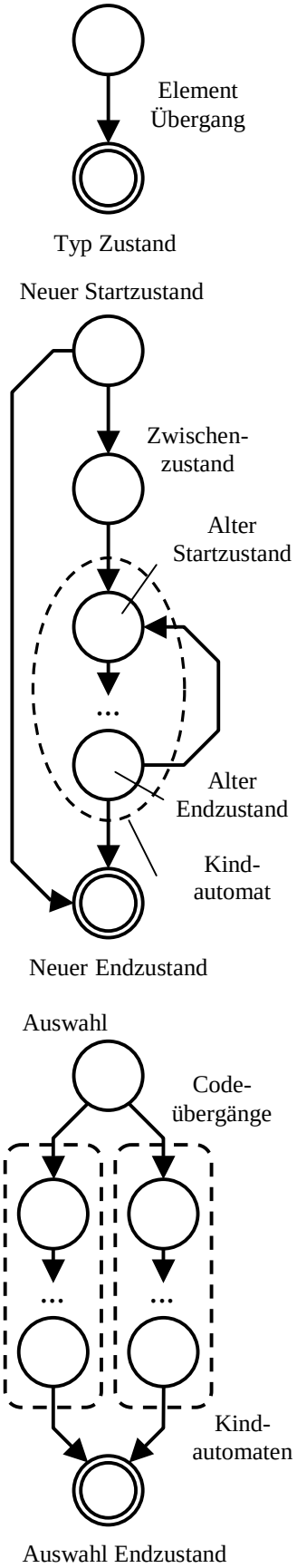


Abb. 4.11: Automatenfragmente

Behandlung von all-Gruppen

Sind Elemente in einer all-Gruppe deklariert, so müssen in einer Instantiierung alle diese Elemente auftreten, wobei die Reihenfolge der Instantiierung beliebig ist. Für eine all-Gruppe mit N Elementen gibt es also N! Möglichkeiten für diese Reihenfolge. In der Payloadcodierung wird die Signalisierung dieser Möglichkeiten dadurch gelöst, dass die all-Gruppe in einen Baum von Auswahlgruppen transformiert wird, bei dem in jeder Auswahlgruppe die Elemente vorhanden sind, welche noch nicht aufgetreten sind, und deshalb noch instantiiert werden können. Dazu ein kurzes Beispiel:

Eine all Gruppe mit drei Elementen “a”, “b” und “c” wird in folgende Struktur von Auswahlgruppen übersetzt:

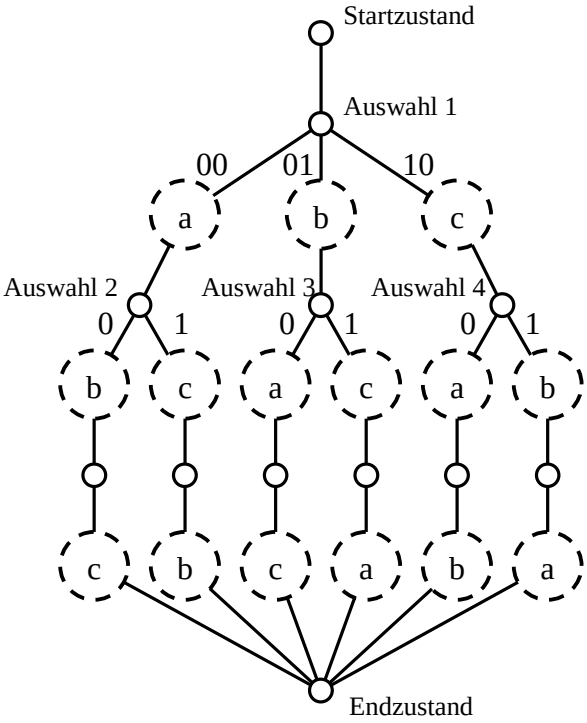


Abb. 4.12: Umsetzung einer all-Gruppe in Auswahlgruppen

Eine all-Gruppe mit N Zweigen wird übersetzt in eine Auswahlgruppe mit N Zweigen, in N Auswahlgruppen mit N-1 Zweigen, in N(N-1) Auswahlgruppen mit N-2 Zweigen usf.

Eine all Gruppe mit 5 Zweigen wird demnach bereits in 86 Auswahlgruppen transformiert (wenn man die Auswahlgruppen mit nur einer Auswahl, welche für das letzte Element formal erzeugt werden vernachlässigt).

Allgemein gilt für die Zahl der Auswahlgruppen:

$$A = 1 + N + N \cdot (N-1) + \ldots + N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot 4 \cdot 3$$
$$= N! \left[\frac{1}{N!} + \frac{1}{(N-1)!} + \ldots + \frac{1}{[N-(N-2)]!} \right]$$

Für große N ist ergibt sich näherungsweise:

$$A = N!(e-2);\ \text{e = Basis des natürlichen Logarithmus}$$

Die Komplexität steigt demnach mit N! Eine Möglichkeit eine all-Gruppe zu implementieren wird in Abschnitt 6.2.4 beschrieben.

Nachdem die Automaten erzeugt wurden, wird jedem Codeübergang ein binärer Code zugeordnet. Der Wert des Codes entspricht der Position des vom Knoten abgehenden Codeübergangs. Die Codeübergänge sind entsprechend ihrer Signatur geordnet. Die Länge des Codes ergibt sich aus der Gesamtzahl der von einem Knoten abgehenden Codeübergänge zu:

$$L_{Code} = \lceil \log 2(Anzahl\_der\_Codeübergänge) \rceil$$

4.4.3 Die Decodierung im Payloadmodus

Decodierung von Attributen

Für die Decodierung der Attribute eines Elements werden alle Attribute des Typs des Elements gesammelt (also auch die Attribute der Basistypen). Die Attribute werden alphabetisch geordnet, um Unabhängigkeit von der willkürlichen Reihenfolge in der textuellen Definition zu erreichen. Für diese Liste an Attributen wird ein Sequenzautomat erzeugt. Für jedes Attribut wird beim Decodieren ein Flag gelesen, falls es optional ist, sowie der Attributwert.

Decodierung von Häufigkeiten

Wenn ein Element mehrfach instanziiert werden kann (Attribut maxOccurs>1), so muss signalisiert werden, wie oft das Element im Dokument tatsächlich auftritt. Dafür sind mehrere Modi vorgesehen, die unterschiedliche Eigenschaften aufweisen.

Im einfachsten Falle kann jedes Auftreten im Bitstrom durch ein Flag zu signalisiert werden. Dies ist für eine geringe Zahl an Instanzen effizient. Wenn das Element jedoch sehr oft auftritt ist diese Lösung nicht mehr geeignet, da die Zahl der Bits der Zahl an Instanzen entspricht. Im Gegensatz dazu würde eine direkte Codierung dieser Zahl nur $\log_2$(Zahl der Instanzen) Bits benötigen, und wäre damit wesentlich effizienter. Eine solche Lösung hat aber den Nachteil, dass die Zahl der Instanzen gleich bei ersten Auftreten des Elements bekannt sein muss, was nicht immer vorausgesetzt werden kann. Deshalb ist es günstig für die Codierung von Häufigkeiten ein variables Format vorzusehen, das mehrere Möglichkeiten unterstützt.

Dazu wird in der Initialisierung des Decoders ein Datenfeld übertragen, das den Modus der Codierung von Häufigkeiten spezifiziert. Ein Modus ist die Häufigkeit des Auftretens des Elements vor der ersten Instanz zu codieren (dafür sind verschiedene Formate für ganze Zahlen definiert, siehe auch Anhang A1). Die anderen Modi organisieren die Instanzen in Gruppen verschiedner Größe. Dies hat den Vorteil, dass mit dem Schreiben des Bitstroms begonnen werden kann, sobald eine Gruppe gefüllt ist. Ein Flag zeigt an, ob noch weitere Gruppen kommen, oder ob die aktuelle Gruppe die letzte ist. Die Zahl der Instanzen in einer Gruppen, welche durch die Variable *Unit Size* in der Decoderinitialisierung festgelegt wird, variiert je nach Modus von 1 (d.h. jede einzelne Instanz wird durch ein Flag angezeigt) bis 128.

Decodierung von Inhalt/Simple Types

Der schemabasierte BiM Algorithmus lässt sich nur für die Codierung und Kompression der Struktur anwenden, für die Codierung von Inhalt sind eigene Lösungen zu suchen. Da der Datentyp eines Elements oder Attributs nach der Decodierung der Struktur, in die es eingebettet ist feststeht, kann für verschiedene Datentypen eine jeweils optimierte Codierung gewählt werden. Fließkommazahlen lassen sich beispielsweise effizient im IEEE 754 Format codieren. Für die Codierung von Zeichenketten hat sich der ZIP Algorithmus bewährt. Hier ist allerdings darauf zu achten, dass dieser die Redundanz innerhalb von Zeichenketten für die Kompression ausnutzt. Deshalb ist es geschickt, nicht den Inhalt jedes einzelnen Elements oder Attributs zu codieren, sondern diese in größere Container zusammenzufassen, und erst dann zu komprimieren. In der Strukturcodierung der Payload werden nur Referenzen in diesen Inhaltscontainer geliefert. Prinzipiell lassen sich auch für andere Datentypen, etwa Matrizen, Histogramme etc. optimierte Codecs entwerfen. Das MPEG Binärformat sieht eine Möglichkeit zur Spezifikation solcher optimierter Codecs vor (s. „7.2 Binary DecoderInit“ in [MPEG01a]).

Parserkomplexität

Bei der Codierung einer Beschreibung mit dem in diesem Abschnitt beschriebenen Verfahren gilt es eine Besonderheit zu beachten:

Es ist nicht immer trivial den richtigen Pfad in den Zustandsautomaten zu finden, welcher der Instanziierung eines Dokuments entspricht. Dazu ein Beispiel:

```
<complexType name=”ambigous”>
  <choice>
    <sequence>
      <element name=”a” type=”Ta”/>
      <element name=”b” type=”Tb”/>
      <element name=”c” type=”Tc”/>
    </sequence>
    <sequence>
      <element name=”a” type=”Ta”/>
      <element name=”b” type=”Tb”/>
      <element name=”d” type=”Td”/>
    </sequence>
  </choice>
</complexType>
```

Abb. 4.13: Mehrdeutige Syntaxdefinition

Bei der Instanziierung eines Dokuments mit den Elementen <a><b><d> wäre erst bei Element <d> klar, dass nicht der erste Zweig der Auswahl der richtige ist, sondern der zweite. Der Encoder müsste in diesem Fall zurückgehen, und feststellen, wo er in den falschen Zustand verzweigt ist. Es ist jedoch auch möglich das Schema so zu formulieren, dass es syntaktisch äquivalent ist, und obige Ambivalenz nicht auftritt:

```
<complexType name="unambigous">
  <sequence>
    <element name="a" type="Ta"/>
    <element name="b" type="Tb"/>
  </sequence>
  <choice>
    <element name="c" type="Tc"/>
    <element name="d" type="Td"/>
  </choice>
</complexType>
```

Abb. 4.14: Eindeutige Syntaxdefinition

Im MPEG-7 Schema sind keine mehrdeutigen Definitionen dieser Art vorhanden. Da es im Extremfall möglich ist, dass erst nach dem letzten codierten Element eines Dokuments klar ist, welche Codes signalisiert werden müssen könnte ein solches Dokument nicht gestreamt werden wenn der Aufbau des Dokuments am Encoder nicht schon beim Beginn der Übertragung vollständig bekannt ist. Es ist deshalb darauf zu achten, dass beim Entwurf eines Schemas solche Fälle vermieden werden, wenn die BiM Codierung verwendet werden soll.

4.5 Synthese von Pfad- und Payloadcodierung

Wie in der Beschreibung zur Pfadcodierung und zur Payloadcodierung deutlich geworden ist, haben beide Werkzeuge unterschiedliche Eigenschaften. Der Flexibilität der Pfadcodierung steht die Kompressionseffizienz der Payloadcodierung gegenüber. Eine Kombination aus den Verfahren verbindet die Vorteile beider Ansätze.

4.5.1 Dynamische Codierung

Um der Forderung nach flexibler partieller Aktualisierung einer XML Beschreibung gerecht zu werden, kann der Beschreibungsbaum in mehrere Fragmente unterteilt werden. Mit einem Pfad kann die Wurzel eines solchen Baumfragments spezifiziert werden. Das Fragment selbst besteht aus einem Teilbaum der XML Beschreibung, und wird mit der Payloadcodierung erfasst. Die Granularität der Unterteilung und die Übertragungsreihenfolge der Fragmente kann dabei im Prinzip beliebig gewählt und an die Anwendung angepasst werden.

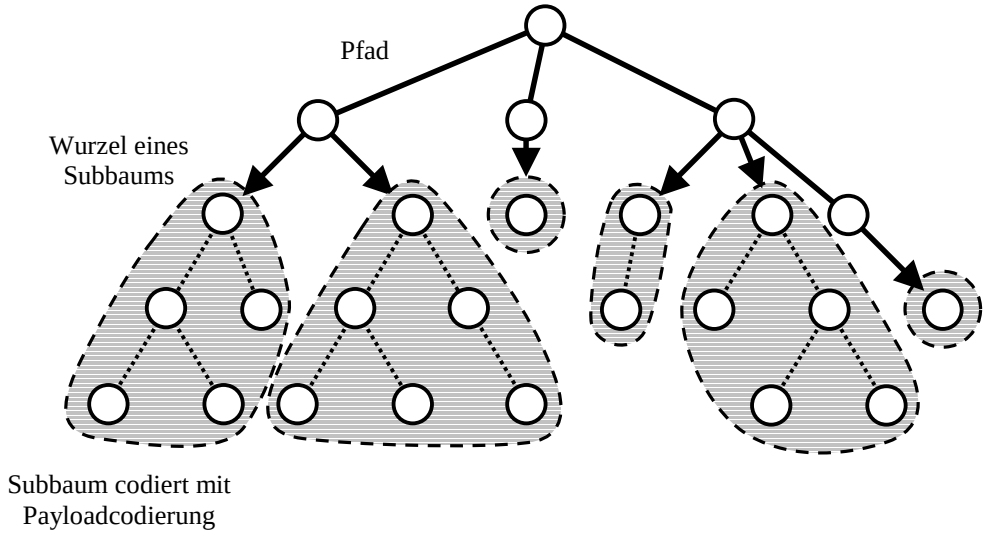


Abb. 4.15: Unterteilung eines Beschreibungsbaums in Fragmente

Die Operation, welche an dem Knoten durchgeführt werden soll, der durch den Pfad adressiert wurde, wird durch ein Kommando, den “fragment update command” angegeben. Möglich sind “add” für das Hinzufügen und “delete” für das Löschen eines Teilbaums, “update”, eine Kombination aus “delete” und “add”, sowie das Kommando “reset”, durch das der aktuelle Beschreibungsbaum vollständig gelöscht wird, und durch eine anfänglich übertragene Rumpfbeschreibung ersetzt wird, vgl. 4.5.2.

4.5.2 Aufbau des Bitstroms

Das Binärformat ist organisiert in einer Folge aus Access Units, die wiederum aus Fragment Update Units bestehen. Eine Access Unit enthält die Information, mit der die Beschreibung zu einem gegebenen Zeitpunkt aktualisiert werden soll. Die Fragment Update Units spezifizieren dann den Ort im Baum und den Inhalt.

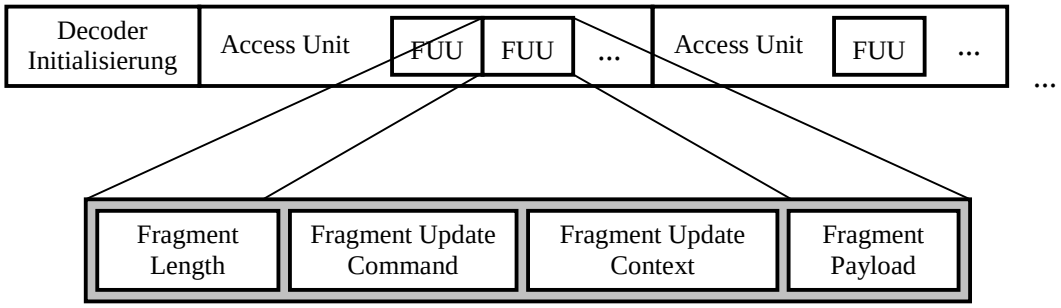


Abb. 4.16: Access Unit und Fragment Update Unit

Abbildung 4.16 zeigt den Aufbau des Binärformats aus Access Units. Der obere Teil zeigt den Aufbau des vollständigen Bitstroms. Die Decoder-Initialisierung enthält Information über die der Beschreibung zugrundeliegenden Schemas, eine Liste von optimierten Datentypcodecs für einfache Typen, und eine “Initial Description”, die eine Rumpfbeschreibung des Dokuments enthält. Diese Rumpfbeschreibung wird dann durch die folgenden FUUs aktualisiert, deren Aufbau im unteren Teil der Abbildung angegeben ist. Jede

FUU beginnt mit der Länge dieser FUU. Diese Längenangabe ist in einem Format spezifiziert, das Vielfache von einem Byte lang ist. Das ermöglicht ein Überspringen der Daten, wenn die FUU Datentypen enthält, deren Schema am Decoder unbekannt, oder die nach einem evtl. am Decoder spezifizierten Profil nicht von Interesse sind. Es folgt das vorhin beschriebene Aktualisierungskommando. Der “Fragment Update Context” enthält einen absoluten oder relativen Pfad, und die Fragment Payload den Subbaum. Am Ende einer FUU werden Füllbits in den Bitstrom eingefügt, damit die folgende FUU oder Access Unit nicht bei einem Bruchteil eines Bytes beginnt.

4.5.3 Filterung

Die beschriebene Struktur des Bitstroms ermöglicht noch eine weitere Funktionalität: jede FUU spezifiziert im Pfad das Wurzelelement des folgend codierten Subbaums, und gibt damit eine Art Inhaltsangabe dieses Datenpakets, nämlich um welchen Datentyp es sich handelt, und in welchem Zusammenhang er in der Beschreibung auftritt (es muss anhand des Update Commands natürlich auch überprüft werden, dass der Subbaum hinzugefügt oder aktualisiert, und nicht etwa gelöscht wird). Diese Inhaltsangabe ist sehr kompakt in Form eines Bitmusters am Anfang der FUU verfügbar, und eignet sich deshalb dafür eine schnelle Filterung der Daten vorzunehmen, ohne jedes Datenpaket auch wirklich vollständig zu decodieren. Das Bitmuster, welches Elemente von Interesse spezifiziert, kann einfach berechnet, und mit den Pfaden verglichen werden. Wenn der Pfad als absoluter Pfad spezifiziert ist, kann der Vergleich Bit für Bit erfolgen. Bei einem relativen Pfad muss dieser erst decodiert und z.B. in einen absoluten Pfad umgewandelt werden, bevor der Vergleich erfolgen kann. In jedem Fall ist aber die Decodierung des Subbaums nicht mehr erforderlich, wenn der Inhalt nicht von Interesse ist. Dieser Vergleich mit einem vorberechneten Muster ist nur deshalb einfach möglich, da die Positionscodes gesammelt am Ende des Pfades eingesetzt sind. Die Positionscodes haben unter Umständen (vgl. 4.3.3) eine variable Länge, und würden deshalb die Berechnung eines Musters (auch wenn man Lücken an den Stellen der Positionscodes im Muster zulassen würde) unmöglich machen. Durch die Datenpartitionierung ist also das Filtern von Elementen möglich unabhängig von der Position, an der sie instanziiert werden.

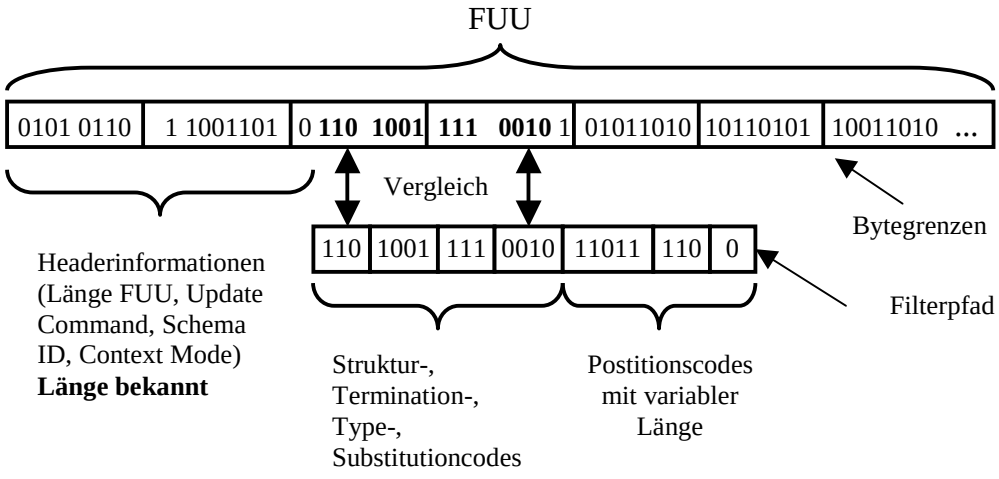


Abb. 4.17: Filterung des Bitstroms nach einem Bitmuster

4.5.4 Granularität der Beschreibung

Die in den vorigen Abschnitten beschriebenen Funktionalitäten “Dynamische Codierung” und “Filterung” lassen sich mit einer Granularität einsetzten, die durch die Feinheit der Unterteilung eines Beschreibungsbaums gewählt werden kann. Das eine Extrem ist nur die Wurzel des vollständigen Beschreibungsbaums durch einen Pfad zu spezifizieren, und den Baum in eine große Payload zu verpacken. In diesem Fall erreicht man die beste Kompression der Daten. Das andere Extrem ist, jedes Element und Attribut durch einen Pfad zu spezifizieren. In diesem Fall kann jedes Element und Attribut einzeln aktualisiert werden, oder das Ziel eines Filtervorgangs sein. Man erreicht maximale Flexibilität, jedoch macht sich dann die geringere Kompression der Pfadcodierung und außerdem der Überhang durch die zusätzlichen Headerinformationen der Access- und Fragment Update Units bemerkbar. Es ist Aufgabe des Encoders unter Berücksichtigung der Randbedingungen der Anwendung eine geeignete Aufteilung zu finden.

4.6 Bewertung des Algorithmus

4.6.1 Kompression

Eine wichtige Kennzahl für die Leistungsfähigkeit von Verfahren zur binären Codierung von Daten ist der Kompressionsfaktor. Hier wird das in diesem Kapitel entwickelte Verfahren bezüglich der Effizienz seiner Kompression für die verschiedenen Werkzeuge (absoluter/relativer Pfad, Payload) ausgewertet und mit anderen Verfahren verglichen.

Tabelle 4.3 gibt eine Übersicht über verschiedene Kompressionsverfahren für vier Testdatensätze. Alle Zahlen sind Dateigrößen in Bytes. Um die Strukturkompression getrennt von der Kompression des Inhalts auszuwerten wurden alle Dateien in einen Strukturteil und einen Teil mit Nutzdaten aufgeteilt. Der Strukturteil enthält die Tags der XML Beschreibung, d.h. die Element- und Attributnamen, der Nutzdatenteil die Werte von Attributen und die Zeichenketten innerhalb der Tags. Man erkennt dass die vollständigen Testdaten zu 75% - 82% aus Strukturinformation bestehen. Die Struktur wird von der Payload recht ordentlich komprimiert, auf nur noch 2,0% - 2,3% des ursprünglichen Umfangs. Auch der Standardalgorithmus für Textkompression ZIP schneidet hier gut ab, weil die Tags oft hochgradig redundant sind, z.B. unterscheiden sich die öffnenden und schließenden Tags nur wenig, außerdem werden in den Testdatensätzen Elemente oft mehrfach instanziiert. Die Kompression mit ZIP reduziert die Strukturinformation auf 4,0% - 12,1%. XMill baut auf ZIP auf, nur ist es auf die Besonderheiten von XML optimiert (vgl. Kapitel drei). Für die Auswertung der Strukturkompression der Pfade wurde jedes Element und Attribut des Dokuments in der Reihenfolge der Instanziierung mit einem absoluten oder relativen Pfad codiert. Der absolute Pfad erreicht eine Kompression auf 10,4% - 16,7%, der relative Pfad auf 4,6% - 6,1% der ursprünglichen Größe. Hier muss man berücksichtigen, dass diese Daten nur den theoretischen Fall betreffen, dass jedes Element/Attribut mit einem Pfad spezifiziert wird. In der Praxis werden größere Teile der Beschreibung zu Subbäumen zusammengefasst und mit einer Payload codiert, so dass die im Vergleich schlechtere Strukturkompression v.a. mit dem absoluten Pfad wenig ins Gewicht fällt.

Testdatensatz	Größe	Payload	Pfad abs	Pfad rel.	XMill	ZIP
MonsterJr4Ver3	Vollst. 87 466	8 217	-	-	9 582	10 444
	Strukt. 65 825	**1 526**	**10 948**	**3 495**	**3 179**	**4 392**
	Nutzd. 21 621	6 691	-	-	-	5 501
MdsExamples-Clause4_7	Vollst. 27 711	3 757	-	-	5 513	5 778
	Strukt. 20 758	**488**	**3 090**	**1 272**	**2 008**	**2 513**
	Nutzd. 6 953	3 269	-	-	-	3 038
MdsExamples-Clause11_12	Vollst. 136 501	10 488	-	-	12 070	13 000
	Strukt. 104 094	**2 047**	**17 429**	**5 711**	**2 812**	**4 168**
	Nutzd. 32 407	8 441	-	-	-	7 427
MdsExamples-Clause13_15	Vollst. 65 209	7 986	-	-	7 101	7 616
	Strukt. 52 878	**1 052**	**5 511**	**2 411**	**3 058**	**3 889**
	Nutzd. 12 331	6 934	-	-	-	3 431

Tab. 4.3: Vergleich der Dateigröße für verschiedene Kompressionsverfahren

Bei der Codierung des gesamten Dokuments (also Struktur- und Nutzdaten) schneidet die Payloadcodierung besser ab als XMILL oder ZIP. Das Payloadverfahren verwendet für die Codierung des Inhalts eine ZIP-Kompression wenn es sich um Zeichenketten handelt, außerdem werden Zahlen nicht als Zeichenketten sondern z.B. als IEEE Floating Point codiert, was normalerweise ebenfalls deutlich kompakter ist als eine Repräsentation als Zeichenkette. Andere optimierte Typcodecs für spezielle simpleTypes werden nicht verwendet, hier kann noch eine Verbesserung der Kompression der Nutzdaten erzielt werden.

4.6.2 Der BiM Algorithmus und die Anforderungen von MPEG

Das Binärformat für XML bestehend aus einer Kombination von Pfad- und Payloadcodierung, welche in diesem Kapitel dargestellt wurde, ist in einer Arbeitsgruppe bei MPEG entwickelt worden, und hat sich gegen alternative Vorschläge [MPEG00d-f] und bereit bestehende Technologie (s. Kapitel drei) durchgesetzt. Sie ist nun Teil des MPEG-7 Standards (Multimedia Content description interface – Part 1: Systems, ISO/IEC 15938-1). Der Autor war an der Entwicklung der Pfadcodierung und der zugehörigen Referenzsoftware beteiligt.

Die hier vorgestellte Lösung hat sich im Vergleich mit den von MPEG definierten Anforderungen am besten bewährt. Das Binärformat kann an die jeweilige Anwendung angepasst werden, wobei der beste Kompromiss aus Kompressionseffizienz und Flexibilität erreicht werden kann.

Die Forderung nach einer kompakten Darstellung ist mit einer Kompression von 40-50 für die Struktur und etwa 10 für ein durchschnittliches Dokument mit textuellem Inhalt gut erfüllt.

Die dynamische Veränderung bereits übertragener Dokumente kann durch die Aktualisierungskommandos durchgeführt werden und zwar, wie ebenfalls gefordert, unter wahlfreiem Zugriff auf einzelne Elemente und Attribute der Beschreibung.

Durch die Organisation des Binärformats in Access Units sind Vorkehrungen für das Streaming von Beschreibungen in Form von schrittweiser Übertragung überschaubarer Datenpakete getroffen. Diese Datenpakete enthalten bereits einen in sich konsistenten und aussagekräftigen Teil der gesamten Beschreibung.

Das Verfahren lässt sich außerdem so implementieren, dass schnelles Parsen des Bitstroms und eine schnelle Verarbeitung (v.a. der Codierung und Decodierung, siehe Kapitel 6) möglich ist.

Die aus dem BiM Format decodierte textuelle Beschreibung ist kanonisch äquivalent zur ursprünglichen Beschreibung. Die genaue Definition dieses Begriffs lässt sich unter [CAN01] nachlesen. Im Prinzip bedeutet der Begriff, dass lediglich für die Aussage der Beschreibung unwichtige Details, wie die Reihenfolge von Attributen, oder gewisse Formatierungen (z.B. Anzahl von Leerzeichen, Zeilenumbrüche zwischen Tags etc.) verloren gehen können.

Die syntaktische Richtigkeit der Beschreibung ist im Falle der Payload durch die Art der Codierung bereits gewährleistet. Zusätzlich ist festgelegt, dass bei der dynamischen Codierung das Dokument nach der Aktualisierung durch eine Access Unit wieder vollständig der Syntaxdefinition des Schemas entsprechen soll: beispielsweise müssen alle Elemente, die nicht optional sind auch vorhanden sein

4.6.3 Andere Ansätze für ein Binärformat

Im Zuge der Entwicklung des hier vorgestellten Verfahrens wurden bei MPEG auch andere Varianten für ein Binärformat diskutiert.

Ein Vorschlag [MPEG00b] weist Eigenschaften auf, die in etwa in der Mitte zwischen Pfad und Payloadcodierung liegen. Er beruht darauf das XML Dokument im wesentlichen wie mit dem relativen Pfad zu codieren, nur ohne die Verwendung von Positionscodes. Dadurch lässt sich ein Dokument “depth-first” übertragen, ähnlich wie bei der Payload. Da die Positionscodes einen erheblichen Teil (ca. 20-60%) des relativen Pfades ausmachen, ermöglicht dieses Verfahren jedoch eine deutlich höhere Kompression als der relative Pfad und ist fast so effizient wie die Payloadcodierung: die Struktur wird auf nur 2% bis 3% der ursprünglichen Größe reduziert. Da in den Tabellen ein Code für jedes mögliche Kind reserviert ist, wäre sogar die Reihenfolge, in der die Elemente instanziiert werden beliebig, jedoch geht die Information über die Position der Elemente verloren. Zwar könnte der Encoder dem Decoder die Positionen im instanziierten Dokument durch eine Art “absoluten Positionscode” übermitteln, jedoch setzt dies voraus, dass der Encoder ein absolut korrektes Abbild des augenblicklichen Dokumentzustands am Decoder hat. Die Fehlerrobustheit ist damit nicht mehr gewährleistet, weshalb diese Variante zu dem hier beschriebenen Verfahren weiterentwickelt wurde.

Ein anderer Vorschlag [MPEG00d] verwendet einen Ansatz, der ähnlich wie WBXML Kürzel für die Tags der XML Beschreibung spezifiziert. Die Tags werden aus dem Schema abgeleitet. Sie sind hierarchisch aufgebaut, aus einem Teil, der die Klasse (Video, Audio, MDS) symbolisiert, sowie einen Teil für die Subklasse und einen Teil für den individuellen Typen, insgesamt 16 Bit, zusammen mit einem 8 Bit „Startcode“ und einer optionalen 4 Bit „Instanznummer“ bis zu 28 Bit. Das Schema liefert also auch hier die Grundlage für den binären Code, aber im Gegensatz zur BiM Codierung sind die Codes global: zu jedem Zeitpunkt kann jeder der (relativ langen) Codeworte signalisiert werden, die Syntaxdefinition wird nicht richtig ausgereizt. Das Verfahren ist deshalb weniger effizient, es erreichte für die Testdaten nur eine Kompression auf 25%-50% der ursprünglichen Größe. Zudem ist der Algorithmus speziell auf die Struktur des MPEG-7 Schemas zugeschnitten und nicht allgemein anwendbar.

Ein dritter Vorschlag [MPEG00e] leitet aus dem Schema für jede Elementdeklaration einen Identifizierungscode ab, den sogenannten „Structuring Key“, der ausgehend von den globalen Elementen, ähnlich wie ein absoluter Pfad, eine Element- oder auch

Attibutdeklaration identifiziert. Allerdings gibt es Elemente, deren Typdefinition das Element selbst enthält. Es ist deshalb eine unendliche Rekursion möglich. Da der Satz an Identifizierungscodes begrenzt sein muss wird dieser Fall durch einen „Self Containment Key“ abgefangen. Die Identifizierungscodes werden in einer einzigen Codetabelle (der „Element-Declaration-Table“), die für das ganze Schema gültig ist eingetragen. Das Verfahren erreichte für die Struktur der Testdaten eine Kompression auf etwa 4% bis 16%. Da es weniger effizient war als andere Vorschläge und wegen der umständlichen Behandlung der Rekursion wurde es im Standard nicht berücksichtigt.

Ein weiterer Vorschlag schließlich [MPEG00f] bietet im Gegensatz zu den bisher beschriebenen Alternativen ein Verfahren an, das ohne die Syntaxdefinition auskommen kann: die Codetabellen, welche für die Signalisierung der XML-Tags, der Attribute und Attributwerte Verwendung finden werden aus dem bisher übertragenen Dokument gewonnen (es gibt allerdings optional auch die Möglichkeit die Schemainformation für vordefinierte Codes zu nutzen). Die Codes sind eindeutig, weshalb der Kontext, in dem sie im Bitstrom stehen nicht bekannt sein muss. Ähnlich wie bei Xmill wird die Struktur des Dokuments vom Inhalt getrennt. Zahlen für die Kompressionsrate wurden nicht angegeben. Besonderer Wert wurde auf die Übertragungseigenschaften gelegt: dafür wurde der Bitstrom in *structure packets*, *text packets*, *header packes*, *trailer packets* und *command packets* organisiert.

4.6.4 Allgemeine Anwendbarkeit der syntaxbasierten Codierung

Wie bereits in Kapitel drei bei der Beschreibung bekannter Codierverfahren deutlich geworden ist, sind verschiedene Ideen, welche im BiM Algorithmus verwendet werden in ähnlicher Form schon an anderer Stelle eingesetzt worden. Das Grundprinzip der syntaxbasierten Codierung ist, dass der Encoder aus einem durch eine Syntaxdefinition des codierten Dokuments eingeschränkten Satz an Möglichkeiten diejenige Alternative signalisiert, welche im aktuellen Dokument an der aktuellen Stelle auftritt. Bei der Codierung von XML ist dies im BiM Format an vier Stellen notwendig: bei der Signalisierung von Auswahlgruppen (oder entsprechend transformierten all Gruppen), bei optionalen Elementen, beim Polymorphismus und bei Ersetzungsgruppen. Ferner gibt es mehrere Möglichkeiten wie rigoros man die Syntaxdefinition ausnutzt: bei der Pfadcodierung ist jedes mögliche Kind eines Typs eine Alternative die codiert wird, bei der Payloadcodierung nur die nächste Auswahlgruppe oder das nächste optionale Element. Neben Fällen, in denen die Syntaxdefinition einen begrenzten Satz an Möglichkeiten vorgibt, gibt es auch Fälle, in denen eine Auswahl aus einer unbegrenzten Zahl an Möglichkeiten vorgenommen werden muss, nämlich der Zahl der Instanzen eines Elements, das beliebig oft auftreten kann. Für diesen Fall sollte eine Repräsentation gewählt werden, die sowohl kompakt ist als auch die Zahl der möglichen Instanzen nicht einschränkt.

Dieses allgemeine Grundprinzip lässt sich auch auf andere strukturierte Dokumente anwenden, die nicht nach der XML Syntax aufgebaut sind, wenn das strukturierte Dokument einer strikten Syntaxdefinition folgt. Wenn diese Syntaxdefinition bekannt ist lässt sich im Prinzip eine “BiM –artige” Codierung für diese Klasse von Dokumenten entwerfen. Das besondere der BiM Codierung ist dabei, dass sie die syntaxbasierte Codierung in zwei verschiedenen Ausprägungen anwendet (Pfad/Payload), welche einen Kompromiss zwischen Kompressionseffizienz und Flexibilität bei der Übertragung sowie dynamische Aktualisierungen ermöglicht.

Kapitel 5 - Vergleich und Bewertung des Binärformats

Die Eigenschaften der bekannten Codierverfahren im Hinblick auf die Anforderungen an ein Binärformat für strukturierte Dokumente, welche sich aus den geplanten Anwendungen ergeben wurden schon in Kapitel drei diskutiert. Hier soll nun noch ein Vergleich mit dem neu entwickelten BiM Verfahren durchgeführt werden. Außerdem wird für das MPEG-7 Schema das Verbesserungspotential von statistischen Optimierungen bei der Codevergabe untersucht. Eine informationstheoretische Analyse der syntaxbasierten Codierung, welche Anhaltspunke für die Grenzen der Kompressionsleistung solcher Verfahren liefert bildet den dritten Teil des Kapitels.

5.1 Vergleich des Binärformats mit den bekannten Verfahren

Die bekannten Verfahren sollten zunächst in syntaxbasierte und syntaxfreie Verfahren unterschieden werden. Der Hauptvorteil von syntaxfreien Verfahren ist die Unabhängigkeit der Codierung des Dokuments vom zugrundeliegenden XML Schema. Es braucht keine Rücksicht darauf genommen zu werden welches XML Schemas in welcher Version am Decoder bekannt ist.

Als mögliche Anwendungen werden Dateitransfer (meist in einem Datenblock) oder Archivierung ins Auge gefasst, weshalb die Kompression der einzige betrachtete Bewertungsmaßstab ist. Andere Eigenschaften, wie inkrementelle Übertragung, dynamische Aktualisierung oder Filterung spielen meist keine Rolle. Die Kompressionswirkung wird durch Ausnutzung der Redundanzen im XML Dokument erzielt. Bei den auf XML optimierten Verfahren werden verschiedene Strategien angewendet, um diese Redundanzen zu erkennen und zu eliminieren. Der Rechenaufwand zur Erkennung der Redundanzen ist normalerweise größer, als die Bestimmung der richtigen Alternative in einer Syntaxdefinition, weil durch diese der Suchraum bereits sehr eingeschränkt ist: es ist einfacher aus einer handvoll möglicher Alternativen die zu codierende auszuwählen, als in einem 10KB großen Suchfenster zu überprüfen, ob ein Wort schon einmal aufgetreten ist [XMLZ].

Bezüglich der Strukturkompression erzielt kein Verfahren die Leistung der syntaxbasierten Codierung. Für die Kompression des Inhalts enthalten die hier vorgestellten Verfahren Ideen, die auch für den BiM Algorithmus eingesetzt werden können: die bei XMILL praktizierte Gruppierung von Daten gleichen Typs in Container zur Vergrößerung der Redundanz lässt sich auch im BiM Verfahren gut anwenden, da für jedes Datenfeld mit Inhalt bekannt ist, um welchen Datentyp es sich handelt. Allerdings müsste für einen effizienten Einsatz eine größere Menge an Inhalt gleichen Typs in einem Container gesammelt werden, um die Redundanzen gut auszunützen. Wenn diese Container sich über die Grenzen von Access Units hinweg erstrecken würden dadurch Datenabhängigkeiten zwischen Access Units entstehen, die sich auf die Übertragungseigenschaften und die Fehlerrobustheit negativ auswirken.

Die syntaxbasierten Verfahren unterscheiden sich vom BiM Verfahren in erster Linie durch die statistischen Optimierungen bei der Vergabe der Codeworte. Schon in Kapitel drei wurde auf die entscheidenden Nachteile hingewiesen: die Codierung ist bei einer fest eingebauten Statistik auf einen speziellen Satz an Dokumenten optimiert, wie bei XML Xpress. Wenn die Statistik aus dem Dokument selbst dynamisch gewonnen wird leidet die Fehlerrobustheit, weil das statistische Modell, welches zur Encodierung verwendet wird genau dem Modell am Decoder entsprechen muss. Fehler bei der Übertragung, oder ein

verspätetes Einschalten in eine laufende Übertragung würden dann zu nicht überbrückbaren Fehlern führen.

5.2 Statistische Optimierungen

Da in anderen Ansätzen zur Codierung von XML Dokumenten statistische Optimierungen zur Kompression verwendet wurden, soll in diesem Abschnitt abgeschätzt werden, ob durch die Anwendung dieses Konzepts auch für die BiM Codierung noch weitere Verbesserungen zu erwarten wären.

In den meisten Fällen wird der größte Teil eines XML Dokuments beim BiM Verfahren mit dem Payloadalgorithmus codiert. Deshalb hätte eine Verbesserung der Codiereffizienz in diesem Teil eine größere Auswirkung als eine Verbesserung bei der Pfadcodierung und wird darum hier bevorzugt betrachtet.

Die Codierung der Struktur eines Dokuments findet bei der Payloadcodierung durch vier Arten von Codes statt: die Codierung von Häufigkeiten bzw. optionalen Elementen, die Codierung der Zweige von Auswahlgruppen (und der umgeformten all-Gruppen), die Codierung von Type Codes und die Codierung von Ersetzungsgruppen.

Bei der Codierung von Häufigkeiten lässt sich eine statistische Optimierung nicht ohne weiteres anwenden, da nicht aus einem fest definierten Satz an Möglichkeiten, für die eine statistische Verteilung bekannt ist, eine Auswahl zu treffen ist. Eine Möglichkeit wäre lediglich den Modus der Codierung der Häufigkeit an die zu erwartenden Anzahl von Instanzen anzupassen (vgl. 5.3.2). Für eine statistische Optimierung bieten sich aber die Auswahlgruppen, die Typecodes und die Ersetzungsgruppen an.

5.2.1 Verteilung der Bitrate in XML Dokumenten des MPEG-7 Schemas

Um beurteilen zu können, wie stark sich Verbesserungen in den einzelnen Bereichen der Strukturcodierung auf die gesamte Kompression auswirken ist es notwendig abzuschätzen welcher Anteil des gesamten Bitstroms eines Dokuments auf Auswahlgruppen, Typecodes und Häufigkeiten entfällt (Ersetzungsgruppen kommen im MPEG-7 Schema nicht vor, und werden deshalb hier nicht betrachtet). Für ein 104 KB große Testdokument aus dem MPEG-7 Testdatensatz ist die Verteilung wie folgt:

Bits für Auswahlgruppen:	1440
Bits für Typecodes:	1717
Bits für Häufigkeiten:	6308
(Bits für Ersetzungsgruppen	0)
Gesamt:	9465

Tab. 5.1: Verteilung der Information auf Codetypen

Dabei wird jede Instanz eines Elements, das mehr als einmal auftreten kann mit einem Bit signalisiert. Wird ein anderer Modus (vgl. 4.4.3) für die Codierung von Häufigkeiten verwendet, bei dem die Zahl der Instanzen eines Elements bei der ersten Instanz als ganze Zahl mit variabler Länge codiert wird (oder mit der Zahl an Bits, die aus dem maxOccurs Attribut abgeleitet ist), so ist die Zahl der Bits für die Codierung der Häufigkeiten noch dominanter. Dies liegt daran, dass viele Elemente, die unendlich oft auftreten können, aber

nur einmal instanziiert werden mit einem 5 Bit Datenfeld des vluimsbf5 Formats (s. Anhang A1) codiert werden.

Auch unabhängig von einem speziellen Dokument kommt man zu einer ähnlichen Abschätzung, wenn man die Verteilung von Auswahlgruppen, Häufigkeiten und vererbten Typen im MPEG-7 Schema betrachtet: 118 Auswahlgruppen stehen 660 Occurrences gegenüber, viele davon mit unbegrenzter obere Schranke. Die Zahl an Typen, für die ein Typecast möglich ist beträgt für das MPEG-7 Schema 89 von etwa 770, also knapp 12%.

Aus der Aufstellung geht hervor, dass der mit Abstand größte Teil der Bitrate auf die Codierung von Häufigkeiten entfällt. Die Wirksamkeit von Optimierungen bei Auswahlgruppen oder Typecodes ist deshalb begrenzt.

Die Verteilung der Codewortlänge bei Auswahlgruppen

Um das Potential einer statistischen Optimierung der Codierung für die einzelnen Bereiche der Strukturcodierung einzugrenzen soll zunächst für das MPEG-7 Schema die Verteilung der Zahl der Zweige der Auswahlgruppen betrachtet werden.

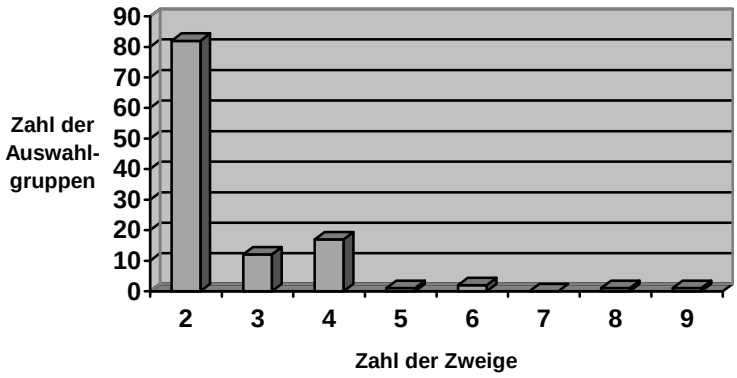


Abb. 5.1: Verteilung Zahl der Zweige bei Auswahlgruppen

Aus der Verteilung geht hervor, dass 70,7% der Auswahlzustände mit nur einem Bit codiert werden (solche mit 2 Alternativen), 25,0% mit zwei Bit (drei und vier Alternativen), 3,4% mit 3 Bit (5 bis 8 Alternativen) und 0,8% mit 4 Bit (neuen Alternativen). Die durchschnittliche Zahl an Bits für die Signalisierung des Zweiges einer Auswahl ist 1,34.

Die Verteilung der Längen der Typecodes

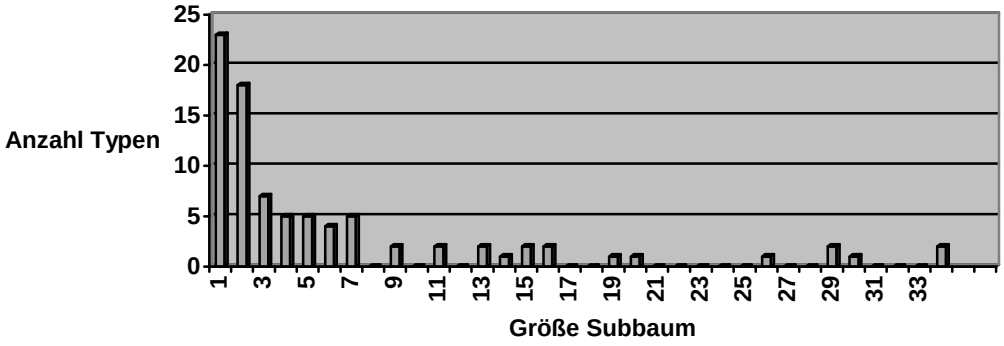


Abb. 5.2: Verteilung Zahl der abgeleiteten Typen

Zusätzlich zu den in dem Diagramm abgebildeten existieren noch drei Typen, bei denen der Unterbaum mehr als 34 Knoten hat: Mpeg7RootType (243 abgeleitete Typen), DType (60) und DSType (216). Damit ergibt sich folgende Verteilung der Zahl der Bits, die zum signalisieren eines Typecodes notwendig sind:

Bit für Typecode	Zahl der abgeleiteten Typen	Prozent
0	23	25,8%
1	18	20,2%
2	12	13,5%
3	14	15,7%
4	11	12,4%
5	8	9,0%
6	1	1,1%
8	1	1,1%
9	1	1,1%
	Summe: 89	

Tab. 5.2: Verteilung der Länge des Typecodes

Außerdem muss noch bei jedem Typ bei dem ein Typecast möglich ist mit einem Flag im Bitstrom signalisiert werden, ob der Typecast durchgeführt wird oder nicht. Die durchschnittliche Länge eines Typecodes ist 2,15 Bit, mit dem Flag für den Typecast ergibt sich also ein Wert von 3,15 Bit für jeden Typecast. Null Bit für den Typcode sind dann möglich, wenn es nur einen abgeleiteten Typen gibt. In diesem Fall reicht bereits das Flag aus, um eindeutig den neuen Typen zu signalisieren.

5.2.2 Anwendung einer Huffmancodierung

Die Grundidee der Huffmancodierung ist es die statistische Verteilung der Häufigkeit, mit der die Symbole eines Alphabets in einem Text auftreten für eine Reduzierung der Information zu nutzen. Die Wirksamkeit des Verfahrens ist umso größer, je ungleicher die Symbole verteilt sind. Häufig auftretenden Symbolen werden kurze Codeworte zugewiesen, selten auftretenden Symbole entsprechend längere. Die Codeworte haben eine variable Länge. Durch den Aufbau der Codes ist sichergestellt, dass ermittelt werden kann, wie lange ein Codewort ist, d.h. wie viele Bits für eine Codewort aus dem Bitstrom gelesen werden müssen.

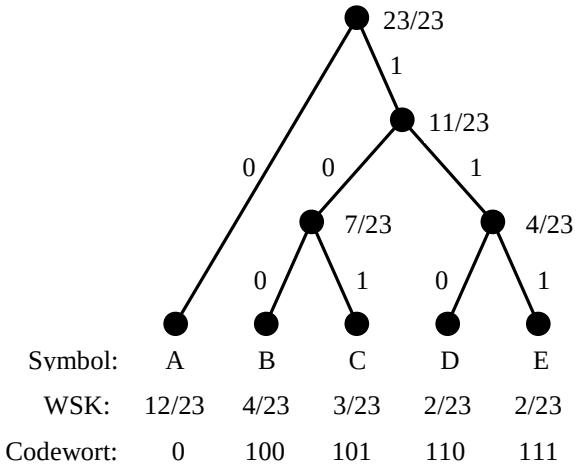


Abb. 5.3: Codevergabe bei der Huffmancodierung

Abbildung 5.3 verdeutlicht die Codezuweisung: Das Alphabet besteht im Beispiel aus den fünf Symbolen A bis E, mit einer Wahrscheinlichkeitsverteilung wie angegeben. Die Knoten der beiden am wenigsten wahrscheinlichen Symbole werden in einen neuen Knoten zusammengefasst, dessen Wahrscheinlichkeit die Summe der Wahrscheinlichkeiten der beiden zusammengefassten Knoten ist. Dies wird wiederholt, bis alle Knoten zu einem Koten mit der Wahrscheinlichkeit eins vereinigt sind. Es entsteht ein binärer Baum. Jedem von einem Knoten abgehenden linken Zweig wird eine Codewortfragment “0” zugewiesen, jedem rechten Zweig eine “1”. Das Codewort eines Symbols wird gebildet durch eine Verkettung der Codewortfragmente des Pfads von der Wurzel des Baums bis zum Knoten des Symbols. Weil kein Codewort das Präfix eines anderen Codewortes bildet ist sichergestellt, dass die einzelnen Codeworte in einem Bitstrom unzweideutig identifiziert werden.

Die durchschnittliche Bitlänge, die für die Codierung eines Symbols in einem Text erforderlich ist, welcher der zugrundegelegten Wahrscheinlichkeitsverteilung, bzw. Häufigkeit der Symbole entspricht, berechnet sich durch die Länge des Codeworts multipliziert mit der Wahrscheinlichkeit des Symbols.

Für das Beispiel ist die zu erwartende Bitlänge pro Symbol:

$$\frac{12}{23}\cdot 1+\frac{4}{23}\cdot 3+\frac{3}{23}\cdot 3+\frac{2}{23}\cdot 3+\frac{2}{23}\cdot 3=1{,}96Bit$$

Eine Huffmancodierung lässt sich auch auf die Codierung von Zweigen einer Auswahlgruppe oder auf die Codierung von Typecodes anwenden. Da die Wahrscheinlichkeitsverteilung der Zweige von Auswahlgruppen oder Typecodes für Dokumente, die nach dem MPEG-7

Schema codiert wurden nicht bekannt ist, müssen hier Annahmen gemacht werden um das Einsparpotential einer Huffmancodierung abzuschätzen. Das maximale Einsparpotential würde erreicht werden, wenn (was eine unrealistische Annahme ist) ein Zweig bei jeder Auswahl, bzw. ein Typecode praktisch immer vorkommt, und alle anderen mit einer Häufigkeit, die zu vernachlässigen ist. In diesem Fall könnte jeder Zweig und jeder Typecode mit einem Bit signalisiert werden, da der Algorithmus der Huffmancodierung für die Zuweisung von Codes zu Symbolen in diesem Fall einen eigenen Zweig von der Wurzel des Entscheidungsbaums für dieses häufigste Symbol generiert.

Unter dieser Annahme werden die Auswahlgruppen um 26% sparsamer codiert (es wird ein Bit benötigt, anstatt 1,35 Bit), der Polymorphismus um 37% sparsamer (2 Bit statt 3,15 Bit). Das Beispieldokument würde

$$1440\frac{1}{1{,}35}+1717\frac{2}{3{,}15}+6308=8464$$

Bits für die Strukturcodierung benötigen, eine Ersparnis von 10,6%. Dabei wird angenommen, dass der Modus für die Codierung von Häufigkeiten nicht verändert wird.

Die zweite Annahme ist, dass bei einer Auswahl oder einem Typecode ein Symbol immer doppelt so häufig auftritt, wie das nächst wahrscheinlichere Symbol. Die Wahrscheinlichkeitsverteilung einer Auswahl mit fünf Zweigen wäre also:

$$\frac{16}{31},\frac{8}{31},\frac{4}{31},\frac{2}{31},\frac{1}{31}$$

Auch diese Wahrscheinlichkeitsverteilung weist eine deutliche Asymmetrie auf, ist aber als Modellverteilung realistischer als die erste Annahme. Die Codevergabe nach der Huffmancodierung weist dem wahrscheinlichsten Zweig einen Code mit einem Bit zu. Jeder Zweig hat ein um ein Bit längeres Codewort, als der nächstwahrscheinlichere Zweig, die beiden unwahrscheinlichsten Zweige haben die gleiche Codewortlänge (gibt es nur zwei Zweige, werden beide mit einem Bit codiert).

Damit ergibt sich für die Auswahlgruppen folgende Verteilung:

Zahl Auswahl	Zweige	Bits
82	2	1,00
12	3	1,43
17	4	1,67
1	5	1,81
2	6	1,89
1	8	1,96
1	9	1,98
Gesamt: 116		1,18

Tab. 5.3: Codierung der Auswahlgruppen

Im gewichteten Mittel werden bei dieser Verteilung 1,18 Bit für die Signalisierung von Auswahlgruppen benötigt.

Eine entsprechende Berechnung für die Verteilung der Bitlänge bei Typecodes liefert einen Wert von 1,20 Bit für den Typecode, zusammen mit dem Typecast Flag also 2,20 Bit. Die zu erwartende Dokumentgröße für das Beispiel ist:

$$1440\frac{1{,}18}{1{,}35}+1717\frac{2{,}20}{3{,}15}+6308=8765$$

was einer Ersparnis von 7.4% entspricht.

Man sieht, dass das Potential einer Huffmancodierung für die Strukturcodierung einer Payload begrenzt ist. Selbst im günstigsten Fall einer extrem ungleichen Verteilung der Elemente in XML Dokumenten ist nur eine Verbesserung der Codierung in der Größenordnung um 10% zu erwarten. Problematisch ist dabei auch die Generierung der Statistik, welche beim Entwurf der Huffmancodes zugrunde gelegt wird. Im Normafall kann sie nur aus einem Satz an Beispieldokumenten gewonnen werden. Dies würde aber bedeuten, dass die Codevergabe auf einen speziellen Anwendungsfall, oder einen speziellen Anwender optimiert ist, was für ein allgemein anzuwendendes Verfahren ungünstig ist. Außerdem ist zu bedenken, dass bei einer Huffmancodierung für jede Auswahl und jeden Typecode eine eigene Codetabelle zu speichern ist, da sonst nicht bekannt ist, wie viele Bits gelesen werden müssen, und welchem Codewort jede der Alternativen entspricht. Für die beim BiM Algorithmus vorgesehene Codevergabe kann die Länge des Codeworts dagegen einfach aus der Gesamtzahl der Codeworte berechnet werden.

5.3 Informationstheorie der Syntaxbasierten Codierung

Neben praktischen Gesichtspunkten wie Einfachheit der Berechnung der Codes, Fehlertoleranz und Übertragungseigenschaften welche bei der Entwicklung der BiM Codierung im Vordergrund standen ist auch die Frage interessant, wie effizient dieses Verfahren im Vergleich zum theoretisch Erreichbaren einer syntaxbasierten Codierung ist. Im Prinzip unterscheidet sich diese Syntaxbasierte Codierung nicht von der Codierung eines Textes mit festem Alphabet. Der Unterschied ist lediglich, dass sich das “Alphabet” mit jedem Codierten Symbol ändert. Die Codierung von Auswahlgruppen, Typecodes oder Ersetzungsgruppen ist nur eine Folge von Entscheidungen zwischen einer begrenzten und wohldefinierten Anzahl von Alternativen. Wenn für jede dieser Alternativen eine Wahrscheinlichkeits- bzw. Häufigkeitsverteilung bekannt ist lässt sich die Entropie berechnen und damit angeben mit wie vielen Bits eine solche Auswahl mindestens zu codieren ist. Bei der Codierung von Häufigkeiten ist die Situation etwas anders: hier handelt es sich um einen Vorgang, bei dem im Allgemeinen nicht zwischen einem festen Satz an Alternativen gewählt wird. Dies tritt auf, wenn die obere Schranke für die Zahl der Instanzen unbegrenzt ist. Der BiM Algorithmus sieht für diesen Fall eine Codierung mit einem Format für ganze Zahlen vor, welches sich der Größenordnung dieser Zahl anpasst (vgl. Anhang A1). Die Frage ist nun, ob es für die Codierung von Häufigkeiten eine bessere Darstellung gibt. Bei der Auswertung der Verteilung der Codebits bei Dokumenten des MPEG-7 Schemas ist aufgefallen, dass der größte Teil auf die Codierung von Häufigkeiten entfällt. Die Beantwortung dieser Frage ist deshalb besonders interessant.

5.3.1 Entropie der Auswahlgruppen

Das folgende Beispiel zeigt das Modell eines komplexen Typen mit fünf Auswahlgruppen “A” bis “E”, die zum Teil kaskadiert (“A” vor “B”), zum Teil verschachtelt sind (“E” in “D” in “B”).

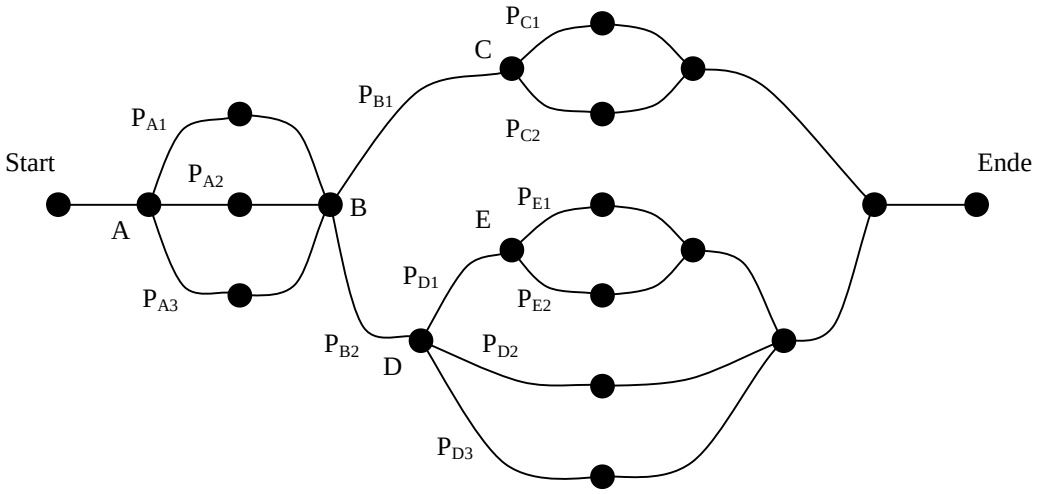


Abb. 5.4: Inhaltsmodell eines komplexen Typen mit Wahrscheinlichkeiten

Die Entropie und damit der Informationsgehalt einer Auswahl ist nach Shannon:

$$H = \sum_{n} P_n \cdot \log_2\left(\frac{1}{P_n}\right) \tag{5.1}$$

Wobei die P_n die Wahrscheinlichkeiten für die einzelnen Alternativen der Auswahl sind, und der Index n über alle Alternativen läuft. Eine solche Auswahl muss im Mittel mit mindestens H Bits Codiert werden.

Die Wahrscheinlichkeiten einer Auswahl könnten zudem vom Kontext abhängen, d.h. von den bis dahin getroffenen Entscheidungen. Beispielsweise könnte die Wahrscheinlichkeitsverteilung am Punkt “B” davon abhängen in welchen Zweig am Punkt “A” verzweigt worden ist. Die Wahrscheinlichkeit für solche mehrstufigen Entscheidungen ist das Produkt der Wahrscheinlichkeiten der einzelnen Stufen. Im Allgemeinen ist der Informationsgehalt eines Weges durch den Inhalt eines komplexen Typen also:

$$H = \sum_{j,j,k\ldots} P_V \log_2\left(\frac{1}{P_V}\right) \quad \text{mit } P_V = P_{1.Stufe(i)} \cdot P_{2.Stufe(j)|1.Stufe=i} \cdot P_{3.Stufe(k)|1.Stufe=i\&2.Stufe=j} \ldots. \tag{5.2}$$

Dabei treten die bedingten Wahrscheinlichkeiten „Entscheidung bei Stufe S ist ‚*y*’ unter der Bedingung Entscheidung bei Stufe S-1 war ‚*x*’ “ usf. als Faktoren auf. Solch ein Zusammenhang könnte durch eine Arithmetische Codierung erfasst werden.

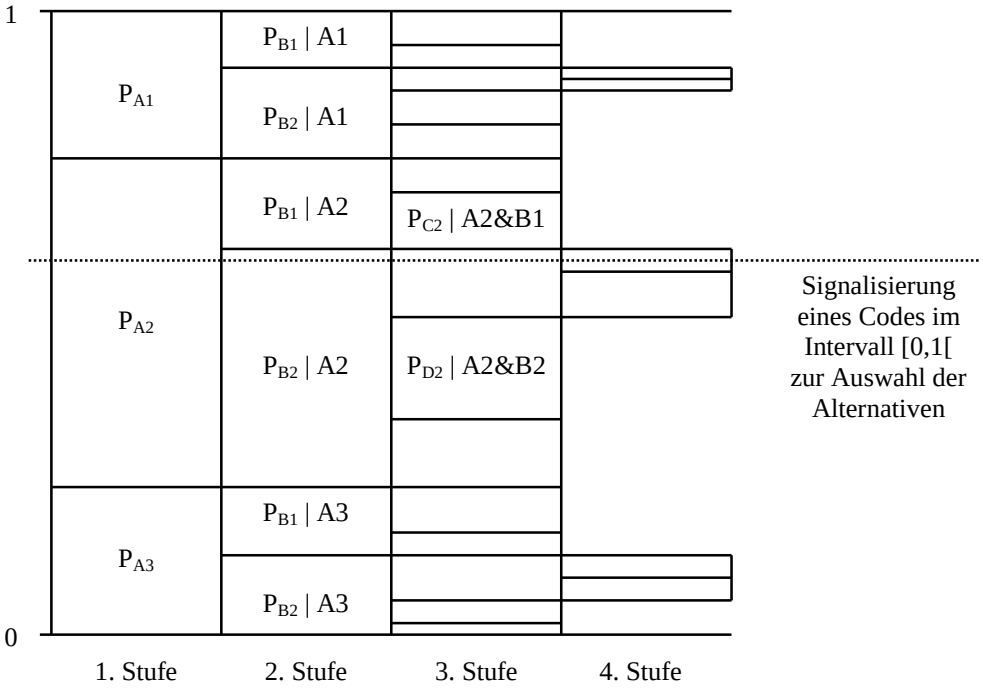


Abb. 5.5: Arithmetische Codierung für Alternativen

Die Abbildung 5.5 zeigt wie das Modell für den Inhalt des komplexen Typen aus Abbildung 5.4 mit einer Arithmetischen Codierung behandelt werden könnte. Das Intervall [0,1[wird durch die erste Entscheidung am Punkt “A” in drei Teilintervalle aufgeteilt. Jedes Intervall ist so groß wie es der Wahrscheinlichkeit P_{Ax} für die Auswahl des korrespondierenden Zweiges im Modell entspricht. Durch die zweite Entscheidung am Punkt “B” werden die Teilintervalle P_{A1} bis P_{A3} entsprechend der Wahrscheinlichkeiten für den oberen und den unteren Zweig weiter unterteilt. Dabei kann die bedingte Wahrscheinlichkeit verwendet werden, z.B. die Wahrscheinlichkeit dafür, dass der obere Zweig am Punkt “B” ausgewählt wird, unter der Bedingung, dass in Punkt “A” ebenfalls der oberste Zweig ausgewählt worden ist: $P_{B1}|A1$. Die Aufteilung der Teilintervalle P_{A1}, P_{A2} und P_{A3} durch die Auswahl in B braucht nicht im gleichen Verhältnis erfolgen. Auf diese Weise werden die weiteren Entscheidungen in das Diagramm eingetragen, wodurch nach der letzten Stufe das Intervall [0,1[in 18 Subintervalle zerfallen ist, deren Größe die Wahrscheinlichkeit für den entsprechenden Weg durch den Inhalt des komplexen Typen darstellt. Je unwahrscheinlicher eine Alternative ist, umso geringer ist die Größe des Intervalls. Bei der Arithmetischen Codierung wird der Binärbruch mit der geringsten Stellenzahl signalisiert, so dass der Zahlenbereich, welcher durch die Genauigkeit des Binärbruchs vorgegeben wird vollständig in dem betreffenden Intervall liegt.

Wird beispielsweise 0.011 signalisiert, so darf im Bereich

$$0.011 - 0.011\bar{1}$$

nur noch eine Alternative liegen. Je kleiner das Intervall ist, umso länger muss der Binärbuch sein, um in dieses Intervall zu treffen. Unwahrscheinliche Alternativen erhalten damit lange Codeworte.

Umfang des Kontextes

Theoretisch könnte man die Hierarchie der komplexen Typen flach machen und die Berücksichtigung des Kontextes fortführen, wenn die Häufigkeit der Instanz in die hierarchisch verzweigt wird eins ist (sonst würde die Codierung von Häufigkeiten mit der Codierung von Alternativen vermischt). Allerdings würde dies in einer nicht mehr überschaubaren Menge an Subintervallen für die Arithmetische Codierung enden. Es erscheint vernünftig die Berücksichtigung von Kontexten auf den Inhalt eines komplexen Typen zu beschränken.

5.3.2 Codierung von Häufigkeiten

Die Codierung von Häufigkeiten wird getrennt von der Codierung von Alternativen untersucht. Es werden nur Häufigkeiten betrachten die von Null bis Unendlich reichen. Dies ist ein Fall, der im MPEG-7 Schema häufig auftritt (es kommt auch vor, dass die Häufigkeit von eins bis Unendlich oder von einer kleinen ganzen Zahl bis Unendlich reicht, solche Details sollen aber in dieser Betrachtung vernachlässigt werden). Für eine Zahl, deren Wertebereich von Null bis Unendlich reicht lässt sich ebenfalls mit der Formel (5.1) die Shannon-Entropie berechnen. Im Normalfall wird die Formel verwendet, wenn ein Symbol codiert wird, das aus einem endlichen Alphabet mit definierter Auftrittswahrscheinlichkeit für jedes Symbol stammt. Im Fall der Codierung von Häufigkeiten muss ein Symbol aus einem unbegrenzten “Alphabet” gewählt werden, nämlich dem Körper der natürlichen Zahlen mit der Null. Die Summe zur Berechnung der Shannon-Entropie läuft deshalb von Null bis Unendlich. Im MPEG-7 Schema gibt es gut 300 Elemente oder Gruppen, die unendlich oft auftreten können. Jede Einzelne davon wird eine Wahrscheinlichkeitsverteilung haben, die vom Typ des Elements und vom Kontext abhängt, in dem das Element oder die Gruppe verwendet wird. Beispielsweise könnten die Häufigkeiten, welche die Benutzerdaten im “ContentManagmentType” annehmen in der Datenbank einer Mediengesellschaft ohne weiteres Zahlen im Millionenbereich enthalten. Das “RelatedMaterial” im “CreationType” dagegen dürfte in vielen Fällen nur ein-, bis maximal zweistellig sein. Zudem kommt, dass auch ein und derselbe Deskriptor je nachdem von wem er in welchem Kontext verwendet wird unterschiedlichen Verteilungen gehorchen kann: abhängig davon, wer der Autor eines Videos ist, und was bezweckt wird, kann man ein Video gröber oder feiner in Videosegmente unterteilen, was natürlich die Gesamtzahl der Segmente beeinflusst. Um solche Zusammenhänge zu erfassen müsste man also menschliches Verhalten unter verschiedenen Rahmenbedingungen in einem Wahrscheinlichkeitsmodell abbilden, was sehr schwer sein dürfte. Ohne Wahrscheinlichkeits- bzw. Häufigkeitsverteilung ist jedoch die Entropie der Verteilung nicht definiert. Deshalb ist im Allgemeinen auch keine Abschätzung der zu erwartenden Zahl an Bits für verschiedene Varianten der Häufigkeitscodierung möglich. In einigen Fällen kann es dennoch gelingen eine Wahrscheinlichkeitsverteilung aus dem Kontext abzuleiten.

Eine zweite Möglichkeit wäre aus der Praxis Daten zu sammeln, indem man entsprechende Beispielbeschreibungen auswertet. Nachdem MPEG-7 noch zu neu ist, und das XML -Schema zum jetzigen Zeitpunkt (Mitte 2003) noch nicht breiter verwendet wird, gibt es allerdings kein aussagekräftiges Datenmaterial. Selbst wenn ein solches zur Verfügung stünde wäre es schwierig die Allgemeingültigkeit der verwendeten Daten zu belegen.

Um das Problem behandeln zu können muss zunächst eine Wahrscheinlich-keitsverteilung zugrunde gelegt werden. Einige Varianten sind denkbar: eine Exponentialverteilung, eine Gaußverteilung, eine Poissonverteilung etc. Weil der Wertebereich unendlich groß ist muss die Verteilung für große Werte gegen Null streben.

Einige Verteilungen haben die größten Wahrscheinlichkeiten bei Null, wie die Exponentialverteilung, andere, beispielsweise die Poissonverteilung steigen zunächst an, um dann ebenfalls gegen Null zu fallen. Neben der Charakteristik der Verteilung ist noch mindestens ein Parameter zu wählen, nämlich der Erwartungswert der Verteilung.

Vermutlich werden in der Praxis oft eher kleine Zahlen bei der Codierung von Häufigkeiten auftreten, also etwa kleiner als 100. Im folgenden sollen einige Wahrscheinlichkeitsverteilungen betrachtet, und die Eigenschaften verschiedener Codierschemata für diese Verteilungen untersucht werden. In Abbildung 5.6 sind die behandelten Verteilungen dargestellt. Alle Verteilungen in der Abbildung haben einen Erwartungswert von zehn.

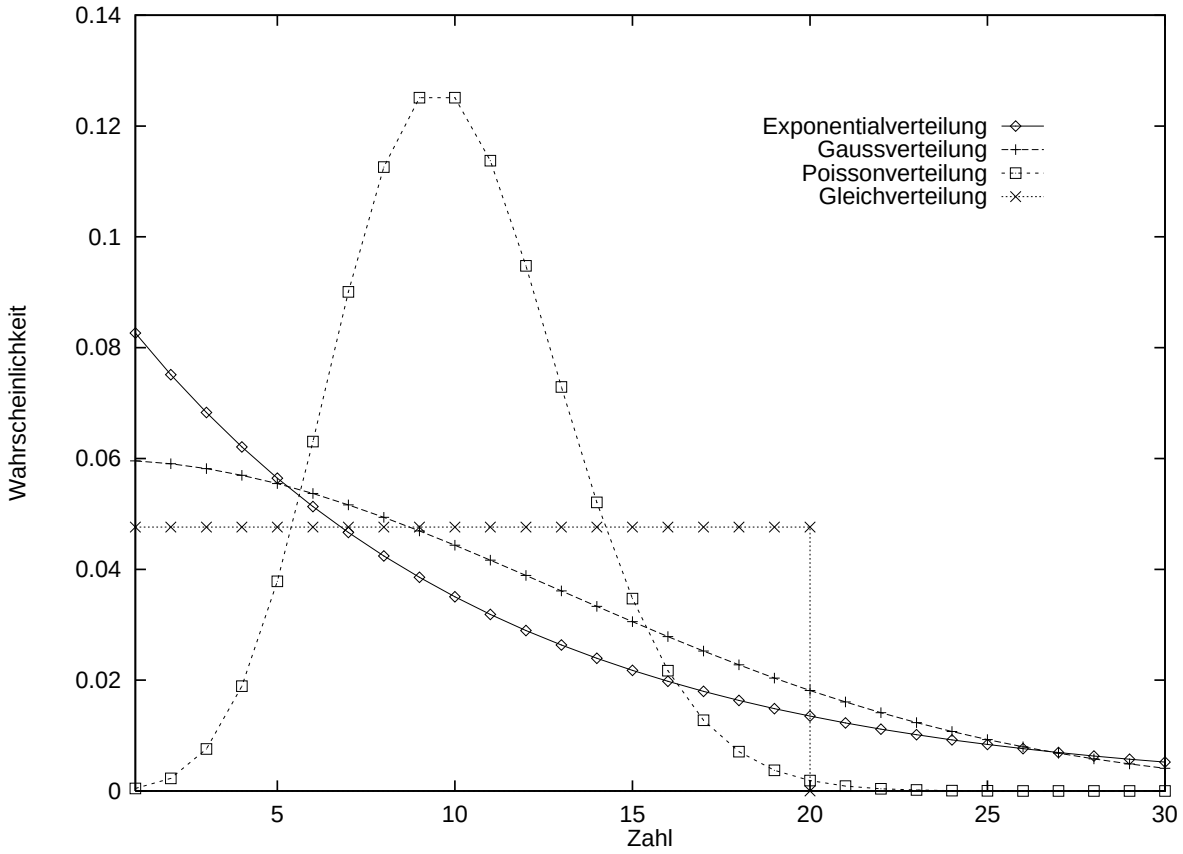


Abb. 5.6: Verschiedene Verteilungen im Vergleich

Theoretische Grenze

Für alle untersuchten Verteilungen wird als Referenz für die Effizienz der Codierverfahren die Shannon-Entropie nach Formel 5.1 für eine Verteilung mit der entsprechenden Charakteristik und dem Erwartungswert, welcher als Variable auf der Abszisse angetragen ist, verwendet. Die Grafik in Abbildung 5.7 zeigt die Entropie der behandelten Verteilungen und damit den Erwartungswert für die Bits als Funktion des Erwartungswertes der entsprechenden Verteilung. Je größer der Erwartungswert ist, umso breiter sind die Wahrscheinlichkeiten verteilt, und umso größer ist auch die Entropie. Es ist einleuchtend, dass eine Poissonverteilung, bei der sich die größten Wahrscheinlichkeitswerte nicht bei Null, sondern um den Erwartungswert konzentrieren bei größeren Erwartungswerten eine geringere Entropie aufweist, als eine Exponential- oder Gaußverteilung mit dem gleichen Erwartungswert. Als Vergleich ist noch eine Gleichverteilung aufgetragen. Für den Erwartungswert λ hat diese Verteilung im Bereich 0 bis 2λ den Wert $1/(2\lambda+1)$, sonst 0. Die

Entropie der Gleichverteilung ist anders als man zunächst vermuten würde geringer als die Entropie der Exponentialverteilung, da die Gleichverteilung eine obere Grenze hat, jenseits der die Verteilung Null ist. Die sich über einen viel breiteren Zahlenbereich erstreckende Kurve der Exponentialverteilung hat dagegen eine größere Entropie. Die Exponentialverteilung kann also für einen gegebenen Erwartungswert als besonders "ungünstige" Verteilung gelten, in dem Sinne, dass eine besonders hohe Zahl an Bits für die Codierung erforderlich ist.

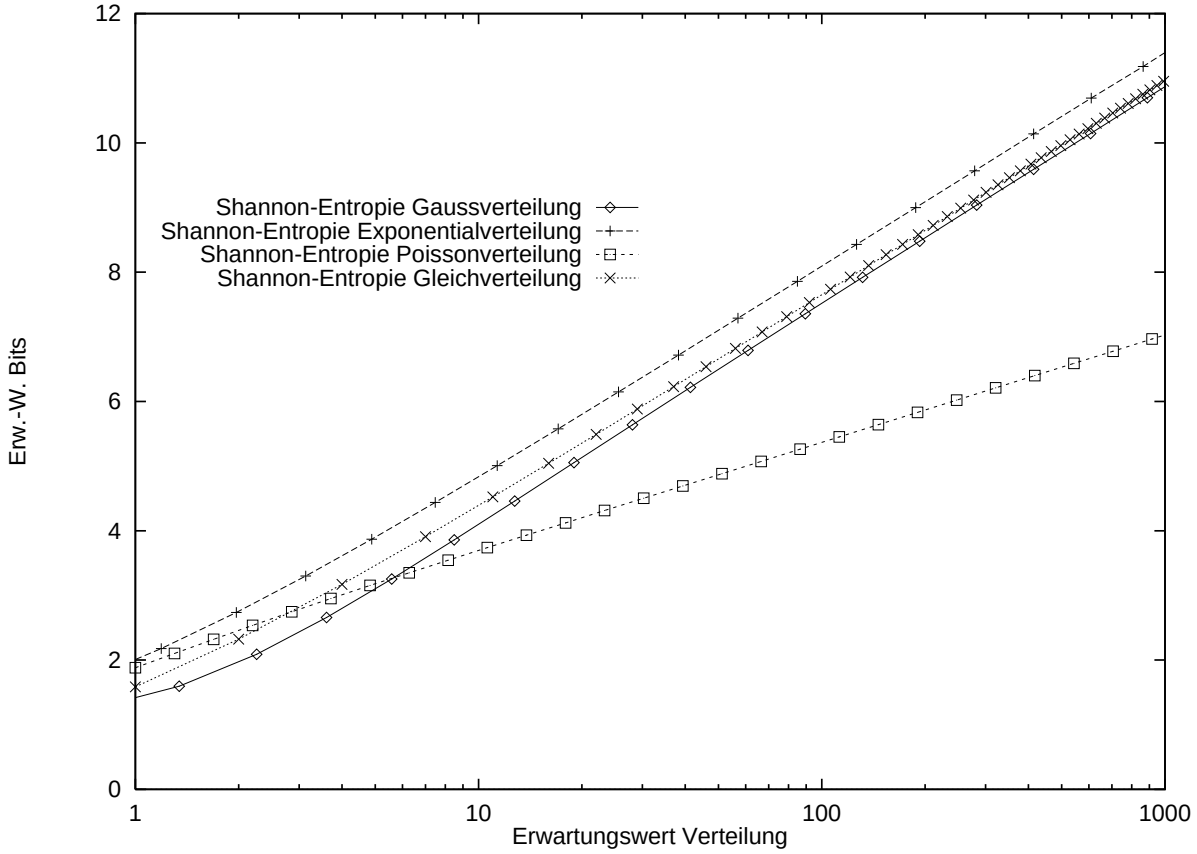


Abb. 5.7: Shannon-Entropie verschiedener Verteilungen

Exponentialverteilung

Da die Exponentialverteilung unter den betrachteten Verteilungen eine besonders hohe Entropie hat wird sie als kritischer Fall bei der Analyse vorrangig verwendet. Die Wahrscheinlichkeitsverteilung der Häufigkeit der Zahl der Instanzen folgt dabei einer geometrischen Reihe mit der Basis 0<a<1. Da die Summe aller Wahrscheinlichkeiten eins ergeben muss, und

$$S = \lim_{x\to\infty} \sum_{n=0}^{x} a^n = \lim_{x\to\infty} \frac{1-a^{x+1}}{1-a} = \frac{1}{1-a} \tag{5.3}$$

muss die geometrische Verteilung mit dem Faktor (1-a) normiert werden:

$$P[n] = (1-a)\cdot a^n; a<1 \tag{5.4}$$

Der Erwartungswert der Verteilung abhängig vom Parameter a ist:

$$E = \sum_n n \cdot P[n] = \sum_n n(1-a)a^n \tag{5.5}$$

Der Wert dieser Summe lässt sich einfach durch das Integral abschätzen Je näher der Wert von “a” an eins liegt, umso besser ist die Abschätzung.

$$E = \sum_{n=0}^{\infty} n \cdot (1-a)a^n \approx (1-a)\int_0^{\infty} xe^{\ln(a)\cdot x}dx = (1-a)\left|\frac{a^x(\ln(a)\cdot x - 1)}{\ln^2 a}\right|_0^{\infty} = \frac{1-a}{\ln^2 a} \tag{5.6}$$

Das Format für die Codierung von Häufigkeiten mit unbegrenzter oberer Schranke im BiM Algorithmus ist die Aneinanderreihung von Bitfeldern der Länge fünf. Das erste Bit signalisiert (wenn eins), dass noch ein weiteres Bitfeld folgt. Die übrigen vier Bits codieren die Zahl[2]. Im folgenden soll der Erwartungswert für die Zahl der Bits für die Codierung von Häufigkeiten berechnet werden, wenn obige Wahrscheinlichkeitsverteilung angenommen wird. Im speziellen soll untersucht werden, ob eine andere Aufteilung (etwa 1 Bit für die Signalisierung der Fortsetzung, aber nur 3 Bits für die Zahl) oder ein anderes Zahlenformat zu einer effizienteren Codierung von Häufigkeiten führt.

Der Erwartungswert für die Länge einer Zahl in Bits ist:

$$L = \sum_{n=0}^{\infty} Bits[n] \cdot P[n] \tag{5.7}$$

Bei der Codierung einer Zahl mit (k) Bitfeldern vorgegebener Länge zerfällt diese Summe in einzelne Bereiche, wobei die Zahl der Bits für einen Bereich in Abhängigkeit der Bitfeldlänge LB ist:

$$L_{Bitfeld}[k] = k \cdot LB \tag{5.8}$$

Der Bereich an Zahlen, die mit k Bitfeldern codiert werden ist:

$$Intervall = [2^{(k-1)\cdot(LB-1)}; 2^{k\cdot(LB-1)} - 1] \quad \text{für k > 1} \tag{5.9a}$$

$$Intervall = [0; 2^{k\cdot(LB-1)} - 1] \quad \text{für k = 1} \tag{5.9b}$$

Dabei ist berücksichtigt, dass nur LB-1 Bits eines Bitfeldes die Zahl codieren. Für Zahlen, die mit bis zu 5 Bits codiert werden müssen also die Wahrscheinlichkeiten der Zahlen von 0 bis 15 aufsummiert werden, für Zahlen mit einer Länge bis zu 10 Bits die Wahrscheinlichkeiten von 16 bis 255. Die Verteilung der Wahrscheinlichkeiten in einer geometrischen Reihe liefert einen einfachen Ausdruck für die Wahrscheinlichkeit, mit der die Zahl in einem bestimmten Bereich liegt:

[2] Die Flags zur Signalisierung der Zahl der Bitfelder werden im Bitstrom allerdings gesammelt und den Feldern, welche die Zahl codieren vorangestellt.

$$P_{Intervall} = (1-a)\cdot\sum_{n=x}^{y} a^n = (1-a)\cdot\frac{a^x - a^{y+1}}{1-a} = a^x - a^{y+1} \qquad (5.10)$$

Der Ausdruck

$$L = \sum_{k=1}^{\infty} L_{Bitfeld}[k]\left(a^x - a^{y+1}\right), \qquad (5.11)$$

wobei die untere und obere Grenze x und y eines Bereiches abhängig von k und der Länge eines Bitfeldes LB sind:

$$\begin{aligned} x &= 2^{(k-1)(LB-1)} && \text{für k>1} \\ x &= 0 && \text{für k=1} \\ y &= 2^{k(LB-1)} - 1 \end{aligned} \qquad (5.12)$$

gibt damit den Erwartungswert für die Länge einer Zahl in Bits an, die einer Exponentialverteilung mit dem Parameter a gehorcht.

Folgende Berechnung gibt den Erwartungswert für Bitfelder der Länge fünf für verschiedene Parameter a an, wobei die Beiträge der einzelnen Bereiche der Zahlen aufgeschlüsselt sind. In der ersten Zeile stehen Zahlen, die mit einem Bitfeld codiert werden, also Zahlen von 0 bis 15, in der zweiten Zeile Zahlen, die mit zwei Bitfeldern codiert werden, also 16 bis 255 usw. Der Erwartungswert der Verteilung für den entsprechenden Parameter "a" ist ebenfalls aufgelistet:

a:	0.50000	0.90000	0.95000	0.99000	0.99900	0.99990
Erw.-W:	1	9	19	99	999	9999
5 Bit:	4.99992	4.07349	2.79937	0.74271	0.07940	0.00799
10 Bit:	0.00015	1.85302	4.40125	7.75143	2.10077	0.23678
15 Bit:	0.00000	0.00000	0.00003	1.14472	11.36157	4.66232
20 Bit:	0.00000	0.00000	0.00000	0.00000	0.33210	13.24955
25 Bit:	0.00000	0.00000	0.00000	0.00000	0.00000	0.03561
30 Bit:	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
Summe:	5.00008	5.92651	7.20064	9.63886	13.87384	18.19226

Tab. 5.4: Verteilung der Bits für Bitfelder der Länge fünf

Wenn man die Länge der Bitfelder variiert ergibt sich folgendes Bild (aufgelistet ist nun die Zeile "Summe" aus der vorigen Aufstellung für verschiedene Bitfeldlängen):

a:	0.50000	0.90000	0.95000	0.99000	0.99900	0.99990
2 Bit:	**2.63284**	6.23477	8.10640	12.64780	19.26863	25.91012
3 Bit:	3.18755	**5.52774**	6.87648	10.24200	15.20316	20.18583
4 Bit:	4.01562	5.72658	**6.80378**	9.81666	14.18300	18.57821
5 Bit:	5.00008	5.92651	7.20064	**9.63886**	**13.87384**	18.19226
6 Bit:	6.00000	6.20602	7.16227	10.35009	13.96478	17.62327
7 Bit:	7.00000	7.00825	7.26267	10.67918	13.68206	18.60266
8 Bit:	8.00000	8.00001	8.01126	10.21001	15.03838	**17.45245**
9 Bit:	9.00000	9.00000	9.00002	9.68683	15.96639	17.78533

Tab. 5.5: Erwartungswert der Bits für verschiedene Bitfeldlängen

Das Minimum für jeden Parameter “a” ist fett markiert. Erwartungsgemäß sind kurze Bitfeldlängen günstig, wenn häufig kleine Zahlen vorkommen, also für Verteilungen mit kleinem Erwartungswert. In diesem Falle hätte eine Codierung mit 9 Bit (8 Bit für den Zahlenwert) eine erhebliche Zahl an ungenutzten Bits zur Folge: die Werte für a=0.5, 0.9 und 0.95 zeigen, dass bei der entsprechenden Verteilung praktisch alle vorkommenden Werte mit nur einem Bitfeld codiert werden. Für den Fall, dass der Erwartungswert der Verteilung eines Elements bekannt ist könnte man für dieses Element ein anderes Format als das in der BiM Codierung gewählte Bitfeld der Länge 5 wählen.

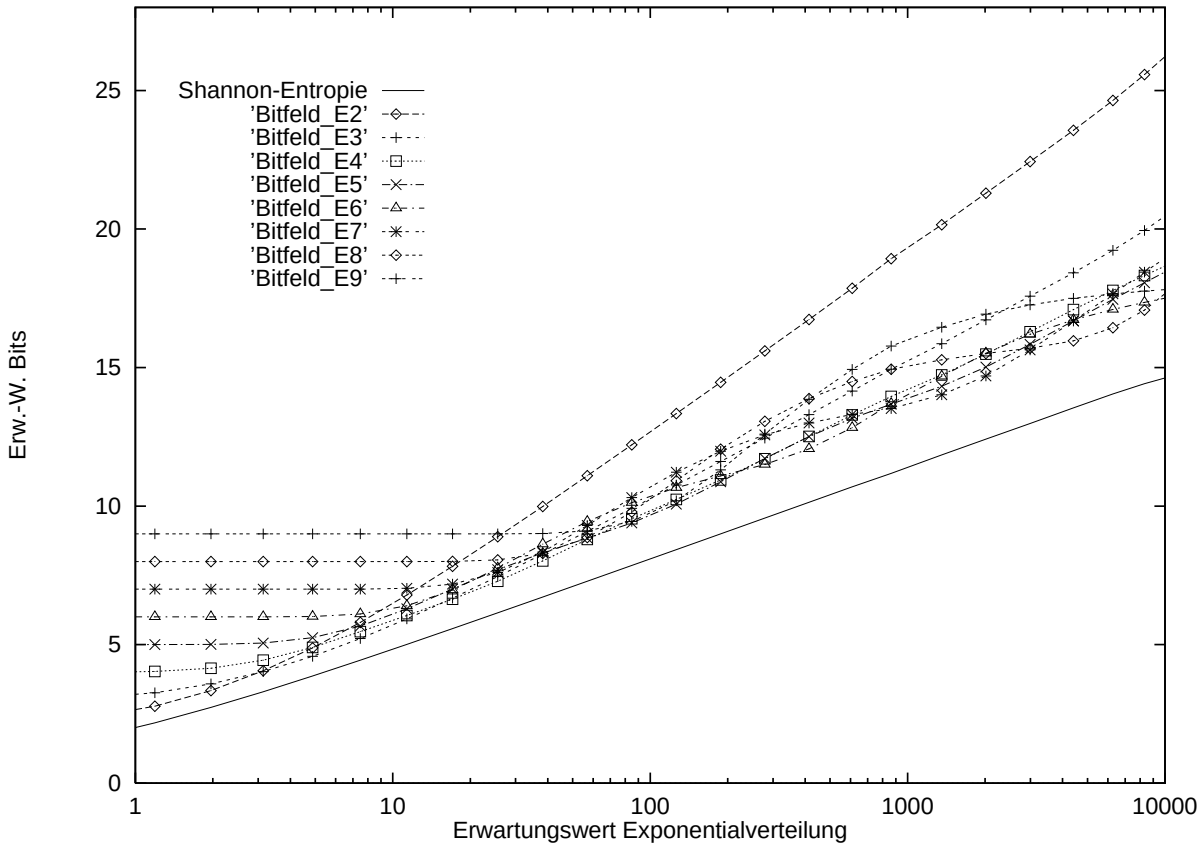


Abb. 5.8: Erwartungswert Bits für Codierung mit Bitfeldern

Abbildung 5.8 zeigt den Erwartungswert der Zahl der Bits für die Codierung einer Zahl als Funktion des Erwartungswertes der Exponentialverteilung für verschiedene Bitfeldlängen von 2 bis 9. Aus dem Verlauf der Kurven am linken Rand der Grafik geht hervor, dass eine Zahl natürlich nicht mit einer geringeren Zahl an Bit codiert werden kann, als der Länge eines Bitfeldes entspricht. Im Bereich bis zu einem Erwartungswert von etwa drei ist eine Bitfeldlänge von zwei am günstigsten. Dafür ist diese bei größeren Erwartungswerten deutlich ineffizienter als andere. Welches Codierschema im Bereich zwischen 1 und 10000 am günstigsten ist variiert abhängig vom genauen Erwartungswert. Die Codierung mit Bitfeldern der Länge fünf stellt aber einen guten Kompromiss dar.

Gibt es noch andere Formate für Zahlen, die eine noch kompaktere Darstellung ermöglichen? Dazu sollen nun noch drei Varianten untersucht werden.

Variable Bitfeldlänge

Eine Möglichkeit ist die Länge der Bitfelder variabel zu gestalten. Beispielsweise könnte man das erste Feld mit der Länge von zwei Bits codieren, das nächste Feld mit 3 Bits usw. Die Idee dabei ist, dass kleine Zahlen kompakt dargestellt werden, und bei größeren Zahlen der Überhang durch das Flag zur Signalisierung des Endes einer Zahl nicht übermäßig ins Gewicht fällt.

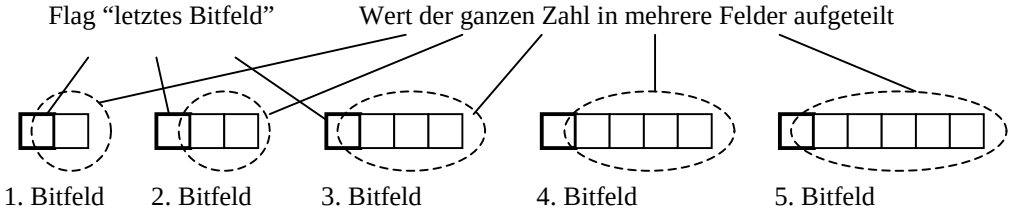


Abb. 5.9: Codierung mit variabler Bitfeldlänge

Die Auswertung dieses Codierschemas führt zu folgendem Ergebnis:

```
       a:  0.50000    0.90000    0.95000    0.99000    0.99900    0.99990
           ------------------------------------------------------------
Var Bits:  2.76       6.15       7.54      11.25      15.80      19.64
```

Wenn man dies mit der Tabelle 5.5 vergleicht stellt man fest, dass durch dieses Codierschema in weiten Bereichen der Erwartungswerte keine Verbesserung erzielt wird. Nur bei einem Parameter von a=0.5 ist diese Schema besser als das Schema mit einer festen Bitfeldlänge von fünf. Eine Abbildung für dieses Codierschema findet sich in Anhang A4.

Huffmancodierung für kleine Zahlen

Eine weitere Idee ist für einen begrenzten Bereich kleiner Zahlen eine Huffmancodierung anzuwenden. Zahlen die außerhalb dieses Bereichs liegen würden bei der Huffmancodierung durch einen Escapecode angekündigt. Der Wert einer solchen Zahl könnte wiederum mit einer entsprechenden Zahl von Bitfeldern einer vorgegebenen Länge codiert werden. Problematisch bei diesem Ansatz ist, dass die Codevergabe bei einer Huffmancodierung von den genauen Wahrscheinlichkeiten der einzelnen Alternativen abhängt. Die Codeworte müssten also auf die Wahrscheinlichkeitsverteilung und den entsprechenden Erwartungswert maßgeschneidert werden. Wenn die Zahl nicht der angenommenen Wahrscheinlichkeitsverteilung gehorcht hätte man schnell eine Verschlechterung der Codiereffizienz erreicht. Die Abbildung 5.10 zeigt eine Huffmancodierung, welche für Zahlen von 0 bis 14 einen VLC Code verwendet. Zahlen ab 15 werden signalisiert, indem ein Escapecode eine Bitfeldcodierung mit Feldlänge fünf ankündigt. Für große Erwartungswerte hat dieser Escapecode eine Länge von eins, weswegen das Codierschema in diesem Falle um ein Bit ineffizienter ist als eine reine Codierung mit Bitfeldern der Länge fünf. Bei geringen Erwartungswerten ist das Codierschema dagegen sehr effizient. In der Abbildung wird für jeden Wert des Erwartungswertes der Verteilung eine auf diese Verteilung angepasste Huffmancodierung verwendet. Die Kurve zeigt also nicht das Verhalten einer einzigen Codierung.

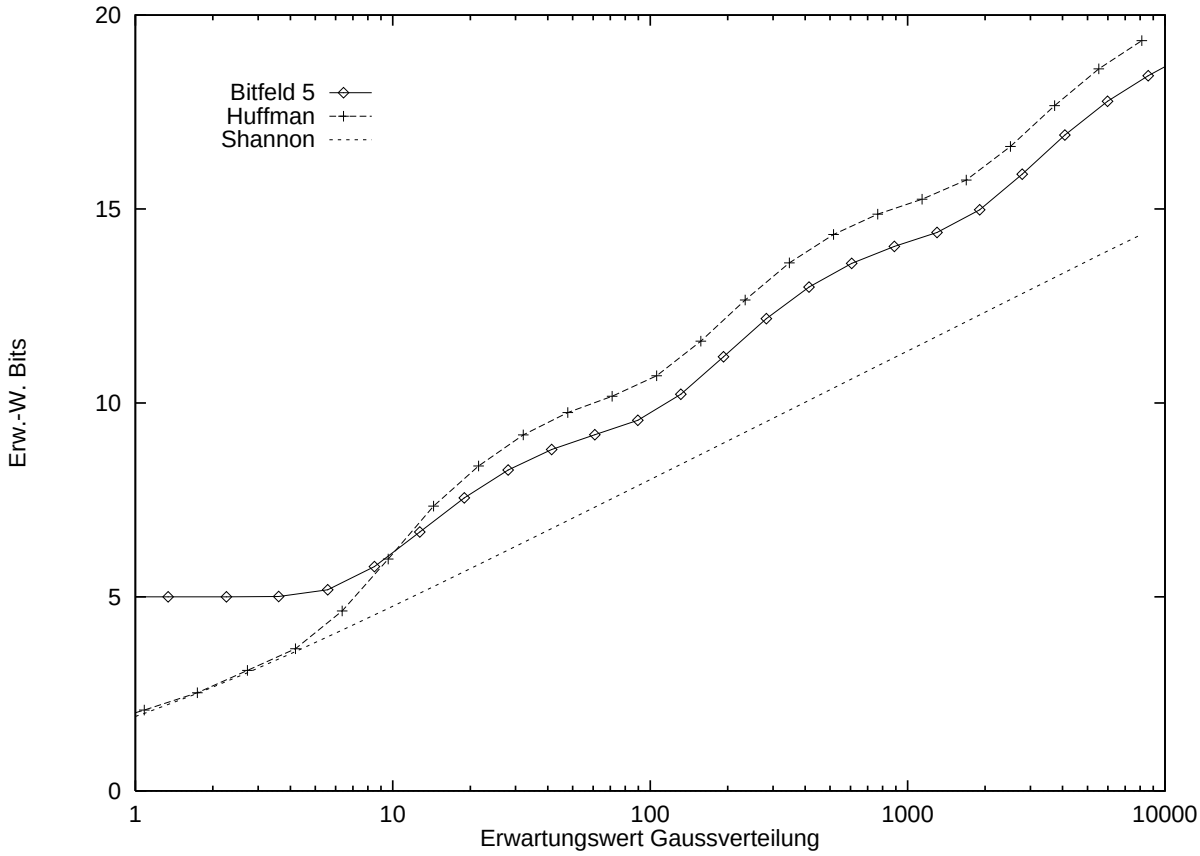


Abb. 5.10: Huffmancodierung für kleine Zahlen

Arithmetische Codierung

Für eine gegebene Wahrscheinlichkeitsverteilung lässt sich auch eine Arithmetische Codierung entwerfen. In Abbildung 5.11 ist ein Beispiel dargestellt mit einer Verteilung für die Auftrittswahrscheinlichkeit der Zahlen, die einer Exponentialverteilung nachempfunden ist. Je größer die Zahl ist, umso kleiner ist das dieser Zahl zugeordnete Wahrscheinlichkeitsintervall, und umso länger ist der Binärbruch der signalisiert werden muss, um in dieses Intervall zu treffen.

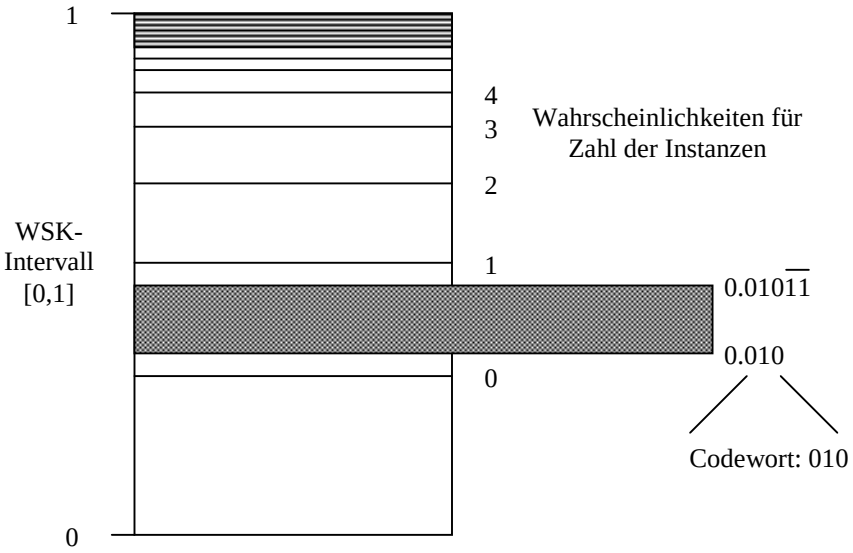


Abb. 5.11: Arithmetische Codierung für Häufigkeiten

Wie bei der Huffmancodierung muss die Arithmetische Codierung speziell für eine bestimmte Wahrscheinlichkeitsverteilung entworfen werden. Wenn die tatsächliche Verteilung mit der für die Berechnung der Codes verwendeten übereinstimmt erreicht man eine sehr effiziente Codierung, nahe an der Shannongrenze. Wenn jedoch die tatsächliche Verteilung abweicht sinkt die Effizienz. Wird beispielsweise eine Arithmetische Codierung für eine Exponentialverteilung mit geringem Erwartungswert entworfen, so haben große Zahlen sehr kleine Wahrscheinlichkeitsintervalle, die mit langen Codeworten codiert werden. Kommen in der tatsächlich auftretenden Verteilung große Zahlen häufiger vor so treiben diese langen Codeworte den Erwartungswert für die Zahl der Bits in die Höhe. Abbildung 5.12 vergleicht Arithmetische Codierungen, die für spezielle Erwartungswerte entworfen wurden mit einer Kurve, bei der die Arithmetische Codierung auf den betreffenden Erwartungswert der Exponentialverteilung optimiert wurde (Kurve "Erw.-Wert variabel"). Für jeden Punkt auf dieser Kurve wird also ein anderer Satz an Codeworten verwendet. Deutlich erkennt man die Abweichung vom Optimum, wenn der tatsächliche Erwartungswert größer oder kleiner ist, als der für die Berechnung der Arithmetischen Codierung verwendete. Nur an einem Punkt berührt die Kurve für jeden Erwartungswert die theoretisch für eine Arithmetische Codierung erreichbare Grenze, nämlich dann, wenn die Erwartungswerte übereinstimmen. Vor allem wenn der tatsächliche Erwartungswert größer ist als der für die Arithmetische Codierung verwendete steigt der Erwartungswert für die Zahl der Bits stark an.

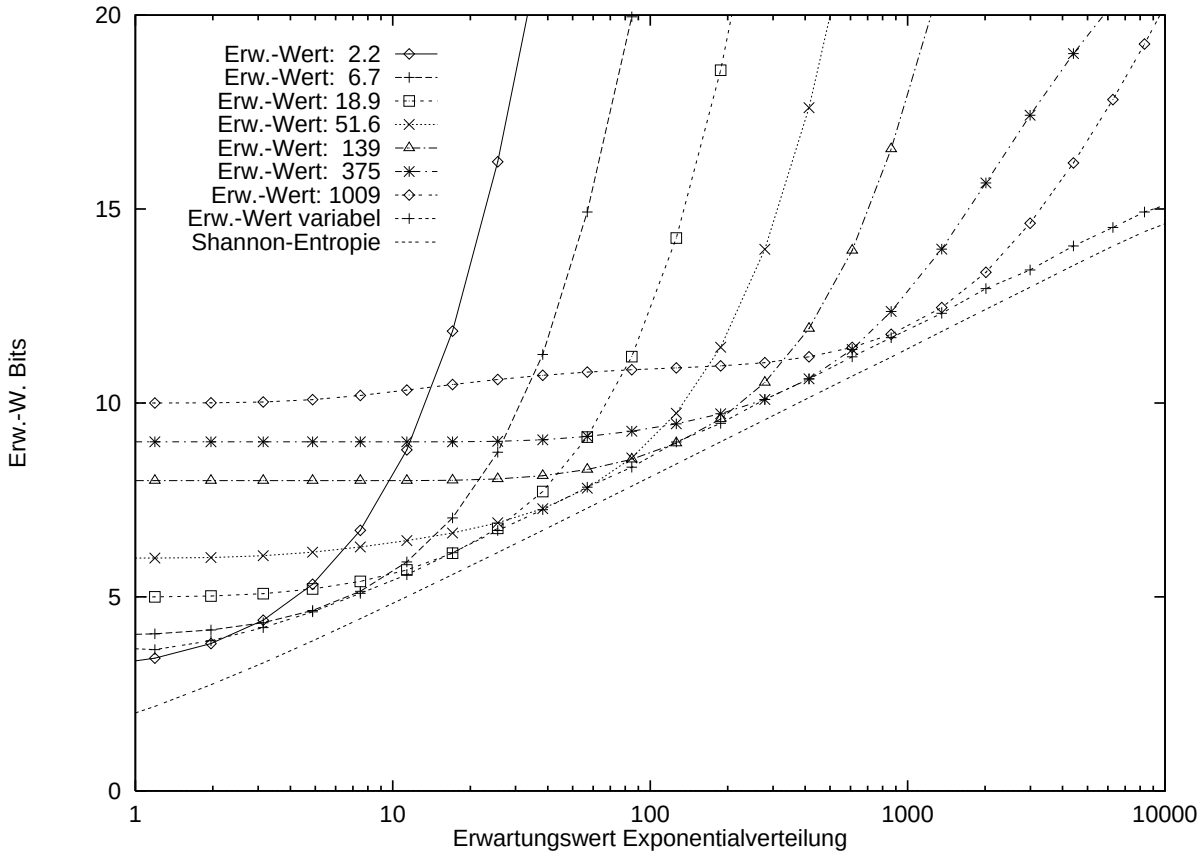


Abb. 5.12: Erwartungswert der Bits für die Arithmetische Codierung

Diese Überlegungen machen noch einmal das grundlegende Problem bei der Codierung von Häufigkeiten bei der syntaxbasierten Codierung deutlich: ohne die Kenntnis der exakten Wahrscheinlichkeitsverteilung für die Zahl der Instanzen eines speziellen Elements lässt sich

keine optimal effiziente Codierung entwerfen. Die Syntaxbasierte Codierung ermöglicht aber im Prinzip für verschiedene Elemente innerhalb des Schemas verschiedene speziell auf die Wahrscheinlichkeitsverteilung des entsprechenden Elements optimierte Codierungen zu verwenden.

Gaußverteilung

Da für Häufigkeiten nur Zahlen größer oder gleich Null vorkommen können wird nur der positive Ast der Gaußverteilung verwendet. Der Scheitel wird auf Null gesetzt. In diesem Fall hat man nur einen Parameter für die Verteilung, welcher den Erwartungswert bestimmt. Unter diesen Bedingungen hat die Gaußverteilung die Form:

$$P[n] = Fe^{-\lambda n^2}, n \geq 0 \tag{5.13}$$

Dabei bestimmt der Parameter λ den Erwartungswert, und der Parameter F normiert die Summe von Null bis Unendlich der Wahrscheinlichkeiten auf eins.

Wie verhalten sich nun die obigen Codierschemata bei einer Gaußverteilung? Eine numerische Auswertung der Codierung mit Bitfeldern verschiedener Länge und der Huffmancodierung zeigt den Erwartungswert für die Zahl an Bits. Die Abbildung 5.13, bei der die Gaußverteilung der Exponentialverteilung gegenübergestellt wird zeigt, dass keine wesentlichen Unterschiede bezüglich des Erwartungswertes für die Bits auftreten.

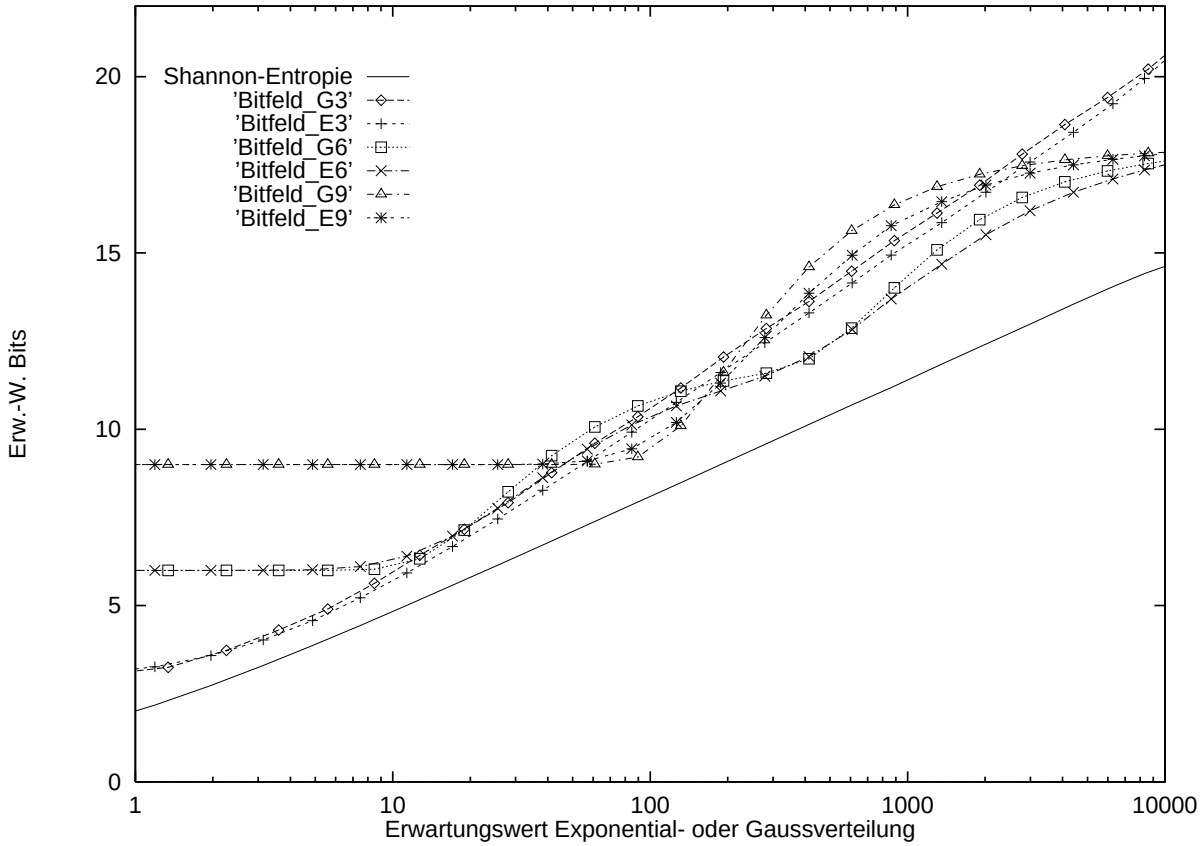


Abb. 5.13: Exponentialverteilung gegenüber Gaußverteilung

In Abbildung 5.13 steht in der Legende Bitfeld_G<x> für die Kurve der Bitfeldcodierung der Länge x bei einer Gaußverteilung, Bitfeld_E<x> entsprechend für eine Exponentialverteilung.

Poissonverteilung

Anders als bei der Exponential- oder Gaußverteilung konzentrieren sich bei einer Poissonverteilung die größten Wahrscheinlichkeitswerte nicht um Null, sondern um den Erwartung der Verteilung. Eine Zufallsgröße ist poissonverteilt, wenn gilt:

$$P[n] = \frac{\lambda^n}{n!} e^{-\lambda} \tag{5.14}$$

Dabei ist Lambda der Erwartungswert der Verteilung. Die Poissonverteilung kann unter gewissen Bedingungen als Annäherung an eine Binominalverteilung verwendet werden[3]. Das Verhalten der verschiedenen Bitfeldcodierungen bei einer Poissonverteilung ist in Abbildung 5.14 dargestellt. Man erkennt am Verlauf der Kurven das "Umschaltverhalten" klarer als bei den anderen Verteilungen: wenn der Erwartungswert der Verteilung in einen Bereich kommt, der nicht mehr mit einer gewissen Anzahl an Bitfeldern codiert werden kann wird die nächsthöhere Anzahl an Bitfeldern verwendet, und der Erwartungswert der Bits springt stufenförmig um die Größe des Bitfeldes. Der Abstand zur Shannon-Entropie ist größer als bei den anderen Verteilungen, da die Tatsache, dass die Zahl um den Erwartungswert gruppiert sein wird durch die Bitfeldcodierungen nicht ausgenutzt wird. Günstiger wäre hier die Codierung der Differenz zwischen der Zahl und dem Erwartungswert. Es würden dann zwar auch negative Zahlen auftreten, doch könnte man dies durch eine Zweierkomplement-Darstellung in das Schema der Bitfeldcodierung leicht integrieren.

[3] Beschrieben wird dadurch eine Stichprobe von Umfang N, bei der die Zufallsgröße ein Merkmal mit der Wahrscheinlichkeit p aufweist. Die Wahrscheinlichkeit n Zufallsgrößen mit dem Merkmal in der Stichprobe zu finden ist etwa Poissonverteilt, wenn N groß und p klein ist. Der Erwartungswert Lambda ist dann das Produkt aus N und p

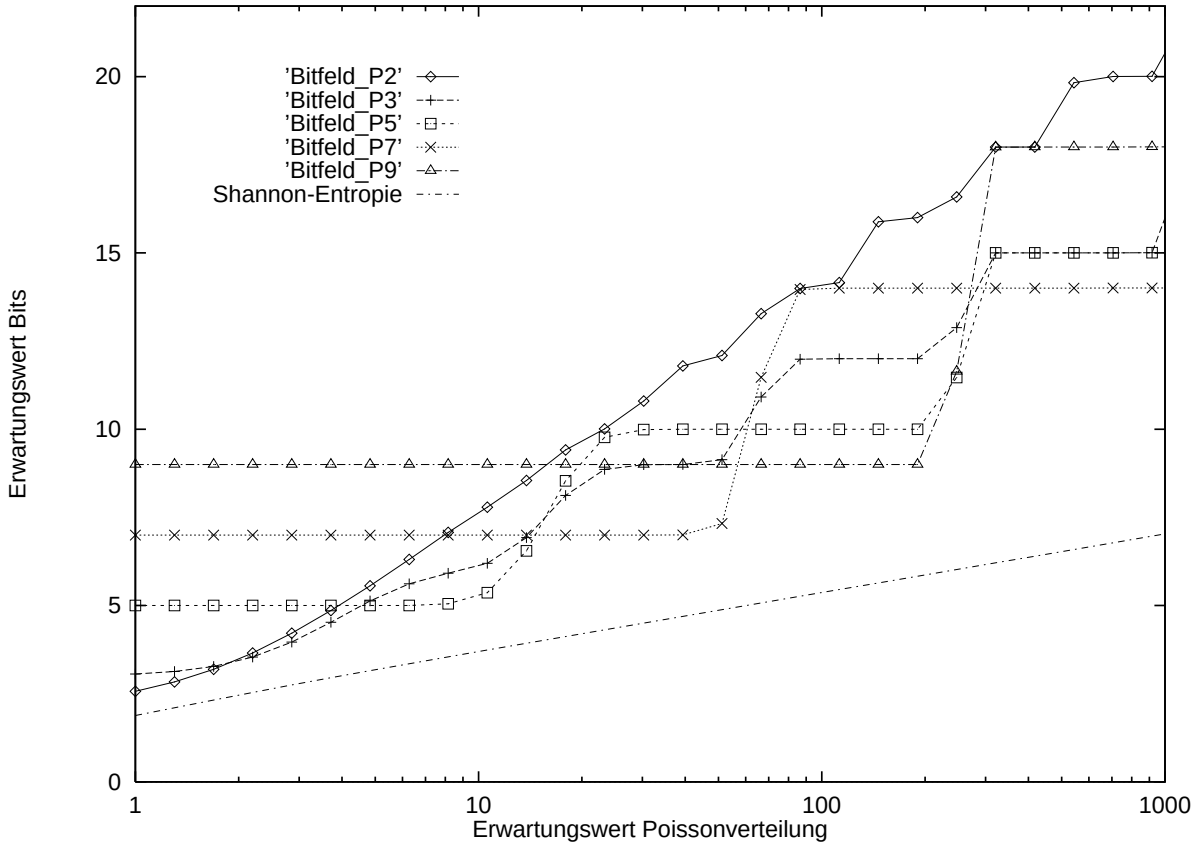


Abb. 5.14: Bitfeldcodierung für Poissonverteilungen

Fazit

Das grundlegende Problem bei der Codierung von Häufigkeiten mit unbegrenzter oberer Schranke ist, dass es nicht möglich ist eine optimierte Codierung für diesen Fall zu verwenden wenn über die Wahrscheinlichkeitsverteilung nichts bekannt ist, weder die Charakteristik der Verteilung noch der Erwartungswert. Durch die Syntaxbasierte Codierung ist für jeden einzelnen Fall, in dem eine solche unbegrenzte Häufigkeit codiert werden muss jedoch bekannt welcher Sachverhalt beschrieben wird. Daraus lässt sich eventuell eine Verteilung und ein Erwartungswert für die Häufigkeit ableiten oder experimentell bestimmen, wenn man gleichzeitig den Kontext kennt in dem die Codierung verwendet wird (beispielsweise weil man der Medienfirma angehört, welche die Beschreibungen generiert). Wenn sich Encoder und Decoder über das Format einigen (entweder durch unabhängig von der Codierung ausgetauschte Konfigurations-Scripts, oder weil man in der Decoderinitialisierung vor einem Dokument die entsprechenden Informationen überträgt), könnte man deshalb für jede einzelne unbegrenzte Häufigkeit ein angepasstes Codierverfahren verwenden. Für die Auswahl eines optimierten Codierverfahrens ist die Kenntnis des Erwartungswerts wichtiger als die Charakteristik der Verteilung. Zusätzliches Vorwissen, beispielsweise dass nur gerade Zahlen auftreten, könnte man natürlich ebenfalls berücksichtigen.

Wenn keine Wahrscheinlichkeitsverteilung bekannt ist, oder keine Verständigung über das Codierverfahren möglich ist sollte man eine Codierung verwenden, welche über einen breiten Bereich an Verteilungen und Erwartungswerten gleichmäßig gute Ergebnisse erzielt. Das im BiM-Algorithmus vorgesehene Verfahren der Bitfeldcodierung mit Länge fünf kann durchaus als solcher Kompromiss gelten, wie die Simulationen gezeigt haben.

Kapitel 6 – Ein optimiertes Codec-Modell

Neben dem Algorithmus für die Codierung von Daten, durch welchen Eigenschaften wie Kompressionsrate, Bitstromsyntax oder Verhalten bei statistischen Besonderheiten des Quellenmaterials festgelegt sind, ist auch die Implementierbarkeit eines solchen Verfahrens von Interesse.

In diesem Kapitel wird ein optimiertes Encoder- Decodermodell für das in Kapitel 4 vorgestellte Verfahren zur Codierung von strukturierten Dokumenten entwickelt. Dabei wurde sowohl auf die Geschwindigkeit des Codiervorgangs als auch auf die Effizienz der Speichernutzung wert gelegt.

Das Modell bietet auch eine Grundlage für eine allgemeine Abschätzung der Komplexität des Algorithmus. Dies wird in Kapitel 7 ausführlicher diskutiert.

6.1 Potential für Optimierungen

Bei der Betrachtung der beiden Kernbestandteile des BiM Verfahrens, der Pfad- und der Payloadcodierung fällt auf, dass beide ihre Codes aus dem XML Schema ableiten, wenn sich auch die genauen Werte der Codes unterscheiden. Es wäre deshalb günstig, eine gemeinsame Datenstruktur für diese Schemainformation zu entwerfen, welche von beiden Verfahren gleichermaßen genutzt werden kann. Dadurch könnte es vermieden werden, dass die Schemainformation unnötigerweise doppelt im Codec vorhanden ist.

In einem syntaxbasierten Codierverfahren ist ein Arbeitsgang erforderlich, welcher die Codes der Datenelemente aus dem Schema bestimmt. Um den Aufwand beim eigentlichen Encodieren oder Decodieren so gering wie möglich zu halten bietet es sich deshalb an, diesen Schritt in einen Kompilationsprozess auszulagern. Dieser erzeugt aus dem textuellen XML Schema ein Zwischenformat, das den eigentlichen Codiervorgang möglichst gut vorbereitet, und so für eine schnelle Ausführung der Codierung und Decodierung sorgt.

Je nach Schwerpunkt der Optimierung wird man einen Kompromiss aus möglichst schneller Ausführung des Programms und möglichst speichereffizienter Darstellung der Schemainformation suchen. Der Grundkonflikt ist, dass diejenigen Arbeitsschritte, welche in den Kompilationsprozess ausgelagert werden, und deshalb die Geschwindigkeit der Codierung erhöhen im Normalfall Daten produzieren, welche im Zwischenformat des Schemas gespeichert werden müssen.

6.2 Das Bytecodemodell

6.2.1 Grundkonzept

Das in dieser Arbeit entworfene Modell wird im folgenden Bytecodemodell genannt, in Anlehnung an die Programmiersprache Java, bei der ebenfalls ein textuelles JAVA Programm in Bytecode übersetzt wird, welcher dann von einem Interpreter, der Java Virtual Machine, ausgeführt wird. Die Grundidee des Java Bytecodes ist die, dass man das Programm in eine Form bringen will, die einfacher ausgeführt werden kann, als das ursprüngliche Programm, welches in einer Hochsprache formuliert ist. Interpretersprachen wie BASIC in seiner ursprünglichen Form sind so aufgebaut: der BASIC Quelltext wird direkt gelesen und die

Befehle vom BASIC Interpreter ausgeführt. Andererseits soll das Format nicht an einen speziellen Prozessor gebunden sein, wie es bei einem Kompilat von Hochsprachen wie C der Fall ist.

Das Bytecodemodell für den BiM Algorithmus gliedert sich in zwei Teile. Der eine erzeugt in einem Kompilationsprozess aus dem textuellen XML Schema den Bytecode, welcher alle relevanten Schemainformationen sowohl für die Pfadcodierung als auch für die Payloadcodierung enthält. Ein Interpreter der die Schemainformation im Bytecode lesen kann übernimmt die Encodierung und Decodierung von XML Dokumenten. Folgende Arbeitsschritte müssen von dem Bytecodewerkzeug ausgeführt werden können:

- Kompilieren eines textuellen Schemas in das Bytecodeformat
- Encodieren einer Payload: es wird der Bytecode für das in dem Dokument verwendete Schema geladen, und das textuelle Dokument in das Binärformat codiert.
- Decodieren einer Payload: der Bytecode und das Binärformat des Dokuments werden geladen, der binäre Strom decodiert und das decodierte Dokument (z.B. im Textformat) ausgegeben.
- Encodieren eines Pfades (absolut oder relativ): der Bytecode und der Pfad in einem textuellen Format werden geladen, ein binärer Pfad ausgegeben.
- Decodieren eines Pfades (absolut oder relativ): der Bytecode wird geladen, und der binäre Pfad decodiert und in einem Textformat ausgegeben.

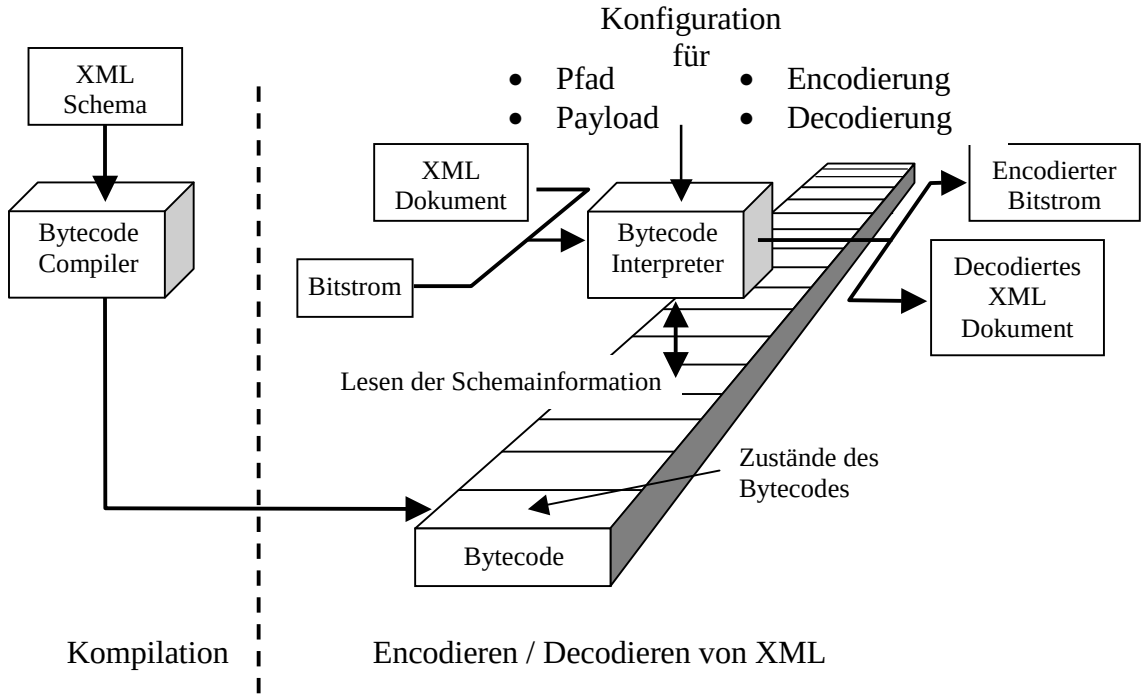


Abb. 6.1: Grundprinzip des Codecmodells

6.2.2 Anforderungen an den Bytecode

Die Anforderungen, welche beim Entwurf des Bytecodes berücksichtigt werden müssen sind:

- Alle Funktionen, die im BiM-Verfahren vorgesehen sind müssen unterstützt werden
- Der Bytecode soll dazu geeignet sein vorkompiliert zu werden. Zum Encodieren oder Decodieren wird nur der Bytecode geladen.
- Der Bytecode muss die Informationen enthalten, welche sowohl die Pfadcodierung als auch die Payloadcodierung benötigen
- Der Bytecode sollte modular aufgebaut sein, d.h. auch funktionsfähig sein, wenn nicht das gesamte Schema in den Codec geladen wird

6.2.3 Aufbau des Bytecode

Das *Bytecodeformat* besteht aus verschiedenen Typen von Zuständen, durch deren Vernetzung die Struktur des XML Schemas nachgebildet werden kann. Jeder Zustand ist aus einer ganzen Anzahl von Bytes aufgebaut, die Zustände haben aber eine unterschiedliche Länge. Auch die Zustände eines Typs haben keine feste Länge, da unter Umständen eine vom Schema abhängige Anzahl an Datenfeldern aufgenommen werden muss. Das erste Byte jedes Zustands ist ein Header, welcher den Typ des Zustands identifiziert (da dafür nicht alle 8 Bit benötigt werden sind im Header auch noch einige Flags untergebracht, die optionale Informationsfelder oder verschiedene Varianten der Zustandstypen anzeigen). Jeder Zustand wird an einer bestimmten Adresse im Bytecodevektor abgelegt. Der Bytecodevektor ist eine Datenstruktur bestehend aus aufeinanderfolgenden Bytes. Die Zustände werden durch Zeiger, welche Adressen in diesem Bytecodevektor enthalten miteinander verknüpft. Mit folgendem Satz an Zuständen können die XML Syntaxkonstrukte abgebildet werden.

- Kopfzustand eines Typs
- Elementzustand
- Attributzustand
- Strukturzustand (für Auswahl-, Sequenz- und all-Gruppen)
- Häufigkeitszustand
- Endzustand eines Typs

Außer den Zuständen selbst müssen noch alle Namen, die im Schema auftreten (z.B. Typnamen, oder die Namen von Elementen oder Attributen) eine Repräsentation erhalten. Da das direkte Einsetzen der Namen in den Zuständen aufgrund der unterschiedlichen Länge die Bestimmung der Größe eines Bytecodezustands erschweren würde, werden alle Namen in einer zentralen Datenstruktur, dem Stringvektor, gesammelt und von den Bytecodezuständen mit Zeigern referenziert.

Neben den Zuständen und dem Stringvektor gibt es noch eine dritte Datenstruktur. Diese beinhaltet den Vererbungsbaum, der für die Unterstützung von Polymorphismus erforderlich ist.

6.2.4 Die Zustände

Im folgenden Abschnitt wird der Aufbau der einzelnen Zustandstypen beschrieben. Die grundlegende Funktion wird kurz erklärt. Die Details werden später bei der Beschreibung der En- und Decodierung durch den Bytecodeinterpreter noch klarer werden. Die Tabellen zu jedem Zustandstyp enthalten eine detaillierte Aufschlüsselung der einzelnen Informationsfelder des Typs.

Angegeben sind :

- die Länge des Informationsfeldes in Bytes. Ein "+" bedeutet dabei, dass das Datum mit einer variablen Anzahl an Bytes codiert ist. Wenn das MSB eines Bytes "1" ist folgt jeweils noch ein weiteres Byte, das zu dem Informationsfeld gehört.
- der Datentyp, dabei ist:

- ZB: Zeiger auf eine Adresse im Bytecodevektor
- ZS: Zeiger auf einen Namen im Stringvektor
- ZV: Zeiger auf einen Zustand im Vererbungsbaum
- ZC: Zeiger auf einen Codec für spezielle Datentypen
- PI: positive Ganze Zahl (Integer)
- BF: Bitfeld (z.B. für Flags)

- die Funktion

Kopfzustand eines Typen (Variante für *complexTypes*)

Der Kopfzustand eines Typen ist der Einsprungpunkt, wenn ein Datenelement von diesem Typ codiert oder decodiert werden soll, und enthält einige wichtige Informationsfelder. Der Name des Typs steht durch einen Zeiger auf den Stringvektor zur Verfügung. Der Zeiger auf die Liste der Attribute referenziert eine Bytecodestruktur (Sequenz), welche alle Attribute dieses Typs enthält. Sie ist für die Behandlung der Payloadcodierung erforderlich. Der Zeiger auf den Basistyp referenziert den Kopfzustand eines anderen Typen, wenn der Typ von jenem vererbt wurde. Dieses Datenfeld ist nur vorhanden, wenn es durch ein Flag im Header angezeigt wird. Der Zeiger auf den Vererbungsbaum ist immer vorhanden. Er gibt jene Stelle im Vererbungsbaum an, an der die Information über die Länge des Typecode und die von diesem Typ vererbten Typen abgelegt ist. Die Anzahl der Kinder von komplexem und einfachem Typ wird bei der Pfadcodierung für die Bestimmung der Länge der Schema Branch Codes benötigt. Der letzte Eintrag ist ein Zeiger auf den ersten Zustand des Typen, an dem mit der Codierung oder Decodierung begonnen wird.

Länge	Typ	Funtion
1	BF	Header
2	ZS	Zeiger auf Name in Stringvektor
2	ZB	Zeiger auf Liste der Attribute
2	ZB	Zeiger auf Basistyp
2	ZV	Zeiger auf Vererbungsbaum
1	PI	Länge der des Codes für Context-BVC
1	PI	Länge der des Codes für Operand-BVC
1	ZB	Zeiger auf ersten Zustand des Typs

Tab. 6.1: Kopfzustand eines Typs

Headerbitfeld

B8	B7	B6	B5	B4	0	0	1

Das Headerbitfeld enthält in den untersten drei Bits den Identifizierungscode für Kopfzustände eines Typen (001). Die Bedeutung der weiteren Bits sind:

- B4: Wert 1 => Typ ist abgeleitet, deshalb Zeiger auf Basistyp vorhanden
- B5: Wert 1 => Typ verwendet globale Postitionscodes, 0 => lokale Positionscodes
- B6: Zeigt simple Type an, für complexType Null
- B7, B8: unbenutzt

Kopfzustand eines Typs (Variante für simpleTypes)

Elemente von simpleType dienen bei der Codierung dazu die Nutzinformation aufzunehmen. Dafür enthält dieser Typ einen Zeiger auf einen spezifischen Codec, der Inhalt von entsprechendem Typ codieren kann. Wenn kein spezifischer Codec zur Verfügung steht kann der Inhalt aber auch immer als Zeichenkette (String) behandelt werden. Wenn Inhalt in Form von Zeichenketten codiert werden kann wird dies mit einem Flag angezeigt.

Länge	Typ	Funtion
1	BF	Header
2	ZS	Zeiger auf Name in Stringvektor
2	ZC	Zeiger auf Codec für Inhalt des simpleType
2	ZV	Zeiger auf Vererbungsbaum

Tab. 6.2: Kopfzustand eines Typs, Variante für simpleType

Headerbitfeld

B8	B7	B6	B5	B4	0	0	1

Die untersten drei Bits identifizieren wieder den Kopfzustand eines Typen (001). Die Bedeutung der weiteren Bits sind:

- B4: Wert 1 => Typ ist abgeleitet, deshalb Zeiger auf Basistyp vorhanden
- B5: unbenutzt
- B6: Zeigt simple Type an, Wert ist 1
- B7: Typ hat Inhalt
- B8: unbenutzt

Elementzustand

Der Elementzustand bildet eine Elementdeklaration im Schema ab. Neben dem Header ist noch ein weiteres Bitfeld erforderlich, in dem optionale Informationsfelder angezeigt werden. Ein Zeiger auf den Stringvektor referenziert den Namen des Elements. Ein Zeiger auf den Typ des Elements referenziert einen Kopfzustand eines Typs. Mit seiner Hilfe wird festgestellt welcher Typ in der nächsten Hierarchiestufe des Dokument die Codierung oder Decodierung kontrolliert. Die Zahl *Länge des Positionscodes* gibt an, mit wie vielen Bits der Positionscode bei der Pfadcodierung im Bitstrom codiert ist. Der Zeiger auf default/fixed Werte ist nur vorhanden, wenn es im Bitfeld angezeigt ist. Ebenso die Anzahl der Substitution Group Mitglieder. Der Zeiger auf den Folgezustand gibt jene Stelle im Bytecode an, wo nach dem Elementzustand fortgefahren werden soll.

Länge	Typ	Funktion
1	BF	Header
2	ZS	Zeiger auf Name in Stringvektor
2	ZB	Zeiger auf Typ des Elements
1	PI	Länge des Positionscodes
2	ZS	Zeiger auf default / fixed Werte
1+	PI	Anzahl Substitution Group Mitglieder
2	ZB	Zeiger auf Folgezustand

Tab. 6.3: Elementzustand

Headerbitfeld

B8	B7	B6	B5	B4	0	1	0

Die untersten drei Bits identifizieren einen Elementzustand (010). Die Bedeutung der weiteren Bits sind:

- B4: Element ist Headelement einer Substitution Group
- B5: Element ist nillable => Modifikation beim Lesen
- B6: Element ist fixed => Zeiger auf fixed Wert vorhanden
- B7: Element hat default Wert => Zeiger auf default Wert vorhanden
- B8: unbenutzt

Attributzustand

Der Attributzustand bildet eine Attributdeklaration im Schema ab. Ebenso wie beim Elementzustand werden Name und Typ über entsprechende Zeiger referenziert. Die default/fixed Werte sind wiederum optional, und werden durch ein Flag im Header angezeigt.

Länge	Typ	Funktion
1	BF	Header
2	ZS	Zeiger auf Name in Stringvektor
2	ZB	Zeiger auf den Typ des Attributs
2	ZS	Zeiger auf default / fixed Werte

Tab. 6.4: Attributzustand

Headerbitfeld

B8	B7	B6	B5	B4	0	1	1

Die untersten drei Bits identifizieren einen Attributzustand (011). Die Bedeutung der weiteren Bits sind:

- B4: Attribut ist optional
- B5: Attribut hat einen fixed Wert => Zeiger auf fixed Wert vorhanden
- B6: Attribut hat default Wert => Zeiger auf default Wert vorhanden
- B7, B8: unbenutzt

Strukturzustand (Variante Auswahl Startzustand)

Dieser Zustand bildet eine Auswahlgruppe im Schema nach. Die Anzahl der Zweige der Auswahl ist nach dem Header abgelegt, und bestimmt die Länge des Zustandes. Es folgt eine Liste mit Zeigern auf den jeweils ersten Zustand der Alternativen der Auswahl. Wenn der Bytecodeinterpreter auf einen Auswahlzustand trifft muss bei der Payloadcodierung ein Codewort aus dem Bitstrom gelesen werden, um den richtigen Zweig für die weitere Decodierung zu ermitteln.

Länge	Typ	Funktion
1	BF	Header
1+	PI	Anzahl der Zweige der Auswahl (=n)
2	ZB	Zeiger auf Zustand 1 der Auswahl
...	...	...
2	ZB	Zeiger auf Zustand n der Auswahl

Tab. 6.5: Strukturzustand, Auswahl Start

Headerbitfeld

B8	B7	B6	B5	B4	1	0	0

Die untersten drei Bits identifizieren einen Strukturzustand (100). Die Bedeutung der weiteren Bits sind:

- B4, B5: 0,0 für Auswahl Startzustand
- B6-B8: unbenutzt

Strukturzustand (Variante Auswahl Endzustand)

Dieser Zustand fasst die einzelnen Zweige einer Auswahlgruppe wieder zusammen, und enthält eine Referenz auf den der Auswahl folgenden Zustand. Für die Encodierung ist auch ein Zeiger auf den Startzustand der Auswahl erforderlich, damit weitere Zweige überprüft werden können, wenn das gesuchte Element in einem Zweig nicht gefunden wurde.

Länge	Typ	Funktion
1	BF	Header
2	ZB	Zeiger auf Startzustand der Auswahl
2	ZB	Zeiger auf Folgezustand

Tab. 6.6: Strukturzustand, Auswahl Ende

Headerbitfeld

B8	B7	B6	B5	B4	1	0	0

Die untersten drei Bits identifizieren einen Auswahlzustand (100). Die Bedeutung der weiteren Bits sind:

- B4, B5: 1,0 für Auswahl Endzustand
- B6-B8: unbenutzt

Strukturzustand (Variante Sequenz)

Dieser Zustand wird ausschließlich bei den Attributlisten für die Payloadcodierung verwendet, der in den Kopfzuständen eines Typs referenziert wird. Das Syntaxelement „sequence“ das innerhalb von Typdefinitionen auftritt wird einfach durch die Verkettung von Zuständen mit dem Zeiger auf den Folgezustand im Bytecode umgesetzt.

Länge	Typ	Funktion
1	BF	Header
1+	PI	Anzahl der Zustände der Sequenz (=n)
2	ZB	Zeiger auf Zustand 1 der Sequenz
...	...	...
2	ZB	Zeiger auf Zustand n der Sequenz

Tab. 6.7: Strukturzustand, Sequenz

Headerbitfeld

B8	B7	B6	B5	B4	1	0	0

Die untersten drei Bits identifizieren einen Strukturzustand (100). Die Bedeutung der weiteren Bits sind:

- B4,B5: 0,1 Für Sequenzzustand
- B6-B8: unbenutzt

Strukturzustand (Variante für all-Gruppen)

Dieser Zustand wird für die Abbildung von all-Gruppen verwendet. Dabei wird eine all-Gruppe in genau einen der hier beschriebenen Strukturzustände übersetzt. Eine Transformation in einen Baum aus Auswahlzuständen (vgl. 4.4.2) ist nicht erforderlich, da nur die Information über die enthaltenen Alternativen zur Verfügung gestellt werden muss. Die Umsetzung der Codestruktur erfolgt durch den Bytecodeinterpreter).

Länge	Typ	Funktion
1	BF	Header
1+	PI	Anzahl der Zweige der all-Gruppe (=n)
2	ZB	Zeiger auf Zustand 1 der all-Gruppe
...	...	...
2	ZB	Zeiger auf Zustand n der all-Gruppe

Tab. 6.8: Strukturzustand für all-Gruppen

Headerbitfeld

B8	B7	B6	B5	B4	1	0	0

Die untersten drei Bits identifizieren einen Strukturzustand (100). Die Bedeutung der weiteren Bits sind:

- B4,B5: 1,1 für all-Gruppe
- B6-B8: unbenutzt

Häufigkeitszustand

Der Häufigkeitszustand wird angelegt, wenn ein Element oder eine Gruppe mehrfach auftreten kann, oder optional ist. Dies wird im textuellen Schema durch die Attribute “minOccurs” und “maxOccurs” angezeigt. Das Element oder die Gruppe wird im Zeiger auf den Inhalt des Häufigkeitszustand referenziert. Wenn der Bytecodeinterpreter auf einen Häufigkeitszustand trifft, muss der Bitstrom gelesen werden, um festzustellen, ob das Element (ein weiteres Mal) auftritt oder nicht. Falls nicht, wird in den Zustand verzweigt, der im Zeiger auf den Folgezustand referenziert wird.

Länge	Typ	Funktion
1	BF	Header
2	ZB	Zeiger auf Inhalt des Occurrencezustands
2	ZB	Zeiger auf Folgezustand
1+	PI	Wert von „minOccurs“
1+	PI	Wert von „maxOccurs“

Tab. 6.9: Häufigkeitszustand

Headerbitfeld

B8	B7	B6	B5	B4	1	0	1

Die untersten drei Bits identifizieren den Häufigkeitszustand. Die weiteren Bits sind unbenutzt.

Endzustand eines Typs

Wird beim decodieren einer Payload der Endzustand eines Typs erreicht, so zeigt dies dem Bytecodeinterpreter an, dass entweder der aktuelle Typ in der Liste der Basistypen eines Typs beendet ist, und mit dem nächsten fortzufahren ist, oder (wenn diese Liste leer ist) dass die aktuelle Hierarchiestufe beendet ist, und mit dem aufrufenden Elementzustand fortgefahren werden muss. Der Endzustand enthält außerdem eine Liste mit Zeigern auf die eigenen Attribute dieser Typdefinition. Beim Decodieren eines Pfades wird diese Liste beim Abzählen der Kindelemente des Typs verwendet.

Länge	Typ	Funktion
1	BF	Header
1+	PI	Anzahl der Attribute des Typs (=n)
2	ZB	Zeiger auf Attribut 1
...	...	...
2	ZB	Zeiger auf Attribut n

Tab. 6.10: Endzustand eines Typs

Headerbitfeld

B8	B7	B6	B5	B4	1	1	1

Die untersten drei Bits identifizieren den Endzustand eines Typen (111). Die weiteren Bits sind unbenutzt.

6.2.5 Darstellung der Schemainformation

Die Bytecodezustände werden beim Kompilieren durch die Zeiger miteinander vernetzt und bilden die Syntaxdefinitionen im XML Schema ab. Folgendes Beispiel (Abb. 6.2) illustriert die Umsetzung der Schemainformation in Bytecodezustände. Betrachtet werden zwei komplexe Typen *CT_A* und *CT_B*, wobei *CT_B* von *CT_A* abgeleitet ist.

XML Definition
eines komplexen Typen “CT_B”:

```
<complexType name=”CT_B”>
   <extension base=CT_A”>
      <sequence>
         <element name= “E_d” type=”Bit”/>
         <element name= “E_e” type=”Bit” maxOccurs=”4”/>
      </sequence>
      <attribute name=”A_b” type=”string”/>
      <attribute name=”A_c” type=”string”/>
   </extension>
</complexType>
```

XML Definition
eines komplexen Typen “CT_A”:

```
<complexType name=”CT_A”>
   <element name=”E_a” type=”string” />
   <choice>
      <element name=”E_b” type=”string”/>
      <element name=”E_c” type=”integer”/>
   </choice>
   <attribute name=”A_a” type=”string”/>
</complexType>
```

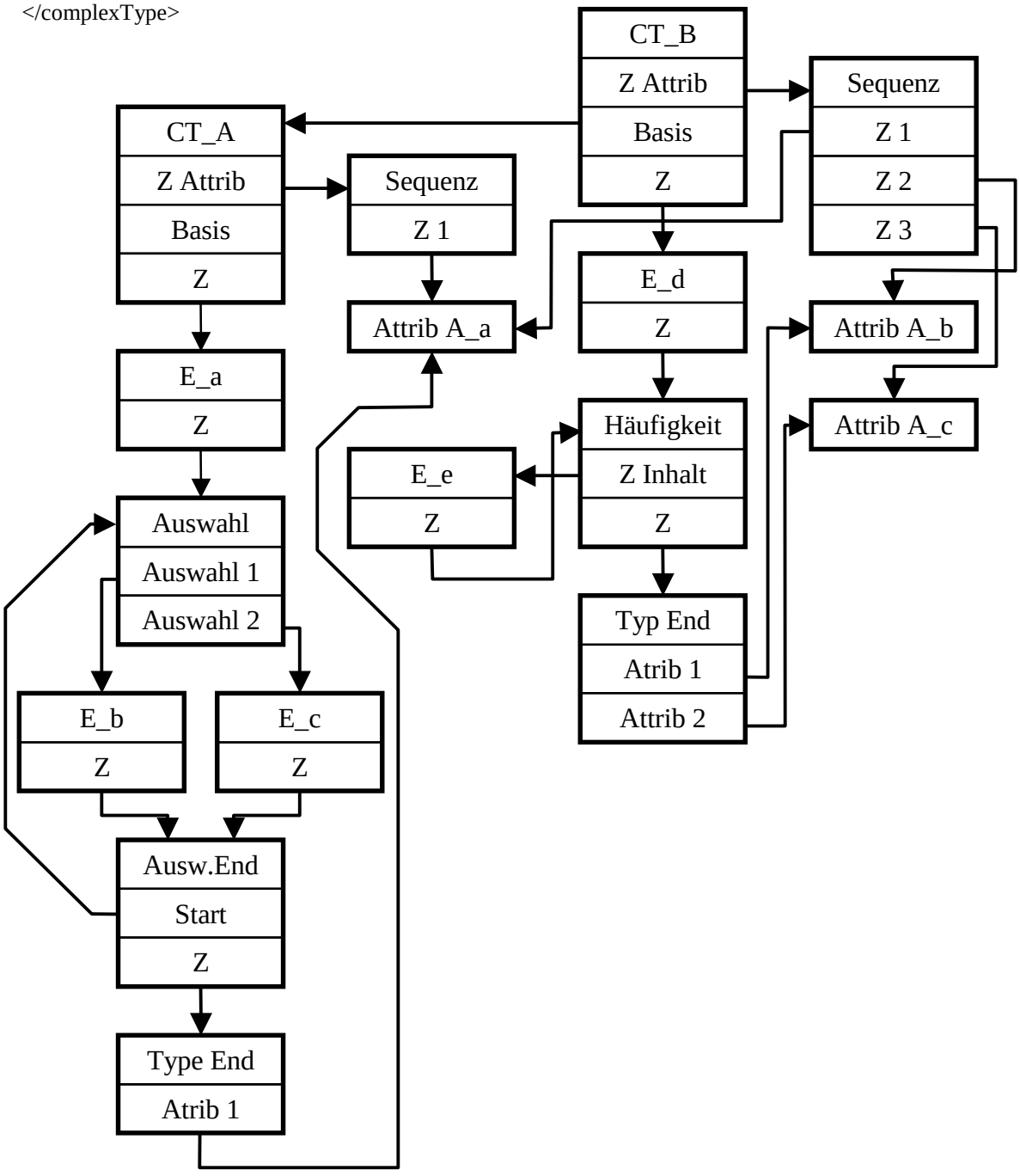


Abb. 6.2: Syntaxdefinition und Umsetzung in eine Bytecodestruktur

Die Rechtecke symbolisieren die Zustände mit den einzelnen Informationsfeldern. Jedes dieser Datenfelder ist an einer Adresse im Bytecodevektor abgelegt. Durch Vergleich mit den textuellen Typdefinitionen lässt sich die Abbildung der XML Syntaxelemente auf die Bytecodezustände nachvollziehen. Jeder Typ beginnt mit seinem Kopfzustand *CT_A* bzw. *CT_B*. Beide Kopfzustände enthalten einen Zeiger auf einen Sequenzzustand, der die Attribute des Typs auflistet. Der Kopfzustand von *CT_B* enthält einen Zeiger auf den Kopfzustand *CT_A*, da *CT_B* von *CT_A* abgeleitet ist.

Das Attribut maxOccurs bei Element *E_e* wird in einen Häufigkeitszustand umgesetzt, dessen Inhaltszeiger auf den Elementzustand *E_e* zeigt. Die Auswahl *choice* in *CT_A* zwischen *E_b* und *E_c* wird in einen entsprechenden Auswahlzustand übersetzt, der als Inhalt die beiden Elementzustände referenziert.. Das Konstrukt *sequence* in *CT_B* wird implizit umgesetzt, durch die Anordnung des Elementzustands *E_d* und des Häufigkeitszustands. Die Endzustände enthalten Zeiger auf die Attribute, welche in dem entsprechenden Typ definiert sind. Das erscheint redundant, da die Attributzustände schon bei den Kopfzuständen der entsprechenden Typen im Sequenzzustand referenziert werden. Dies ist jedoch deshalb erforderlich, weil die Behandlung von Attributen sich bei Pfadcodierung und der Payloadcodierung unterscheidet.

6. 3 Der Bytecodecompiler

Der Bytecodecompiler übersetzt ein textuelles Schema in die Bytecodezustände. Das Kompilieren gliedert sich grob in folgende Arbeitsschritte:

- Parsen des textuellen Schemas

Hier werden die Syntaxkonstrukte des Schemas (etwa “complexType”, “choice”, “maxOccurs”) detektiert und Daten wie Elementnamen und Häufigkeiten extrahiert.

- Anlegen der Zustandsmatrizen

Gewisse Syntaxkonstrukte lösen das Anlegen eines neuen Zustandes aus. Da zu diesem Zeitpunkt noch nicht alle Information zur Verfügung steht, können einige Informationsfelder innerhalb des Zustandes leer bleiben oder werden mit Zwischendaten belegt, die später durch die entgültigen Daten ersetzt werden. Dennoch erhält jeder Zustand bereits eine feste Adresse im Bytecodevektor.

- Nachverarbeitung

Nach dem vollständigen Parsen des Schemas können die noch fehlenden Informationen in die Bytecodezustände eingesetzt werden (z.B. Zeiger innerhalb eines Elementzustands auf den Typ des Elements). Außerdem wird der Vererbungsbaum und die Attributlisten für die Payloadcodierung angelegt.

6.3.1 Parsen des Schemas

Ein XML Schema hat eine hierarchische Struktur. Der Inhalt der XML Typ Definitionen besteht aus verschachtelten Syntaxkonstrukten, welche den möglichen Inhalt eines Typs festlegen. Eine hierarchische Struktur wird üblicherweise durch einen Stapel (Stack) abgebildet. Jede Stufe enthält eines der aktuell verwendeten Syntaxkonstrukte. Dieser Stapel (der "Parser Stack") ist so entworfen, dass er alle möglichen Informationen, die für eine Stufe relevant sind erfassen kann. Die Syntaxkonstrukte, welche zum Anlegen einer neuen Stufe im Stapel führen sind:

- ComplexType Definition
- SimpleType Definition
- Elementdeklaration
- Attributdeklaration
- Sequence- , choice- , all- Gruppen

Die Konstrukte „extension" und „restriction", welche die Vererbung des Typs anzeigen führen nicht zum Anlegen einer eigenen Stufe. Es wird lediglich im zuletzt eingesetzten complex- oder simpleType ein Datenfeld mit dem Namen des Basistypen angelegt.

6.3.2 Aufbau der Zustände

Beim Parsen wird jedes mal, wenn ein relevantes Syntaxkonstrukt gefunden wird eine neue Stufe im Parser Stack anlegt, und die verfügbaren Informationen (etwa die Werte der maxOccurs- und minOccurs Attribute) eingetragen. Erst wenn das entsprechende Syntaxelement wieder verlassen wird kann der Bytecode erzeugt werden.

Bei den einzelnen Syntaxkonstrukten läuft folgendes ab:

Syntaxelement "sequence"

Bei einer Sequenz müssen die Kindelemente nacheinander verkettet werden, das heißt der Zeiger auf den Folgezustand jedes Knotens muss auf den Kopfzustand des nächsten Zustands in der Sequenz zeigen.

Dazu ein Beispiel:

```
<sequence>
   <element a>
   <element b>
   <element c>
</sequence>
```

Trifft der Parser auf diese Struktur, wird in Stufe (i) des Stack die Sequenz eingetragen. Es folgt *element a*, welches in Stufe (i+1) angelegt wird. Da *element a* keinen weiteren Inhalt hat (wie etwa einen anonymen Typen), wird ein Elementzustand angelegt und die Stufe (i+1) wieder freigegeben. Dabei wird ein Datenelement (genannt "HeadEnd") erzeugt, das die

Bytecodeadresse des Headers des Elementzustands enthält und die Adresse des Zeigers auf den Folgezustand (*NextStatePointer*). Dieses Datum wird in die Liste der Kindelemente der Stufe (i) eingetragen. Ebenso wird mit Elementen b und c verfahren.

Wenn der Parser das Endtag der Sequenz erreicht, können die Zustände der in der Sequenz enthaltenen Konstrukte (in diesem Falle der drei Elemente) verkettet werden.

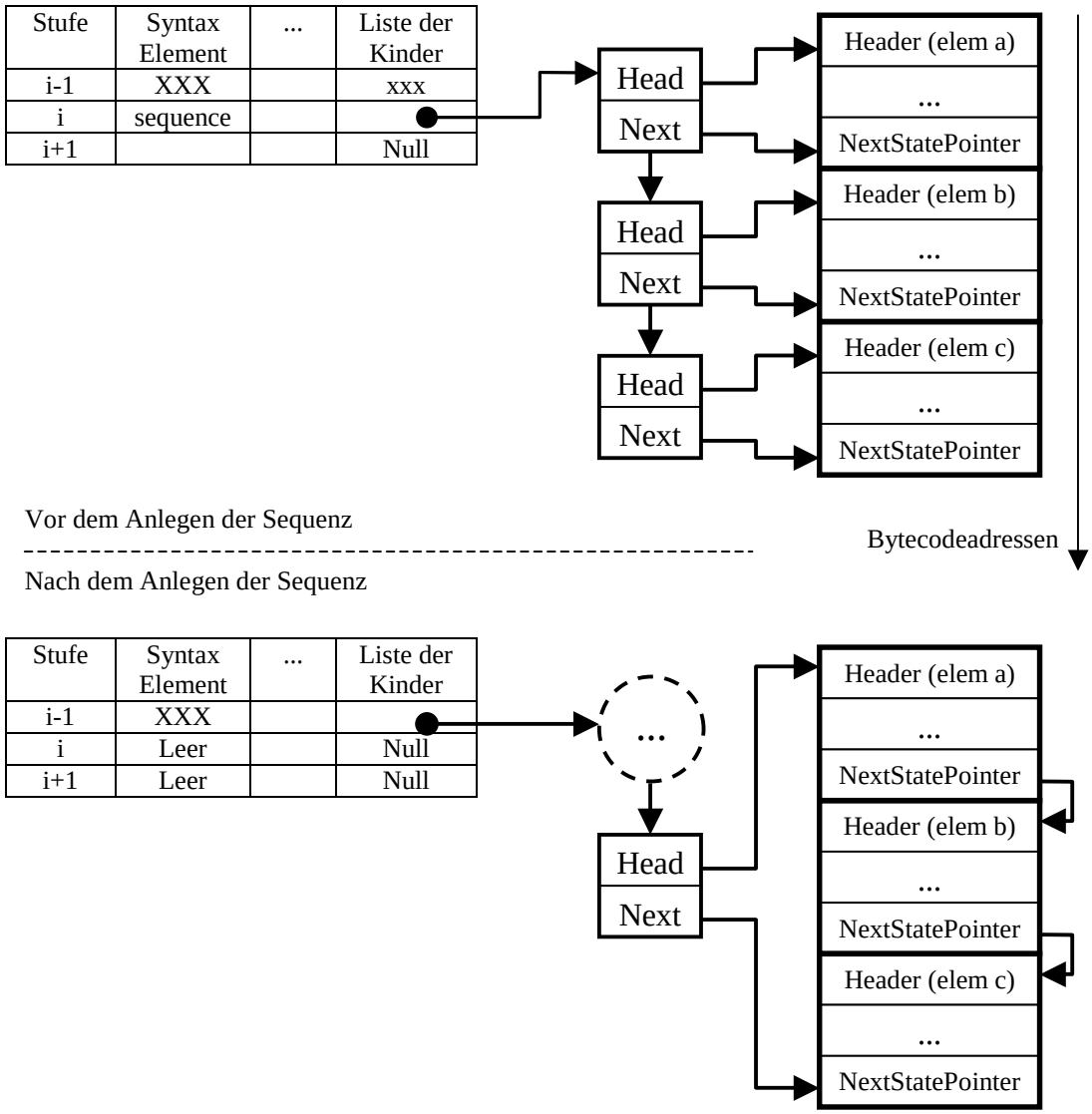


Abb. 6.3: Hierarchische Erzeugung der Bytecodestruktur

Dazu wird an der Adresse des *NextStatePointer* von *element a* die Adresse des Headers von *element b* eingetragen, und an der Adresse des *NextStatePointer* von *element b* der Header von *element c*. Der Header von *element a* und der *NextStatePointer* von *element c* werden zu einem neuen *HeadEnd* Datensatz zusammengefasst. Dieser wird in die Liste der Kindelemente der Stufe (i-1) eingetragen. Der gesamte Inhalt der Sequenz kann damit über ein kleines Datenelement referenziert werden. Der übergeordneten Stufe steht die Information zur Verfügung, wo in die Sequenz eingesprungen, und wo ein Zeiger auf den der Sequenz folgenden Zustand eingetragen werden muss. Dieses Verfahren ist auch anwendbar, wenn der Inhalt der Sequenz nicht nur aus einer Reihe von Elementen besteht, sondern aus beliebig tief verschachtelten Strukturen.

Syntaxelement “choice”

Die Vorgehensweise bei der Erzeugung der Strukturen einer Auswahl ist dieselbe wie bei der Sequenz. Die Elemente der Auswahlgruppe werden in die Liste der Kindelemente des Parser Stack in jener Stufe eingetragen, in der die Auswahl angelegt ist. Beim Anlegen des Auswahlzustands werden sie lediglich anders vernetzt: in die Liste der Folgezustände des Auswahl Startzustands werden die Adressen der Header aller Kindelemente eingetragen. Die Zeiger auf den Folgezustand innerhalb der Kindelemente werden auf den Header des Endzustands der Auswahlgruppe gelegt. Zur darüberliegenden Stufe des Parserstack wird ein HeadEnd Datenelement übergeben, das aus der Adresse des Headers des Auswahl-Startzustands und dem *NextStatePointer* des Auswahl-Endzustands besteht.

Syntaxelement “complexType”

Eine complexType Definition im XML Schema führt zur Neuanlage eines Typ-Kopf-Zustands und eines Typ-End-Zustands. Dabei ist es unerheblich, ob es sich um einen anonymen Typen oder Typen mit Namen handelt. Der Inhalt des komplexen Typen wurde in den vorangegangenen Schritten erzeugt. Die Einsprungadresse des Inhalts (der aus vielen Zuständen bestehen kann), und die Adresse des *NextStatePointer* des letzten Zustands des Inhalts stehen wieder durch ein *HeadEnd* Datenelement zur Verfügung, welches in die Liste der Kindelemente im Stack eingetragen wurde. Beim Anlegen des Typ-Kopf-Zustands wird dessen *NextStatePointer* auf die Einsprungadresse des Inhalts, und der *NextStatePointer* des Inhalts auf den Endzustand dieses Typen gesetzt.

Der Typ-Kopfzustand enthält auch ein Informationsfeld, welches den Modus für den Positionscodes zur Verfügung stellt (globaler/lokaler Positionscode). Beim Abarbeiten des Inhalts wurde dieses Informationsfeld berechnet, und kann nun eingetragen werden. Details dazu finden sich im Anhang A3.

Da der Inhalt des komplexen Typen bekannt ist, kann die Liste der eigenen Attribute dieses Typs vollständig im Typ-Endzustand eingetragen werden. Wenn der Datentyp von einem anderen Typ vererbt wurde (das ist bekannt durch das <extension> oder <restriction> Tag im Inhalt des Typen), so wird der Zeiger auf den Namen des Basistypen eingetragen. Dieser wird später, wenn allen Typen eine Adresse zugewiesen wurde, durch einen Zeiger auf den Typ selbst ersetzt.

Syntaxelement “simpleType”

Im Gegensatz zum complexType hat der simpleType nur einen Typ-Kopfzustand. Datenelemente von simpleType dienen nicht dazu den Inhalt eines XML Dokuments zu strukturieren, wie die complexTypes, sondern die Nutzinformation, etwa eine Zeichenkette, Zahlen oder Matrizen zu codieren. Um diese Datentypen effizient darzustellen sind oft spezifische Codecs zweckmäßig. Entsprechend enthält der simpleType eine Referenz, mit dem so ein Codec aufgerufen werden kann.

Syntaxelement “attribute”

Ein Attributzustand wird angelegt. Es wird der Zeiger auf den Namen des Attributs eingetragen, und der Zeiger auf den Namen des einfachen Typen des Attributs. Dieser wird bei der Nachverarbeitung, wenn die Adresse des einfachen Typen bekannt ist durch einen

Zeiger auf dessen Kopfzustand ersetzt. Außerdem wird evtl. ein Zeiger auf default- oder fixed- Werte angelegt. Diese werden bei der Kompilation in den Stringvektor eingetragen.

Syntaxelement "element"

Ein Elementzustand wird angelegt. Wie beim Attribut werden Zeiger auf den Namen und auf den Namen des Typen des Elements eingetragen. Wenn das Element ein maxOccurs Attribut ungleich *0* oder *1* hatte wird im Datenfeld "Länge des Positionscodes" die entsprechende Zahl $\log_2$(maxOccurs) oder eine Markierung für Unendlich eingetragen. Dies ist relevant, wenn lokale Positionscodes verwendet werden.

Attribute "maxOccurs" und "minOccurs"

Wenn ein Element oder eine Gruppe (choice, sequence) ein maxOccurs oder minOccurs Attribut ungleich eins enthält wird zusätzlich ein Häufigkeitszustand angelegt. Der darüber liegenden Stufe des *Parser Stack* wird als *HeadEnd* Datenelement die Adresse des Headers und des *NextStatePointers* des Häufigkeitszustands übermittelt. Der Zeiger auf den Inhalt des Häufigkeitszustands wird auf das entsprechende Element oder die Gruppe gelegt, der *NextStatePointer* dieses Inhalts wird wieder auf den Häufigkeitszustand gesetzt.

6.3.3 Anlegen der Zustandsmatrizen

Für jedes Syntaxelement wird eine Funktion aufgerufen, die den entsprechenden Zustand des Syntaxelements im Bytecodevektor anlegt. Die Struktur des Zustands ist in dieser Funktion fest angelegt, da jeder Zustand eines gegebenen Typs immer die gleiche Art von Informationsfeldern beinhaltet. Wenn Informationsfelder nur manchmal gebraucht werden, und dies beim Parsen schon bekannt ist (etwa die Vererbung von einem anderen Typ), so wird das Vorhandensein dieses Informationsfeldes durch ein Flag im Header angezeigt. Bei gewissen Typen (z.B. der Auswahlzustand und der Endzustand eines Typen) muss eine Zeigerliste mit variabler Länge angelegt werden. Dies ist möglich, da die Länge dieser Listen bekannt ist. Da die Zustände im Bytecodevektor lückenlos aufeinanderfolgen ist es auch entscheidend, dass die Länge jedes Zustands beim Anlegen schon feststeht. Die niedrigste freie Adresse im Bytecodevektor wird durch eine zentrale Variable erfasst.

6.3.4 Behandlung von Vererbung

Wenn ein Typ T1 von einem anderen Typ T2 (dem Basistypen) abgeleitet ist, so bedeutet dies, dass der Inhalt des Basistypen auch in T1 präsent ist. Das kann auch verschachtelt auftreten, d.h. der Basistyp T2 ist seinerseits von einem anderen Typen (T3) abgeleitet. Um den Bytecode möglichst kompakt zu halten wird der Inhalt der Basistypen nicht erneut in Bytecodezuständen angelegt, sondern es wird lediglich im Kopfzustand des Typen ein Zeiger auf den Basistypen angelegt. Beim Encodieren und Decodieren werden diese Zeiger verwendet, um an die entsprechende Stelle im Bytecode zu verzweigen, an welcher der Inhalt des Basistypen abgelegt ist.

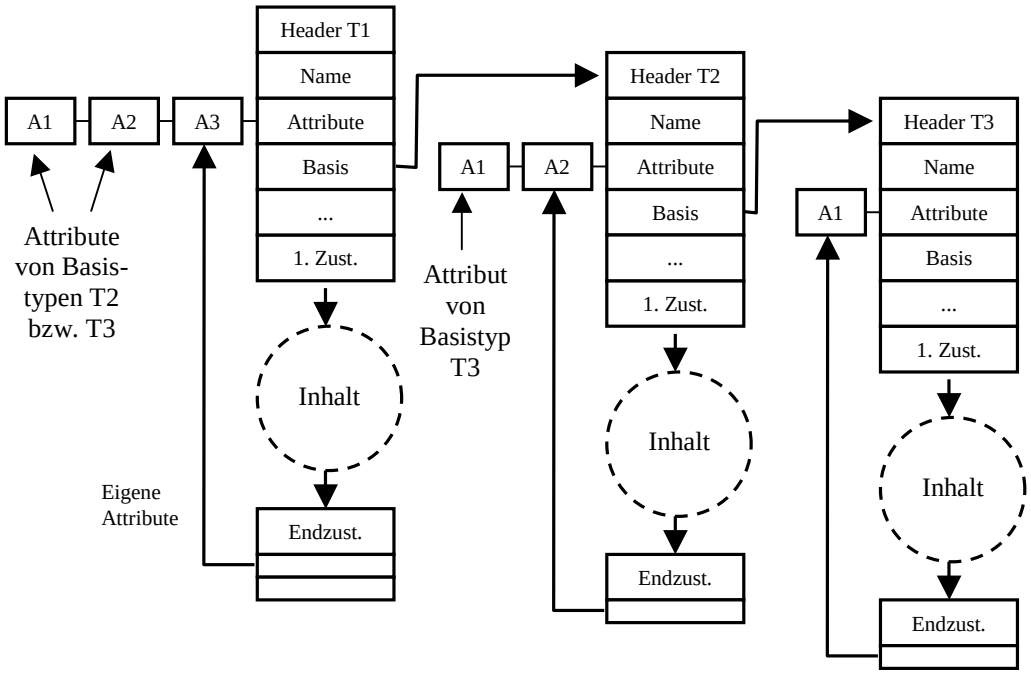


Abb. 6.4: Behandlung von Vererbung

Aufgrund der Vererbung erfordern die Attribute für die Pfadcodierung und die Payloadcodierung eine getrennte Behandlung (s. Abbildung 6.4). Bei der Payloadcodierung werden alle Attribute, auch die aller Basistypen alphabetisch geordnet und am Anfang eines Typs codiert. Bei der Pfadcodierung erscheinen die Attribute eines Typs am Ende der *BVC-Tabelle*, ebenfalls alphabetisch geordnet, wobei jedoch die Attribute der Basistypen nicht enthalten sind. Diese finden sich and der entsprechenden Stelle vorher in der Tabelle. Deshalb werden die Attribute im Bytecode durch zwei unterschiedliche Strukturen repräsentiert. Für die Payloadcodierung ist im Kopfzustand der Typen eine Liste referenziert, die Zeiger auf alle Attribute (auch die der Basistypen) eines Typs enthält. Diese Liste ist alphabetisch geordnet. Für die Pfadcodierung findet sich eine solche Liste im Endzustand eines Typs, ebenfalls alphabetisch geordnet. Dadurch wird das En- und Decodieren für beide Verfahren sehr gut vorbereitet.

6.3.5 Nachverarbeitung

Die Nachverarbeitung wird durchgeführt, nachdem das XML Schema geparst und alle Typzustände angelegt wurden. Verschiedene Aufgaben können nun erledigt werden, die vorher noch nicht möglich waren, weil die notwendigen Informationen noch nicht vollständig zur Verfügung standen. Dazu werden alle Zustände im Bytecodevektor in der Reihenfolge aufsteigender Bytecodeadressen abgearbeitet. Der Header identifiziert den Typ, woraus (mit der Information in den Flags über evtl. vorhandene optionale Datenfelder) die Länge berechnet werden kann. Damit ist die Adresse des nächsten Zustands bekannt.

Die durchzuführenden Aufgaben sind:

- Referenzen auf Typen einsetzen
- Attributlisten für die Payloadcodierung erstellen
- Länge der *BVC-Tabellen* für die Pfadcodierung ermitteln
- Den Vererbungsbaum anlegen
- Das Inhaltsmodell für die Länge der Positionscodes berechnen

Zeiger auf Typen

Da das Anlegen der Kopfzustände der Typen in der Reihenfolge erfolgt, wie sie vom Parser detektiert werden, und dies wiederum davon abhängt, in welcher Reihenfolge sie im Schema definiert sind, ist nicht sichergestellt, dass ein Typ, der als Typ eines Elements oder Attributs oder als Basistyp referenziert wird bereits im Bytecodevektor angelegt ist. Seine Adresse ist deshalb nicht unbedingt bekannt. Darum wird beim Anlegen des Zustands in den Zeiger auf den Typ stattdessen ein Zeiger auf die Position im Stringvektor abgelegt, an dem der Name des Typs steht. Falls der Name noch nicht eingetragen wurde wird er im Stringvektor neu angelegt.

Bei der Nachverarbeitung wird nun mit dem Namen des zu referenzierenden Typs in einer Liste, dem *Dereferenzvektor* die Adresse des Typ-Kopfzustands ermittelt und anstelle des Zeigers auf den Namen eingetragen. Dadurch muss diese Referenz beim Codieren nicht erst durch den Namen aufgelöst werden, sondern es kann direkt in den entsprechenden Zustand verzweigt werden.

Attributlisten anlegen

Der Kopfzustand eines Typs enthält einen Zeiger auf eine Liste mit allen Attributen dieses Typs, die von der Payloadcodierung benötigt wird. Diese Liste enthält (anders als die Liste der Attribute in einem Typ-Endzustand) auch die Attribute aller Basistypen, und zwar alphabetisch geordnet. Deshalb müssen auch zwei verschiedene Datenstrukturen für die Referenzierung der Attribute für die Pfad- und die Payloadcodierung angelegt werden. Da lediglich Zeigerlisten auf die eigentlichen Attributzustände angelegt werden, ist der Mehraufwand für die unterschiedliche Repräsentation der Attribute begrenzt.

Nach dem Anlegen aller Typen sind auch die Adressen der Basistypen bekannt. In den Endzuständen der Typen steht eine Liste mit den in dem entsprechenden Typ deklarierten Attributen zur Verfügung. Diese Listen werden für den Typ und alle seine Basistypen gesammelt, in eine gemeinsame Liste eingetragen, diese alphabetisch geordnet, und ein

Sequenzzustand angelegt, der eine Zeigerliste auf alle Attributzustände enthält. Diese Zeigerliste wird im Kopfzustand des komplexen Typen referenziert.

Länge der BVC Tabellen ermitteln

Wenn alle Typen angelegt sind kann auch die Länge der BVC Tabellen ermittelt werden, und in den Kopfzustand eines Typen eingetragen werden. Dies liefert bei der Pfadcodierung die Länge des SBC.

Vererbungsbaum anlegen

Der Vererbungsbaum ist eine eigene Datenstruktur, also nicht im Bytecodevektor abgelegt. Er ist ebenfalls aus Knoten und Listen aufgebaut. Jeder Knoten entspricht einem Typen des Schemas. In ihm ist der Name des Typen sowie der Wert des Typecodes bezüglich des Wurzelknotens und die Größe des Unterbaums abgelegt. Um die Struktur des Baums zu erfassen, hat jeder Knoten eine Referenz auf eine verkettete Liste, welche die Knoten der von diesem Typen abgeleiteten Typen enthält. Das Anlegen dieser Listen erfolgt beim Parsen des Schemas: ein base-Attribut (base=”B”) in einem Typ “X” führt dazu, dass in dem Knoten des Vererbungsbaums für den Typ “B” ein Verweis auf den Knoten für den Typ “X” in die Liste der Kinder eingetragen wird. Wenn einer der Knoten noch nicht im Vererbungsbaum vorhanden ist, so wird er neu angelegt. Um festzustellen, ob einer der beiden Knoten bereits angelegt wurde, wird der Vererbungsbaum in der Reihenfolge, in der die Knoten angelegt wurden durchsucht.

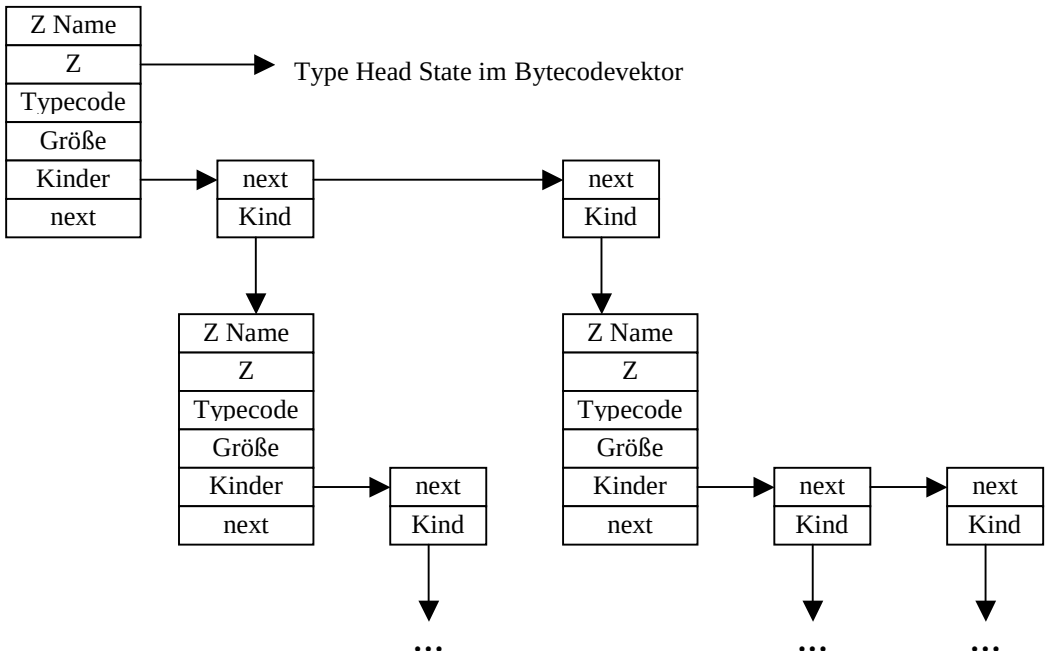


Abb. 6.5: Implementierung des Vererbungsbaums

Sobald der Vererbungsbaum vollständig angelegt worden ist, kann die Berechnung der Typecodes und der Größe des Unterbaums eines Knotens erfolgen. Dazu wird der Baum depth-first abgearbeitet, die passierten Knoten mitgezählt, und der Wert, welcher der Typecode bezüglich des Wurzelknotens ist, in das Feld *Typecode* eingetragen. Wird ein Knoten in Richtung seines Vater wieder verlassen kann auch das Feld *Größe des Subbaums*

eingetragen werden, nämlich die Differenz aus dem aktuellen Zählerstand und dem bereits eingetragenen Wert für den Typecode. Die Größe des Subbaums wird für die Berechnung der Länge des Typecodes verwendet. Aus dem Typecode bezüglich der Wurzel kann der Typcode bezüglich jedes anderen Typen berechnet werden, wie in 4.3.2 beschrieben.

Inhaltsmodell für die Länge der Positionscodes berechnen.

Für die Positionscodes ist zunächst festzustellen, ob globale oder lokale Positionscodes verwendet werden müssen (vgl. 4.3.3). Dabei ist zu beachten, das der Codiermodus global/lokal für einen Typen auf global gesetzt werden muss, wenn nur einer der Basistypen globale Positionscodes benötigt. In diesem Fall wird die Größe des Inhaltsmodells des Typen und aller Basistypen aufaddiert. Die Größe des Inhaltsmodells sagt aus, wie viele Instanzen von Elementen in einem Typen maximal auftreten können. Aus dieser Zahl wird die Bitlänge des globalen Positionscodes berechnet, und in die Elementzustände des Typen eingetragen. Dabei wird ein evtl. vorhandener Wert für einen lokale Positionscode überschrieben. Die Entscheidung, ob globale oder lokale Positionscodes verwendet werden ist erst möglich, wenn alle Basistypen eines Typs bekannt sind. Deshalb kann dies erst in der Nachverarbeitung durchgeführt werden.

6.4 Der Bytecode Interpreter

Der Bytecode Interpreter übernimmt das eigentliche codieren und decodieren der XML Daten. Je nach Konfiguration kann eine Payload oder ein (absoluter oder relativer) Pfad codiert werden.

Im Falle der Encodierung besteht die Aufgabe darin, die Elemente und Attribute des zu codierenden XML Dokuments im Inhaltsmodell der komplexen Typen zu finden (es gibt z.B. im Falle einer Auswahlgruppe mehre Alternativen, die überprüft werden müssen), und die entsprechenden Codes zu ermitteln.

Der Bytecodeinterpreter auf der Decoderseite wird durch den binären Strom des encodierten Dokuments gesteuert, d.h. wenn mehrere Alternativen möglich sind, welcher Zustand als nächstes aufgerufen wird, kann durch Lesen des Bitstroms der richtige Folgezustand im Bytecode ermittelt werden.

6.4.1 Aufbau des Interpreters

Da ein XML Dokument ebenso wie ein XML Schema eine hierarchische Struktur besitzt benötigt auch der Bytecode Interpreter ein mehrstufiges Gedächtnis, wobei jede Stufe die aktuelle Konfiguration der entsprechenden Hierarchiestufe sichert. Eine Hierarchiestufe speichert die Information für die Codierung oder Decodierung eines komplexen Typen. Wenn ein Element innerhalb dieses komplexen Typen codiert wird, wird die Bytecodeadresse dieses Elementzustands in der Hierarchiestufe eingetragen, der Typ des Elements ermittelt (durch den Zeiger auf den Typen im Elementzustand) und eine neue Hierarchiestufe mit dem neuen komplexen Typen angelegt. Dies gilt sowohl für die Payload- als auch für die Pfadcodierung. Beide Werkzeuge verwenden dieselbe Datenstruktur, den *Interpreterstack*.

	Aktuelles Element	Aktueller Typ	Liste der Basistypen	Encoderspur	Flags
Daten-struktur	Zeiger in Bytecode	Zeiger in Bytecode	Verkettete Liste mit Zeiger in Bytecode	Verkettete Liste mit Zeiger in Bytecode und Codewerten	Byte

Tab. 6.11

Der *Interpreterstack* ist die zentrale Datenstruktur des Interpreters. Aus den eingetragenen Daten wird bei der Pfadcodierung der Bitstrom erzeugt. Jeder Stufe im *Interpreterstack* entspricht dabei ein Schritt im Pfad.

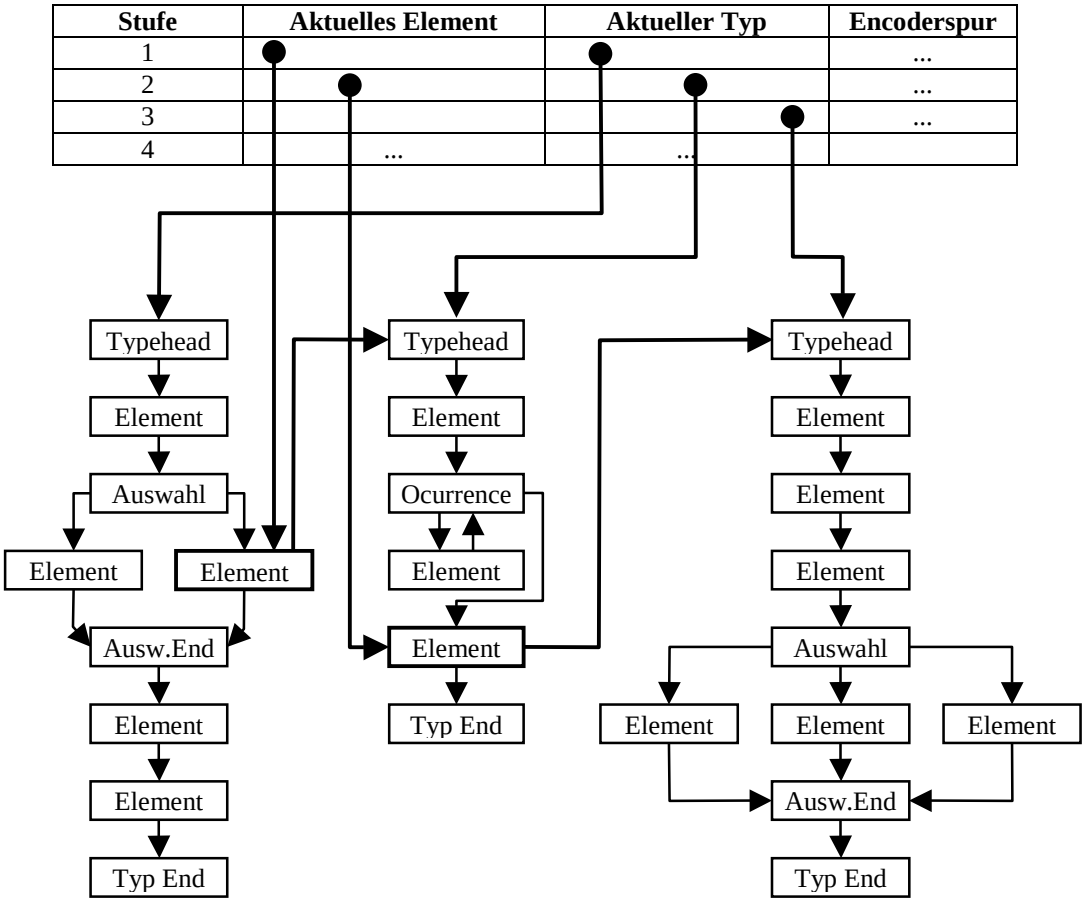


Abb. 6.6. Beispiel für eine Hierarchie der Bytecodemodelle im Bytecodeinterpreter

Abbildung 6.6 stellt das Prinzip an einem Beispiel graphisch dar. Jeder Hierarchiestufe des *Interpreterstack* ist ein komplexer Typ zugeordnet, der durch einen Zeiger referenziert wird. Das aktuell codierte Element eines Typen bestimmt über seinen Typ die nächste angelegte Hierarchiestufe, der aktuelle Elementzustand des aufrufenden Typen wird im *Interpreterstack* gesichert. Dadurch kann nach der Beendigung der Codierung oder Decodierung des aufgerufenen Elements wieder im richtigen Zustand fortgefahren werden.

Datenstruktur „Encoderspur“

Die Datenstruktur *Encoderspur* enthält bei der Encodierung einer Payload die Information, welche Zustände bei der Suche nach den zu codierenden Elementen schon geprüft wurden. Es werden in einer verketteten Liste allen Zuständen die eine Wahlmöglichkeit bezüglich des folgenden Zustands lassen mitprotokolliert, nämlich die Strukturzustände für Auswahl- und all-Gruppen sowie Häufigkeitszustände. Aus diesem Protokoll wird dann auch der Bitstrom erzeugt. Die einzelnen Einträge in der verketteten Liste sichern:

- Für einen Auswahlzustand die Nummer des aktuell geprüften Zweiges
- Für einen Häufigkeitszustand die Nummer des Aufrufs des entsprechenden Inhalts. Dies ist erforderlich, weil bei einem Modus für die Codierung von Häufigkeiten, wie sie in der Payloadcodierung verwendet wird, die Zahl der Instanzen des Inhalts des Häufigkeitszustands in den Bitstrom geschrieben wird.
- Für den Strukturzustand einer all-Gruppe das Bitmuster der bisher instanziierten Zweige.

Datenstruktur Bitstrompuffer

Da bei gewissen Modi der Codierung von Häufigkeiten die Zahl der folgenden Instanzen vor der ersten Instanz in den Bitstrom geschrieben werden muss (vgl. 4.4.3), ist es erforderlich den Bitstrom der für diese Instanzen erzeugt wird zu puffern. Erst nach der letzten Instanz ist die Zahl bekannt und der Bitstrom kann in den Ausgangspuffer geschrieben werden, oder in den Puffer der darüber liegenden Hierarchiestufe, wenn auch diese einen Häufigkeitszustand mit einer solchen Codierung enthält. Da das Schreiben von Codeworten variabler Länge auf einem Standardprozessor rechenaufwändig ist, wird der Bitstrom im Puffer nicht in Form von Bytes geschrieben, da im entgültigen Bitstrom die Ausrichtung der Codeworte innerhalb der Oktetts wahrscheinlich anders ist als im Puffer, und deshalb der Bitstrom auf die neuen Bytegrenzen umcodiert werden müsste. Vielmehr ist es günstiger den Puffer als Folge von Anweisungen aufzubauen, welche die Länge der einzelnen Codeworte in Bit und den Wert der Codeworte enthalten. Für diese Datenstruktur bietet sich beispielsweise eine verkette Liste mit Einträgen an, welche Datenfelder der Art (Länge des Codeworts, Wert des Codeworts) enthalten. Diese Liste kann durch Zeigerreferenzen einfach der nächsthöheren Stufe übergeben werden, welche diese Anweisungen sammelt (s. Abb. 6.7). Wenn im Interpreterstack von der aktuellen Stufe bis zur ersten Stufe keine Häufigkeitszustände aktiv sind, die eine solche Codierung erfordern, kann die Liste mit Anweisungen in den Ausgangsbitstrom umgesetzt werden.

XML Definition:

```
<element name="E_A" type="T_A" maxOccurs="unbounded">
```

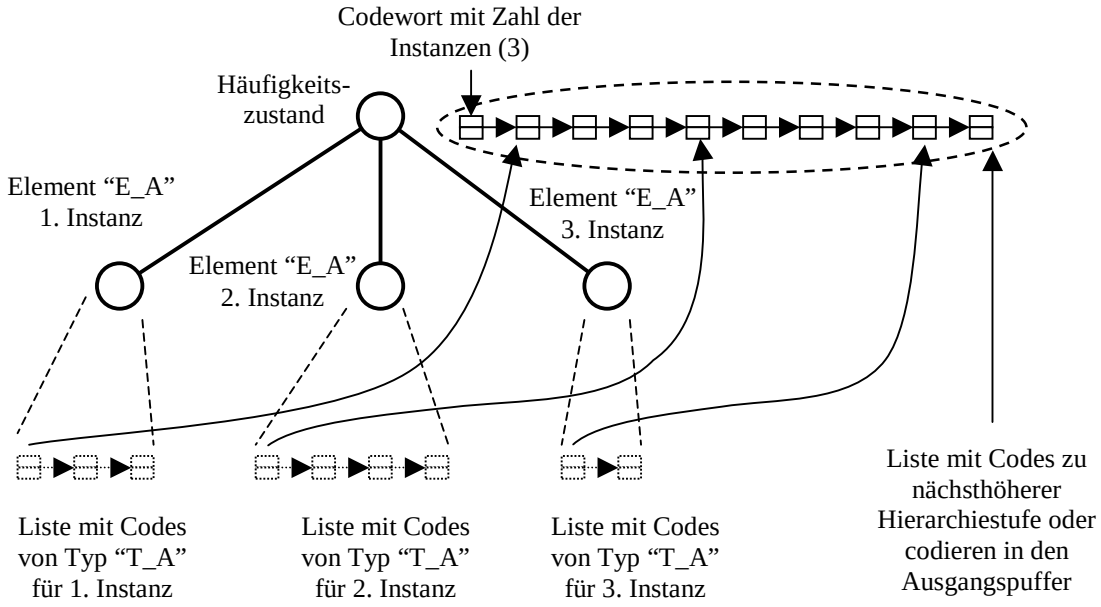


Abb. 6.7: Erzeugung eines Bitstrompuffers

6.4.2 Allgemeine Aufgaben

Codierung von all-Gruppen

In Abschnitt 4.4.2 wurde beschrieben, wie im XML-Schema definierte all-Gruppen bei der Payloadcodierung behandelt werden: eine all-Gruppe wird in einen Baum aus Auswahlgruppen transformiert, welche einen mehrstufigen Entscheidungsprozess für die nächste zu codierende Option implementieren. Die direkte Abbildung dieser Transformation im Bytecode würde das Anlegen einer entsprechend aufgebauten Struktur von Auswahlzuständen erfordern. Da die Anzahl dieser Entscheidungsgruppen mit der Fakultät der Einträge in der all-Gruppe wächst wäre eine enorme Anzahl an Auswahlzuständen erforderlich. Deshalb wird dieser Entscheidungsbaum beim Kompilieren nicht statisch erzeugt. Vielmehr wird nur ein einziger Strukturzustand der Variante "all" angelegt, der alle notwendigen Informationen über die eingeschlossenen Elemente in Form von Zeigern enthält. Die Berechnung der Codes für die Alternativen und der Codewortlängen wird vom Interpreter durchgeführt, und zwar mit einem Datenfeld in der Struktur *Encoderspur*. Für jede Alternative wird im Datenfeld ein Flag reserviert, das anzeigt, ob der Zweig in der all-Gruppe der dem Flag zugeordnet ist bereits beschritten wurde oder nicht. Die Zahl der noch nicht beschrittenen Zweige entspricht der Länge der Auswahlgruppe. Damit kann die Länge des Codeworts für diese Auswahl berechnet werden. Der Wert des Codes ist die Nummer der im aktuellen Schritt ausgewählten Alternative unter den bis dahin noch unverarbeiteten Alternativen.

6.4.3 Encodierung einer Payload

Suchen des Elements

Die Kernaufgabe beim Encodieren einer Payload besteht darin, das gesuchte Element im Inhalt des komplexen Typen, der beliebig tief verschachtelt sein kann, zu finden. Eine Wahlmöglichkeit bezüglich des einzuschlagenden Pfades gibt es dabei bei Häufigkeitszuständen (befindet sich das Element im Inhalt des Häufigkeitszustands) und bei Auswahlzuständen (welcher Zweig?). Alle möglichen Pfade müssen nach dem zu codierenden Element durchsucht werden. Dabei kann es auch notwendig sein mehrstufige Entscheidungen zu treffen (z.B. eine Auswahl in einem Häufigkeitszustand etc.). Wenn an einem Gabelpunkt ein möglicher Pfad durchsucht wird, so muss mitprotokolliert werden, welche Entscheidungen getroffen wurden, damit ein neuer Pfad durchsucht werden kann, wenn das Element nicht gefunden wurde. Dieses Protokoll (Datenstruktur Encoderspur) ist auch erforderlich, um den Bitstrom zu schreiben, da es alle Zustände in der richtigen Reihenfolge enthält an denen eine Wahl zu treffen ist.

Das oben erwähnte Protokoll ist im Interpreter durch eine verkette Liste repräsentiert. Jede Stufe des *Interpreterstack* hat einen Zeiger auf eine solche Liste. Wenn der Bytecodeinterpreter beim Encodieren auf eine Auswahl oder einen Häufigkeitszustand trifft, so wird die Adresse dieses Zustands in die Liste eingetragen. Jedes Listenelement hat auch einen Zähler, der bei der Auswahl den aktuellen Zweig speichert. Wird ein Häufigkeitszustand neu in die Liste eingetragen, so wird zunächst der Inhalt des Häufigkeitszustands überprüft. Wird dort das gesuchte Element gefunden, so wird dieser Listeneintrag für den Häufigkeitszustand mit einer eins markiert (bzw. der Zähler für die Nummer der Instanz erhöht), falls nicht wird die gesamte Liste, die seit dem Eintrag des Häufigkeitszustands angelegt wurde, gelöscht und der Zustand mit einer Null markiert. Beim Auftreten eines Auswahlzustands wird ebenfalls ein neuer Listeneintrag angelegt, und der erste Zweig der Auswahl nach dem Element durchsucht. Beim Erreichen des Endzustands wird überprüft, ob die Auswahl noch weitere Zweige enthält, falls nicht wird die Auswahl wieder aus der Liste gelöscht (dieser Fall kann z.B. auftreten, wenn die Auswahl in einem Häufigkeitszustand enthalten ist. In diesem Fall wird dann bereits der Häufigkeitszustand mit einer Null markiert). Wenn das Element dagegen in einem Zweig der Auswahl gefunden wird, so wird die Nummer dieses Zweiges in der Liste eingetragen, und beim Schreiben des Bitstroms codiert.

Beispiel: Codieren von Element d

Notation:

- Zahl hinter Occurrence
- [1]: gehe in Inhalt,
- [0]: gehe zu Folgezustand
- Zahl hinter Auswahl: Nummer des Zweiges

Reihenfolge der Zustände	Liste *Encoderspur* mit Markierungen
⇨ Typehead	
⇨ Occurr A	Occurr A [1]
⇨ Auswahl A	Occurr A [1] => Ausw A [0]
⇨ Element a	Occurr A [1] => Ausw A [0]
⇨ AuswA End	Occurr A [1] => Ausw A [0]
⇨ Auswahl A	Occurr A [1] => Ausw A [1]
⇨ Element b	Occurr A [1] => Ausw A [1]
⇨ AuswA End	Occurr A [1] => Ausw A [1]
⇨ Occurr A	Occurr A [0]
⇨ Occurr B	Occurr A [0] => Ocurr B [1]
⇨ Auswahl B	Occurr A [0] => Ocurr B [1] => Ausw B [0]
⇨ Element c	Occurr A [0] => Ocurr B [1] => Ausw B [0]
⇨ Ausw B End	Occurr A [0] => Ocurr B [1] => Ausw B [0]
⇨ Auswahl B	Occurr A [0] => Ocurr B [1] => Ausw B [1]
⇨ Element d	Occurr A [0] => Ocurr B [1] => Ausw B [1]
Codieren: 0,1,1	

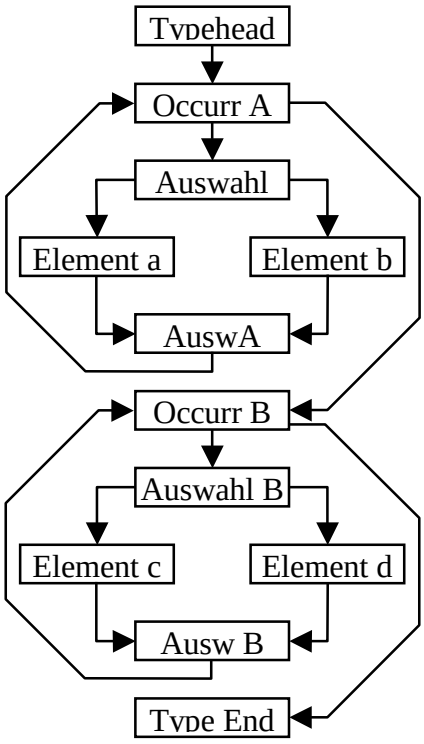


Abb. 6.8: Suchvorgang beim Encodieren einer Payload

Abbildung 6.8 zeigt ein Beispiel für die Vorgänge beim Codieren eines einfach aufgebauten komplexen Typen. Man beachte, dass der Listeneintrag für die Auswahl A gelöscht wird, nachdem das gesuchte Element dort nicht gefunden wurde, und der darüber liegende Häufigkeitszustand mit einer Null markiert wird.

In Abb. 6.9 ist das Ablaufschema beim Suchen eines Elements im Inhalt eines Typen etwas vereinfacht dargestellt. Die Grundstruktur besteht aus einer Schleife, in der die Zustände des Bytecodes der Reihe nach aufgerufen werden. Die erste Operation innerhalb der Schleife ist die Auswahl des richtigen Zustandtyps. Je nachdem, ob es sich um eine Auswahl, einen Auswahl-Endzustand einen Häufigkeitszustand oder einen Elementzustand handelt werden verschiedene Operationen durchgeführt. Bei einem Auswahlzustand wird im wesentlichen der nächste noch nicht geprüfte Zweig ausgewählt, und in dem Protokoll *Encoderspur* eingetragen. Im Auswahl-Endzustand wird geprüft, ob noch weitere Zweige möglich sind, oder ob in den Folgezustand der Auswahl verzweigt werden muss. Außerdem werden alle Einträge im Protokoll gelöscht, die seit dem Auswahl- Startzustand angelegt wurden, da das Element in dem Zweig nicht gefunden wurde. Bei einem Häufigkeitszustand wird überprüft, ob das Element bereits einmal aufgetreten ist, und es wird die aktuelle Nummer der Instanz im Protokoll eingetragen, oder der Zustand wird für ein Überspringen des Inhalts markiert, wenn das Element dort nicht gefunden wurde. Bei einem Elementzustand entscheidet sich, ob das Element gefunden wurde, oder ob noch weitere Durchläufe nötig sind. Wenn die Suche beendet ist wird der Bitstrom aus dem Protokoll in der *Encoderspur* erzeugt, und das Protokoll geeignet für den nächsten Durchlauf initialisiert.

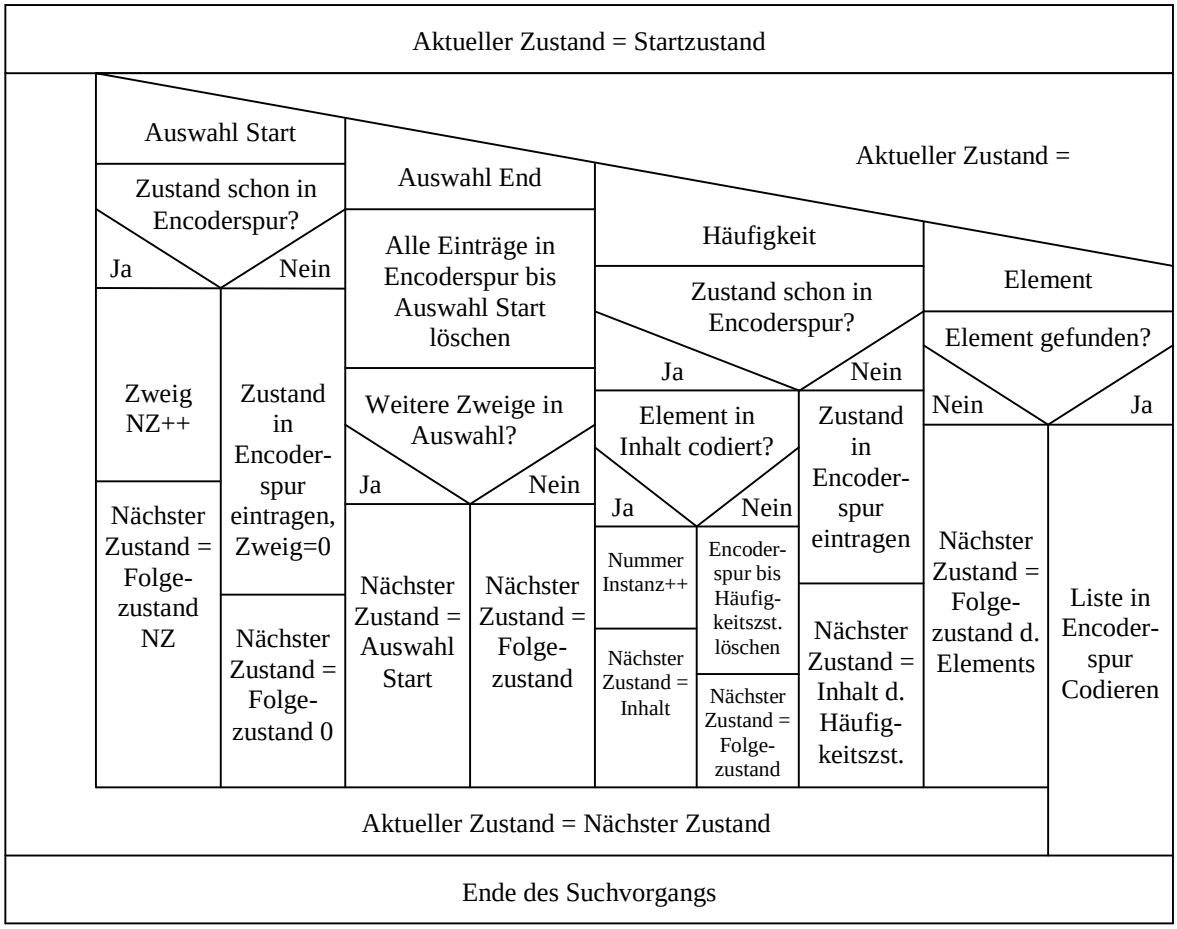


Abb. 6.9: Ablaufschema beim Encodieren eines Elements

Ende des Typs suchen

Es gibt in der Payloadcodierung keine Codes, die anzeigen, wann der Inhalt eines komplexen Typen vollständig codiert wurde. Da dies jedes Element betreffen könnte, dem nur noch optionale Elemente folgen, müsste ein solcher Code bei jedem solchen Element vorgesehen werden. Statt dessen muss, wenn das textuelle Dokument ein Endtag aufweist, und noch weitere optionale Elemente in der Typdefinition folgen ein Pfad durch den komplexen Typen gefunden werden, der keine weiteren Elemente codiert. Dazu wird in Häufigkeitszuständen nicht in den Inhalt verzweigt, sondern sofort in den Zweig des Folgezustands. Das gilt auch wenn das aktuelle Element im Inhalt eines Häufigkeitszustandes steht.

Attribute Codieren

Jeder Kopfzustand eines Typen enthält eine Referenz auf eine Liste mit Zeigern auf alle Attributzustände. Da die Attribute im Starttag eines textuellen XML Dokuments auftreten, und somit beim Codieren des Elements am Anfang bekannt sind, kann diese Liste einfach überprüft, und alle optionalen Attribute, die im Dokument tatsächlich auftreten mit einer Eins markiert werden. Die nicht auftretenden Attribute werden mit einer Null markiert. Der Inhalt der Attribute (auch jener, die nicht optional sind und deren Auftreten deshalb nicht explizit signalisiert wird) wird entsprechend ihres Typs (ein simpleType wie z.B. string) codiert.

Typecast

Für Elemente, deren Typ durch einen Typecast geändert werden kann, muss durch ein Bit im Bitstrom signalisiert werden, ob der Typecast tatsächlich durchgeführt wird, oder nicht. Ein Typecast ist möglich, wenn andere Typen von dem Typ des Elements abgeleitet sind. Der Encoder und der Decoder stellen das fest, indem sie die Größe des Unterbaums des Typs im Vererbungsbaum abfragen. Ein Wert größer Null erfordert das Schreiben und Lesen des Typecast Flag.

Wenn der Typ tatsächlich geändert wird, so muss ein entsprechender Typecode in den Bitstrom eingefügt werden. Seine Länge ist durch die Gesamtzahl der abgeleiteten Typen gegeben, sein Wert durch die Differenz aus dem Typecode des neuen Typs und dem Typecode des Basistyps, vgl. 4.3.2. Alle diese Informationen sind in der Datenstruktur des Vererbungsbaums vorhanden. Der neue Typ muss im Teil des Vererbungsbaums, der unter dem Basistyp angelegt ist gesucht werden.

6.4.4 Decodieren einer Payload

Das Decodieren einer Payload ist weniger aufwändig als das Encodieren, weil nicht verschiedene Alternativen im Inhalt eines komplexen Typen durchsucht werden müssen. Das ist auch sinnvoll, da in praktischen Anwendungsszenarien der Encoder häufig in einer professionellen Umgebung (z.B. einer Rundfunkanstalt) eingesetzt wird, und der Decoder (z.B. ein mobiles Endgerät) unter Umständen mit geringeren Ressourcen auskommen muss.

Im Decoder finden dieselben Strukturen Verwendung wie im Encoder. Der *Interpreterstack* bildet wieder die Hierarchie des Dokuments nach, und sichert denjenigen Zustand im Inhalt eines komplexen Typen, bei dem nach der Decodierung eines Element fortgefahren werden muss. Wenn der Bytecodeinterpreter beim Decodieren auf einen Häufigkeitszustand oder eine Auswahl trifft liest er den Bitstrom, um festzustellen, ob im Häufigkeitszustand in den Inhalt oder in den Folgezustand verzweigt werden soll (es sei denn es wurde schon vorher, abhängig vom Modus der Häufigkeitscodierung (vgl. 4.4.2), die Zahl der Instanzen ermittelt, und es ist daher bekannt, wie oft der Inhalt des Häufigkeitszustandes noch instanziiert werden muss). Bei einer Auswahl wird der Bitstrom gelesen, um mit dem richtigen Zweig fortzufahren.

Wenn ein Typendzustand erreicht wird, so gibt es zwei Möglichkeiten:

1. Die Liste der Basistypen enthält keinen weiteren Eintrag. Der aktuelle Typ wurde komplett decodiert. In diesem Fall wird die aktuelle Hierarchiestufe im *Interpreterstack* beendet, und mit der Decodierung des Typs in der nächst niedrigeren Hierarchiestufe fortgefahren.

2. Die Liste der Basistypen enthält noch mindestens einen weiteren Eintrag. Der aktuelle Typ ist zwar beendet, jedoch muss mit dem Typen fortgefahren werden, der von dem beendeten Typ abgeleitet wurde. Es wird in den Kopfzustand des nächsten Typen in der Liste verzweigt und mit der Decodierung fortgefahren. Die aktuelle Hierarchiestufe im *Interpreterstack* wird beibehalten.

6.4.5 Encodieren eines Pfades

Der Bytecode ist so entworfen, dass mit denselben Strukturen, welche die Codierung einer Payload steuern auch der Pfad codiert werden kann. Anders als bei der Payoadcodierung beginnt der Suchvorgang immer beim Kopfzustand eines Typen, und nicht beim zuletzt

codierten Element. Außerdem werden beim Pfad die Codeworte nicht bei den Häufigkeits- oder Auswahlzuständen geschrieben. Diese Zustände dienen nur dazu, die Elemente zu vernetzen, und ihnen eine Ordnung zu geben. Da Sowohl bei der Payload als auch bei der Pfadcodierung die Elemente alphabetisch geordnet sind, können die durch Zeiger vernetzten Zustände für die Pfadcodierung verwendet werden. Wenn der BVC eines Elements ermittelt werden soll, muss der Inhalt eines komplexen Typen sequentiell abgearbeitet werden. Das bedeutet: der Inhalt jedes Häufigkeitszustands wird einmal abgearbeitet. Die Zweige einer Auswahl werden in aufsteigender Reihenfolge durchlaufen. Alle durchlaufenen Elemente von komplexem und alle Elemente von einfachem Typ und Attribute werden getrennt mitgezählt. Wird das gesuchte Element gefunden, so kann aus dem Zählerstand der *Operand-* oder *Context-BVC* Code berechnet werden.

Wie bei der Payloadcodierung werden die Basistypen eines Typs zuerst behandelt. Ist der Zeiger auf den Basistyp ungleich Null, so wird der aktuelle Typ in die Liste der Basistypen (vgl. 6.4.1) eingetragen, und in den Kopfzustand des Basistyps verzweigt. Hat auch der neue Typ einen Basistyp, so wird auch dieser in der Liste gesichert usf.. Das Abzählen beginnt in dem ersten Typen, der keinen Basistyp hat. Wird der Endzustand eines Typen erreicht, so wird der letzte Typ aus der Liste geholt, und bei ihm fortgefahren.

Reihenfolge der Zustände	**Code**
TypeHead A	
TypeHead B	
Auswahl B	
Element E	C 1
Element F	S 1
Element G	S 2
Ausw.End	
Occurr B	
Element H	C 2
Occurr B	
TypeEnd B RefAttr C RefAttr D RefAttr E	 S 3 S 4 S 5
Occurr A	
Element A	C 3
Occurr A	
Auswahl A	
Element B	C 4
Element C	S 6
Ausw. End	
Element D	S 7
TypeEnd A RefAttr A RefAttr B	 S 8 S 9

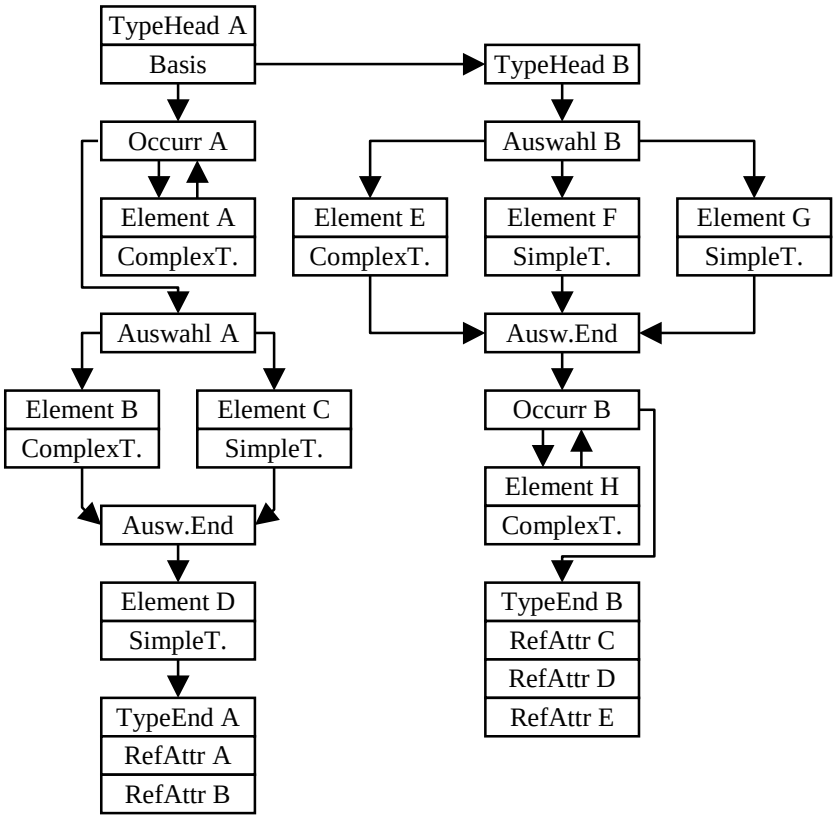


Abb. 6.10: Suchvorgang bei der Pfadcodierung

Abbildung 6.10 enthält ein Beispiel zur Pfadencodierung mit einem Typen A, der von einem anderen Typen B abgeleitet ist, sowie die Liste der Zustände in der Reihenfolge, in der sie abgearbeitet werden, und der Zählerstand für jeden Zustand. Der mehrfache Durchlauf bei

Auswahlgruppen wurde in der Abbildung etwas vereinfacht. Bei der Berechnung eines *Context-BVC* wird nur der Zählerstand für Elemente von komplexem Typ verwendet, bei *Operand-BVC* die Summe aus dem Zählerstand für komplexe und einfache Typen. Das Beispiel verdeutlicht auch, warum die Attribute für die Pfadcodierung am Ende jedes Typs in der Liste erscheinen. Außerdem erkennt man, warum die selben Strukturen wie bei der Payloadcodierung verwendet werden können.

Die Suche im Inhalt eines komplexen Typen liefert den Codewert für das zu codierende Element. Die Länge, mit der Codeworte in den Bitstrom geschrieben werden müssen hängt von der Gesamtzahl der Kinder eines Typen ab. Diese wurde bereits beim Kompilieren in der Nachverarbeitung ermittelt, und in die entsprechenden Datenfelder des komplexen Typen eingetragen. Für die Codierung von evtl. auftretenden Typecodes gilt dasselbe wie bei der Payloadcodierung.

6.4.6 Decodieren eines Pfades

Das Decodieren eines Pfades läuft im Grunde genauso ab, wie das Encodieren. Das gesuchte Element ist durch den Wert des Codes gegeben, der aus dem Bitstrom gelesen wurde (die Länge des Codes wird wieder mit der Gesamtzahl der Kinder im Kopfzustand des Typen ermittelt). Der Durchlauf der Zustände beim Zählen ist identisch mit dem beim Encodieren.

Ist das gesuchte Element gefunden, so wird im *Interpreterstack* eine neue Stufe angelegt. Der Typ der neuen Stufe wird aus dem Zeiger auf den Typen im Elementzustand ermittelt. Eine Sonderstellung nimmt der *Termination Code* ein. Wenn dieser im Bitstrom detektiert wird, braucht kein Element gesucht zu werden. Statt dessen wird ein weiteres Codewort gelesen (mit der Länge des *Operand-BVC*), und das zugehörige Attribut oder Element ermittelt.

Positionscodes Lesen

Beim Kompilieren des Bytecodes wurde in jedem Typen ein Flag gesetzt, das den Modus der Positionscodierung (global/lokal) anzeigt. Gleichzeitig wurde in jedem Element die Länge des Positionscodes in Bit abgelegt. Mit dieser Information kann für jedes Element, das im Interpreterstack als eine Stufe des aktuellen Pfades abgelegt ist, der Positionscode aus dem Bitstrom gelesen werden.

Absoluter und relativer Pfad

Für die Vorgänge beim Suchen eines Elements durch den Bytecodeinterpreter ist es unerheblich, ob ein absoluter oder ein relativer Pfad bearbeitet werden soll. Der absolute Pfad wird vom Wurzelknoten des Schemas aufgebaut, und decodiert. Für die Decodierung eines relativen Pfads muss der Pfad bis zu jenem Element, bei dem der Pfad beginnt bereits im Interpreterstack vorliegen. Ansonsten ist es nicht möglich die Länge des ersten Codeworts zu ermitteln, und es ist darüber hinaus nicht klar, von welchem Elementzustand gestartet werden soll. Deshalb kann ein relativer Pfad nur decodiert werden, wenn vorher bereits ein absoluter oder ein anderer relativer Pfad decodiert worden ist. Wenn am Beginn der Decodierung eines relativen Pfads Codeworte mit dem Wert Null auftreten (Schritt zum Vater), so wird im Interpreterstack eine entsprechende Zahl von Stufen freigegeben, und evtl. vorhandene Einträge (Liste der codierten Zustände etc.) gelöscht.

6.5 Vergleich des Bytecodemodells mit der Referenzsoftware

Wie bereits in Kapitel vier erwähnt ist das BiM Verfahren für die binäre Codierung von XML Teil des MPEG-7 Standards. Für alle bei MPEG aufgenommenen Techniken muss auch eine Software zur Verfügung gestellt werden, welche den Standard implementiert. Die MPEG Referenzsoftware für den BiM Algorithmus wurde von zwei verschiedenen Parteien entwickelt. Das Modul, welches die Payloadcodierung durchführt wurde von Expway zur Verfügung gestellt und in Java implementiert, das Modul für die Pfadcodierung wurde von Siemens und der TU München entwickelt und in C/C++ implementiert. Beide Module sind durch ein Interface (JNI, Java Native Interface) miteinander verbunden.

Wenn auch ein Vergleich der Leistungsfähigkeit von Softwarearchitekturen zwischen völlig verschiedenen Programmierplattformen (Java-C) nur schwer durchzuführen ist, v.a. wenn man lediglich die Ausführungszeiten der Programme für diverse Testdaten misst, so zeigen sich bei der Analyse der verschiedenen Umsetzungen doch prinzipielle Unterschiede, welche eine qualitative Bewertung ermöglichen.

6.5.1 Vergleich des Bytcodemodells mit dem Payloadmodul der Referenzsoftware

Im folgenden wird nur der Decoder behandelt, da nur dieser bei MPEG standardisiert wird, für den Encoder gilt aber Vergleichbares. Der gravierendste Unterschied zwischen den beiden Modulen liegt in der Art der Repräsentation des Schemas im Codec. Die Referenzsoftware repräsentiert die Schemainformation in Form von Syntax-Trees. Diese Syntax-Trees (vgl. 4.4.2) werden durch eine Kompilation des textuellen Schemas erzeugt, und enthalten alle relevanten Syntaxkonstrukte. Sie sind aber nicht dazu geeignet beim Codieren oder Decodieren die Codeworte abzuleiten. Das übernehmen die Finite-State-Automatons, die aus den Syntaxbäumen erzeugt werden, und zwar erst dann, wenn im zu decodierenden Dokument der entsprechende Datentyp auftritt. Die Finite State Automatons werden also “on-the-fly” erzeugt. Diese Alternative wurde gewählt, da die Darstellung der Schemainformation durch die Syntaxbäume kompakter ist als durch die FSAs. Das liegt darin begründet, dass die FSAs durch wenig optimierte Datenstrukturen repräsentiert werden. Allerdings fällt durch diese Aufteilung natürlich bei jedem zu codierenden Typen die Arbeit an, den zugehörigen FSA erst zu erzeugen, was den Rechenaufwand deutlich erhöht. Die FSAs bestehen aus zwei verschiedenen Java Datenstrukturen: es wird unterschieden zwischen den Knoten des Automaten und den Übergängen zwischen den Knoten. Einige dieser Übergänge sind sogenannte Codeübergänge, and denen die Codeworte aus dem Bitstrom gelesen werden. Anders als im Bytecodemodell, in dem Übergänge zwischen Zuständen nur durch Zeigerreferenzen dargestellt werden, benötigen die Übergange der FSAs einige Bytes zur Repräsentation. Die FSAs eignen sich in dieser Implementierung nicht zum Sichern oder Auslagern auf Platte, da sie Referenzen enthalten, die von der Java Virtual Machine zugewiesen werden, und nach einem Abspeichern und erneuten Laden nicht mehr unbedingt gültig sind. Wie beim Bytecodemodell sind die aktuell verwendeten Automaten in einer Stackstruktur angeordnet, welche die Hierarchie des Dokuments nachbildet.

6.5.2 Vergleich des Bytecodemodells mit dem Pfadmodul der Referenzsoftware

Das Schema ist im Pfadmodul in Form eines DOM Baums [W3C] repräsentiert. Dieser wird mit einem Parser aufgebaut und bleibt während der Codierung vollständig im Speicher. Das Parsen muss jedes Mal, wenn das Programm gestartet wird durchgeführt werden. Die Pfadcodierung benötigt im wesentlichen die BVC Tabellen für die Codierung. Wie bei der Payloadcodierung werden diese Tabellen erst aufgebaut wenn ein Element des entsprechenden Typs codiert werden soll. Sie bleiben dann allerdings im Speicher, und

müssen bei einem weiteren Aufruf desselben Typs nicht erneut erzeugt werden. Im Unterschied zum Bytecodemodell (wo nur eine Referenz auf den Basistypen gespeichert wird) werden die Elemente der Basistypen erneut in der *BVC-Tabelle* aufgenommen, sind also mehrfach vorhanden.

In den Einträgen der *BVC-Tabellen* sind keine festen Referenzen auf die Typen der Elemente angelegt, sie müssen in einer Liste der Typen erst mit Hilfe des Typnamens gesucht werden. Die Art der Codierung der Positionscodes (global/lokal) wird erst bei der Anforderung des betreffenden Elements ermittelt.

Für eine Implementierung in C++ ist das Bytecodemodell bei der Pfadcodierung etwa um den Faktor 15 schneller als die Referenzsoftware.

6.6 Alternative Varianten für das Bytecodemodell

Für den Aufbau des Bytecodemodells gibt es mehrere Möglichkeiten, die sich im wesentlichen durch ihren Speicherbedarf und die Geschwindigkeit der Ausführung unterscheiden. Hier wird dargelegt, warum die in diesem Kapitel dargestellte Variante als guter Kompromiss zwischen den verschiedenen Alternativen ausgewählt wurde.

6.6.1 Integration der Strukturen für Pfad und Payload.

Eines der Ziele des Modells ist es die Schemainformation für beide Codierwerkzeuge in einer einheitlichen Struktur zu erfassen. Diese Struktur wird in ihrem Aufbau von den Erfordernissen der Payloadcodierung dominiert. Häufigkeits- und Auswahlzustände haben für die Pfadcodierung keine Bedeutung, sie dienen hier nur dazu den Inhalt eine Typs und die Reihenfolge der Elemente und Attribute festzulegen. Dies wäre mit einer Liste von Zeigern auf die Elementzustände einfacher zu erreichen. Bei der Codierung eines Pfades müsste nur diese Liste abgearbeitet, und nicht über die Häufigkeits- und Auswahlzustände der Payloadcodierung gegangen werden. Allerdings wären diese Listen als eigene Datenstrukturen anzulegen. Die dafür notwendige zusätzliche Information wäre etwa im Bereich von 4 KB (Die Summe der Einträge für Elemente in den *BVC-Tabellen* für alle Typen ist 2035, jedes Element würde mit einem 2 Byte Pointer referenziert. Für Attribute ist im Bytecode schon eine Liste angelegt, zusätzlicher Bedarf an Speicher entsteht dadurch nicht). Die Abarbeitung der Pfadcodierung würde sich beschleunigen, grob geschätzt um 30%, wenn man die Einsparung bei den überprüften Zuständen betrachtet. Da allerdings auf die Pfadcodierung im Vergleich zur Payloadcodierung in praktischen Anwendungsfällen ein geringer Teil entfällt, ist es vernünftig der Einsparung an Speicherbedarf für die Schemainformation den Vorzug zu geben.

6.6.2 Variante bei der Pfadcodierung

Die Informationsfelder für die Anzahl der Einträge in den *BVC-Tabellen* sind nicht zwingend erforderlich. Die Information wird zwar benötigt für die Bestimmung der Länge der BVC-Codes, jedoch könnte man auch die Zahl der Einträge jedes Mal abzählen. In der hier vorgeschlagenen Implementierung wird das nur bei der Kompilation des Bytecodes durchgeführt. Wenn man die Länge der Tabellen erst beim Codieren ermittelt, müsste der Inhalt jedes Typs bis zum Ende abgearbeitet werden, damit würde sich der Rechenaufwand in etwa verdoppeln. Eingespart werden könnten nur etwa 1KB (zwei Informationsfelder mit einem Byte für 522 Komplexe Typen). Deshalb wurden die Informationsfelder vorgesehen.

6.6.3 Variante für den Vererbungsbaum

Für den Vererbungsbaum wurde eine vollständige Baumstruktur entwickelt. Für die Ermittlung des Typecodes ist diese jedoch nicht erforderlich, da die Berechnung des Typecodes bezüglich eines Typs durch einfache Differenzbildung der Typcodes bezüglich des Wurzelelements erfolgen kann (vgl. 4.3.2). Deshalb müsste nur die Liste aller Typen geordnet nach diesem Code und ein Feld für die Länge des Typecodes gespeichert werden. Allerdings geht damit die Baustruktur verloren. Auf der Encoderseite könnte nicht mehr festgestellt werden, ob ein Typ, der als neuer Typ eines Elements im Dokument angegeben ist, auch tatsächlich vom ursprünglichen Typ abgeleitet wurde. Die syntaktische Richtigkeit eines Dokuments könnte damit am Encoder nicht mehr überprüft werden, weshalb sich diese Variante am Encoder nicht anbietet. Auf der Decoderseite dagegen wäre eine Überprüfung nicht mehr zwingend erforderlich, wenn sie vom Encoder sicher durchgeführt wird. Deshalb wäre die reduzierte Version der Vererbungsinformation für den Decoder interessant. Diese würde etwa 2KB Speicher belegen, im Vergleich zu knapp 8KB für die vollständige Bauminformation.

6.7 Weitere Entwicklung

6.7.1 Systemaspekte

Funktionell deckt das Bytecodeformat nur die eigentlichen Codierung und Decodierung ab. Für den praktischen Einsatz gibt es aber noch weitere Aufgaben zu erfüllen um die Eigenschaften der Binärcodierung sinnvoll einzusetzen. Dazu gehört vor allem die Schnittstelle zu den Anwendungen, welche eine Konfiguration der Codieroptionen auf der Encoderseite sowie einen effizienten Zugriff auf die Information auf der Decoderseite ermöglicht. Da viele Vorgänge bei der Encodierung und Decodierung die XML-Schemainformation sowie ein Abbild der aktuellen XML Beschreibung benötigen ist es zweckmäßig die Funktionen zur Kontrolle der Codieroptionen zusammen mit dem Binärcodec in ein Werkzeug zu integrieren.

Schnittstelle am Encoder

Der Encoder sollte eine Schnittstelle zur Verfügung stellen, die es der Anwendung, beispielsweise einer Authoring-Software, ermöglicht die Codierung zu kontrollieren. Dazu zählt:

Aufbau der Beschreibungen
Da für den Aufbau der Beschreibungen das XML Schema erforderlich ist, könnte man eine Schnittstelle zur Verfügung stellen, welche direkt das Bytecodeformat der Schemainformation verwendet, um beispielsweise einem XML Editor die notwendigen Syntaxinformationen zur Verfügung zu stellen. Das aufgebaute Dokument könnte direkt als Baum im internen Dokumentspeicher dargestellt werden, ohne den Umweg über eine textuelle Repräsentation.

Konfiguration der Access Units
Für die Spezifikation der Teile einer XML Beschreibung, welche zusammen in eine Access Unit verpackt werden gibt es mehrere Möglichkeiten. In einigen Fällen kann es notwendig sein dies manuell durchzuführen. In anderen Fällen kann es ausreichen nur die maximale Größe in Bytes oder die Zahl der enthaltenen Elemente anzugeben. Bei der Zusammenstellung der AUs und der darin enthaltenen FUUs ist es auch entscheidend, für

welche Elemente eine Filterfunktion möglich sein soll, da diese von einem Pfad adressiert werden müssen.

Initialisierung der Codierung
Des weiteren gibt es eine Reihe von Parametern für die Codierung, welche in der Initialisierung der Codierung festgelegt werden. Auch diese müssen von der Anwendung abhängig bestimmt werden (z.B. der Parameter *Unit Size*, der die Zahl der in einer Gruppe codierten Instanzen eines Elements festlegt, vgl. 4.4.3, oder die Identifikation der verwendeten XML Schemas).

Für viele Vorgänge bei der Codierung von Dokumenten mit den fortschrittlichen Optionen der BiM Codierung ist eine enge Verzahnung des Binärencoders mit der Schnittstelle zur Anwendung, über die oben genannte Parameter spezifiziert werden sinnvoll. Für die Unterstützung der dynamischen Codierung, d.h. der Aktualisierung der Beschreibung ist es sinnvoll die Beschreibung am Encoder in Form eines Baums vorzuhalten, der praktisch die aktuelle Situation am Decoder abbildet. Für die Aktualisierungen können dann mit Hilfe dieses Baums die notwendigen Informationen für die Codierung bestimmt werden, im speziellen der Pfad zu dem zu aktualisierenden Element. Für häufig zu aktualisierende Elemente könnte auch eine feste Referenz in den Beschreibungsbaum angelegt werden. Um die Reihenfolge der Codierung sinnvoll festzulegen könnte man die Elemente der Beschreibung in Prioritätsklassen unterteilen, wobei Elemente mit höherer Priorität zuerst oder in einem Datenkarussell mit höherer Frequenz übertragen werden.

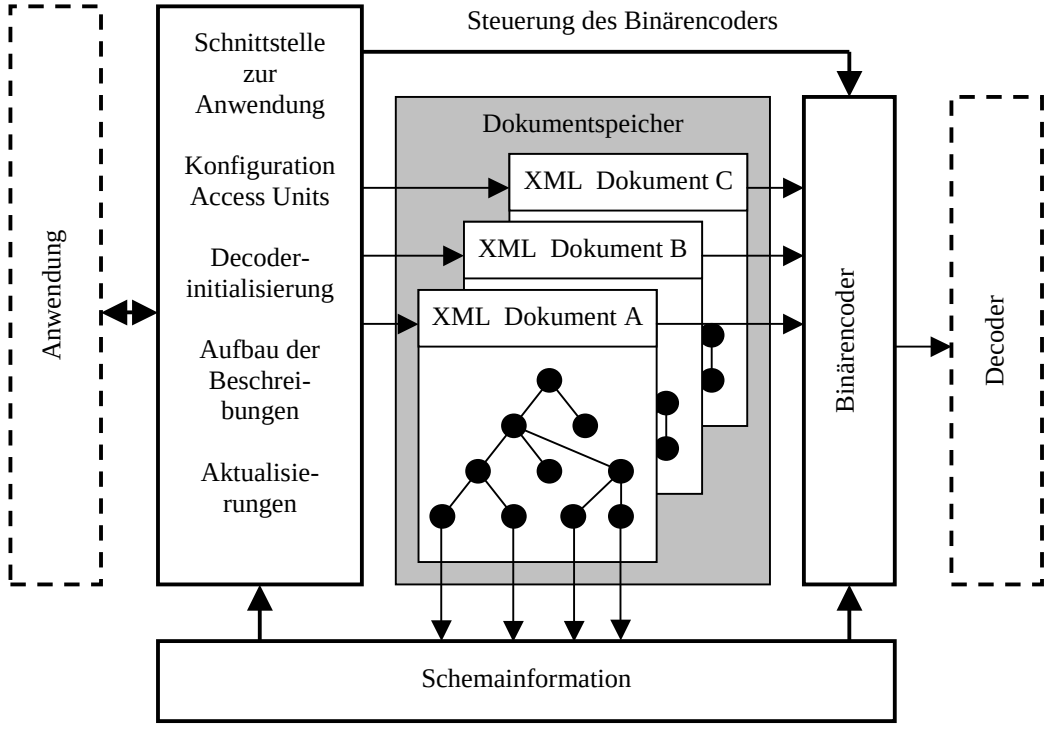


Abb. 6.11: Schnittstelle zur Anwendung auf Encoderseite

Schnittstelle am Decoder

Auf der Decoderseite ist es ebenfalls zweckmäßig die decodierte XML Beschreibung in einer Baumrepräsentation zu speichern und der Anwendung eine Schnittstelle zur Verfügung zu stellen, über die sie Informationen aus dieser Repräsentation aktiv abrufen oder von einer Veränderungen in der Beschreibung unterrichtet werden kann.

Durch eine Baumrepräsentation können im Vergleich zu einer textuellen Darstellung Aktualisierungen der Beschreibungsstruktur besser unterstützt werden. Außerdem kann das XML Dokument auch kompakter repräsentiert werden, da beispielsweise die Namen der Elemente der Beschreibung als Referenzen auf den ohnehin vorhandenen Stringvektor des Bytecodemodells effizient gespeichert werden können.

Außerdem sollte die Möglichkeit geboten werden mehrere Beschreibungen parallel zu Verarbeiten, und die Beschreibungen mit einem einstellbaren Aufwand an Ressourcen (beispielsweise Speicher) zu verfolgen.

Die Funktionen, welche die Decoderschnittstelle der Anwendung zur Verfügung stellt könnten ein:

- Identifikation von Elementen/Attributen, die ein Signal auslösen, wenn sie vom Encoder verändert werden. Die Anwendung kann dann den Nutzer darauf aufmerksam machen, dass neue oder aktualisierte Information zur Verfügung steht.
- Identifikation von Kontexten, die ein Signal auslösen wenn sie in einer Beschreibung auftreten (Filterfunktion). Als Grundlage hierfür dient das XML Schema, mit dem solche Kontexte spezifiziert werden können.
- Durchsuchen und Abrufen der Information im Beschreibungsbaum („Browsing“).
- Spezifikation einer Rangfolge für das Verdrängen von Information aus dem Speicher (bei begrenzten Ressourcen).
- Spezifikation von Relationen zwischen verschiedenen Elementen eines Beschreibungsbaums oder verschiedener Beschreibungsbäume, so dass Aktionen ausgelöste werden, wenn Kombinationen von Bedingungen erfüllt werden.

Wenn ein Rückkanal vom Decoder zum Encoder besteht ist es auch möglich über die Schnittstelle am Decoder direkt Information vom Encoder anzufordern.

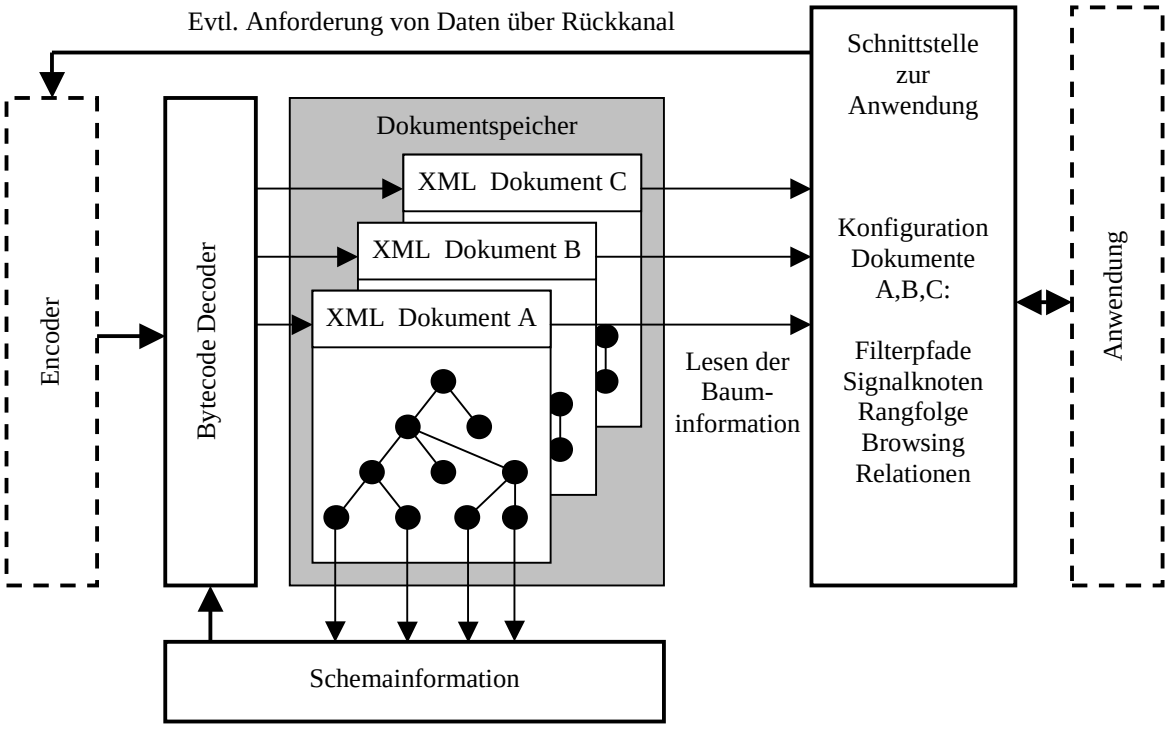


Abb. 6.12: Schnittstelle zur Anwendung auf Decoderseite

Kapitel 7 - Komplexitätsanalyse

In diesem Kapitel wird eine Komplexitätsanalyse des BiM Algorithmus durchgeführt. Als Grundlage dafür dient das Bytecodemodell, welches für diesen Zweck geeignet ist, da es eine in Bezug auf Speicheranforderungen und Rechenleistung optimierte Umsetzung der BiM-Codierung darstellt. Mit Hilfe dieses Modells wird untersucht, wie umfangreich die Schemainformation für ein gegebenes XML Schema im Codec ist, wie sich diese Information zusammensetzt, welche Datenpuffer für die einzelnen Arbeitsschritte erforderlich sind, und wie sich die Rechenleistung auf die Arbeitsschritte aufteilt. Dabei werden sowohl allgemeine theoretische Aussagen gemacht, als auch Messungen mit einer Implementierung des Bytecodemodells diskutiert. Für die Simulationen wird das MPEG-7 Schema und XML Dokumente aus dem MPEG Testdatensatz verwendet.

Die Ergebnisse belegen, dass es sinnvoll ist den Algorithmus aufzuteilen in einen Teil der aus dem textuellen XML Schema ein Zwischenformat mit der Schemainformation erzeugt, und einen anderen Teil, der mit Hilfe dieses Zwischenformats die Enodierung oder Decodierung von Dokumenten durchführt.

Anhand des Bytecodemodells wird eine allgemeine Abschätzung der Komplexität der Codierung und Decodierung eines Dokuments im Payload- oder Pfadmodus entwickelt.

7.1 Speicheranforderungen

Für eine Bewertung von Codecarchitekturen sind die Anforderungen an den Speicher von Interesse. Diese untergliedern sich in Speicher für die Repräsentation der Schemainformation und von Puffern für Zwischendaten während der Programmausführung, sowie in die Eingangs- und Ausgangspuffer für die Nutzdaten. Diese Größen sind vom Umfang des verwendeten textuellen XML Schemas und des XML Dokuments abhängig. Eine allgemeine Angabe ist deshalb nicht möglich. Es ist aber sinnvoll diese Anforderungen in Abhängigkeit der Anzahl der Typdefinitionen und der Element- und Attributdeklarationen im Schema bzw. der Größe des codierten XML Dokuments abzuschätzen und auf diese Weise eine Normierung auf die Größe und Komplexität des Schemas bzw. des XML Dokuments durchzuführen. Prinzipiell besteht auch die Möglichkeit für die Decodierung eines Dokuments nur die verwendeten und nicht alle Datentypen des Schemas zu laden. Nicht zuletzt muss die Größe des ausführbaren Programms berücksichtigt werden.

7.1.1 Speicheranforderungen beim Kompilationsprozess.

Das Kompilierte Schema ist sowohl beim Encoder als auch beim Decoder erforderlich. Die Ressourcen auf Encoderseite stellen für die Ausführung des BiM Algorithmus üblicherweise keine Beschränkung dar. Anders kann sich das beim Decoder darstellen, der auch flexibel auf beliebige Schemas konfigurierbar sein soll und den Kompilationsprozess unter Umständen selbst durchführen muss. Deshalb wird dieser auch hier betrachtet.

Eingangsdaten

Beim Kompilationsprozess stellt das textuelle Schema den Eingangsdatensatz dar. Im Falle des MPEG-7 Schemas sind dies:

Schemafragment	Größe (Byte)
MPEG-7 Master Schema	1.756
Visual Deskriptoren	39.377
Audio Deskriptoren	44.608
MDS Deskriptoren	248.399
DDL	1.860
Gesamt	336.000

Tab. 7.1: Teile des MPEG-7 Schemas

Das MPEG-7 Master Schema bindet die anderen XML Dateien ein. Es ist aber zu beachten, dass ein beträchtlicher Teil der Information in den hier aufgelisteten Dateien aus Kommentaren besteht, die nur der besseren Lesbarkeit dienen, und funktionell ohne Bedeutung sind. Der funktionell relevante Teil umfasst ca. 243 KB.

Für die Erzeugung des Bytecodes ist es nicht erforderlich das gesamte Schema im Speicher vorzuhalten. Jedes Syntaxelement wird nur einmal bearbeitet. Die Reihenfolge der Typen ist dabei beliebig, kann also der Reihenfolge im textuellen Schema entsprechen.

Temporäre Puffer

Während der Kompilation werden einige Pufferspeicher mit Zwischendaten belegt, diese Zwischendaten sind:

Datenstruktur	Aufteilung		Gesamt für MPEG-7 Schema
	Allgemein	für MPEG-7 Schema	
Dereferenzvektor	#AT x 8 Bytes	772 x 8 Bytes	6176
Content Model Vektor	#CT x 12 Bytes	521 x 10 Bytes	5210
Liste der globalen Elemente	#GE x 8 Bytes	1 x 6 Bytes	6
Parser Stack	#ST x ca.30 Bytes	11 x ca. 30 Bytes	330
Gesamt			11722

Tab. 7.2: Temporäre Puffer beim Kompilieren

Dabei bedeutet:

- #AT: Anzahl aller Typdefinitionen im Schema
- #CT: Anzahl der komplexen Typen
- #GE: Anzahl der globalen Elemente
- #ST: Tiefe der Verschachtelung des Schemas. Relevant sind nur Syntaxkonstrukte, die eine eigene Stufe im Parser Stack erfordern.

Im *Dereferenzvektor* werden die Referenzen zu den Typen des Schemas zwischengespeichert, bis allen Typen eine Bytecodeadresse zugewiesen worden ist. Der *Dereferenzvektor* verknüpft den Namen eines Typs durch eine Referenz auf die entsprechende Stelle im Stringvektor (2 Byte) mit der Adresse des Typs im Bytecodevektor (2 Byte). Der nächste Eintrag im *Dereferenzvektor* wir durch einen Pointer (4 Byte) angehängt. Der *Content Model Vektor* speichert das Inhaltsmodell für die Positionscodierung. Da dieses von den Basistypen eines Typs abhängt, kann die Zuweisung eines Inhaltsmodells erst erfolgen, wenn alle Basistypen bekannt und angelegt sind (vg. 6.3.3 „Inhaltsmodell für die Länge der Positionscodes berechnen“). Aufgebaut ist er aus einer Referenz auf die Adresse des Typen im Bytecodevektor, einer Variable für das Inhaltsmodell und der Referenz auf den nächsten Vektoreintrag. Aus der Liste der globalen Elemente wird die Wurzelauswahlgruppe erzeugt, welche im Wurzelknoten eines Dokuments die Auswahl des Dokumentknotens erlaubt (s. Abb. 4.7). Der *Parserstack* bildet die Hierarchie der Syntaxkonstrukte des XML Schemas nach (vgl. 6.3.1). Er wird beispielsweise benötigt um alle Elemente einer Auswahlgruppe in den hierarchisch darüber liegenden Auswahlzustand aufzunehmen. Es besteht aus einem Feld (1 Byte), das den Typ sichert, einer Referenz auf den Namen (des Typs, des Elements), den Werten von evtl. vorhanden min-/maxOccurs Attributen, einer Referenz für ein evtl. vorhandenes base-Attribut, und einem Zeiger auf eine Liste mit allen hierarchisch eingeschlossenen Konstrukten. Diese Liste hat eine variable Länge, weshalb kein fester Wert für die Größe der Datenstruktur angegeben werden kann.

Ausgangsdaten

Der Bytecode ist der Ausgangsdatensatz des Kompilationsprozesses. Drei separate Datenstrukturen werden erzeugt: die eigentlichen Bytecodezustände, der Stringvektor mit den Zeichenketten aller Namen und der Vererbungsbaum. Für das MPEG-7 Schema sehen die Zahlen wie folgt aus:

	Größe (Byte)
Bytecode – Zustände	33867
Zeichenketten unkomprimiert	25563
Vererbungsbaum	7901
Gesamt	**67331**

Tab. 7.3: Aufteilung der Schemainformation im Bytecode

Die Zeichenketten könnten noch auf 9545 Bytes komprimiert werden. Komprimiert man das komplette MPEG-7 Schema mit ZIP, so ergibt sich eine Größe von 21669 Bytes, also etwa ein Drittel der Information des Bytecodemodells. Man muss jedoch berücksichtigen, dass Kompression der Schemainformation hier nur ein untergeordneter Nebenaspekt ist. Wichtiger ist die schnelle Ausführung der Codierung, für die ein ZIP komprimiertes Schema völlig ungeeignet ist.

Aufschlussreich ist auch eine Statistik der erzeugten Bytecodezustände. Daraus lässt sich entnehmen wie viele Definitionen von komplexen und einfachen Typen (einschließlich der anonymen Typen), und wie viele Element- und Attributdeklarationen das Schema aufweist: diese Zahlen entsprechen den Zahlen der korrespondierenden Bytecodezustände.

Etwa ein Drittel des Bytecodes entfällt auf Zustände, welche die Definitionen von complex- und simpleTypes nachbilden, ein weiteres Drittel auf Zustände für Element- und

Attributdeklarationen und das letzte Drittel auf Strukturzustände (Auswahl Start & End, Sequenz- und Häufigkeitszustände).

Zustand	Anzahl	Prozent	Bytes	Prozent
Type Kopfzustand	522	13.1%	6548	19.3%
Type Endzustand	522	13.1%	1916	5.7%
Simple Type Zustand	270	6.7%	2294	6.8%
Elementzustand	1041	26.1%	9369	27.7%
Attributzustand	447	11.2%	4023	11.9%
Auswahlzustand	118	3.0%	1434	4.2%
Sequenzzustand	407	10.2%	3000	8.9%
Häufigkeitszustand	660	16.6%	5280	15.6%
Gesamt	**3987**		**33864**	

Tab. 7.4: Zustände für das MPEG-7 Schema

Da die Zustände eine variable Länge aufweisen (beim Auswahlzustand ist diese z.B. von der Anzahl der Zweige abhängig), kann man keinen festen Wert für den Informationsbedarf pro Zustand angeben. Es lässt sich aber mit dieser Tabelle auch die durchschnittliche Länge der Zustände ermitteln.

Um den Umfang des Bytecode für andere Schemas abzuschätzen kann man den Aufbau der Bytecodezustände, wie er in Kapitel 6 beschrieben ist heranziehen, und mit den entsprechenden Kenngrößen (Zahl der Typdefinitionen etc.) des Schemas einen Richtwert berechnen.

7.1.2 Speicheranforderungen beim Encodieren

Beim Encodieren lassen sich die Anforderungen an den Speicher grob unterteilen in:

- den Eingangspuffer, er enthält das zu codierende XML Dokument oder Fragment
- die Schemainformation, also den Bytecode
- temporäre Puffer, welche in erster Linie den Zustand des Bytecodeinterpreters für die verschiedenen Hierarchiestufen des Dokuments sichern
- den Ausgangspuffer mit dem codierten Bitstrom

Die notwendige Größe für den Eingangspuffer hängt natürlich von der Größe der XML Dokumente ab, die bearbeitet werden sollen. Prinzipiell besteht auch die Möglichkeit das XML Dokument in mehrere Fragmente aufzuteilen, und diese in Access Units zu codieren. Der Puffer braucht in diesem Fall nur eine Access Unit zu speichern.

Die Information für den Bytecode beträgt für des MPEG-7 Schema ca. 67 KB, wenn das Schema vollständig geladen wird (s. Tab.7.3)

Der temporäre Puffer, welcher beim Encodieren zum Einsatz kommt, ist der *Interpreterstack*. Jede Hierarchiestufe im XML Dokument führt zu einer neuen Stufe im Stack. Die Verschachtelungstiefe, welche in der Praxis auftritt ist typischerweise im Bereich zehn bis fünfzehn. Dementsprechend wird diese Anzahl an Stufen im Stack angelegt. Jede Hierarchiestufe enthält die Datenelemente, welche in Tabelle 6.11 angegeben sind. Einige Datenstrukturen innerhalb einer Hierarchiestufe haben eine feste Größe. Die Datenstruktur *Encoderspur* enthält das Gedächtnis des Interpreters, welche Zustände er bei der Suche nach dem zu codierenden Element bereits passiert hat, und wo gegebenenfalls fortgefahren werden

muss, wenn das Element in einem Zweig nicht gefunden wurde. Deshalb ist die Größe dieser Datenstruktur variabel, bleibt aber normalerweise unter einem Wert von zehn Einträgen. Für die Pfadcodierung ist diese Datenstruktur nicht relevant, nur für die Payloadcodierung. Ebenfalls variabel ist die Liste der Basistypen, sie kann maximal so lang werden, wie die Tiefe des Vererbungsbaums des verwendeten Schemas ist, im Falle des MPEG-7 Schemas also sieben.

Der Datenumfang jeder Stufe ist:

Datenstruktur	**Byte**
Aktuelles Element	2
Aktueller Typ	2
Liste der Basistypen	Max. 7 *4
Encoderspur	Ca. 10 *6
Flags	1
Gesamt	**93**

Tab. 7.5: Temporäre Puffer beim Encodieren

Für ein Dokument mit der Schachteltiefe fünfzehn hat der *Interpreterstack* eine Größe von maximal etwa 1400 Bytes. Da es im MPEG-7 Schema Datentypen gibt, welche direkt oder in einer tieferen Ebene Elemente vom selben Typ enthalten ist eine unendliche Verschachtelung von Elementen möglich, weshalb keine theoretische obere Grenze für die notwendige Zahl an Stufen des *Parserstack* angegeben werden kann.

Der Ausgangspuffer nimmt den codierten Bitstrom auf. Je nach Anwendung kann auch dieser in mehrere Teile aufgeteilt werden. Da die Codierung mit einer Kompression verbunden ist, ist der Speicherbedarf naturgemäß wesentlich geringer als für den Eingangspuffer. Die folgende Tabelle gibt Zahlen für ein Beispiel an:

	Größe (Byte)
Textuelles XML Dokument	106357
	Bitstrom
Payload (mit Content)	15566
Pfad absolut	7144
Pfad relativ	2742

Tab. 7.6: Verhältnis von Eingangs zu Ausgangspuffer für ein Beispiel

Größe des Ausführbaren Programms.

Das Ausführbare Programm hat für das Bytecodewerkzeug und eine x86 Plattform unter einem win32 Betriebssystem eine Größe von 148 KB. Allerdings sind hier die Funktionen zum Parsen von XML Dokumenten noch nicht enthalten. (Für diesen Zweck wurde der Xerces Parser [APA] in Form einer dynamisch gelinkten Bibliothek verwendet.). Diese würden jedoch nur einen geringen Beitrag zum Code liefern.

Das Ausführbare Programm könnte noch entsprechend seinen Funktionen aufgeteilt werden in einen reinen Compiler, einen reinen Encoder und einen reinen Decoder. Für einen reinen Decoder bräuchten die Funktionen zum Kompilieren des Bytecodes sowie zum Encodieren nicht mit eingebunden werden.

Insgesamt bleibt festzustellen, dass die Größe des temporären Puffers und des codierten Bitstroms im Vergleich zur Schemainformation und dem zu codierenden XML Dokument wenig ins Gewicht fällt. Auch das ausführbare Programm ist mit 148 KB eine relevante Größe.

7.1.3 Speicheranforderungen beim Decodieren

Die Schemainformation und die Größe der temporären Puffer sind beim Decodieren im wesentlichen die selben wie beim Encodieren. Lediglich die Datenstruktur *Encoderspur* ist beim Decoder nicht erforderlich, da kein Suchvorgang nach Elementen im Bytecode stattfindet. Der temporäre Puffer hat deshalb beim Decodieren nur etwa 30 Bytes pro Stufe.

Im Eingangspuffer wird der Inhalt der binär codierten Access Units angeliefert, im Ausgangspuffer wird das XML Dokument rekonstruiert. Dabei ist es von der Anwendung abhängig, in welcher Form das XML Dokument dargestellt werden soll, als textuelles Dokument oder in Form einer Baumstruktur. Weitere Überlegungen dazu wie ein decodiertes Dokument kompakt dargestellt werden kann finden sich in Absatz 6.7.1.

7.2 Anforderungen an die Rechenleistung

Für die praktische Einsetzbarkeit eines Datenformats für XML ist der Rechenaufwand bei der Codierung neben der Funktionalität und den Speicheranforderungen ein entscheidendes Kriterium. Ein Datenformat ist auch bei noch so hoher Kompression und noch so fortschrittlichen Übertragungseigenschaften kaum anwendbar, wenn hohe Latenzzeiten bei der Codierung oder Decodierung in Kauf genommen werden müssen, weil der Algorithmus eine hohe Rechenleistung benötigt.

Die Komplexität eines Algorithmus analytisch zu berechnen ist schwierig und fehlerträchtig. Verlässlicher, wenn auch nicht mit dem Anspruch das theoretische Minimum zu erfassen ist es die Anforderungen an die Rechenleistung anhand einer optimierten Implementierung zu messen bzw. abzuschätzen.

Die Simulationen, welche in diesem Abschnitt diskutiert werden, sind auf einem Pentium III System mit 650 MHz Taktfrequenz unter dem Betriebssystem Linux durchgeführt worden. Als Testdaten wurden Beschreibungen aus dem MPEG-7 Testdatensatz verarbeitet, welche nach dem MPEG-7 Schema codiert wurden.

7.2.1 Methodik des Profiling

Für das Profiling wurden zwei verschiedene Werkzeuge mit unterschiedlichen Eigenschaften eingesetzt. Zum einen der Standardprofiler *gprof*, der ein statistisches Profiling durchführt. Er ermittelt die Zeit, die in den einzelnen Funktionen bei der Programmausführung verbracht wird. Zum anderen der Profiler *iprof* „Instruction Level Profiler“ [Kuhn98], der die Zahl der ausgeführten Instruktionen aufgeschlüsselt nach den verschiedenen Gruppen der Maschinenbefehle bei der Programmausführung aufzeichnet.

Funktionsprinzip von gprof

Bei *gprof* wird mit einer vorgegebenen Frequenz (z.B. 100 mal pro Sekunde) die Position des Programmzählers ermittelt und in ein Histogramm eingetragen, in welcher Funktion sich der Programmzähler gerade befindet (auf der horizontalen Achse des Histogramms sind die Funktionen gelistet, auf der vertikalen Achse die Zahl der Aufrufe). Der Funktion wird dann eine Ausführungszeit entsprechend dem Zeitintervall der Abtastfrequenz zugeschlagen.

Dieses Verfahren ist mit einer statistischen Unsicherheit behaftet. Kleinere Funktionen, die selten aufgerufen werden erscheinen unter Umständen gar nicht im Histogramm, obwohl sie aufgerufen wurden (allerdings sind diese Funktionen dann auch solche, die wenig ins Gewicht fallen und deshalb z.B. bei Optimierungen nicht bevorzugt betrachtet werden). Man kann diese statistische Unsicherheit reduzieren, indem man das Programm mehrfach aufruft und über die einzelnen Aufrufe mittelt.

Gprof stellt auch den sogenannten „Call-Graph“ auf. Dieser enthält Informationen welche Funktionen innerhalb einer Funktion aufgerufen werden, bzw. von welchen Funktionen eine Funktion aufgerufen wird. Auf diese Weise kann auch die Zeit, welche in den Unterprogrammen einer Funktion verbracht wird zur gesamten Ausführungszeit dieser Funktion hinzuaddiert werden. Außerdem kann ermittelt werden, welcher Anteil der Zeit in einer Funktion selbst, und welcher Anteil in den Unterprogrammen verbracht wird.

Funktionsprinzip von iprof

Ein Maschinenprogramm lässt sich unterteilen in sogenannte Basic Blocks. Das sind Sequenzen von Maschinenbefehlen die nacheinander ausgeführt werden, d.h. in einem Basic Block dürfen keine Sprung- oder (bedingte) Verzweigungsbefehle enthalten sein. Ein solcher Befehl beendet einen Basic Block, und ein neuer beginnt. Durch eine Option, die beim Kompilieren der Quellen eingeschaltet wird, kann die Information, wie oft ein Basic Block durchlaufen wird extrahiert werden. Dafür wird in dem ausführbaren Maschinenprogramm extra Code erzeugt. Durch eine Dissassemblierung des Maschinencodes ist außerdem bekannt, welche Maschinenbefehle in jedem Basic Block auftreten. Damit kann für jeden Befehl angegeben werden, wie oft er bei einem Programmdurchlauf aufgerufen wurde. *Iprof* Gruppiert die Maschinenbefehle in verschiedene Befehlsklassen (Arithmetic, Data, etc.) die eine detailliertere Aussage z.B. über die zu erwartende Speicherbandbreite erlauben.

7.2.2 Messung der Bytecode-Operationen mit iprof

Die folgende Tabelle gibt eine Übersicht der Gesamtzahl und Verteilung der Maschinenbefehle in den verschiedenen Betriebsmodi des Bytecodewerkzeugs. Für die Encodierung / Decodierung wurde als Beispiel ein 104 KB großes Testdokument verwendet.

Operation	Arith-metic	Data	Logic	Rotate	Jump/ Test	Stack	Total
Compile	25.24%	11.94%	1.36%	0.66%	52.50%	8.22%	126.1 M
EncodePayload	17.95%	26.59%	4.09%	3.83%	30.45%	16.71%	10.10 M
DecodePayload	21.34%	26.95%	3.88%	4.8%	28.29%	14.65%	7.17 M
EncodePathABS	17.83%	26.78%	3.99%	2.8%	30.36%	17.78%	10.32 M
DecodePathABS	16.34%	28.36%	4.28%	2.20%	31.54%	16.71%	21.91 M
EncodePathREL	17.71%	27.41%	3.60%	1.84%	30.01%	18.98%	8.61 M
DecodePathREL	17.55%	27.76%	3.67%	2.07%	31.18%	17.29%	7.11 M

Tab. 7.7: Aufteilung der Maschinenbefehle bei der Bytecodeverarbeitung

Ein Kernergebnis dieser Simulation ist, dass der Kompilationsprozess erheblich mehr Rechenleistung benötigt (126 Mio. Instruktionen), als das Encodieren/Decodieren einer 104 KB großen XML-Beschreibung (entspricht etwa 50 Seiten Text). Das ist ein gewichtiges Argument dafür den Algorithmus in einen Kompilationsprozess und einen Codier-/ Decodierprozess aufzuteilen.

Bei der Verarbeitung einer Payload fällt auf, dass die Decodierung nur 70% des Rechenaufwands der Encodierung benötigt. Das ist in erster Linie dadurch begründet, dass beim Decodieren die Folge der durchlaufenen Zustände durch den eingelesenen Bitstrom festgelegt ist, beim Encoder dagegen das zu codierende Element in den Bytecodezuständen erst gesucht werden muss, wobei auch Zustände im Bytecode vergeblich durchlaufen werden können.

Die Decodierung eines Absoluten Pfades benötigt laut Messung den doppelten Rechenaufwand wie die Codierung. Dieses zunächst überraschende Ergebnis liegt an der Art der Durchführung des Experiments: das Beispieldokument wurde depth-first geparst, und der Pfad zu jedem Element und Attribut codiert. Dabei haben zwei aufeinanderfolgende Pfade (die ja beim Absoluten Pfad in der Wurzel des Dokuments beginnen) üblicherweise einen recht großen Teil gemeinsam. Für diesen Teil stehen die notwendigen Codes bereits im *Interpreterstack*, ein Suchen im Bytecode ist nur noch für den neuen Teil des Pfades erforderlich. Anders beim Decoder: hier liegt keine Information vor, welcher Teil vom letzten Pfad übernommen werden kann, deshalb muss der Pfad von der Wurzel ganz neu aufgebaut werden. Im Allgemeinen kann nicht davon ausgegangen werden, dass auf der Encoderseite zwei aufeinanderfolgend codierte Pfade einen großen Teil gemeinsam haben, weil größere Teile der Beschreibung auch mit der Payload codiert werden. Andererseits bietet es sich an die deutliche Reduktion des Rechenaufwands beim Encoder auszunutzen, wenn dieser Fall auftritt.

Bei der Verarbeitung eines Relativen Pfades ist das Verhältnis wieder normal: der Encoder benötigt etwas mehr Maschinenbefehle als der Decoder. Allerdings ist der Unterschied nicht so groß wie bei der Payload, weil bei der Pfadcodierung sowohl beim Encoder als auch beim Decoder immer ein Suchvorgang stattfindet: beginnend mit dem Typkopfzustand wird der Inhalt des Typen abgearbeitet, bis das Element/Attribut gefunden wird (Encoder) oder bis der Zählerstand für die Kinder des Typs dem gelesenen Wert entspricht (Decoder).

7.2.3 Profil nach Arbeitsschritten

Im folgenden werden die verschiedenen Operationsmodi des Bytecodewerkzeugs noch detaillierter nach Arbeitsschritten aufgeschlüsselt, um mehr Information zu erhalten welche Teile des Algorithmus besonders rechenintensiv sind, und gegebenenfalls Potential für weitere Optimierungen bieten. Die Daten wurden mit dem Profiler *gprof* generiert. Um die statistischen Effekte abzudämpfen wurde der untersuchte Programmteil einhundert mal mit identischen Parametern aufgerufen, und die Ergebnisse anschließend durch einhundert geteilt. Da vor allem der relative Anteil der einzelnen Arbeitsschritte interessiert werden hier die absoluten Zeitintervalle nur angegeben, weil sie die Grundlage der Messung bilden.

Kompilierung des Schemas:

Parsen des Schemas	7 ms	3,7 %
Anlegen der Zustände	87 ms	45,8 %
Anlegen Vererbungsbaum	42 ms	22,1 %
Nachverarbeitung Bytecode	51 ms	26,8 %
Speichern des Bytecodes	3 ms	1,6 %
Gesamt:	190 ms	

Tab. 7.8: Verteilung der Rechenleistung bei Kompilieren

Das Anlegen der Bytecodezustände als Kernaufgabe des Kompilationsprozesses benötigt den größten Anteil der Rechenleistung. Hier werden die Zustandsmatrizen im Bytecodevektor angelegt und die bereits vorhandenen Informationen in den Zeigerreferenzen eingetragen. Auch der Stringvektor wird hier erzeugt. Ein gewichtiger Teil der Rechenleistung wird dabei für folgende Aufgabe gebraucht: der Stringvektor soll alle Namen nur genau einmal enthalten. Zum einen, weil ein wiederholtes Erscheinen eines Namens ineffizient ist, zum anderen, weil die Referenzen, welche z.B. innerhalb eines Elementzustands auf den Namen des Typs zeigen eindeutig sein müssen. Sonst werden diese Referenzen bei der Nachverarbeitung nicht richtig aufgelöst. Deshalb muss immer bevor ein neuer Name in den Stringvektor eingesetzt wird überprüft werden, ob der Name schon irgendwo im Stringvektor steht. Da dieser im Verlauf der Kompilation mit vielen Einträgen versehen wird, müssen entsprechend viele Namen überprüft werden. Die Liste der Namen, die bereits im Stringvektor stehen wird im Laufe der Verarbeitung länger. Die Komplexität dieses Arbeitsschrittes wächst also in etwa quadratisch mit der Zahl der Namen, die im Schema auftreten. Hier könnte man noch eine Indexierung einführen, um die Suche zu beschleunigen.

Der zweitgrößte Teil ist die Nachverarbeitung der Bytecodezustände. Die wesentlichen Aufgaben, die durchgeführt werden, sind in Kapitel 6.3.5 nachzulesen (das Anlegen des Vererbungsbaums wurde dort in der Nachverarbeitung aufgenommen, erscheint in obiger Auflistung aber als eigener Punkt).

Das Anlegen des Vererbungsbaums ist der drittgrößte Posten. Der Hauptaufwand liegt hier beim Eintrag eines Knotens in der Liste der Kinder eines anderen Knotens, da wiederum überprüft werden muss, ob einer der beiden Knoten schon angelegt ist, oder neu angelegt werden muss. Dagegen spielt der Durchlauf des fertig angelegten Vererbungsbaums zur Ermittlung der Typecodes und der Größe des Subbaums eines Knotens kaum eine Rolle.

Encodierung Payload Modus:

Laden des Bytecode	1,3 ms	5,1 %
Elemente suchen	8,4 ms	32,8 %
Ende suchen	3,9 ms	15,2 %
Attribute encodieren	3,7 ms	14,5 %
Inhalt encodieren	3,1 ms	12,1 %
VLC Codes schreiben	4,9 ms	19,1 %
Bitstrom schreiben	0,3 ms	1,2 %
Gesamt	25,6 ms	

Tab. 7.9: Verteilung der Rechenleistung beim Encodieren einer Payload

Beim Encodieren einer Payload stellt erwartungsgemäß das Suchen der zu codierenden Elemente im Bytecode die rechenaufwändigste Teilaufgabe dar. Auch das Suchen des Endes eines Typs (vgl. 6.4.3) erfordert gehörigen Aufwand. Dagegen ist der Aufwand zum Laden des Bytecodes mit 1,3 ms sehr gering, wenn man es mit einer Neukompilation des Schemas vergleicht (190 ms, siehe vorherige Tabelle). Besonders auffällig ist auch, dass das Schreiben der VLC Codes stark ins Gewicht fällt. Diese lassen sich auf einem Standardprozessor schwer verarbeiten, da sie häufig auf verschiedene Bytes aufgeteilt, und mittels Schiebe- und Logikoperationen an die Bytegrenzen und die Position innerhalb eines Bytes angepasst werden müssen.

Decodierung Payload Modus:

Laden des Bytecode	1,3 ms	8,1 %
Bitstrom laden	0,3 ms	1,8 %
VLC Codes lesen	5.5 ms	34,2 %
Elemente decodieren/Ende suchen	4,8 ms	29,8 %
Attribute decodieren	3,3 ms	20,5 %
Sonstige	0,9 ms	5,6 %
Gesamt	16.1 ms	

Tab. 7.10: Verteilung der Rechenleistung beim Decodieren einer Payload

Das Decodieren im Payloadmodus ist deutlich weniger aufwändig als das encodieren. Vor allem die Punkte „Elemente decodieren und Ende suchen“ sind mit 4.8 ms im Vergleich zu der entsprechenden Zahl beim Encodieren (8,4 ms + 3,9 ms = 12,3 ms) nur ein drittel so rechenintensiv (wie bereits erwähnt ist der Grund dafür, dass beim Decodieren die Folge der Zustände durch den eingelesenen Bitstrom bestimmt wird, und nicht wie beim Encoder erst der richtige Pfad gesucht werden muss). Das Lesen der VLC Codes fällt deshalb mit 34,2 % noch stärker ins Gewicht als die VLC-Verarbeitung beim Encoder.

Encodierung Absoluter Pfad:

Laden des Bytecode	1,3 ms	5,2 %
Sonstige	2,6 ms	10,5 %
Elemente suchen	14,5 ms	58,5 %
Attribute suchen	3,1 ms	12,5 %
VLC Codes schreiben	3,0 ms	12,1 %
Bitstrom schreiben	0,3 ms	1,2 %
Gesamt	24,8 ms	

Tab. 7.11: Verteilung der Rechenleistung beim Encodieren eines absoluten Pfades

Auch beim Encodieren eines Pfades ist das Suchen der Elemente der gewichtigste Posten. Der Aufwand ist sogar noch größer als bei der Payloadcodierung, da für jedes neu zu codierende Element der Suchvorgang beim Kopfzustand des Typen beginnt (bei der Payloadcodierung dagegen beim letzten codierten Element). Deshalb sind die Pfade, welche im Bytecode durchlaufen werden auch länger.

Decodierung Absoluter Pfad:

Laden des Bytecode	1,7 ms	3,7 %
Bitstrom laden	0,3 ms	0,6 %
VLC Codes lesen	4,0 ms	8,7 %
Elemente/Attribute suchen	27,0 ms	58,7 %
Sonstige	13,0 ms	28,3 %
Gesamt	46,0 ms	

Tab.7.12: Verteilung der Rechenleistung beim Decodieren eines Absoluten Pfades

Beim Decodieren eines Absoluten Pfads ist wiederum das Suchen der Elemente und Attribute der größte Posten. Anders als beim Encodieren kann man hier nicht zuwischen Elementen und Attributen unterscheiden, beides wird in derselben Funktion durchgeführt. Erst das letzte Codewort entscheidet, ob ein Element oder ein Attribute adressiert wurde. Wie in Abschnitt 7.2.2 beschrieben ist der Gesamtaufwand größer als beim Encodieren.

Encodierung Relativer Pfad:

Laden des Bytecode	1,4 ms	7,7 %
Sonstige	2,8 ms	15,5 %
Elemente suchen	10,8 ms	59,7 %
Attribute suchen	2,1 ms	11,6 %
VLC Codes schreiben	0,9 ms	5,0 %
Bitstrom schreiben	0,1 ms	0,5 %
Gesamt	18,1 ms	

Tab. 7.13: Verteilung der Rechenleistung beim Encodieren eines Relativen Pfades

Die Zahlen beim Encodieren des Relativen Pfads ähneln denen des Absoluten Pfads, lediglich das Schreiben der VLC Codes fällt deutlich geringer aus, da die Relativen Pfade im Durchschnitt kürzer sind, und deshalb weniger Codeworte geschrieben werden müssen.

Decodierung Relativer Pfad:

Laden des Bytecode	1,7 ms	12,2 %
Bitstrom lesen	0,1 ms	0,7 %
VLC Codes lesen	1,1 ms	7,9 %
Elemente/Attribute suchen	8,1 ms	58,3 %
Sonstige	2,9 ms	20,9 %
Gesamt	13,9 ms	

Tab.7.14 Verteilung der Rechenleistung beim Decodieren eines Relativen Pfades

Das Decodieren eines relativen Pfads ist sparsamer als das Encodieren, v.a. weil das Suchen der Elemente und Attribute mit 8,1 ms im Vergleich zu (10,8 ms + 2,1 ms = 12,9 ms) schneller erfolgt. Ursache dafür ist, dass der Encoder bei der Suche nach dem zu codierenden Element die Zeichenketten der Namen des aktuellen Elements im Bytecode mit dem gesuchten Element vergleichen muss (das gleiche gilt für Attribute). Der Decoder dagegen zählt nur die Elemente, und muss ganze Zahlen vergleichen, was etwas sparsamer ist.

Bewertung

Einige Operationen haben sich beim Profiling als aufwändiger erwiesen, als man zunächst vermutet hätte. Zum einen benötigt die Verarbeitung von Zeichenketten einen erstaunlich hohen Aufwand. Das häufige Vergleichen von Namen bei diversen Suchvorgängen macht sich hier bemerkbar. Zum anderen ist das Schreiben der Variable Length Codes im Vergleich zur geringen „bewegten Informationsmenge“ rechenintensiv. Das Decodieren ist (bis auf den nicht allgemeingültigen Sonderfall beim codieren Absoluter Pfade, welche durch depth-first Parsen des XML Dokuments generiert wurden) weniger aufwändig als das Encodieren, was im Sinne der Anwendungen ist, da der Decoder häufig weniger leistungsfähig sein wird, als der Encoder.

7.2.4 Normierung der Ergebnisse

Die hier vorgestellten Ergebnisse wurden experimentell durch Verarbeitung eines Beispieldokuments mit dem MPEG-7 Schema generiert. Damit ließ sich die Verteilung der erforderlichen Rechenleistung innerhalb des Algorithmus ermitteln. Die Kernaufgabe sowohl bei der Pfadcodierung als auch bei der Payloadcodierung ist die Suche des zu codierenden Elements im Bytecode. Um diese zum Aufbau des XML Schemas in Beziehung zu setzen und die Betrachtungen in Abschnitt 7.3 vorzubereiten sollen hier einige Kenngrößen für das MPEG-7 Schema diskutiert werden.

Länge der BVC Tabellen

Für die Länge des Suchvorgangs bei der Pfadcodierung ist die durchschnittliche Länge der BVC Tabellen mit maßgebend. Das folgende Histogramm listet die Verteilung der Länge dieser Tabellen für alle komplexen Typen des MPEG-7 Schemas auf.

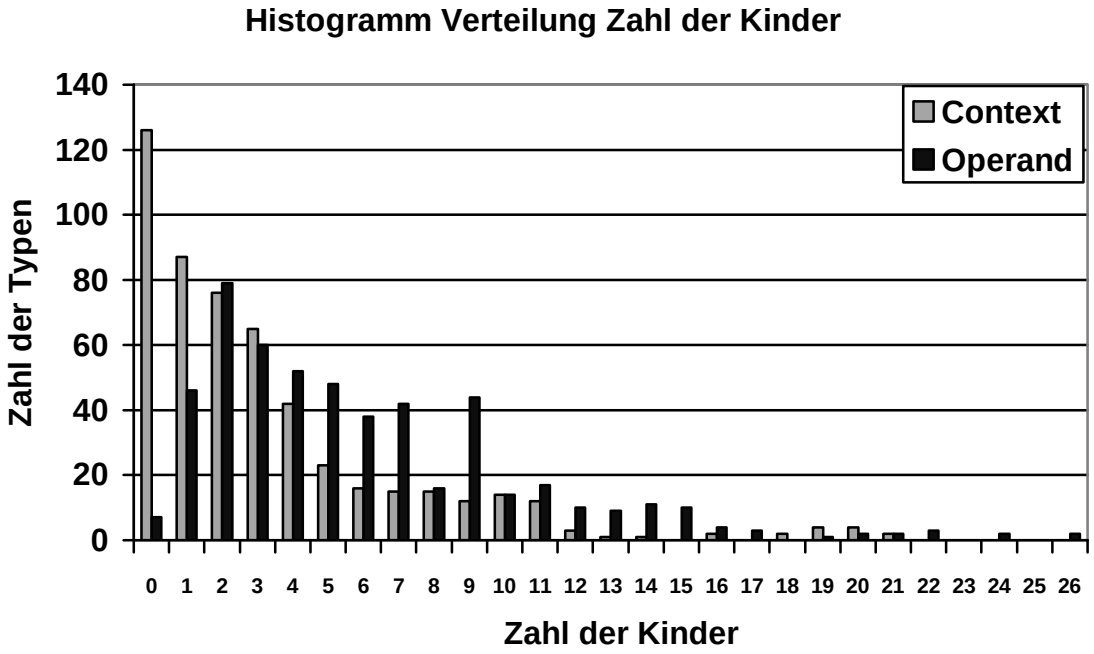


Abb. 7.1: Verteilung der Länge der Codetabellen über den Typdefinitionen

In der Abbildung ist auf der Hochachse die Zahl der Typen angetragen, welche die auf der Querachse angegebene Zahl von Kindern hat, getrennt für *Operand*- und *Context BVC-Tabellen*. Es werden nur die im XML Schema definierten Kinder angegeben, die Einträge in den *BVC-Tabellen* für die Referenz zum Vater, den Erweiterungs- oder den *Termination-Code* sind nicht enthalten, da für diese kein Suchvorgang erforderlich ist. Bei den Typen, welche auch für die *Operand BVC-Tabellen* keine Einträge haben handelt es sich um abstrakte Datentypen, welche nicht als Typen für Elemente in Frage kommen, sondern nur als Basis für andere, vererbte Datentypen dienen. Durchschnittlich enthalten die *Context BVC-Tabellen* 3.3 Einträge und die *Operand BVC-Tabellen* 6.0 Einträge.

Statistik der Zustände

Modus	Elemente	Passierte Zustände	Zustände / Element	Max Zust. / Element	Zust. Ende suchen	Zust. Ende / Element
Encodieren Payload	1782	14868	8.34	41	7386	4.14
Decodieren Payload	1782	10585	5.94	18	7041	3.95
Encodieren Pfad ABS	1782	28696	14.34	65	-	-
Decodieren Pfad ABS	7353	110093	14.97	65	-	-
Encodieren Pfad REL	1782	28696	14.34	65	-	-
Decodieren Pfad REL	1782	28696	14.34	65	-	-

Tab. 7.15: Zahl der passierten Zustände bei verschiedenen Arbeitsschritten

Die Tabelle gibt eine Übersicht über die Zahl der Zustände, die beim Encodieren und Decodieren in den verschiedenen Modi passiert wird. Als XML Dokument dient wiederum die 104 KB große Beschreibung, welche bereits in den vorangegangenen Abschnitten verwendet wurde. In der ersten Spalte ist die Zahl der prozessierten Elemente aufgetragen, in der zweiten Spalte die Zahl der passierten Zustände, in der dritten Spalte die durchschnittliche Zahl der Zustände pro Element, in der vieren Spalte die maximale Zahl von Zuständen welche für die Codierung oder Decodierung eines Elements abgearbeitet wurde, in der fünften Spalte die Zahl der Zustände für das Suchen des Endes eines Typs, in der sechsten Spalte diese Zahl bezogen auf die Zahl der codierten Elemente.

Die Zahl der verarbeiteten Elemente ist fast immer 1782, da es sich um dasselbe Dokument handelt, nur beim decodieren des absoluten Pfades ist diese Zahl deutlich größer, da der Pfad immer von der Wurzel des Dokuments aufgebaut wird, und manche Elemente deshalb mehrfach gesucht werden.

Die nächste Spalte gibt die Zahl der passierten Bytecodezustände an, wie sie bei dem Experiment mitprotokolliert wurde. Erfasst wurden alle Zustandstypen.

Aufschlussreich ist die dritte Spalte, welche die Zahl der durchschnittlich für die Codierung / Decodierung eines Elements passierten Zustände auflistet. Hier sollte sich die qualitative Argumentation aus Abschnitt 7.2.2 und 7.2.3 mit den Profiling-Ergebnissen widerspiegeln. Man sieht, dass beim Decodieren einer Payload wie erwartet weniger Zustände passiert werden (ca. 26%), weil beim Decodieren kein Suchvorgang stattfindet. Ebenfalls aus den Zahlen ersichtlich ist der höhere Aufwand beim Encodieren und Decodieren von Pfaden, da dort der Suchvorgang immer beim Kopfzustand des Typen beginnt, vgl. 6.4.5.

Die maximale Zahl von Zuständen, die für die Verarbeitung eines Elements passiert wurde variiert deutlich: sie ist für die Encodierung einer Payload wesentlich größer als für die Decodierung (bei dem Element mit der größten Zahl an Zuständen wurden viele Bytecodepfade erfolglos durchsucht, bis das Element gefunden wurde), ebenfalls aufwändiger ist hier die Verarbeitung der Pfade.

Das Suchen des Endes eines Typs ist nur für den Payloadmodus relevant, die Zahlen ähneln sich hier für Encoder und Decoder.

Die Zahl der Zustände, die für die Codierung passiert wird kann als Maß für die Komplexität des Suchvorganges verwendet werden. Sie hängt vom Aufbau des Schemas ab. Im folgenden Abschnitt wird dies detaillierter diskutiert.

7.3 Komplexitätsmodell des Algorithmus

Mit Hilfe des Bytecodemodells und den bisherigen Ergebnissen soll nun eine Metrik für die Komplexität des BiM Algorithmus entwickelt werden. Ziel ist es eine allgemeine Abschätzung des Aufwandes für die Encodierung und Decodierung zu erhalten, in welche die wesentlichen Kenngrößen des XML Schemas und des verarbeiteten XML Dokuments eingehen, ohne dass Details des Aufbaus des Schemas oder des Dokuments bekannt sein müssen, etwa wie die Typdefinitionen im einzelnen aufgebaut sind, oder welche Elemente genau im Dokument auftreten.

Für die Abschätzung der Komplexität soll nur das Grundprinzip des Algorithmus betrachtet werden, die syntaxbasierte Codierung. Randaspekte, wie die Codierung der Nutzdaten, welche von spezialisierten Codecs übernommen wird, sind zu spezifisch für eine universelle Betrachtung.

Die wesentlichen Kenngrößen für den Einfluss des Schemas sind:

- *Die Anzahl der Elementeund Attribute pro Typ*
  Diese ist vor allem bei der Pfadcodierung für die Länge der Tabellen und damit für die Länge des Suchvorgangs entscheidend (vgl. Abb. 7.1).
- *Die Anzahl bzw. die Häufigkeit von Syntaxkonstrukten, die eine Auswahl erlauben, d.h. occurrence-Attribute und choice- oder all-Gruppen.*
  Die Anzahl der Zustände, welche bei der Payloadcodierung durchsucht werden hängt davon ab, wie viele Alternativen geprüft werden müssen.
- *Die Anzahl der Typen, für die Vererbung möglich ist, und die Tiefe des Vererbungsbaums*
  Diese Kenngrößen bestimmen den effektiven Inhalt eines Typs, sowie den Aufwand für der Verarbeitung von Polymorphismus.
- *Die Anzahl der möglichen Attribute pro Typ*
  Optionale Attribute müssen überprüft werden, die dafür erforderliche Rechenzeit ist in der Zahl der tatsächlich codierten Attribute (welche das XML Dokument vorgibt) noch nicht enthalten.

Die wesentlichen Kenngrößen für den Einfluss des XML-Dokuments sind:

- *Die Anzahl der enthaltenen Elemente*
  Jedes zu codierende Element liefert einen Beitrag zur Codierzeit, die Schemakomplexität steuert einen Skalierungsfaktor bei. Dabei muss man unterscheiden zwischen Elementen von einfachem Typ und Elementen von komplexem Typ. Bei Elementen von komplexem Typ muss ein Suchvorgang im Inhalt des Typen analysiert werden, bei Elementen von einfachem Typ werden Nutzdaten codiert.
- *Die Anzahl der Attribute*
  Die Attribute liefern ebenfalls einen Beitrag zur Codierzeit.

7.3.1 Konzept und Annahmen

Die genaue Anzahl von Zuständen, die bei der Codierung abgearbeitet wird, hängt vom Aufbau der einzelnen Datentypen der Elemente ab, die im Dokument codiert sind. Es ist jedoch für eine Komplexitätsabschätzung nicht praktikabel auf den Typ eines codierten Elements einzugehen, da dann über 500 verschiedene Fälle berücksichtigt werden müssen. Deshalb wird hier der Rechenaufwand für die Codierung oder Decodierung eines Typs mit einem durchschnittlichen Datentyp abgeschätzt, der sich durch Mittelung der Anzahl der diversen Syntaxkonstrukte im Schema ergibt.

Annahme 1: mittlere Anzahl von Syntaxkonstrukten der Datentypen.
In die Abschätzung wird die mittlere Anzahl von Auswahlzuständen, Häufigkeitszuständen und Elementen pro Typ eingehen. Dazu wird zunächst das Verhältnis der Syntaxkonstrukte im Schema gebildet:

$$\textit{Anzahl der Elementzustände pro Typ} = \frac{\textit{Zahl der Elementdeklarationen}}{\textit{Zahl der komplexen Typen}} = \frac{D_{El}}{D_{KT}}$$

$$\textit{Anzahl der Auswahlzustände pro Typ} = \frac{\textit{Zahl der Auswahlgruppen}}{\textit{Zahl der komplexen Typen}} = \frac{D_{Ch}}{D_{KT}}$$

$$\textit{Anzahl der Häufigkeitszustände pro Typ} = \frac{\textit{Zahl der min/max-Occurs Attribute}}{\textit{Zahl der komplexen Typen}} = \frac{D_{Occ}}{D_{KT}}$$

Annahme 2: mittlerer Umfang des Inhalt eines Typs
Durch Vererbung kann die tatsächliche Zahl der Konstrukte eines Typs größer sein als durch obige Verhältnisbildung berechnet. Ein zusätzlicher Faktor muss eingefügt werden, der die mittlere Tiefe des Vererbungsbaums berücksichtigt. Der durchschnittliche effektive Inhalt eines Typs ist damit das (1 + Tiefe_{VB}) – fache der durch obige Mittelwerte berechneten Zahlen. Die Größe Tiefe_{VB} ist dabei die durchschnittliche Anzahl von Basistypen eines Typs.

Beispiel: Typ A mit zwei Basistypen B und C

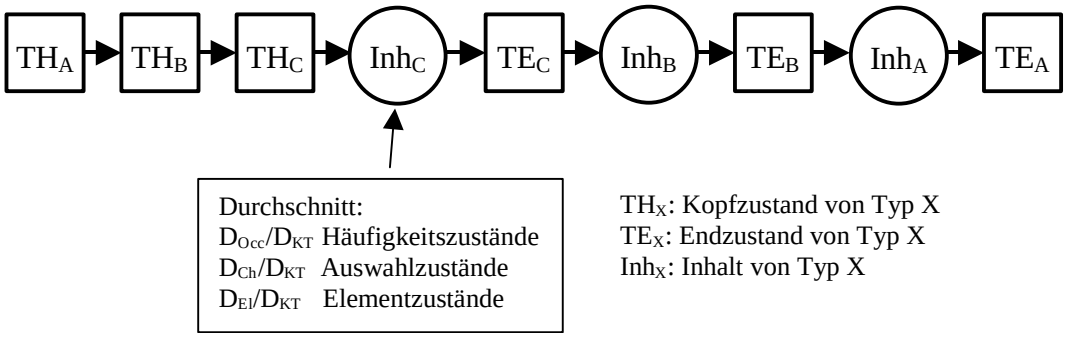


Abb. 7.2: Effektiver Inhalt eines vererbten Typen

Abbildung 7.2 verdeutlicht die Bildung des effektiven Inhalts eines Typs (Zahl der Auswahl-/ Häufigkeits-/Elementzustände) aus dem Schema unter Berücksichtigung der Vererbung. Der Typ C ist dabei der Basistyp von Typ B, dieser wiederum der Basistyp von Typ A.

Annahme 3: Codierung von Auswahlgruppen und Häufigkeiten.
Bei der Codierung von Auswahlgruppen und Häufigkeiten muss der Inhalt der entsprechenden Gruppe überprüft werden. Bei einer Auswahlgruppe werden alle Zweige nacheinander überprüft. Bei der Payloadcodierung wird angenommen, dass das zu codierende Element in einem der Zweige gefunden werden muss. In diesem Fall ist die mittlere Zahl der überprüften Zweige $(L_{ch}+1)/2$, wobei L_{ch} die mittlere Zahl der Zweige der Auswahlgruppen ist. Bei der Pfadcodierung, bei der das gesuchte Element nicht in einem Zweig der Auswahlgruppe auftreten muss, wird angenommen, dass die volle Anzahl an Zweigen überprüft wird. Da das Bearbeiten eines Zweiges immer einen Auswahlzustand und einen Auswahl-Endzustand umfasst muss noch ein Faktor zwei berücksichtigt werden. Häufigkeitszustände werden ebenfalls zweimal durchlaufen (vor und nach der Überprüfung des Inhalts der Häufigkeit).

Annahme 4: mittlere Häufigkeit des Auftretens.
Um den Erwartungswert für bestimmte Größen abzuschätzen wird der minimal mögliche Wert dieser Größe zum maximal möglichen Wert addiert, und diese Summe durch zwei dividiert. Der maximal mögliche Wert kann seinerseits durch Mittlung entstanden sein. Es wird also eine Gleichverteilung der Wahrscheinlichkeit aller Möglichkeiten angenommen.

Annahme 5: Codierzeit ist proportional zur Zahl der verarbeiteten Zustände
Interessant ist letztlich die Rechenzeit, welche für die Codierung eines Dokuments benötigt wird. Diese wird durch die Anzahl der verarbeiteten Bytecodezustände abgeschätzt. Dabei wird angenommen, dass die Verarbeitungszeit für einen Zustand in etwa konstant ist, unabhängig vom Typ des Zustandes. Dies ist gerechtfertigt, da im wesentlichen die selben oder ähnliche Arbeitsschritte durchgeführt werden müssen: Identifikation des Typs des Zustands, Prüfung einer Bedingung oder ein Vergleich sowie Ermittlung des Folgezustands.

Im folgenden werden zwei Arten von Symbolen verwendet: N_{ZustX} ist die Zahl der verarbeiteten Zustände der entsprechenden Kategorie und $N_{Element}$, $N_{ElementComplex}$, $N_{Element\&Attribut}$ sind die Zahlen der im verarbeiteten XML Dokument auftretenden Elemente, Elemente von komplexem Typ, Elemente und Attribute etc. Außerdem werden die in diesem Abschnitt eingeführten Symbole verwendet, die aus dem Schema abgeleitet sind.

7.3.2 Encodierung einer Payload

In die Komplexitätsabschätzung der Payloadcodierung gehen als Kenngrößen des Dokuments die Anzahl der Elemente von komplexem und einfachem Typ ein. Diese werden mit einem schemaabhängigen Faktor skaliert.

Die Anzahl der Zustände für die Codierung eines Elements ergibt sich wie folgt: Wird der Typkopfzustand des Elements aufgerufen, so wird zunächst in die Basistypen verzweigt, entsprechend werden auch die Typkopfzustände der Basistypen aufgerufen. Für die Anzahl an Basistypen, die zu erwarten sind gibt die mittlere Tiefe des Vererbungsbaums einen Anhaltspunkt. Für das MPEG-7 Schema und komplexe Typen ist die mittlere Tiefe 2,59, d.h. jeder Typ hat im Mittel diese Anzahl an Basistypen. Für jeden Typ wird der Kopfzustand und der Endzustand aufgerufen, das ergibt einen Faktor 2 in der Formel 7.1.

$$N_{ZustTypeHead\&End} = N_{ElementComplex}(1 + Tiefe_{VB}) \cdot 2 \tag{7.1}$$

In jedem Typ gibt es eine Anzahl von Häufigkeits- und Auswahlzuständen. Die zu erwartende Anzahl an Häufigkeitszuständen pro Typ wird abgeschätzt durch die Gesamtzahl an Häufigkeitszuständen des Schemas dividiert durch die Zahl der Typdefinitionen. Da jeder Häufigkeitszustand aufgerufen wird, bevor in dessen Inhalt verzweigt wird und wenn er verlassen wird, muss er doppelt berücksichtigt werden.

$$N_{ZustOccurrence} = N_{ElementComplex}(1 + Tiefe_{VB})\left[2\frac{D_{Occ}}{D_{KT}}\right] \tag{7.2}$$

Bei der Auswahlgruppe spielt ebenfalls die mittlere Zahl an Auswahlgruppen pro Typ eine Rolle, zudem die mittlere Anzahl an Zweigen in der Auswahl (2,56 für das MPEG-7 Schema), da diese nacheinander überprüft werden (vgl. auch Annahme drei).

$$N_{ZustChoice} = N_{ElementComplex}(1 + Tiefe_{VB})\left[2\frac{D_{Ch}}{D_{KT}}\frac{\left(\overline{L}_{ch} + 1\right)}{2}\right] \tag{7.3}$$

In den Häufigkeiten und in den Zweigen einer Auswahl sind Zustände angelegt, die beim Suchvorgang geprüft werden, ohne dass das Element gefunden wird. Bei Häufigkeiten wird angenommen, dass ein Zustand vergeblich geprüft wird, entsprechend ist:

$$N_{ZustGeprüftOccurrence} = N_{ElementComplex}(1 + Tiefe_{VB})\left[\frac{D_{Occ}}{D_{KT}}\right] \tag{7.4}$$

Bei Auswahlgruppen wird angenommen, dass ein Zustand in einem Zweig angelegt ist. Minimal werden Null Zweige, maximal $\left(\overline{L}_{ch} - 1\right)$ Zweige vergeblich geprüft, entsprechend ist der Erwartungswert:

$$N_{ZustGeprüftChoice} = N_{ElementComplex}(1 + Tiefe_{VB})\left[\frac{D_{Ch}}{D_{KT}}\frac{\left(\overline{L}_{ch} - 1\right)}{2}\right] \tag{7.5}$$

Schließlich muss noch der Elementzustand des zu codierenden Elements selbst berücksichtigt werden. Die Gesamtzahl der Zustände für die Suche in Elementen von komplexen Typ ist die Summe dieser Beiträge multipliziert mit der Zahl der komplexen Elemente des Dokuments, also:

$$N_{ZustComplex} = N_{ElementComplex}\left[\left(1 + Tiefe_{VB}\right)\left[2 + 3\frac{D_{occ}}{D_{KT}} + 2\frac{D_{Ch}}{D_{KT}}\frac{\left(\overline{L}_{ch} + 1\right)}{2} + \frac{D_{Ch}}{D_{KT}}\frac{\left(\overline{L}_{ch} - 1\right)}{2}\right] + 1\right] \tag{7.6}$$

Bei der Codierung eines Elements von einfachem Typ wird nur der Kopfzustand des einfachen Typen und der Elementzustand selbst aufgerufen, also

$$N_{ZustSimple} = 2 \cdot N_{ElementSimple} \tag{7.7}$$

und die Gesamtzahl der Zustände für ein Dokument ist:

$$N_{ZustGesamt} = N_{ZustSimple} + N_{ZustComplex} \tag{7.8}$$

Die mit dieser Formel berechnete Zahl der Zustände für das Beispieldokument mit 737 Elementen von komplexem und 1045 Elementen von einfachem Typ ist 20751, die experimentell gemessene 22254, was einer Abweichung von etwa 7% entspricht.

7.3.3 Decodieren einer Payload

Die Struktur der Formel zur Abschätzung der Komplexität des Decodiervorgangs ist ähnlich zur eben entwickelten.

Die Unterschiede sind:

Bei Häufigkeitszuständen muss kein Inhalt berücksichtigt werden, da in den Inhalt nur verzweigt wird, wenn ein enthaltenes Element tatsächlich instanziiert wird (die Beiträge der instanziierten Elemente sind durch den Summand „+1“ berücksichtigt). Entsprechend ist der Vorfaktor 2. Bei Auswahlzuständen muss auch kein Inhalt berücksichtigt werden. Außerdem wird nur der tatsächlich verwendete Zweig ausgewählt. Deshalb entfällt der Faktor $\left(\overline{L}_{ch} + 1\right)/2$ und es werden keine Elemente in Auswahlgruppen vergeblich geprüft, weshalb der Summand (7.5) nicht auftritt. Die Formel zum Abschätzen der Zahl der Zustände für das Suchen in Elementen von komplexen Typ ist somit:

$$N_{ZustComplex} = N_{ElementComplex}\left[\left(1 + Tiefe_{VB}\right)\left[2 + 2\frac{D_{occ}}{D_{KT}} + 2\frac{D_{Ch}}{D_{KT}}\right] + 1\right] \tag{7.9}$$

Für Elemente von einfachem Typ gilt Formel (7.7). Die Gesamtzahl der Zustände damit ist:

$$N_{ZustGesamt} = N_{ZustSimple} + N_{ZustComplex} \tag{7.10}$$

Die mit diesen Formeln berechnete Zahl an Zuständen für das Dekodieren ist für das Beispieldokument 16005, die gemessene 17626. Die Zahl wurde somit 9% zu gering geschätzt.

7.3.4 Pfadcodierung

Drei Kenngrößen bestimmen die Komplexität der Pfadcodierung für ein XML Dokument:

- die Anzahl der codierten Pfade
- die durchschnittliche Länge der Pfade,
- die durchschnittliche Länge der TBC Tabellen.

Der erste Punkt ist von der Anwendung abhängig, von der Größe des XML Dokuments und davon, wie fein es in Fragmente unterteilt wird. Der zweite Punkt ist vom Dokument abhängig. Allerdings hat auch die Struktur eines Schemas einen Einfluss darauf, wie Dokumente aufgebaut sind, ob viele oder nur wenige Hierarchiestufen möglich sind, und damit lange oder kurze Pfade auftreten. Außerdem ist wichtig, welche Elemente innerhalb des Dokuments mit Pfaden adressiert werden, und welcher Teil in einer Payload codiert wird. Der dritte Punkt ist nur vom Schema abhängig, und kann deshalb als Grundlage für eine Komplexitätsabschätzung dienen. Da der Bytecodeinterpreter das zu codierende Element bei der Pfadcodierung in derselben Datenstruktur sucht wie bei der Payloadcodierung ist die Formel ähnlich. Unterschiede, welche sich auf die Komplexität auswirken sind (vgl. 6.4.3-6.4.6):

- die Attribute eines Typs sind im Endzustand des Typen als Liste angelegt, die Attribute eines evtl. vorhandenen Basistypen werden im Endzustand des Basistypen referenziert.
- die Suche nach dem Element oder Attribut beginnt immer im Kopfzustand des Typen

Da die Suchvorgänge bei der Pfadcodierung für Encodierung und Decodierung identisch sind, braucht man nicht zwischen diesen beiden Fällen zu unterscheiden. Die Anzahl der passierten Zustände bei der Pfadcodierung wird im folgenden getrennt nach den Zustandstypen abgeschätzt. In die Komplexitätsabschätzung geht die Zahl der Elemente und Attribute ein, die in den einzelnen Stufen aller Pfade einer Dokumentcodierung auftreten.

Die Minimale Anzahl an Kopfzuständen für Element- oder Attributcodierung ist eins, die gemittelte maximale Anzahl ist ($1+Tiefe_{VB}$), entsprechend ist der Erwartungswert:

$$N_{ZustTypeHead} = N_{Elment\&Attribut} \frac{[1+(1+Tiefe_{VB})]}{2} \tag{7.11}$$

Man muss beachten, dass hier zwei Mittelungen durchgeführt werden: die erste mittelt aus dem Schema, wie viele Zustände pro Typ überhaupt vorhanden sind, die zweite wie viele von den Vorhandenen im Mittel geprüft werden.

Bei den Endzuständen der Typen muss zwischen Elementen und Attributen unterschieden werden, da eine Attributcodierung mindestens einen Typendzustand aufruft, eine Elementcodierung dagegen nicht, entsprechend gilt:

$$N_{ZustTypeEnd}[Element] = N_{Element} \frac{[0+(1+Tiefe_{VB})]}{2}$$
$$N_{ZustTypeEnd}[Attribut] = N_{Attribut} \frac{[1+(1+Tiefe_{VB})]}{2} \tag{7.12}$$

Die Häufigkeitszustände werden mit der durchschnittlichen Zahl der Häufigkeitszustände in Typdefinitionen abgeschätzt, zwischen Element- und Attributcodierung muss nicht unterschieden werden:

$$N_{ZustHäufigkeit} = N_{Element\&Atribut}(1+Tiefe_{VB})\left[2\cdot\frac{D_{OCC}}{D_{KT}}\right]\frac{1}{2} \qquad (7.13)$$

Für die Auswahlzustände ergibt sich entsprechend:

$$N_{ZustAuswahl} = N_{Element\&Attribut}(1+Tiefe_{VB})\left[2\cdot\frac{D_{Ch}}{D_{KT}}\bar{L}_{Ch}\right]\frac{1}{2} \qquad (7.14)$$

Die Zahl der durchschnittlich passierten Elementzustände ist:

$$N_{ZustElement} = N_{Element\&Attribut}(1+Tiefe_{VB})\frac{D_{El}}{D_{CT}}\cdot\frac{1}{2} \qquad (7.15)$$

Die Gesamtzahl der Zustände ist damit:

$$N_{Zust} = N_{ZustTypeHead} + N_{ZustTypeEnd} + N_{ZustHäufigkeit} + N_{ZustAuswahl} + N_{ZustElement} \qquad (7.16)$$

Die nach dieser Formel berechnete Zahl von Zuständen ist für das MPEG-7 Schema und das XML Testdokument 27941, die durch Simulation gemessene Zahl ist 31465, also etwa 12.6 % größer. Dies liegt vor allem darin begründet, dass im MPEG-7 Schema viele Datentypen vom abstrakten Datentyp Mpeg7RootType abgeleitet sind der keinen eigenen Inhalt besitzt, weshalb die Statistik für Typkopf- und Endzustände verfälscht wird: das Durchlaufen dieser Zustände für den Mpeg7RootType erhöht die gemessenen Werte. Solche Besonderheiten eines Schemas lassen sich in einer allgemeinen Abschätzung jedoch kaum berücksichtigen.

Polymorphismus

Wie in 6.4.3 beschrieben erfordert das Ändern des Typs eines Elements einen Suchvorgang im Vererbungsbaum. Dieser ist in obigen Formeln nicht berücksichtigt, da das Auftreten von Polymorphismus stark vom Dokument abhängig und selten ist. Will man dennoch eine Abschätzung des Rechenaufwandes pro Typecast machen, so könnte man den durchschnittlichen Umfang der Unterbäume des Vererbungsbaums ermitteln, welcher für die zu erwartende Länge des Suchvorgangs einen Anhaltspunkt liefert. Beim MPEG-7 Schema enthält der Unterbaum eines Typs, für den ein Typecast möglich ist im Durchschnitt 11.98 Knoten, d.h. der Suchvorgang muss im Mittel etwa sechs Knoten im Vererbungsbaum prüfen. Für 89 von 792 Typen des MPEG-7 Schemas ist Vererbung möglich, also bei etwa jedem neunten Typen.

7.3.5 Mehrfachinstanziierungen

Wenn ein Element, das mehrfach auftreten kann sehr oft instanziiert wird, liefert dies ebenfalls einen Beitrag, der vom Dokument abhängt und deshalb nicht getrennt berücksichtigt werden kann. Der Suchvorgang läuft in jenem Typ T_A ab, der das Element mehrfach instanziiert und müsste dort berücksichtigt werden. Wenn das Element sehr oft instanziiert wird würde der schemaabhängige Teil der Formel 7.6 einen zu geringen Beitrag berechnen, da das vielfache Durchlaufen des entsprechenden Häufigkeitszustandes nicht allgemein

berücksichtigt werden kann. Dazu wäre es notwendig zu wissen, wie oft genau das Element auftritt. Andererseits liefert jede der Instanzen des Elements in $N_{Element}$ einen Beitrag. Wenn das Element El_X das Kindelement einer Häufigkeit ist, so wären beim aufrufenden Typ T_A zwei zusätzliche Zustandsprüfungen nötig. Denkt man sich diesen Beitrag auf den schemaabhängigen Teil für den Typ des Elements El_X geschoben, so wäre die Abschätzung entsprechend um diese zwei Zustände zu gering. Man sieht also, dass diese zu geringe Abschätzung der Zustände bei Mehrfachinstanziierungen nicht akkumuliert, sondern lediglich eine Erhöhung in der Größenordnung von zwei Zuständen pro zusätzlicher Instanz verglichen mit der Abschätzung liefert.

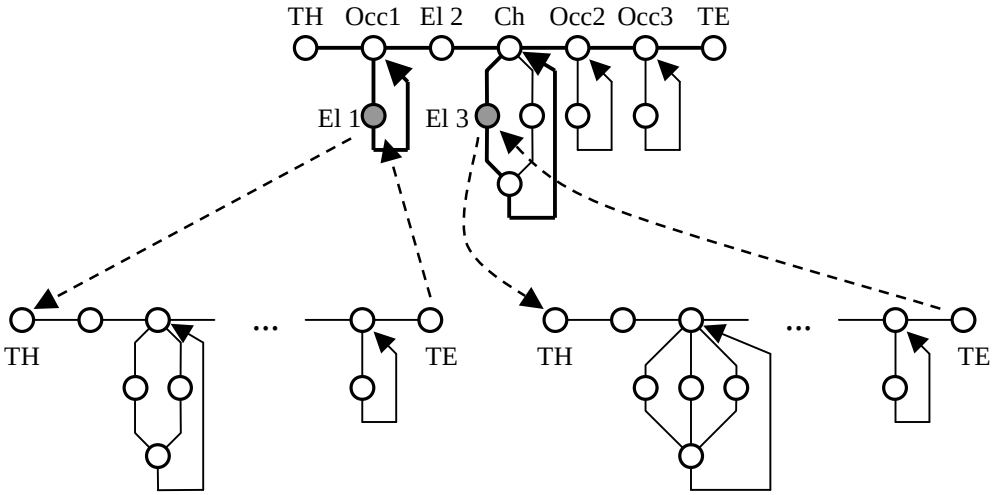


Abb. 7.3: Zählmethode bei Abarbeitung eines komplexen Typen

Abbildung 7.3 zeigt die Abarbeitung der Zustände mit hierarchischer Verzweigung in die Typen der instanziierten Elemente. In den Formeln zur Komplexitätsabschätzung sind für jeden Typ die Zustände berücksichtigt, welche in dem Typ selbst auftreten, vom Kopfzustand TH bis zum Endzustand TE. Die Zustände, die in den Typen der Kindelemente (El1,El2) abgearbeitet werden sind dadurch berücksichtigt, dass diese Elemente im dokumentabhängigen Faktor $N_{ElementComlex/Simple}$ der Formel 7.6 bzw. 7.7 einen Beitrag liefern. Wenn beispielsweise das Element El1 mehrfach auftritt, so müsste die mehrfach durchlaufene Schleife bei Occ1 im aufrufenden Typ berücksichtigt werden. Bei sehr vielen Instanzen wäre die Abschätzung zu klein. Denkt man sich diese Zustände jedoch in den aufgerufenen Typen geschoben, so ergibt sich nur eine gleichbleibende Fehlschätzung von zwei zu wenig berücksichtigten Zuständen. Der prozentuale Fehler ist deshalb gering.

7.4 Differenzierung der Komplexität nach den Bereichen des Schemas

Bei der Entwicklung der Formeln für die Abschätzung des Rechenaufwands beim Codiervorgang eines MPEG-7 konformen XML Dokuments wurde von der Annahme ausgegangen, dass die Komplexität der instanziierten Typen etwa dem Mittel aller im MPEG-7 Schema definierten Typen entspricht. Um die Verteilung der Komplexität der complexType -Definitionen für die drei Bereiche des Schemas (Video, Audio, MDS) abzuschätzen kann man diese noch einmal getrennt analysieren. Dazu wurde für alle Typen des entsprechenden Bereichs ein Suchvorgang simuliert, der vom Typkopf- bis zum Typendzustand das Inhaltsmodell des Typen komplett abarbeitet.

Die Simulation liefert eine Worst-Case Abschätzung für jeden einzelnen Typ wie lange ein Suchvorgang maximal sein kann. Es fallen einige Unterschiede zwischen den drei Bereichen auf. Zunächst stellt man fest, dass der Bereich MDS mit Abstand am umfangreichsten ist. Hier werden einschließlich der anonymen Typen 393 komplexe Typen definiert. Im Bereich Audio sind es 74 und im Bereich Video 54. Die Zahl der abgearbeiteten Zustände ist innerhalb jedes Bereichs relativ breit gestreut, allerdings sind sehr komplexe Typen auch selten. Folgendes Histogramm zeigt die Verteilung für die Video Deskriptoren.

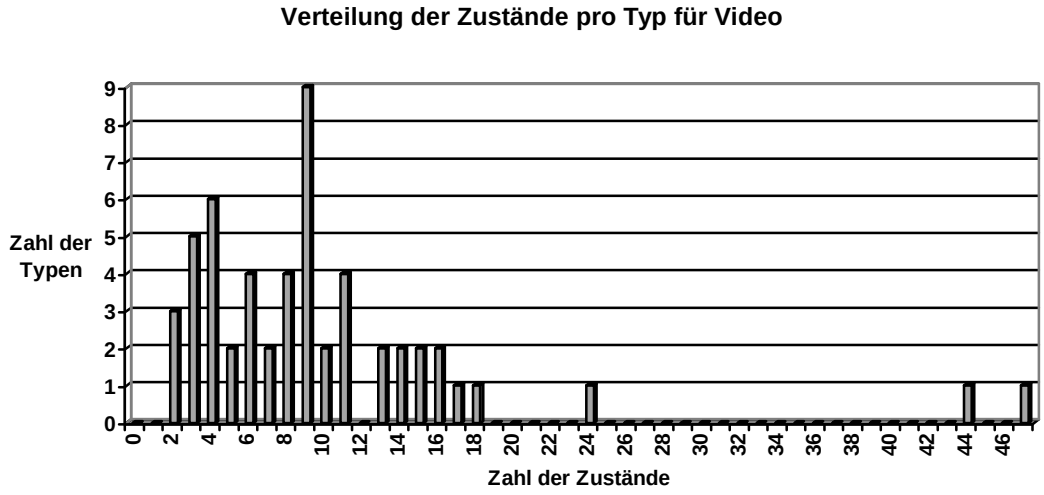


Abb. 7.4: Verteilung des Codieraufwands über die Typen bei Video DS

Der Mittelwert liegt bei 9,93 Zuständen pro Typ, der komplexeste Typ hat 47 Zustände. (Dies ist der „FractionalPresenceType“, der 15 optionale Elemente in einer Sequenz auflistet. Im Extremfall müssen also 15 Alternativen geprüft werden). Für den Bereich Audio ergibt sich folgende Verteilung:

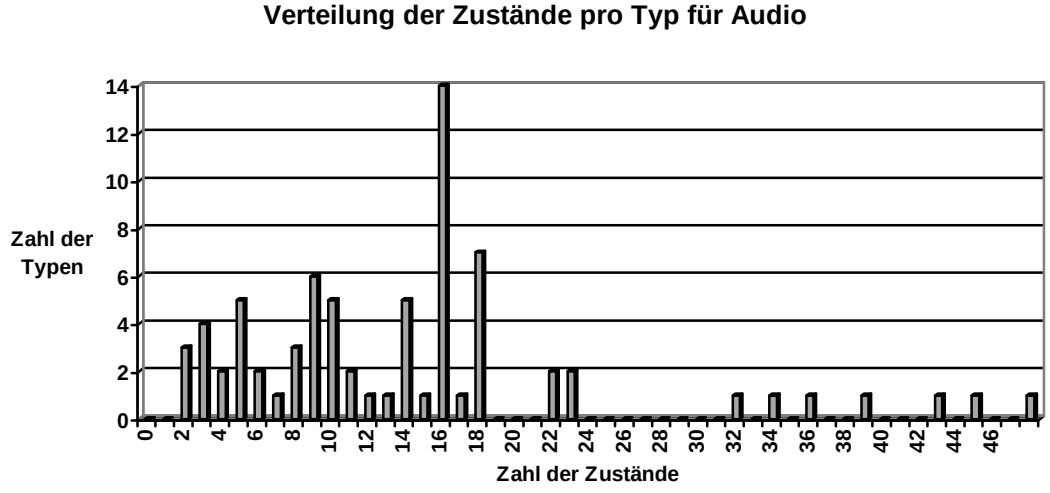


Abb. 7.5: Verteilung des Codieraufwands über die Typen bei den Audio DS

Der Mittelwert für Audio liegt bei 14.38 Zuständen pro Typ, der komplexeste Typ benötigt 48 Zustände. (SeriesOfVectorBinaryType zusammen mit dem Basistypen SeriesOfVectorType enthält ebenfalls eine lange Liste mit optionalen Elementen, die bei der Codierung überprüft werden müssen.)

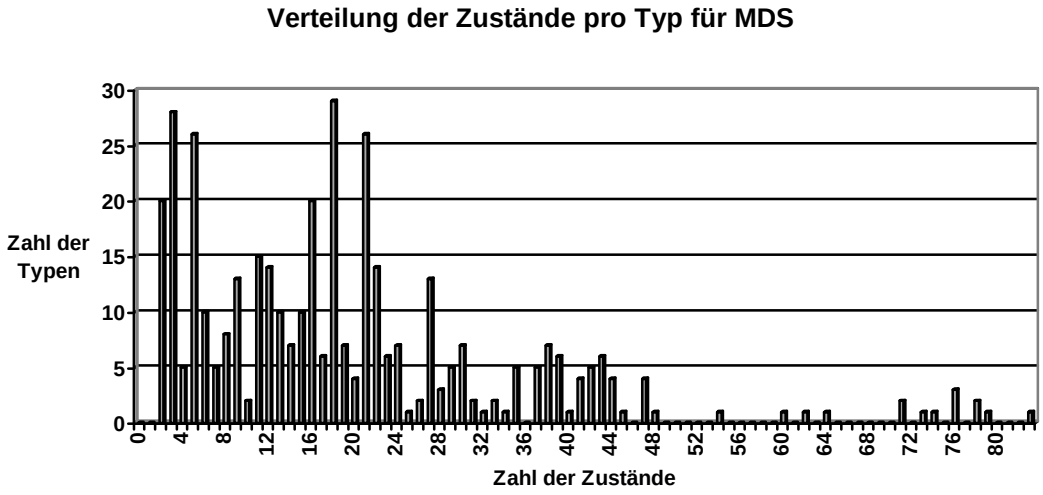


Abb. 7.6: Verteilung des Codieraufwands über die Typen bei MDS

Der Mittelwert für MDS liegt bei 19,65 Zuständen pro Typ, der komplexeste Typ hat 83 Zustände. (Dies ist der „AnalyticClipType“, der eine Kette von Basistypen der Tiefe 5 hat, und wiederum viele optionale Elemente sowie optionale Auswahlgruppen enthält).

Der Mittelwert für alle drei Bereiche liegt bei 17,86 Zuständen pro Typ. Diese Zahl liegt nahe an den experimentell ermittelten Werten. Die durchschnittliche Zahl an Zuständen für die Suche nach den Elementen bei der Codierung einer Payload lag bei 8,34 (vgl. Tab. 7.15). Dazu gezählt werden muss noch der Aufwand für das Suchen nach dem Ende eines Typen mit 4,14 Zuständen pro Typ, zusammen also 12,48. Die hier ermittelte Zahl liegt höher, weil jeder Typ vollständig durchsucht wird. Im Gegensatz dazu wird beim Codieren eines Dokuments der Suchvorgang einfacher wenn das letzte Element eines Typen codiert worden ist, denn dann werden optionale Zweige nicht mehr geprüft, sondern nur noch das Ende des Typen gesucht. Außerdem kann die spezielle Auswahl der Typen, die durch das Dokument getroffen wird zu einer Abweichung vom Mittelwert führen.

Codierung von Attributen.

Die Attribute sind im Bytecode als alphabetisch geordnete Listen von Referenzen auf Attributzustände angelegt. Beim Encodiervorgang werden diese Listen vom Beginn an durchsucht, die Zahl der durchschnittlich dabei geprüften Optionen ist proportional zur Länge der Attributliste. Die Verteilung der Länge der Attributlisten für das gesamte MPEG-7 Schema ist in folgendem Diagramm abgebildet.

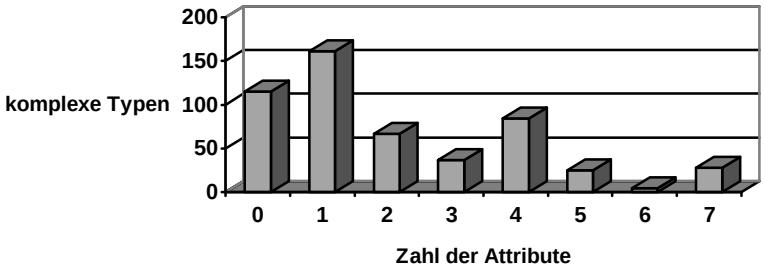


Abb. 7.7: Verteilung der Attribute

Im Mittel hat jeder Typ 2,09 Attribute, die maximale Zahl liegt bei 7 Attributen. Der Suchvorgang für Attribute bei der Payloadcodierung ist also im Vergleich zu der Suche nach Elementen kurz und wird auf einer einfachen Liste durchgeführt.

Weitere Optimierungen

Für das Codieren einer Payload gibt es noch weitere Möglichkeiten zu Optimierungen, die den Suchaufwand beim Encodieren reduzieren können.

- Statistik der codierten Elemente
  Beim Durchsuchen einer Auswahl kann man Zweige, die bisher häufig codierte Elemente enthalten zuerst durchsuchen. Diese Statistik könnte aus dem bis dahin codierten Dokument selbst gewonnen werden, oder auch aus zuvor codierten Dokumenten. Im Gegensatz zu einer statistisch optimierten VLC Codierung hätte eine solche Optimierung keinen Einfluss auf die Bitstromsyntax, und ist deshalb unproblematisch.

- Optimierung des Bytecode
  Im Bytecode können, wenn er direkt aus dem textuellen XML Schema erzeugt wird, Strukturen entstehen, die funktional ohne Bedeutung sind. Beispielsweise treten im MPEG-7 Schema Basistypen auf, die keinen Inhalt haben. Für diese wird dennoch eine Struktur im Bytecode angelegt, und die Basistypen werden im Suchprozess nach Elementen geprüft. Würden diese Typen durch einen speziellen Verarbeitungsschritt bei der Kompilation eliminiert könnte dadurch die Codierung vereinfacht werden.

Kapitel 8 – Ausblick

8.1 Codierung von Schemas

Die syntaxbasierte Codierung von Dokumenten erfordert zwingend die Kenntnis des zugrundeliegenden XML Schemas bei der Encodierung und Decodierung. Wenn das Schema eines Dokuments dem Decoder nicht bekannt ist, so kann es dennoch über eine Datenverbindung beschafft werden. XML Dokumente enthalten normalerweise einen Hinweis auf das Schema welches dem Dokument zugrunde liegt, sowie eine URI, wo dieses Schema zu finden ist. Natürlich kann ein Schema als XML Text übertragen werden. Da jedoch ein XML Schema selbst ein XML Dokument ist, kann auch ein Schema mit einer syntaxbasierten Codierung komprimiert und übertragen werden. Das Schema, welches als Grundlage für die syntaxbasierte Codierung verwendet werden kann, ist das "XML Schema for XML Schemas", das vom W3C spezifiziert wurde [W3C01]. Die Vorgänge bei der Codierung und Übertragung eines Dokuments lassen sich dann wie in Abb. 8.1 veranschaulichen.

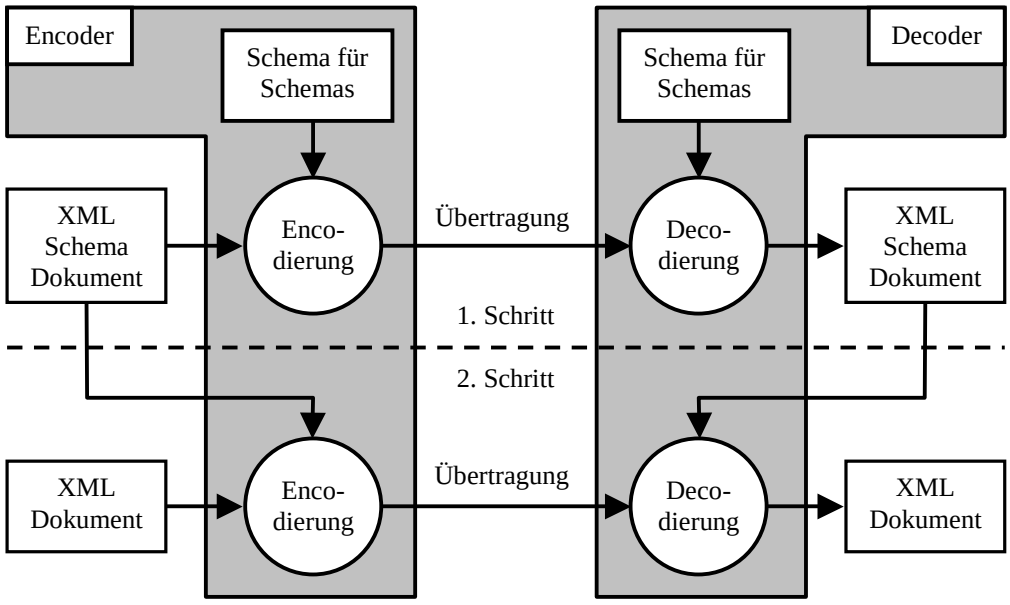


Abb. 8.1: Übertragung von Schemas

Das Schema für Schemas hat einen Umfang von etwa 85 KB. Ein Decoder, der dieses Schema zur Verfügung hat, kann dann auf BiM codierte Schemas zugreifen, und fehlende Datentypen nachladen. Um innerhalb des Namensraums eines Schemas konsistent zu sein muss zusätzlich zu der eigentlichen Syntaxdefinition eines Typen der Typecode bezüglich der Wurzel übertragen werden. Daraus kann der Typecode bezüglich jedes anderen Typen berechnet werden (vgl. 4.3.2). Dies ist erforderlich, da ein Decoder, der nicht alle Typen eines Namensraums zur Verfügung hat den richtigen Typecode sonst nicht ermitteln könnte.

Diese Typecodes könnten in einer Tabelle übertragen werden, welche die Typecodes zu den Typnamen in Beziehung setzt. Für diese Tabelle sind mehrer Formate denkbar, welche entweder die Typecodes direkt oder die Lücken zwischen den Typcodes von zwei aufeinanderfolgenden Typen enthalten. Mit einer solchen Tabelle lässt sich noch eine weitere

Optimierung durchführen: die Typnamen in der Tabelle werden im Schema oft bei der Deklaration von Elementen oder Attributen verwendet. Anstatt diese Namen direkt anzugeben bietet es sich an eine Referenz in die Tabelle zu codieren, dies spart Information bei der Codierung von Namen.

Wenn der Decoder Typen verwendet, welche in einem nachträglich übertragenen Schema definiert sind muss er die Kompilation dieses Schemas selbst übernehmen. Einfaches Laden eines vorkompilierten Schemas ist nicht ausreichend. Bisweilen treten in Schemas sehr tief verschachtelte Strukturen auf, welche vor allem durch die Definition von anonymen Typen entstehen. Um den Decoder bei der Kompilation zu entlasten könnte man schon auf der Encoderseite diese anonymen Typen aus dem Inhalt des Typen, in dem sie definiert sind, herauslösen und als eigenständigen Typen mit einer eindeutigen Kennung übertragen.

8.2 Anwendungsfelder in der Praxis

Die Idee der syntaxbasierten Codierung wurde durch unterschiedliche Entwicklungen motiviert und oft unabhängig voneinander für die verschiedensten Anwendungen vorgeschlagen (vgl. Kapitel drei). Das in dieser Arbeit vorgestellte Verfahren für die syntaxbasierte Codierung von XML wurde im speziellen durch die Entwicklung des MPEG-7 Standards beeinflusst. Es ist vorstellbar und nicht unwahrscheinlich, dass dieser Standard am Markt eine weite Verbreitung finden wird, um die Handhabbarkeit der Explosion an multimedialen Inhalten zu gewährleisten. In diesem Umfeld wird auch die BiM Codierung aufgrund der speziellen Optimierungen für die Filterung und Übertragung der codierten Information, die ein Alleinstellungsmerkmal darstellen, ihre Anwendungsfelder finden.

In der Vergangenheit waren die MPEG Standards recht erfolgreich, wenn sich auch die Verbreitung einzelner Werkzeuge und Eigenschaften nicht ohne weiteres vorhersagen lässt. In spezialisierten Einzelsystemen werden bereits Technologien zum Umgang mit multimedialen Daten eingesetzt. Ein Mobilfunkbetreiber bietet in Deutschland beispielsweise den Service an ein Musikstück zu identifizieren, indem man mit dem Mobiltelefon einige Sekunden der Musik zu einem Analysesystem überträgt, welches unter einer speziellen Rufnummer erreichbar ist. Wenn das Stück identifiziert werden konnte bekommt man eine Kurzmitteilung (SMS) mit den notwendigen Informationen zurück. Auf der Übertragungsstrecke ist zwar bei diesem Beispiel noch keine MPEG-7 Technologie im Einsatz, aber das Beispiel zeigt, dass das Marktpotential vergleichbarer Technologien von den Wettbewerbern in zunehmendem Maße erkannt wird.

Ein Nachteil von solchen Einzelsystemen ist jedoch dass Interoperabilität zwischen verschiedenen Systemen und Anbietern nicht gewährleistet ist. Wenn das Musikstück in der Datenbank des Mobilfunkbetreibers nicht gefunden wird (weil es nicht vorhanden ist, oder aufgrund von technischen Mängeln bei der Analyse oder Störungen bei der Übertragung des Musters) ist der Nutzer auf sich gestellt. Wenn standardisierte Deskriptoren verwendet würden, und die Analyse des Musters schon im Endgerät erfolgte könnte man bei verschiedenen Datenbanken anfragen.

Ein weiteres Problem ist, dass der Entwicklungsaufwand von Einzelsystemen verhältnismäßig groß ist, und nur von wenigen Firmen geschultert werden kann. Wenn dagegen von größeren Unternehmen der Informationstechnologie Module entwickelt werden, welche auf einem Standard aufbauen, können auch kleinere Markteilnehmer fortschrittliche Dienste anbieten, indem sie die für sie relevanten Module (also etwa spezielle Deskriptoren und dazugehörige Extraktions- und Analysewerkzeuge) lizenzieren. Hier sind auch viele Nischenanwendungen denkbar.

Auch von anderen Standardisierungsgremien wurde MPEG-7 Technologie berücksichtigt. Das TV Anytime Konsortium [TVA] verwendet Teile des MPEG Schemas sowie das BiM Format für die Codierung seiner Metadaten. TV Anytime hat sich zum Ziel gesetzt einen Standard für personalisierte TV Dienste zu entwickeln. Kernstück ist ein PVR (Personal Video Recorder), der mit einem hinreichend großen lokalen Massenspeicher (Festplatte ~ 80 GB) ausgerüstet ist und elementare Funktionen wie Aufnahme, Live Pause, und EPG beinhaltet, aber auch fortschrittlichere Funktionen wie Personalisierte Dienste (Benutzerprofile) und Interaktive Funktionen (Programme die einen Rückkanal erfordern) unterstützt.

Man darf also erwarten, dass in absehbarer Zukunft, wenn entsprechende MPEG-7 Software für den Markt entwickelt wurde, eine breite Palette von Anwendungen entsteht, welche von der in dieser Arbeit vorgestellten und mit entwickelten Technologie profitiert.

Zusammenfassung

Die Verbreitung strukturierter Dokumente für den Informationsaustausch und die elektronische Datenverarbeitung wird in Zukunft deutlich zunehmen. Entwicklungen wie das semantische Internet und die Verarbeitung von Multimediadaten, die mit Metadaten angereichert sind, bauen auf solchen Dokumenten auf, deren bekanntester Vertreter XML ist. In internationalen Standards finden sie Verwendung, beispielsweise dem MPEG-7 Standard, der sich zum Ziel gesetzt hat die Semantik von Multimediadaten (z.B. Audio und Videodaten) automatischen Systemen durch eine Hinterlegung mit beschreibenden Zusatzdaten (den „Metadaten“) zugänglich zu machen.

Strukturierte Dokumente binden Datenelemente in einen hierarchischen Kontext ein, der es solchen Systemen erlaubt einfacher Rückschlüsse auf die Bedeutung der Datenelemente zu ziehen. Oft folgen diese strukturierten Dokumente einer Syntaxdefinition (z.B. XML Schema), welche unabhängig von einem konkreten Dokument mit seiner Nutzinformation die Form von einer ganzen Klasse von Dokumenten festlegt. Die Vorteile werden allerdings auch mit einem Preis erkauft: der Umfang von solchen Dokumenten nimmt typischerweise auf das drei- bis fünffache der Datenelemente zu.

Dies liefert eine erste Motivation dafür eine verbesserte, kompaktere Binärdarstellung dieser strukturierten Dokumente anzustreben. Zwar gibt es bewährte und gut funktionierende Kompressionsmethoden für textuelle Information. - Jedoch entstehen durch neue Anwendungen neue Anforderungen an ein binäres Format, welche die klassischen Verfahren und auch neuere auf XML optimierte Kompressionsverfahren nicht erfüllen konnten. Zu diesen Anforderungen zählen die Möglichkeit Dokumente dynamisch zu verändern (zu aktualisieren), eine gute Unterstützung von inkrementeller Übertragung (“Streaming”) und wahlfreie Übertragungsreihenfolge der Datenelemente, Transparenz des Inhalts eines Dokuments im Binärformat (z.B. für Filteranwendungen), einfaches Parsen des Bitstroms, und möglichst geringe Komplexität der Codierung.

In dieser Arbeit wurden die bekannten Verfahren, welche sich potentiell für die Codierung von XML eignen bezüglich dieser Anforderungen analysiert. Dabei kann man unterscheiden zwischen allgemeinen Kompressionsverfahren, auf XML optimierten Kompressionsverfahren und syntaxbasierten Verfahren. Diese Verfahren wenden verschiedene Strategien für die Kompression an: im einfachsten Fall werden die Redundanzen in einem Dokument durch Referenzen auf bereits Bekanntes eliminiert. Ein anderes Verfahren verwendet Tabellen mit Kürzeln für häufig auftretenden XML Symbole. Wieder ein anderes fasst ähnliche Teile eines XML Dokuments zusammen, um die Wirksamkeit der anschießenden Redundanzreduktion zu verbessern. Allen Verfahren ist gemeinsam, dass sie in erster Linie auf eine hohe Kompression ausgelegt sind. Die Unterstützung von dynamischer Aktualisierung und die wahlfreie Übertragungsreihenfolge finden sich aber bei keinem der Verfahren. Auch die anderen Anforderungen sind zum Großteil nicht oder nur in unzureichender Form vorhanden.

Bei MPEG wurde deshalb ein neues binäres Datenformat für XML entwickelt, welches speziell auf diese Anforderungen zugeschnitten ist, genannt “Binary Format for Metadata” (BiM). In der vorliegenden Arbeit wurden wesentliche Beiträge in diesen Standart [MPEG01a] eingebracht.

Die wesentliche Idee des neuen Verfahrens ist dabei die einem strukturierten Dokument zugrundeliegende Syntaxdefinition für eine Codierung und Kompression zu nutzten: aus dieser werden die Codeworte abgeleitet, welche die einzelnen Datenelemente eines

Dokuments identifizieren. Diese Codeworte werden Tree Branch Codes (TBC) genannt. Sie wählen an jeder Verzweigung des Datenbaums eines strukturierten Dokuments den aktuell auftretenden Ast. Durch eine Verkettung der Codes lässt sich ein Pfad im Datenbaum beschreiben, der das Datenelement identifiziert, an dem eine Veränderung im Dokument vorgenommen werden soll. Möglich sind das Löschen, das Hinzufügen und das Aktualisieren (eine Kombination aus Löschen und Hinzufügen) von Information an der durch den Pfad identifizierten Stelle. Da die Pfade frei im Dokument verlaufen können, kann die Reihenfolge, in der die Datenelemente codiert werden beliebig gewählt werden. Die Pfade codieren die Struktur eines Dokuments. Kompressionsfaktoren für die Struktur von 6 bis 10 für absolute Pfade (diese beginnen bei der Wurzel des Datenbaums) und etwa 20 für relative Pfade (sie beginnen beim letzten codierten Element) sind möglich.

Für die hinzugefügte oder aktualisierte Information wird eine andere, ebenfalls aus der Syntaxdefinition abgeleitete Codierung (die Payloadcodierung) verwendet. Zusammen mit der Pfadcodierung bildet sie das BiM Datenformat. Die Payloadcodierung erreicht eine noch höhere Kompression der Struktur, lässt allerdings nur die Übertragung in der durch das textuelle Dokument vorgegebenen Reihenfolge zu.

Die Syntaxbasierte Codierung wurde schon in anderen Zusammenhängen vorgeschlagen, etwa für die Codierung von Quelltexten höherer Programmiersprachen. Dabei wurde auch die Idee der statistischen Optimierung für die Vergabe der Codeworte vorgebracht, welche im BiM Verfahren nicht zum Einsatz kommt. Das Problem statistischer Optimierungen ist, dass entweder die Codierung nicht allgemein ist, oder die Fehlerrobustheit leidet. Ersteres tritt dann auf, wenn die Codierung neben der Syntaxdefinition von einem festen Satz an Beispieldokumenten abhängig ist auf welche die Vergabe der Codeworte optimiert wurde (wenn die Statistik fest eingebaut ist), was bei einem Algorithmus, der in einem Standard zum Einsatz kommen soll unerwünscht ist. Letzteres tritt auf, wenn die Statistik dynamisch aus dem codierten Dokument gewonnen wird: das statistische Modell muss für eine funktionierende Übertragung auf der Encoderseite dasselbe sein wie auf der Decoderseite. Dies setzt voraus dass das Dokument fehlerfrei von Anfang an empfangen wurde, was bei mobilen Anwendungen oder bei inkrementeller Übertragung (Streaming) nicht immer gegeben ist.

Zudem ist das Verbesserungspotential von statistischen Optimierungen in der Praxis begrenzt, wie eine in dieser Arbeit durchgeführte informationstheoretische Analyse der syntaxbasierten Codierung zeigt. Bei dem Verfahren treten im wesentlichen zwei Fälle auf: entweder muss ein Datenelement aus einem begrenzten und durch die Syntaxdefinition vorgegebenem Satz an Möglichkeiten ausgewählt werden, oder der Wert einer potentiell unbegrenzten Zahl muss codiert werden. In diesem Fall gibt es für die Auswahl unendlich viele Möglichkeiten. Aus der Entropiedefinition von Shannon geht hervor, dass der Informationsgehalt einer solchen Entscheidung von der Wahrscheinlichkeits- bzw. Häufigkeitsverteilung der einzelnen Alternativen abhängt. Je unterschiedlicher die einzelnen Alternativen gewichten sind, umso geringer ist der Informationsgehalt, und umso besser lässt sich diese Auswahl komprimieren. Die Wirksamkeit von statistischen Optimierungen setzt also eine unsymmetrische Verteilung voraus. Für das MPEG-7 Schema ließe sich im theoretischen Extremfall einer vollkommen unsymmetrischen Verteilung (es wird immer nur eine Alternative ausgewählt) bei der Auswahl aus einem begrenzten Satz an Alternativen insgesamt eine Verbesserung der Effizienz in der Größenordnung um 10% erzielen. Auch für die Auswahl unter einer unbegrenzten Anzahl an Alternativen muss eine Wahrscheinlichkeitsverteilung zugrunde gelegt werden um eine Analyse durchführen zu können. Für die Codierung einer solchen Auswahl kann man Codierverfahren wählen, die für einen kleinen Erwartungswert der Wahrscheinlichkeitsverteilung oder für höhere

Erwartungswerte gute Ergebnisse erzielen. Dies setzt allerdings voraus, dass man den Erwartungswert der Wahrscheinlichkeitsverteilung abschätzen kann. Wenn darüber keine Aussage gemacht werden kann ist ein Verfahren günstiger, das in allen Fällen einen brauchbaren Kompromiss darstellt. Ein solches Verfahren ist in den BiM Standard aufgenommen worden um Einfachheit und Allgemeinheit zu gewährleisten.

Neben dem Codierverfahren an sich ist die effiziente Implementierung eines Algorithmus ein wichtiger Gesichtspunkt für den praktischen Einsatz eines Verfahrens. Für den BiM Algorithmus wurde bei MPEG zwar eine Referenzsoftware entwickelt, jedoch ist diese nur dafür vorgesehen syntaktisch korrekte Bitströme zu erzeugen. Sie wurde nicht auf Geschwindigkeit oder geringe Größe optimiert. In dieser Arbeit wurde eine optimierte Softwarearchitektur entwickelt, welche signifikante Verbesserungen im Vergleich zur Referenzsoftware enthält. Der neue Architekturansatz teilt den Codieralgorithmus in zwei Teile auf. Der erste Teil erzeugt in einem Kompilationsprozess aus der Syntaxdefinition ein Zwischenformat (den “Bytecode”), aus dem sich die Codeworte leicht gewinnen lassen, welche für den anderen Teil, die eigentliche Codierung oder Decodierung verwendet werden. Diese wird von einem Modul durchgeführt bei dem ein Interpreter die im Kompilationsschritt angelegte Information liest, wobei er durch das zu codierende Dokument bzw. den zu decodierenden Bitstrom gesteuert wird. Der Bytecode ist so entworfen, dass die Codierung mit maximaler Geschwindigkeit erfolgen kann: alle Informationen werden durch feste Referenzen, die bei der Kompilation angelegt werden dem Interpreter zur Verfügung gestellt. Das Ziel beim Entwurf des Bytecodemodells war es möglichst viel Komplexität in den Kompilationsprozess zu verlagern um die Codierung zu entlasten. Ein weiterer wesentlicher Vorteil ist, dass das Zwischenformat sowohl die Pfadcodierung als auch die Payloadcodierung in einem System effizient unterstützt. Die Schemainformation muss deshalb nicht unnötigerweise doppelt im Codec vorhanden sein.

Anhand dieser Softwarearchitektur lässt sich auch eine Komplexitätsanalyse des Codieralgorithmus durchführen und durch Simulationen verifizieren. Dabei bestätigt sich, dass ein erheblicher Anteil der Komplexität und damit der erforderlichen Rechenleistung schon im Kompilationsprozess steckt, der nur durchgeführt werden muss, wenn ein Dokument mit einer neuen Syntaxdefinition verarbeitet werden soll. Ansonsten genügt es das Zwischenformat zu laden. Beim relativ umfangreichen MPEG-7 Schema (etwa 240KB) ist die Kompilation etwa so aufwändig wie die Codierung eines 500-seitigen XML Dokuments (durch die Strukturierung ist allerdings erheblich weniger Information enthalten als beispielsweise in einem 500-seitigen Buch, etwa 1MB). Der erzeugte Bytecode hat für das MPEG-7 Schema einen Umfang von etwa 68KB, also 28% des Schematextes. Das Modell benötigt bei der Codierung neben der Schemainformation keine großen Pufferspeicher, nur die Konfiguration des Interpreters für jede Hierarchiestufe des XML Dokuments muss gesichert werden wozu beim Encoder etwa bis zu 100 Bytes pro Stufe erforderlich sind, beim Decoder nur 30 Bytes. Das ausführbare Programm hat für eine x86 Plattform 148KB. Darin enthalten sind Bytecodecompiler, Encoder und Decoder. Bei der Encodierung ist der rechenintensivste Teil das Suchen eines zu codierenden Datenelements in der Bytecoderepräsentation der Syntaxdefinition. Allerdings ist diese Suche insgesamt deutlich weniger aufwändig als die Suche nach Redundanzen in Dokumenten, wie sie bei anderen Kompressionsalgorithmen durchgeführt werden muss, da durch die Syntaxdefinition der Suchraum bereits sehr eingeschränkt ist. Beim BiM Verfahren ist der Aufwand für die Suche nach Elementen davon abhängig, wie häufig in der Syntaxdefinition Konstrukte auftreten, die eine Wahlmöglichkeit lassen, also etwa Auswahlgruppen oder optionale Datenelemente. Wenn man die Häufigkeit und Art dieser Konstrukte in einem Schema analysiert, kann man damit eine Abschätzung des zu erwartenden Rechenaufwands für die Codierung von

Dokumenten durchführen, die nach diesem Schema aufgebaut sind. Diese Ergebnisse können verwendet werden um schon beim Entwurf einer Syntaxdefinition darauf zu achten, dass der Codiervorgang später mit möglichst geringem Aufwand abläuft, oder um Anhaltspunkte bei der Auslegung v.a. von mobilen Geräten mit begrenzter Rechenleistung zu erhalten. Die Codierung eines 100KB großen Dokuments (mit etwa 1780 XML Datenelementen) benötigt auf einem mit 650 MHZ getakteten Standardprozessor etwa 27ms. Insgesamt wird die Implementierung des Codierverfahrens auch auf mobilen Geräten wenig Probleme bereiten, da die Anforderungen an den Speicher, sowie die Rechenleistung moderat sind. Die notwendige Schemainformation ließe sich darüber hinaus noch reduzieren, wenn nur der Bytecode der tatsächlich verwendeten Datentypen geladen wird.

Das neue Codierverfahren hat bereits (neben MPEG) das Interesse von anderen Organisationen wie dem TV Anytime Konsortium geweckt. Die damit erzielbaren hohen Kompressionsraten sowie die flexiblen Codieroptionen und die Möglichkeit kompakte und schnelle Implementierungen umzusetzen dürften für eine weite Verbreitung bei neuen Anwendungen strukturierter Dokumente sorgen.

Anhang

A1 - Zahlenformat für unbegrenzte Zahlen

Im MPEG-7 Systems Standard [MPEG01a] ist ein Zahlenformat für ganze Zahlen definiert, welches bei der Codierung der Positionscodes in Pfaden und der Codierung von Häufigkeiten verwendet wird, wenn die zugrundeliegende Syntaxdefinition eine unbegrenzte obere Schranke vorsieht. Ein festes Format (z.B. 32 Bit Integer) ist nicht geeignet, da die zu codierende Zahl auch größer sein könnte, vor allem aber weil bei vielen Zahlen, die wesentlich kleiner sind als 2^{32} unnötig Bits spendiert würden. Das Format besteht aus mehren Bitfeldern der Länge vier, in welchen die Zahl mit dem höchstwertigen Bit zuerst codiert wird. Wie viele Bitfelder verwendet werden wird vorab signalisiert, indem n Einsen codiert werden, und eine Null. Es folgen dann n+1 Bitfelder für die Zahl. Das Zahlenformat wird vluimsbf5 genannt („variable length unsigned integer most significant bit first“).

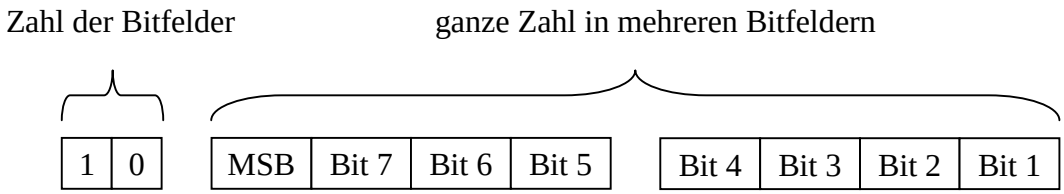


Abb. A1: Zahlenformat für unbegrenzte Zahlen

A2 - Signatur

Für die Anordnung der Elemente in choice- oder all-Gruppen muss eine unzweideutige Ordnung festgelegt werden, um sicherzustellen, dass ihnen die gleichen Codeworte zugewiesen werden wenn sie in verschiedenen aber äquivalenten Schemas definiert sind. Dafür ist ein rekursiver Algorithmus vorgesehen, der auf den transformierten Syntaxbäumen operiert, und jedem Knoten eine Signatur zuweist, bezüglich der eine alphabetische Ordnung vorgenommen werden kann: die Signatur jedes Knotens des Syntaxbaums ist aufgebaut aus einem Schlüsselwort für die Gruppe („choice“, „all“, „sequence“) wenn der Knoten des Syntaxbaums eine Gruppe repräsentiert, oder dem Namen des Elements, wenn er ein Element repräsentiert, sowie der mit Leerzeichen verketteten Liste der Signaturen aller Kindelemente des Knotens. Im Falle einer all- oder choice- Gruppe wird diese Liste vorher alphabetisch geordnet.

A3 - Rechenvorschrift für den globalen Positionscode

Um festzustellen mit wie vielen Bits ein globaler Positionscode zu codieren ist, um alle möglichen Instanzen von Elementen innerhalb eines Typs adressieren zu können muss berechnet werden, wie viele Instanzen maximal auftreten können. MPA bezeichnet die maximale Häufigkeit, die in einem Partikel instanziiert werden kann.

Für eine Sequenz gilt:

Wenn für einen Index ‘i' die maximale Häufigkeit MPA_i = 'unendlich' oder $m_{sequence}$ = 'unendlich' ist gilt:

$MPA_{sequence}$ = 'unendlich'

sonst

$$MPA_{sequence} = m_{sequence} * \sum_{1}^{nb\_of\_children} MPA_i$$

Wobei die “MPA_i” die maximale Zahl an Instanzen ist, die der i-te Partikel der Sequenz instanziieren kann. “$m_{sequence}$” hat den Wert des maxOccurs - Attributs der Sequenz.

Für eine Auswahl gilt:

Wenn für einen Index ‘i' die maximale Häufigkeit MPA_i = 'unendlich' ist oder m_{choice} = 'unendlich' ist gilt:

MPA_{choice} = 'unendlich'

sonst

$MPA_{choice} = m_{choice} * \max(MPA_i)$

Wobei die “MPA_i” die maximale Zahl an Instanzen ist, die der i-te Partikel der Auswahl instanziieren kann. “m_{choice}” hat den Wert des maxOccurs –Attributs der Auswahl.

Für ein all-Gruppe gilt:

Wenn m_{all}= 'unendlich' ist gilt:

MPA_{all} = 'unendlich'

sonst

$MPA_{all} = m_{all} * \max(MPA_i)$

Wobei die “MPA_i” die maximale Zahl an Instanzen ist, die der i-te Partikel der all-Gruppe instanziieren kann. “m_{all}” hat den Wert des maxOccurs Attributs der all-Gruppe

Für eine Elementdeklaration gilt:

$MPA_{element} = m_{element}$

“$m_{element}$” hat den Wert des maxOccurs – Attributs des Elements.

Mit diesen Regeln kann die maximal mögliche Zahl von Instanzen eines Typs berechnet werden. Ist diese Zahl kleiner als 65536, so wird der globale Positionscode mit $\lceil \log_2(MPA) \rceil$ codiert, sonst mit dem vluimsbf5 Format.

A4 - Bitfeldcodierung mit variabler Länge

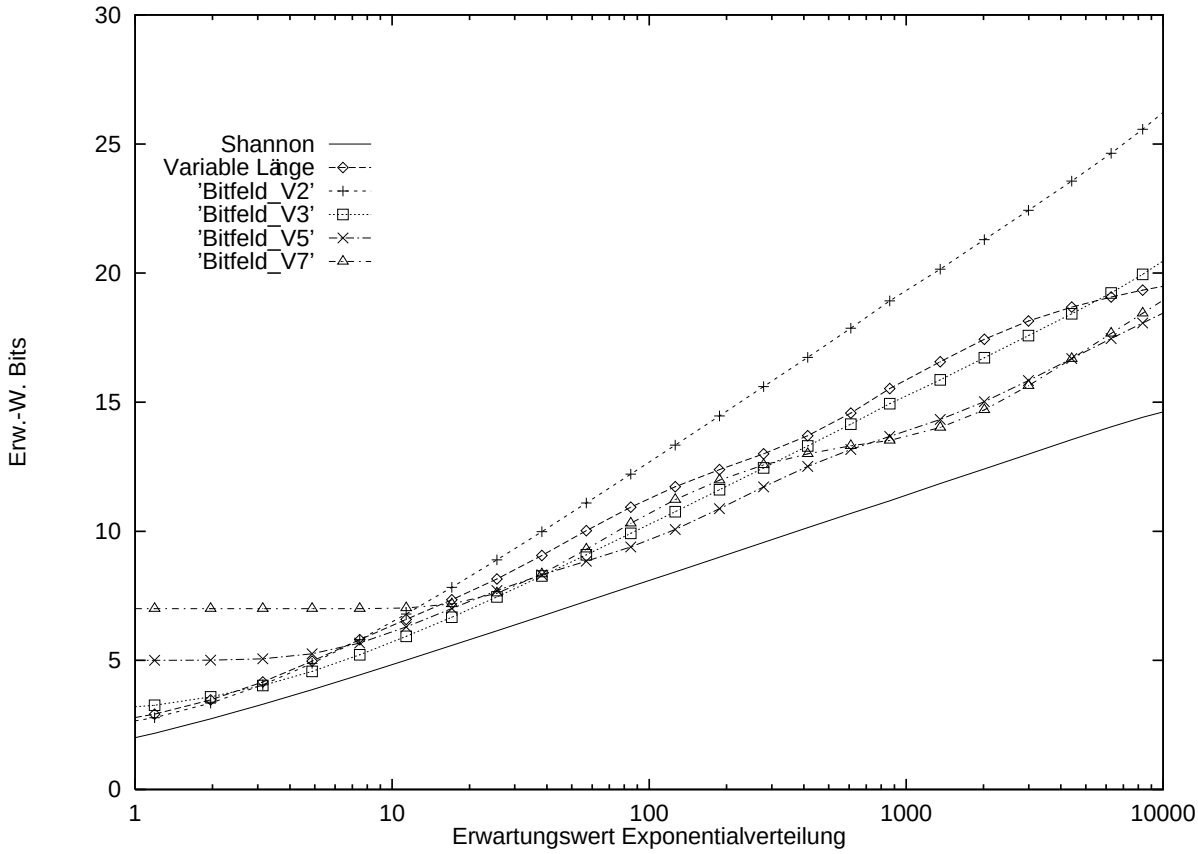


Abb. A2: Codierung von Häufigkeiten mit Bitfeldern variabler Länge

Die Abbildung zeigt das Verhalten der Codierung von unbegrenzten Zahlen mit variabler Bitfeldlänge aus Abschnitt 5.3.2. Bei Erwartungswerten unter etwa sieben ist dieses Codierschema effizienter als das Referenzverfahren mit Bitfeldlänge 5. Außerdem würde es bei sehr großen Erwartungswerten wieder effizienter (es macht sich dann der geringere Überhang durch die Flags bei den langen Bitfeldern bemerkbar).

A5 - Suchprotokoll für die Payloadcodierung

Folgendes Protokoll zeigt einen Ausschnitt aus dem Suchvorgang der Payloadcodierung.
Th: Typkopfzustand, TE: Typ Endzustand, St: SimpleType Kopfzustand, El: Elementzustand, Oc: Häufigkeitszustand, Ch: Auswahl Startzustand, Ce: Auswahl Endzustand.

```
search element: Mpeg7 Th, Ch, El, num states is: 3
search element: DescriptionMetadata Th, Th, Oc, El, num states is: 4
search element: Confidence Th, Th, Th, Te, Te, Oc, El, num states is: 7
search end: St,
search element: Version Oc, Oc, El, num states is: 3
search end: St,
search element: LastUpdate Oc, Oc, El, num states is: 3
search end: St,
search element: Comment Oc, Oc, El, num states is: 3
search element: FreeTextAnnotation Th, Oc, Ch, El, num states is: 4
search end: Th, St,
search end: Ce, Oc, Te,
search element: PublicIdentifier Oc, El, Oc, Oc, El, num states is: 5
search end: Th, St,
search element: PrivateIdentifier Oc, El, Oc, Oc, El, num states is: 5
search end: St,
search element: Creator Oc, El, Oc, Oc, El, num states is: 5
search element: Role Th, El, num states is: 2
search element: Name Th, Th, Th, Th, Te, Oc, El, Oc, Te, Oc, El, #: 11
search end: Th, Th, St,
search end: Oc, Oc, Oc, Te, Te,
search element: Agent Ch, El, num states is: 2
search element: Name Th, Th, Th, Th, Te, Oc, El, Oc, Te, Te, Oc, El, #: 12
search end: Th, Th, St,
search element: Kind Oc, El, Oc, Oc, El, Oc, Oc, El, num states is: 8
search element: Name Th, Th, Th, Th, Te, Oc, El, Oc, Te, Oc, El, #: 11
search end: Th, Th, St,
search end: Oc, Oc, Oc, Te, Te,
search element: Contact Oc, El, Oc, Oc, Ch, El, num states is: 6
search element: Name Th, Th, Th, Th, Te, Oc, El, Oc, Te, Te, Oc, Ch, El, 13
search element: GivenName Th, Oc, Ch, El, num states is: 4
```

Der Suchvorgang findet zum Teil (bis “Creator”) in dem komplexen Typen DescriptionMetadataType statt, dieser ist wie folgt definiert:

```
<!-- Definition of DescriptionMetadata Header -->
<complexType name="DescriptionMetadataType">
  <complexContent>
    <extension base="mpeg7:HeaderType">
      <sequence>
        <element name="Confidence" type="zeroToOneType" minOccurs="0"/>
        <element name="Version" type="string" minOccurs="0"/>
        <element name="LastUpdate" type=" timePointType" minOccurs="0"/>
        <element name="Comment" type=" TextAnnotationType" minOccurs="0"/>
        <element name="PublicIdentifier" type="UniqueIDType" minOccurs="0"
                                                              maxOccurs="unbounded"/>
        <element name="PrivateIdentifier" type="string" minOccurs="0" maxOccurs="unbounded"/>
        <element name="Creator" type="CreatorType" minOccurs="0" maxOccurs="unbounded"/>
        <element name="CreationLocation" type=" PlaceType" minOccurs="0"/>
        <element name="CreationTime" type="timePointType" minOccurs="0"/>
        <element name="Instrument" type=" CreationToolType" minOccurs="0"
                                                              maxOccurs="unbounded"/>
        <element name="Rights" type="RightsType" minOccurs="0"/>
        <element name="Package" type="mpeg7:PackageType" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Literaturverzeichnis

[APA] *The Apache XML Project*, http://xml.apache.org/

[ASN02] Spezifikation der International Telecommunication Union, ITU-T X.680-X.683 /ISO/IEC 8824-1 bis 4
ASN.1 Informationsseite: http://asn1.elibel.tm.fr/en

[BER02] Spezifikation der International Telecommunication Union, *ASN.1 Basic Encoding Rules (BER)*, ITU-T Rec. X.690 (2002) | ISO/IEC 8825-1:2002

[Cha01] James Cheney, *Compressing XML with Multiplexed Hierarchical PPM Models* Proceedings von Data Compression Conference, 27-29 März 2001, Snowbird, Utah, USA

[Eck98] Peter Eck, Xie Changsong, Rolf Matzner, *A New Compression Scheme for Syntactically Structured Messages (Programs) and its Application to the Internet*, Proceedings von International Conference on Data Compression, 30. März - 1.April 1998 Snowbird, Utah, USA

[Gir00] M. Girardot, N. Sundaresan, *Millau: an encoding format for efficient representation and exchange of XML over the web*, Proceedings of the 9th International World Wide Web Conference (WWW9), Amsterdam, 2000

[Heu01a] Jörg Heuer, Andreas Hutter, Ulrich Niedermeier, *Improved filtering functionality for the binary representation of MPEG-7 descriptions (BiM)*, ISO/IEC JTC1/SC29/WG11/M6795, Januar 2001, Pisa, Italien

[Heu01b] Jörg Heuer, Andreas Hutter, Ulrich Niedermeier, *Results of the mini experiments for binary MPEG-7 data representation(BiM)*, ISO/IEC JTC1/SC29/WG11/M7024, März 2001, Singapur

[Heu03] Jörg Heuer: *Verfahren zur Beschreibung von Bild- und Videomerkmalen und deren Codierung*, Dissertation, Erlangen 2003

[HTML] *HTML 4.01 Specification*, W3C Recommendation 24 December 1999, http://www.w3.org/TR/html4/

[Klein02] Shmuel T. Klein, Dana Shapira, *Searching in Compressed Dictionaries*, Proceedings, Data Compression Conference 2002, Snowbird, Utah

[Kuhn98] Peter Kuhn, *A Complexity Analysis Tool: iprof (version 0.41)* ISO/IEC JTC1 Doc. SC29/WG11 M3551, Dublin, Irland, 1998

[Lie00] H. Liefke und D. Suciu. *Xmill: an efficient compressor for XML data*. Aus Proceedings of the 2000 ACM SIGMOD international Conference on Management of Data, S. 153-164, 2000

[MHP] *Multimedia Home Platform*, www.mhp.org

[MPEG00a] *Core experiment description for binary and dynamic MPEG-7 data representation*, ISO/IEC JTC1/SC29/WG11/N3577, Peking, Juli 2000

[MPEG00b] J. Heuer, A. Hutter, U. Niedermeier, E. Wan, *Results of CE on BiM: A generic context sensitive encoding scheme for MPEG-7 descriptions and MPEG-7 description schemes*, ISO/IEC JTC1/SC29/WG11/M6613, La Baule, FR Oktober 2000

[MPEG00c] Claude Seyrat, Cedric Thienot, Gregoire Pau, *Xbin – a generic and extensible binary encoding format for MPEG-7 descriptions*, ISO/IEC JTC1/SC29/WG11/M6554, La Baule, Frankreich, Oktober 2000

[MPEG00d] Michael Wollborn, Sven Bauer,Andreas Engelsberg, Jens Rainer Ohm, *BiM - A binary format for MPEG-7 data*, ISO/IEC JTC1/SC29/WG11/M6574, La Baule, Frankreich, Oktober 2000[4]

[MPEG00e] Benoit Mory, Nicolas Santini, Franck Laffargue, *Binary format for MPEG-7 descriptions (BiM) : a new approach*, ISO/IEC JTC1/SC29/WG11/M6585, La Baule, Frankreich, Oktober 2000

[MPEG00f] Ernest Wan, Jörg Heuer, Andreas Hutter, Ulrich Niedermeier: *A Context-free Method for Generating Bitstreams for MPEG-7 Descriptions*, ISO/IEC JTC1/SC29/WG11/M6537, La Baule, Frankreich, Oktober 2000

[MPEG01a] International Organization for Standardization, *MPEG-7 – Multimedia Content Description Interface – Part 1: Systems*, ISO/IEC 15938-1

[MPEG01b] International Organization for Standardization, *MPEG-7 – Multimedia Content Description Interface – Part 2: Description Definition Language*, ISO/IEC 15938-2

[MPEG01c] International Organization for Standardization, *MPEG-7 – Multimedia Content Description Interface – Part 3: Visual*, ISO/IEC 15938-3

[MPEG01d] International Organization for Standardization, *MPEG-7 – Multimedia Content Description Interface – Part 4: Audio*, ISO/IEC 15938-4

[MPEG02] ISO/IEC JTC1/SC29/WG11 N4980, *MPEG-7 Overview*, Klagenfurt, Juli 2002 http://mpeg.telecomitalialab.com/standards/mpeg-7/mpeg-7.htm

[MPGF] *Pro-MPEG Forum*, www.pro-mpeg.org

[Nap02] Milind Naphade, Ching-Yung Lin, John Smith, *Learning Semantic Multimedia Representations from a Small Set of Examples*, Proceedings, 2002 IEEE International Conference on Multimedia and Expo, Lausanne, Schweiz

[4] Anmerkung: obwohl der Name des Vorschlags für ein Binärformat dem im Standard entspricht handelt es sich nicht um das in dieser Arbeit vorgestellte und in den Standard aufgenommene Codierverfahren. Michael Wollborn war der Leiter der Arbeitsgruppe, der bei MPEG die Arbeiten am Binärformat koordiniert hat.

[Nat02] Paul Natsev, John Smith, *A Study of Image Retrieval by Anchoring*, Proceedings, 2002 IEEE International Conference on Multimedia and Expo, Lausanne, Schweiz

[Nie02a] U. Niedermeier, J. Heuer, A. Hutter, W. Stechele, *MPEG-7 Binary Format for XML Data*, Proceedings, Data Compression Conference, 2-4 April 2002 Snowbird, Utah, USA

[Nie02b] U. Niedermeier, J. Heuer, A. Hutter, W. Stechele, A. Kaup, *An MPEG-7 Tool for Compression and Streaming of XML Data*, Proceedings IEEE International Conference on Multimedia and Expo, 26-29 August 2002, Lausanne, Schweiz

[PER02] Spezifikation der International Telecommunication Union, *ASN.1 Packet Encoding Rules (PER)*, ITU-T Rec. X.691 (2002) | ISO/IEC 8825-2:2002

[SCI01] Scientific American, Artikel *The semantic Web*, Ausgabe Mai 2001, auch online unter: http://www.sciam.com/

[SGML86] International Organization for Standardization, ISO 8879, Information processing -Text and office systems, *Standard Generalized Markup Language (SGML)*, Genf 1986

[Smi02] John Smith, *Spatial and Feature Normalization for Content-Based Retrieval*, Proceedings, 2002 IEEE International Conference on Multimedia and Expo, Lausanne, Schweiz

[TVA] TV Anytime Forum, http://www.tv-anytime.org/

[W3C] World Wide Web Consortium, http://www.w3.org/

[W3C01] World Wiede Web Consortium, *Canonical XML, Version 1.0*, W3C Recommendation 15 March 2001 http://www.w3.org/TR/xml-c14n

[W3C01] World Wide Web Consortium, *XML Schema Part 1: Structures, Appendix A* W3C Recommendation 2. Mai 2001, *http://www.w3.org/TR/xmlschema-1/*

[W3C99] World Wide Web Consortium, *WAP Binary XML Content Format (WBXML)*, W3C NOTE 24 Juni 1999 http://www.w3.org/TR/wbxml/

[Wel84] T. A. Welch, *A Technique for High Performance Data Compression*, IEEE Computer, Vol. 17 No. 6, 1984

[XML00] World Wide Web Consortium
Extensible Markup Language (XML) 1.0 (Second Edition)
W3C Recommendation 6 October 2000, http://www.w3.org/TR/REC-xml

[XML01a] World Wide Web Consortium, *XML Schema Part 0: Primer*,
W3C Recommendation, Mai 2001,
http://www.w3.org/TR/2001/REC-xmlschema-0-20010502

[XML01b] World Wide Web Consortium, *XML Schema Part 1: Structures*,
W3C Recommendation, Mai 2001,
http://www.w3.org/TR/2001/REC-xmlschema-1-20010502

[XML01c] World Wide Web Consortium, *XML Schema Part 2: Datatypes*,
W3C Recommendation, Mai 2001,
http://www.w3.org/TR/2001/REC-xmlschema-2-20010502

[XML02] World Wide Web Consortium, *Extensible Markup Language (XML) 1.1*
W3C Candidate Recommendation 15 October 2002,
http://www.w3.org/TR/xml11/

[XMLX00] Intelligent Compression Technologies, *XML-Xpress, "White Paper"*,
http://www.ictcompress.com/products_xmlxpress.html

[XMLZ] XMLZIP – von XML Solutions, http://xmls.com , inzwischen von Vitria aufgekauft

[Zie03] Peter-Michael Ziegler, *Adlerauge - Europas größte Gesichtserkennungsanlage im Zoo Hannover*, Magazin für Computertechnik, c't 9/2003, S.26: Biometire

[ZIV77] J. Ziv und A. Lempel: *A universal algorithm for sequential data compression.*
IEEE transactions on Information Theory, 23(3): S. 337-343, 1977