UNIVERSITÄT
KOBLENZ · LANDAU

Institut für
Softwaretechnik

Studienarbeit

BinaryGXL

vorgelegt von:
Tassilo Horn

Betreuer: Dr. Volker Riediger

Erklärung

Hiermit bestätige ich, Tassilo Horn, dass ich die vorliegende Studienarbeit selbstständig erarbeitet habe. Als Quellen wurden ausschließlich die im Literaturverzeichnis und den Fußnoten angegebenen Dokumente und Internetseiten verwendet.

Ort, Datum	Unterschrift

Kurzfassung

XML ist aus der heutigen IT-Welt nicht mehr wegzudenken und wird für vielfältige Zwecke verwendet. Die Befürworter von XML führen insbesondere die Menschenlesbarkeit und leichte Editierbarkeit als Gründe für diesen Erfolg an. Allerdings hat XML gerade dadurch auch in manchen Anwendungsgebieten gravierende Nachteile: XML ist sehr verbos, und das Parsen von XML-Dateien ist relativ langsam im Vergleich zu binären Dateiformaten. Deshalb gibt es Bestrebungen ein binäres XML-Format zu schaffen und gegebenenfalls zu standardisieren.

Die vorliegende Studienarbeit beschreibt einige schon existierende binäre XML-Formate und zeigt auf, wie weit der Entwurf und die Standardisierung eines allgemein akzeptierten Binary-XML-Formats fortgeschritten sind.

Der praktische Teil der Studienarbeit umfasst den Entwurf und die Implementierung eines eigenen Binärformats für den XML-Dialekt GXL. Zur Bewertung der entwickelten Lösung wird diese mit den heute üblichen Kompressionstools verglichen.

Abstract

To imagine IT-industry without XML is impossible. It's widely used for diverse purposes. XML's supporters explain its success with its human-readability and the ease of modification. But these two properties have a drawback, too. XML is very verbose and parsing XML documents is slow compared to binary file formats. Because of this there are efforts going on which try to define and standardize a binary XML format.

This student research project examines some existing binary XML formats and points out how far conception and standardization of a generally accepted binary XML format has proceeded.

The project's practical part includes design and implementation of an binary format for the XML dialect GXL. To see how well this developed solution works it is compared to usual compression tools.

Danksagung

Besonderer Dank gebührt meinem Betreuer Volker Riediger, der stets ein offenes Ohr für meine Fragen hatte und mir mit gutem Rat zur Seite stand.

Ebenfalls darf Lev Walkin, der Entwickler des ASN.1-Compilers *asn1c*, nicht unerwähnt bleiben, der mir bei meinen Fragen zu ASN.1 im Allgemeinen und *asn1c* im Speziellen mit fachkundigen Antworten und kurzen Codebeispielen half.

Fürs Korrekturlesen darf ich mich bei meiner Freundin Nicole Reinbach bedanken, die sich als Fachfremde (BWLer!) vermutlich mit allerlei Aufputschmitteln wach halten musste.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Problembeschreibung

Die GXL (*Graph Exchange Language*, [1]) ist ein vom Institut für Softwaretechnik[1] (IST) et. al. der Universität Koblenz entwickeltes XML-basiertes (eXtensible Markup Language, [XML]) Austauschformat für Graphen. Die Syntax wird mit einer DTD (*Document Type Definition*, [2]) spezifiziert.

Das Ziel der Entwickler von GXL war es, im Bereich des Software-Reengineering ein gemeinsames Austauschformat für die verschiedenen Tools zu etablieren und somit deren Interoperabilität zu steigern.

Der große Vorteil von GXL ist, dass man jedes konzeptuelle Datenmodell (*Schema*) auf einen typisierten, attributierten und gerichteten Graphen abbilden kann. Aber auch Instanzmodelle lassen sich so beschreiben. Dies gestattet es, sowohl Schemata als auch ihre Instanzen in GXL zu beschreiben, bzw. die in anderen Tools benutzen Repräsentationen, z.B. Relationen, in GXL zu überführen.

Das Software-Reengineering-Tool GUPRO des IST Koblenz[1] benutzt GXL als Austauschformat. Zur internen Faktenrepräsentation benutzt es ebenfalls gerichtete, attributierte, typisierte Graphen, welche dann als GXL exportiert werden können. Je nach Granularität des zur Faktenextraktion benutzten Schemas und der Größe des zu untersuchenden Quelltextes werden diese Graphen aber sehr groß, teilweise in einer Größenordnung mehrerer Millionen Knoten und Kanten.

Exportiert man einen solchen Graph nun nach GXL, so entstehen aufgrund der Verbosität von XML Dateien in Gigabytegröße. Da GXL ein Austauschformat sein soll, ist neben der Adaptierbarkeit und Flexibilität eben auch die Dateigröße ein wichtiger Faktor.

1.2 Aufgabenbeschreibung

Genau hier soll diese Studienarbeit Verbesserung bringen. Ziel soll es sein, die Größe der auszutauschenden Dateien drastisch zu verkleinern ohne jedoch den Informationsgehalt zu verringern.

[1] `http://www.uni-koblenz.de/~ist`, 05.07.2006

Dazu müssen die folgenden Aufgabenstellungen bearbeitet werden.

1. Untersuchung alternativer Binär-XML-Formate
2. Binäre Kodierung und Dekodierung von GXL in BinaryGXL
3. Vergleich von BinaryGXL und gepacktem GXL

1.2.1 Untersuchung alternativer Binär-XML-Formate

Zuerst soll die Arbeit der XML Binary Characterization Working Group [XBC] untersucht werden, welche versucht ein für möglichst alle Anwendungsfälle adäquates Binary-XML-Format zu finden.

Danach sollen alternative Binary-XML-Formate untersucht werden. Dazu muss zuerst recherchiert werden, welche Formate es in diesem Bereich zur Zeit gibt, welche davon evtl. sogar schon standardisiert sind und welche sich in der Planung befinden. Außerdem soll die Frage nach der Toolunterstützung des jeweiligen Formats geklärt werden.

1.2.2 Binäre Kodierung von GXL

Es soll ein binäres Dateiformat für GXL (*BinaryGXL*) entwickelt werden, welches den Größennachteil von XML wett macht, ohne dabei Informationen zu verlieren.

Die BinaryGXL-Datentypen sind zuerst in ASN.1 (*Abstract Syntax Notation One*, [11]) zu spezifizieren. ASN.1 wurde gewählt, da es sich insbesondere in der Telekommunikationsindustrie als vielversprechende Möglichkeit für die Implementierung binärer Dateiformate herausgestellt hat. Als Beispiele sind hier GSM (*Global System for Mobile Communications*) und UMTS (*Universal Mobile Telecommunications System*) zu nennen. Andere Anwendungsfelder sind digitale Zertifikate (X.509), das SNMP-Protokoll oder Verzeichnisdienste wie X.500 und LDAP.

Dazu soll ein Kommandozeilen-Konvertierungstool geschrieben werden, welches GXL-Dateien in BinaryGXL und zurück überführt. Im Idealfall soll die hin und zurück kodierte GXL-Datei identisch mit der ursprünglichen GXL-Datei sein.

Zusätzlich soll die Ausgabe auch auf `stdout` statt in eine Datei erfolgen können, damit man sie direkt über eine Pipe oder Umleitung in anderen Tools verwenden kann.

1.2.3 Vergleich von BinaryGXL und gepacktem GXL

Um die Güte der entwickelten Lösung abzuschätzen, soll sie mit gängigen Komprimierungsprogrammen verglichen werden, und zwar hinsichtlich der Dateigröße, des Speicherverbrauchs und der Geschwindigkeit.

Mögliche Komprimierungstools, die hier betrachtet werden können, wären beispielsweise:

- compress[2]
- gzip[3]
- bzip2[4]
- 7-Zip[5]
- rar[6]

Im nun folgenden zweiten Kapitel wird zunächst die Arbeit der XML Binary Characterization Working Group beleuchtet und dann zwei alternative, binäre XML-Formate vorgestellt und beschrieben.

[2] `ftp://ftp.ibiblio.org/pub/linux/utils/compress/ncompress-4.2.4.tar.Z`, 05.07.2006
[3] `http://www.gnu.org/software/gzip/gzip.html`, 05.07.2006
[4] `http://www.bzip.org/`, 05.07.2006
[5] `http://www.7-zip.org/`, 05.07.2006
[6] `http://www.rarsoft.com/`, 05.07.2006

Kapitel 2

Binary-XML – Standards und Formate

In diesem Kapitel soll ein Überblick über Standards und Formate im Rahmen von Binary-XML gegeben werden. Besondere Beachtung finden hier die Arbeiten der *XML Binary Characterization Working Group* ([XBC]), welche genaue Anforderungen an ein einheitliches Binär-XML-Format aufgestellt und viele mögliche Anwendungsfälle aufgezeigt hat.

Darüber hinaus werden in Kapitel 2.3 auch einige alternative, schon existierende XML-Kodierungen vorgestellt und erörtert.

2.1 Einführung

Die *Standard Generalized Markup Language* ([SGML]) ist eine so genannte *Metasprache*, d.h. mit ihr ist es möglich andere Auszeichnungssprachen (*Markup Languages*) zu definieren, die unabhängig von Plattform und Anwendung sind. Dazu wird in einer *Document Type Definition* (DTD) der strukturelle Aufbau von gültigen Dokumenten beschrieben. Ziel ist es die in einem Dokument enthaltenen Daten logisch zu strukturieren und das unabhängig von ihrer Darstellung.

Anwendungen von SGML sind beispielsweise HTML oder DocBook.

Aufgrund der enormen Komplexität der SGML konnte sie sich nie wirklich durchsetzen. Stattdessen etablierte sich eine stark vereinfachte Teilmenge der SGML – die vom World Wide Web Consortium standardisierte *eXtensible Markup Language* ([XML]).

Als große Vorteile von XML werden gerne

- Menschenlesbarkeit,
- Einfachheit,
- Flexibilität und Erweiterbarkeit,
- Lizenzfreiheit und
- Plattformunabhänigkeit

angeführt. Menschenlesbarkeit bedeutet dabei, dass ein XML-Dokument mit einem beliebigen Editor betrachtet werden kann, da XML ein Textformat ist. Man benötigt keine Spezialapplikationen dazu. Oft wird XML auch als selbsterklärend bezeichnet, was daher rührt, dass man in XML-Dokumenten eigene Tags, die sinnvoll benannt werden sollten, zur Strukturierung verwendet. Dass es sich bei `<name>Hans Müller</name>` um einen Namen handelt, ist somit leicht zu erkennen. Allerdings liegt dies weniger an XML an sich, sondern an der Wahl sinnvoller Tag-Namen und damit am Designer eines XML-Formats.

XML ist eine Untermenge der SGML. Die SGML-Sprachdefinition umfasst ungefähr 500 Seiten, die von XML nur 26. Trotzdem lässt sich der Großteil aller Anwendungsfälle von SGML auch mit XML realisieren. Diese Einfachheit ermöglicht es vielen Firmen und Projekten Parser und Tools bereitzustellen, ohne die kein Format weite Verbreitung finden kann.

Außerdem lassen sich XML-Formate problemlos erweitern, z.B. indem man ein Format mit einem anderen kombiniert. Damit bei der Kombination verschiedener Formate keine Namenskollisionen entstehen, gibt es in XML ein Namensraum-Konzept.

XML spezifiziert nur, was Tags und Attribute sind und welche Eigenschaften ein Dokument besitzen muss, um wohlgeformt und gültig zu sein. Mittlerweile ist XML aber zu einer ganzen Familie an Techniken für verschiedenste Anwendungsbereiche angewachsen, von denen die Wichtigsten hier kurz erwähnt werden sollen:

- XPointer [26], Verweise auf Teile von XML-Dokumenten
- CSS [27] und XSL [28] zur Erstellung von Style-Sheets
- XSLT [29], zur Transformation von XML nach den Regeln eines Style-Sheets
- XML Schema [30], zur präzisen Definition eigener XML-Formate
- DOM [31] und SAX [32], zur Bearbeitung von XML-Dokumenten mit Programmiersprachen
- XQuery [33], zur Abfrage von XML-Datenbanken

Heute gibt es fast keine Bereiche in der IT mehr, in denen XML noch nicht Einzug gehalten hat. Einige Anwendungen sind:

- XHTML [18], Webseitengestaltung
- WML [19], Gestaltung von WAP-Seiten
- XMPP [21], Instant Messaging
- GXL [1], Graphbeschreibungssprache
- Konfigurationsdateien
- OpenDocument [22]
- SVG [23], Beschreibungssprache für Vektorgrafiken
- MPEG-7 [24], Ergänzung von Multimediadaten mit Metainformationen
- SOAP [25], Remote Procedure Calls und Datenaustausch zwischen vernetzten Systemen
- RDF [34], Framework zur Darstellung von Informationen von WWW-Ressourcen

Man sieht also, dass XML gerade im Bereich des Datenaustauschs zwischen verschiedenen Systemen eine große Rolle spielt und auch in Zukunft spielen wird.

Zur Bearbeitung von XML-Dokumenten gibt es zwei verschiedene standardisierte APIs,

- DOM ([31])
- SAX ([32])

Implementierungen davon finden sich für fast jede Programmiersprache und jede Plattform.

Da in vielen Anwendungsgebieten die Verbosität von XML ein großer Nachteil ist, aber auf die anderen Vorteile nicht verzichtet werden soll, wurden schon früh binäre XML-Kodierungen entwickelt, die abgestimmt auf die Anforderungen einer bestimmten Domäne sind.

2.2 Die XML Binary Characterization Working Group (XBC)

Die *XML Binary Characterization Working Group* ([XBC]) wurde als Resultat des *W3C Workshop on Binary Interchange of XML Information Item Sets*, welcher vom 24. bis 26. September 2003 in Santa Clara, Kalifornien, USA auf dem Gelände von Sun Microsystems stattfand, gegründet.

Für die XML-Lobby ist die Menschenlesbarkeit von XML-Dokumenten ein großer Vorteil. Jedes Element ist durch ein Paar von sinnvoll benannten Tags eingeschlossen, und meist wird durch Einrückungen die Baumstruktur eines XML-Dokuments noch hervorgehoben. Allerdings sind XML-Dokumente dadurch auch sehr verbos, was sich negativ auf die Dateigröße, den Verarbeitungsaufwand und die Lese- und Schreibgeschwindigkeit auswirkt.

In vielen Anwendungsgebieten sind die Prioritäten jedoch so gesetzt, dass die Dateigröße minimiert und die Verarbeitungsgeschwindigkeit möglichst hoch sein soll oder eine besonders effiziente Suche innerhalb des Dokuments ermöglicht werden soll. Im Laufe der Zeit entstanden so für viele Bereiche schon zueinander inkompatible Nischenlösungen.

Um solchem Wildwuchs vorzubeugen und aufgrund der großen Nachfrage der Industrie nach einem allgemeingültigen, binären XML-Format wurde die XBC gegründet. Ihre Aufgaben sind

- das Finden von Anwendungsfällen, in denen der Aufwand des Verarbeitens von XML zu hoch ist,
- das genaue Beschreiben der Eigenschaften von XML, sowie der Eigenschaften, die in jedem Anwendungsfall benötigt werden und
- das Erstellen von objektiven Messungen, die eine Entscheidung darüber liefern, ob XML 1.x oder eine alternative (binäre) Kodierung die nötigen Eigenschaften, die die Anwendungsfälle voraussetzen, bieten.

2.2.1 Definition der von der XBC benutzten Eigenschaften

In diesem Abschnitt wird eine genaue Definition der von der XBC benutzen Eigenschaften gegeben. Diese wurden durch genaue Betrachtung verschiedenster Anwendungsfälle [7] extrahiert. Für genaue, aktualisierte Definitionen und Erklärungen ist das Originaldokument der XBC [8] einzusehen.

Algorithmische Eigenschaften

Geringe Prozessorgröße *(Small Footprint)* Die Größe eines Prozessors für das neue Format soll im Vergleich zu einem XML-Prozessor gering sein.

Speichereffizienz *(Space Efficiency)* Ein Prozessor für das neue Format soll im Vergleich zu einem XML-Prozessor nur wenig Speicheranforderungen haben.

Verarbeitungseffizienz *(Processing Efficiency)* Die Verarbeitungseffizienz gibt an, wie schnell ein Format im Vergleich zu XML verarbeitet, d.h. generiert oder gelesen, werden kann.

Format-Eigenschaften

Abgeschlossenheit *(Self Contained)* Ein Format ist abgeschlossen, wenn die einzigen Informationen, die zur Reproduktion einer Datenmodellinstanz notwendig sind, (i) die Darstellung der Instanz und (ii) die Spezifikation des XML-Formats sind.

Allgemeingültigkeit *(Generality)* Ein Format heißt allgemeingültig, wenn es über einen weiten Bereich von XML-Dokumenten, Anwendungen und Anwendungsfällen mit Alternativen konkurrenzfähig ist.

Beschleunigter sequentieller Zugriff *(Accelerated Sequential Access)* Beschleunigter sequentieller Zugriff ist die Eigenschaft, die es erlaubt eine bestimmtes Element in einer XML-Datei schneller zu finden, als es mit einem Vergleich Byte für Byte möglich wäre.

Content-Type-Management *(Content Type Management)* Das Format integriert sich in die bestehende Medientyp- und Kodierungsinfrastruktur. Es definiert also einen oder mehrere Medientypen und/oder Kodierungen und bestimmt wann und wie diese genutzt werden sollen.

Deltas *(Deltas)* Ein Delta ist eine Repräsentation von beliebigen Änderungen von einer Instanz eines Basisdokuments, welches zusammen mit dem Basisdokument den neuen Zustand des Basisdokuments beschreibt. Das jeweilige Basisdokument wird global oder lokal eindeutig identifiziert.

Direkte Schreib-/Lesbarkeit *(Directly Readable and Writable)* Ein Format heißt direkt schreib-/lesbar, wenn eine gegebene Instanz eines Datenmodells direkt serialisiert und eine Datei des Formats direkt in eine Instanz des Datenmodells geparst werden kann, ohne sie vorher zuerst in eine Zwischenform zu transformieren.

Effizientes Update *(Efficient Update)* Effizientes Update bedeutet, dass Änderungen an einer Instanz eines Datenmodells effizient gespeichert werden können. Diese Eigenschaft ist besonders wichtig bei Applikationen, welche Änderungen an

speziellen Elementen von Datenmodellinstanzen auf eine Art vornehmen müssen, die schneller als ein kompletter Deserialisierungs-Modifizierungs-Serialisierungs-Zyklus ist.

Erweiterungspunkte *(Extension Points)* Ein Erweiterungspunkt ist eine Methode zur einfachen Erweiterung eines Formats und seiner Implementierung.

Explizite Typisierung *(Explicit Typing)* Ein Format heißt explizit typisiert, wenn Datentypinformationen der Elemente eines Datenmodells untrennbarer Bestandteil des Formats sind.

Formatversionsidentifizierung *(Format Version Identification)* Die Version des Formats kann effizient festgestellt werden.

Fragmentierbarkeit *(Fragmentable)* Ein Format heißt fragmentierbar, wenn es möglich ist Instanzen, die nicht die Gesamtheit eines Dokuments repräsentieren, zusammen mit dem zur Dekodierung benötigten Kontext zu kodieren.

Inkrementelle Verarbeitbarkeit *(Streamable)* Ein Prozessor beherrscht inkrementelle Verarbeitung, wenn er aus korrektem aber unvollständigem Input, korrekten, partiellen Output erzeugen kann. Ein Format heißt dann inkrementell verarbeitbar, wenn man dafür einen solchen Prozessor implementieren kann.

Integrierbarkeit in den XML-Stack *(Integratable into XML-Stack)* XML ist umgeben von einer ganzen Familie von Technologien, die zusätzliche Fähigkeiten und Eigenschaften wie Validierung, Transformation, Querying oder Programmierung über standardisierte APIs ermöglichen. Alle diese Technologien zusammen bilden den sogenannten XML-Stack. Ein binäres Format lässt sich gut in diesen integrieren, wenn nur wenig Spezifikationsaufwand nötig ist, um bereits darin existierende Technologien auf das neue Format anzupassen.

Keine Beschränkungen *(No Arbitrary Limits)* Diese Eigenschaft bezieht sich auf den Grad, in dem ein Format gegebenen Größen Beschränkungen auferlegt. Größen können z.B. Längen, Anzahl verschiedener Zeichenkodierungen, etc. sein. Die meisten Formate führen solche Limitierungen ein, indem sie nur eine feste Anzahl von Bits für die Speicherung verschiedener Größen vorsehen.

Kompaktheit *(Compactness)* Die Kompaktheit eines Formats bezieht sich auf die Größe der Repräsentation eines XML-Formats, sowohl im Speicher als auch anderweitig gespeichert. Kompaktheit wird dadurch erzielt, dass ein Format möglichst wenige zusätzliche Informationen, welche nicht zur korrekten und vollständigen Verarbeitbarkeit notwendig sind, enthält.

Lizenzkostenfreiheit *(Royality Free)* Genau wie XML soll das alternative binäre Format lizenzkostenfrei sein.

Lokal begrenzte Änderungen *(Localized Changes)* Ein Format weist nur lokal begrenze Änderungen auf, wenn eine Änderung an einer einzigen Stelle in einer Instanz eines XML-Datenmodells in der entsprechenden binären Repräsentation ebenfalls nur eine genau lokalisierbare, begrenzte Anzahl von Bytes verändert.

Genauso hat ein Format nur lokal begrenzte Änderungen, wenn kleine Änderungen des Datenmodells nur kleine Änderungen im binären Format bedingen.

Die Delta-Eigenschaft ist verwandt mit dieser Eigenschaft.

Neutralität bzgl. natürlicher Sprache *(Human Language Neutrality)* Ein Format ist neutral im Bezug auf natürliche Sprache, wenn es keine natürliche Sprache gibt, für die es signifikant besser geeignet ist als für die Übrigen. Beispielsweise wäre ein Format, welches nur ASCII-Zeichen speichern kann, signifikant besser für Englisch als für Chinesisch geeignet und damit nicht neutral bezüglich natürlicher Sprache.

Plattformneutralität *(Platform Neutrality)* Ein Format heißt plattformneutral, wenn es keine spezielle Plattform oder Architektur gibt, für die das Format signifikant besser geeignet ist als für die Übrigen.

Robustheit *(Robustness)* Ein Format heißt robust, wenn es Prozessoren erlaubt fehlerhafte Sektionen derart abzuschirmen, dass die Fähigkeit nachfolgende Sektionen zu verarbeiten nicht gefährdet wird.

Roundtrip-Unterstützung *(Roundtrip Support)* Ein Format unterstützt Roundtripping, wenn das Konvertieren eines XML-Dokuments in dieses Format und zurück wieder in einem äquivalenten XML-Dokument resultiert.

Ein Format unterstützt Roundtripping via XML, wenn das Konvertieren einer Datei diesen Formates in XML und wieder zurück in einer Datei resultiert, die äquivalent zur Originaldatei ist.

Schemaerweiterungen und Abweichungen *(Schema Extensions and Deviations)* Schemaerweiterungen und Schemaabweichungen werden unterstützt, wenn auch Informationen, die entweder nicht im Schema enthalten sind oder diesem nicht entsprechen, repräsentiert werden können (Stichwort: *open content*).

Signierbarkeit *(Signable)* Ein Format heißt signierbar, wenn es die Erzeugung und Validierung von digitalen Signaturen unkompliziert und interoperabel unterstützt.

Spezialisierte Codecs *(Specialized Codecs)* Ein Format unterstützt spezialisierte Codecs, wenn ein Prozessor für dieses Format spezielle Teile eines Dokuments erkennen und mit Erweiterungen (Plugins/Codecs) verarbeiten kann. Diese Erweiterungen erlauben es dem Prozessor die Verarbeitung optimal zu gestalten.

Übertragungsunabhängigkeit *(Transport Independence)* Ein Format heißt übertragungunabhängig, wenn die einzige Anforderung an das Übertragungsmedium oder das Transportprotokoll fehlerfreie Zustellung in korrekter Reihenfolge ist, ohne Begrenzung der Nachrichtenlänge.

Unterstützung von Einbettung *(Embedding Support)* Ein Format unterstützt Einbettung in dem Maße, in dem es den Austausch und die Verwaltung von eingebetteten Dateien beliebigen Formats unterstützt.

Unterstützung von Fehlerkorrekturen *(Support for Error Correction)* Ein Format unterstützt Fehlerkorrekturen, wenn Fehlerkorrektur-Codes darauf angewendet werden können. Diese ermöglichen (a) das Lokalisieren von fehlerhaften Sektionen und (b) die Wiederherstellung der nicht betroffenen Sektionen.

Formate zur Repräsentation von XML Datenmodellen können in drei Klassen aufgeteilt werden, die sich dadurch unterscheiden, ob und wie gut sich einzelne Sektionen isolieren lassen.

1. Eine Aufteilung der Repräsentation ist nicht möglich.
2. Eine Aufteilung der Repräsentation ist möglich.
3. Eine Aufteilung der Repräsentation nach der Wichtigkeit der Informationen ist möglich.

Die Eigenschaft der Fragmentierbarkeit ist eine Voraussetzung für die Unterstützung von Fehlerkorrekturen.

Verschlüsselbarkeit *(Encryptable)* Ein Format heißt verschlüsselbar, wenn es die Ver- und Entschlüsselung unkompliziert und interoperabel unterstützt.

Wahlfreier Zugriff *(Random Access)* Wahlfreier Zugriff ist die Fähigkeit, auf Elemente des Datenmodells in einem Dokument in konstanter Zeit zuzugreifen. Der wahlfreie Zugriff ist also das Gegenteil der sequentiellen Suche nach Elementen durch Traversierung der XML-Strukturen.

Widerstandsfähigkeit gegenüber Schemaänderungen *(Schema Instance Change Resilience)* Ein Format ist widerstandsfähig gegenüber Schemaänderungen, wenn ein Empfänger die nach seiner Schemadefinition gültigen Teile des Dokuments extrahieren kann, auch wenn die vom Sender geschickten Instanzen nach einem älteren oder neueren Schemamodell kodiert wurden.

Zusätzliche Überlegungen

Implementierungskosten *(Implementation Costs)* Gerade wenn sich ein Format weit verbreiten soll, dürfen die Kosten, die zur Implementierung eines Prozessors für dieses Format aufgewendet werden müssen, nicht allzu hoch sein. Insbesondere dürfen sie nicht höher als die Kosten zur Implementierung eines XML-Prozessors sein.

Vorwärtskompatibilität *(Forward Compatibility)* Ein Format ist vorwärtskompatibel, wenn es die Evolution zu neuen Versionen des Formats unterstützt. Es muss die Evolution von Datenmodellen und die Implementierung korrespondierender, geschichteter Standards unterstützen. Beispiele für die Evolutionen von Datenmodellen und Formaten sind das Hinzufügen von neuen Elementen und Zeichenkodierungen.

Eigenschaften, die direkt mit dieser Überlegung zusammenhängen sind

- Erweiterungspunkte,
- Keine Beschränkungen und
- Formatversionsidentifizierung.

Weitverbreiteter Einsatz *(Widespread Adoption)* Das Format soll auf einer großen Anzahl von Plattformen und Geräten und in einer Vielzahl von Applikationen genutzt werden.

Diese Eigenschaft wird unter anderem stark von den Implementierungskosten beeinflusst. Ein zu hoher Entwicklungsaufwand und damit hohe Entwicklungskosten, stehen einer weiten Verbreitung entgegen.

2.3 Alternative binäre XML-Formate

2.3.1 WBXML – WAP Binary XML

2.3.1.1 Einführung

Das *Wireless Application Protocol (WAP)*[1] ist eine Sammlung von Protokollen und Technologien, welche die Verfügbarmachung von Internetdiensten auf mobilen Geräten wie Handys und PDAs zum Ziel hat. Insbesondere die Anpassung von Internetinhalten an kleine Displays und den Umgang mit den Bandbreitenbeschränkungen im Mobilfunk waren zur Zeit der höchsten Aktivität der WAP-Initiative (1998-2002) besonders wichtige Aufgaben.

Zur Lösung der erstgenannten Anforderung wurde eine speziell zugeschnittene Beschreibungssprache, die *Wireless Markup Language (WML)* [19], spezifiziert, welche eine Untermenge von XHTML [18] darstellt. Zur effizienten und bandbreitenschonenden Übertragung dieser WML-Seiten, wurde zusätzlich die binäre XML-Kodierung WBXML [20] erstellt, die in diesem Kapitel nun erläutert und bewertet werden soll.

2.3.1.2 Eigenschaften von WBXML

Da die Kodierung mittels WBXML nicht sonderlich kompliziert und die Spezifikation [20] leicht verständlich und kurz ist, werden in diesem Abschnitt nur die Grundzüge der Kodierung mit WBXML erläutert, anstatt die gesamte Spezifikation übersetzt wiederzugeben. Im folgenden Abschnitt wird die Kodierung dann noch mittels eines konkreten Beispiels demonstriert und veranschaulicht.

Der Hauptgrund für die Verbosität von XML sind die sich immer wiederholenden Strukturdaten, also Tags und Attribute. Um diese möglichst platzsparend zu kodieren, werden in WBXML drei Tabellen definiert: eine Tag-Tabelle, eine Attribut-Tabelle und eine Wert-Tabelle. Um einen neuen XML-Dialekt als WBXML kodieren zu können, müssen in diesen geeignete Einträge vorgenommen werden. Im Rahmen dieser Studienarbeit wurde Unterstützung für GXL in die freie `libwbxml`-Bibliothek[2] integriert.

- In der Tag-Tabelle werden alle Elemente des jeweiligen XML-Dialekts aufgelistet und jedem davon einen Integer-Wert zugeordnet, der das entsprechende Tag eindeutig identifiziert. Dafür sind 6 Bit vorgesehen. Die höchstwertigen 2 Bit zeigen an, ob dem öffnenden Tag Attribute folgen und ob es Inhalt einschließt. Damit mehr als 64 verschiedene Tags möglich sind, gibt es 256 sogenannte *code pages*, in denen sie definiert werden können. GXL kommt allerdings mit der Default-Code-Page 0 aus, so dass hier nicht genauer auf den Code-Page-Mechanismus eingegangen wird. Mit den Zuordnungen dieser Tabelle (siehe Tabelle aus Abbildung 2.1) ist es möglich, alle XML-Tags mit nur einem Byte zu kodieren.
- Die Attribut-Tabelle (siehe Abbildung 2.2) beschreibt analog die Kodierung der Attribute. Da viele Attribute nicht beliebigen Text enthalten können, sondern

[1] `http://www.wapforum.org/` bzw. `http://www.openmobilealliance.org/tech/affiliates/wap/wapindex.html`, 05.07.2006

[2] `http://libwbxml.aymerick.com`, 11.09.2006

Tag	Hex-Value	Tag	Hex-Value
gxl	05	bool	0E
type	06	int	0F
graph	07	float	10
node	08	string	11
edge	09	enum	12
rel	0A	seq	13
relend	0B	set	14
attr	0C	bag	15
locator	0D	tup	16

Abbildung 2.1: Kodierungstabelle der GXL-Tags

nur bestimmte Werte, können die möglichen Attribut-Wert-Kombinationen explizit definiert werden. Damit werden Attribute bzw. Attribut-Wert-Paare analog zu den Tags mit einem Byte kodiert. Zulässig sind hier Werte kleiner 128 (dezimal). Genau wie bei der Kodierungstabelle der Tags kann auch hier mit einem Code-Page-Mechanismus die Anzahl der möglichen Attribute bzw. Attribut-Wert-Kombinationen erhöht werden.

Attribute	Attribute-Value	Hex-Value
xmlns:xlink	http://www.w3.org/1999/xlink	05
xlink:type	simple	06
xlink:href		07
id		08
role		09
edgeids	true, false	0A, 0B
hypergraph	true, false	0C, 0D
edgemode	directed, undirected, defaultdirected, defaultundirected	0E, 0F, 10, 11
from		12
to		13
fromorder		14
toorder		15
isdirected	true, false	16, 17
target		18
direction	in, out, none	19, 1A, 1B
startorder		1C
endorder		1D
name		1E
kind		1F

Abbildung 2.2: Kodierungstabelle der GXL-Attribute

- In der Wert-Tabelle können zuletzt noch häufig vorkommende Attributwerte definiert werden (*common value tokens)*. Da dies sehr oft URLs sind, wurden die in Abbildung 2.3 aufgelisteten Zuordnungen getroffen. Es sind Werte größer oder gleich 128 (dezimal) zulässig. Auch hier lässt sich der mögliche Werteraum durch den Code-Page-Mechanismus erweitern.

Attribute-Value	Hex-Value	Attribute-Value	Hex-Value
.com/	85	http://www.	8F
.edu/	86	http://	8E
.net/	87	https://www.	91
.org/	88	https://	90
.de/	89		

Abbildung 2.3: Kodierungstabelle häufiger GXL-Attributwerte

Ein weiterer Grund für die Verbosität von XML neben den Strukturdaten (Tags, Attribute) sind Wiederholungen von Zeichenketten, die entweder als Inhalt zwischen Tags oder als Attributwert im Dokument enthalten sind. Damit keine oft vorkommende Zeichenkette mehrfach gespeichert wird, kann ein WBXML-Encoder Strings in einer sogenannten String-Tabelle speichern. An den Stellen der Benutzung wird dann nur noch ein Offset in dieser Tabelle angegeben. Nur einmal vorkommende Zeichenketten dürfen auch an Ort und Stelle (*inline*) gespeichert werden. Wie ein Encoder damit umgeht, ist nicht genau spezifiziert, jedoch soll immer die kompakteste Form gewählt werden.

2.3.1.3 Beispiel zu WBXML

In diesem Abschnitt soll das WBXML-Format an einem Beispiel erklärt werden. Dazu wird das einfache GXL-Dokument aus Listing 2.1 mit der WBXML-kodierten Variante verglichen.

Listing 2.1: Die GXL-Beispieldatei `wbxml_test.gxl`

```
1  <?xml version="1.0"?>
2  <!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.dtd">
3  <gxl xmlns:xlink="http://www.w3.org/1999/xlink">
4    <graph id="attributes-instance">
5      <type xlink:href="http://www.foo.org/schema.gxl#attributesSchema"/>
6      <node id="a-node">
7        <type xlink:href="http://www.foo.org/schema.gxl#Prog"/>
8        <attr name="file">
9          <tup>
10           <string>a string</string>
11           <bool>true</bool>
12         </tup>
13       </attr>
14     </node>
15     <node id="another-node">
16       <type xlink:href="http://www.foo.org/schema.gxl#Prog"/>
17       <attr name="file">
18         <float>-17.63</float>
19       </attr>
20     </node>
21     <edge id="an-edge" from="a-node" to="another-node">
22       <type xlink:href="http://www.foo.org/schema.gxl#Ref"/>
23       <attr name="line">
24         <int>27</int>
25       </attr>
26     </edge>
27   </graph>
28 </gxl>
```

Zur (De-)Kodierung wurden die Tools `xml2wbxml` und `wbxml2xml` der `libwbxml`-Bibliothek verwendet. Im `tools`-Verzeichnis dieser Studienarbeit liegt ein Patch, der die Version 0.9.2 dieser Bibliothek GXL-fähig macht, indem er die erforderlichen Tabellen (siehe Abschnitt 2.3.1.2) implementiert.

In der folgenden Tabelle wird die WBXML-Datei Byte für Byte durchgegangen und erläutert.

Bytes	**Erläuterung**
`03`	Versionsnummer: WBXML Version 1.3
`00 19`	Public Identifier steht in der String-Tabelle beginnend mit Offset $19_{16} = 25$.
`6a`	Charset UTF-8 (MIBEnum $106 = 6a_{16}$)
`32`	Längenfeld der Stringtabelle: Länge ist $32_{16} = 50$
`61 2d 6e 6f 64 65 00`	1. Eintrag der String-Tabelle (Offset 0, nullterminiert): "`a-node`"
`66 69 6c 65 00`	2. Eintrag der String-Tabelle (Offset 7, nullterminiert): "`file`"
`61 6e 6f 74 68 65 72 2d 6e 6f 64 65 00`	3. Eintrag der String-Tabelle (Offset $12 = 0c_{16}$, nullterminiert): "`another-node`"
`2d 2f 2f 47 58 4c 2f 2f 44 54 44 20 47 58 4c 20 56 31 2e 30 2f 2f 45 4e 00`	4. Eintrag der String-Tabelle (Offset $25 = 19_{16}$, nullterminiert): "`-//GXL//DTD GXL V1.0//EN`"
`c5`	`gxl`-Element, Attributliste und Inhalt folgen
`05`	`xmlns:xlink`-Attribut mit Wert `http://www.w3.org/1999/xlink`
`01`	Ende der Attributliste (des `gxl`-Tags)
`c7`	`graph`-Element, Attributliste und Inhalt folgen
`08`	`id`-Attribut
`03`	Inline-String folgt
`61 74 74 72 69 62 75 74 65 73 2d 69 6e 73 74 61 6e 63 65 00`	Inline-String: "`attributes-instance`"
`01`	Ende der Attributliste (des `graph`-Tags)
`86`	`type`-Element, Attributliste folgt aber kein Inhalt
`07`	`xlink:href`-Attribut
`8f`	*common value token*: "`http://www.`"
`03`	Inline-String folgt
`66 6f 6f 00`	Inline-String: "`foo`"
`88`	*common value token*: "`.org/`"
`03`	Inline-String folgt

73 63 68 65 6d 61 2e 67 78 6c 23 61 74 74 72 69 62 75 74 65 73 53 63 68 65 6d 61 00	Inline-String: "schema.gxl#attributesSchema"
01	Ende der Attributliste und des type-Elements
c8	node-Element, Attributliste und Inhalt folgen
08	id-Attribut
83	String-Tabellen-Referenz folgt
00	Offset in String-Tabelle ist 0, also der 1. Eintrag: "a-node"
01	Ende der Attributliste (des node-Tags)
86	type-Element, Attributliste folgt aber kein Inhalt
07	xlink:href-Attribut
8f	*common value token*: "http://www."
03	Inline-String folgt
66 6f 6f 00	Inline-String: "foo"
88	*common value token*: ".org/"
03	Inline-String folgt
73 63 68 65 6d 61 2e 67 78 6c 23 50 72 6f 67 00	Inline-String: "schema.gxl#Prog"
01	Ende der Attributliste und des type-Elements
cc	attr-Element, Attributliste und Inhalt folgen
1e	name-Attribut
83	String-Tabellen-Referenz folgt
07	Offset in String-Tabelle ist 7, also 2. Eintrag: "file"
01	Ende der Attributliste (des attr-Tags)
56	tup-Element, keine Attributliste, Inhalt folgt
51	string-Element, keine Attributliste, Inhalt folgt
03	Inline-String folgt
61 20 73 74 72 69 6e 67 00	Inline-String: "a string"
01	Ende des string-Elements
4e	bool-Element, keine Attributliste, Inhalt folgt
03	Inline-String folgt
74 72 75 65 00	Inline-String: "true"
01	Ende des bool-Elements
01	Ende des tup-Elements
01	Ende des attr-Elements
01	Ende des node-Elements
c8	node-Element, Attributliste und Inhalt folgen
08	id-Attribut
83	String-Tabellen-Referenz folgt

0c	Offset in String-Tabelle ist $0c_{16} = 12$, also 3. Eintrag: "another-node"
01	Ende der Attributliste (des node-Tags)
86	type-Element, Attributliste folgt aber kein Inhalt
07	xlink:href-Attribut
8f	*common value token*: "http://www."
03	Inline-String folgt
66 6f 6f 00	Inline-String: "foo"
88	*common value token*: ".org/"
03	Inline-String folgt
73 63 68 65 6d 61 2e 67 78 6c 23 50 72 6f 67 00	Inline-String: "schema.gxl#Prog"
01	Ende der Attributliste und des type-Elements
cc	attr-Element, Attributliste und Inhalt folgen
1e	name-Attribut
83	String-Tabellen-Referenz folgt
07	Offset in String-Tabelle ist 7, also 2. Eintrag: "file"
01	Ende der Attributliste (des attr-Tags)
50	float-Element, ohne Attributliste, Inhalt folgt
03	Inline-String folgt
2d 31 37 2e 36 33 00	Inline-String: "-17.63"
01	Ende des float-Elements
01	Ende des attr-Elements
01	Ende des node-Elements
c9	edge-Element, Attributliste und Inhalt folgen
08	id-Attribut
03	Inline-String folgt
61 6e 2d 65 64 67 65 00	Inline-String: "an-edge"
12	from-Attribut
83	String-Tabellen-Referenz folgt
00	Offset in String-Tabelle ist 0, also 1. Eintrag: "a-node"
13	to-Attribut
83	String-Tabellen-Referenz folgt
0c	Offset in String-Tabelle ist $0c_{16} = 12$, also 3. Eintrag: "another-node"
01	Ende der Attributliste (des edge-Tags)
86	type-Element, Attributliste folgt aber kein Inhalt
07	xlink:href-Attribut
8f	*common value token*: "http://www."
03	Inline-String folgt
66 6f 6f 00	Inline-String: "foo"
88	*common value token*: ".org/"
03	Inline-String folgt

73 63 68 65 6d 61 2e 67 78 6c 23 52 65 66 00	Inline-String: "schema.gxl#Ref"
01	Ende der Attributliste und des type-Elements
cc	attr-Element, Attributliste und Inhalt folgen
1e	name-Attribut
03	Inline-String folgt
6c 69 6e 65 00	Inline-String: "line"
01	Ende der Attributliste (des attr-Tags)
4f	int-Element, keine Attributliste, Inhalt folgt
03	Inline-String folgt
32 37 00	Inline-String: "27"
01	Ende des int-Elements
01	Ende des attr-Elements
01	Ende des edge-Elements
01	Ende des graph-Elements
01	Ende des gxl-Elements
Ende der Tabelle	

2.3.1.4 Fazit und Bewertung

WBXML ist ein relativ einfaches Format, welches keine wirkliche Kompression bietet, sondern nur die Verbosität von XML durch Beseitigung von langen Element- und Attribut-Namen und die Vermeidung von Wiederholungen versucht wett zu machen.

Von den von der XBC aufgestellten wünschenswerten Eigenschaften erfüllt WBXML nur geringe Prozessorgröße, direkte Schreib-/Lesbarkeit, Formatversionsidentifizierung, Lizenzkostenfreiheit, lokal begrenzte Änderungen, Neutralität bezüglich natürlicher Sprache, Plattformneutralität, Übertragungsunabhängigkeit und geringe Implementierungskosten. Günstig ist allerdings, dass mit der libwbxml eine freie Implementierung existiert.

Viele besonders wichtige Anforderungen wie die Unterstützung von wahlfreiem Zugriff, beschleunigtem sequentiellen Zugriff, Fehlerkorrektur, Content-Type-Management, Widerstandsfähigkeit gegenüber Schemaänderungen und Erweiterungspunkten sind allerdings nicht erfüllt.

Außerdem ist das Interesse am Haupteinsatzgebiet von WBXML, also WAP und WML, weitestgehend abgeklungen, so dass keine neuen Innovationen und Entwicklungen mehr in WBXML einfließen.

2.3.2 BiM – Binary Format for MPEG-7

2.3.2.1 Einführung

In vielen Anwendungsfällen ist die ressourcenschonende Übertragung von Zusatzinformationen zu den eigentlichen Inhaltsdaten wichtig. Beim digitalen Fernsehen sollen neben den Audio- und Videodaten beispielsweise noch Metadaten wie Untertitel und Programmübersichten übertragen werden.

Der von der Moving Pictures Expert Group entwickelte MPEG-7 ISO/IEC Standard (*Multimedia Content Description Interface*, [24]) bietet Mittel zur Beschreibung der Inhalte multimedialer Daten. Aber auch weitere Metadaten wie z.B. Copyright- und DRM-Informationen, Altersfreigaben, Preis, Referenzen zu vergleichbarem, für den Benutzer interessantem Material, Zeichenkodierung und Speicherformat werden von MPEG-7 berücksichtigt.

MPEG-7-Beschreibungen sind XML-Dateien, welche durch sogenannte Beschreibungsschemata (*Description Schemes*) spezifiziert werden. Ein solches Beschreibungsschema beschreibt die Struktur und Semantik der Beziehungen seiner Komponenten, welche wiederum Beschreibungsschemata oder *Descriptors* sein können. Ein Descriptor beschreibt die Syntax und die Semantik der Darstellung eines Merkmals, beispielsweise die Farbe eines Video-Objekts oder die Tonart eines Musikstücks.

Mit einer *Description Definition Language (DDL)* kann man neue Beschreibungsschemata und Descriptors definieren. Diese DDL baut auf XML Schema auf und fügt einige Erweiterungen ein, die besondere Wichtigkeit im Zusammenhang mit multimedialen Daten haben, wie z.B. neue Datentypen für Arrays und Matrizen. Ebenfalls führt die DDL primitive Datentypen für Zeitpunkte und Zeitspannen ein.

Zusammenfassend und vereinfacht kann man also sagen, dass MPEG-7 eine auf XML Schema basierende Sprache zur Definition von Metadaten bereitstellt. Es gibt eine große Anzahl von standardisierten Elementen und zugehörigen Attributen, die nach Anwendungsgebieten gruppiert sind (eigene Gruppen für Audio-/Videoinhalte, Gruppen für Elemente, die die Navigation und den Zugriff erleichtern, wie z.B. Zusammenfassungen, etc.), und man kann zusätzlich noch eigene Elemente definieren.

Die resultierenden Beschreibungen sind valide XML-Daten. Um die in Abschnitt 2.2.1 auf Seite 14 aufgezeigten Eigenschaften zu befriedigen, liefert der MPEG-7-Standard eine leistungsfähige binäre Serialisierung dieser XML-Daten – das Binary Format for MPEG-7 (BiM), auf welches nun genauer eingegangen werden soll.

2.3.2.2 Eigenschaften von BiM

BiM ist ein generisches Framework zur binären Serialisierung von beliebigen XML-Daten, um diese auf effiziente Weise zu übertragen und zu verarbeiten. Besonders hervorzuheben ist, dass sich beliebige XML-Daten mit BiM binär kodieren lassen, indem aus der Schemadefinition (XML Schema, bzw. der darauf aufbauenden DDL oder DTD) automatisch das Binärformat abgeleitet wird. Durch die Kenntnis des Schemas kann die Dokumentstruktur im Durchschnitt um 98% komprimiert werden. Die Kompression für Strukturdaten basiert auf endlichen Automaten und ist dadurch einfach, effizient und ermöglicht Validierung.

Die eigentlichen Daten, also Element- und Attributwerte, werden mit spezialisierten Codecs kodiert. Die Spezifikation beinhaltet schon eine ganze Reihe von Codecs für einfache und wichtige Datentypen, z.B. UTF-8 für Zeichenketten. Weiterhin definiert der BiM-Standard einen Typ-Codec-Zuordnungsmechanismus, welcher die einfache Einbindung neuer Codecs für bestimmte Typen ermöglicht. Im Vergleich mit einer textuellen XML-Datei hat eine BiM-Datei durchschnittlich nur noch 15% der ursprünglichen Größe.

Eine weitere Besonderheit ist, dass die Daten schon bei der Kodierung validiert werden können. Normalerweise findet die Validierung auf Empfängerseite nach Erhalt des gesamten Dokuments statt. Diese Eigenschaft ist es, die BiM zu einem sogenannten *pre-validated* Format machen. Bei der Validierung werden den Komponenten des XML-Dokuments (Elemente, Attribute) Typinformationen, Defaultwerte etc. zugeordnet. Deshalb wird BiM ein *pre-parsed* Format genannt. Das verbessert einerseits die Kompressionsfähigkeit, denn jedem Typ kann eine bestmögliche Kodierungsform zugeordnet werden und erleichtert andererseits die Verarbeitung des Dokuments auf Empfängerseite.

Zwei weitere in Abschnitt 2.2.1 aufgeführte Eigenschaften sind die Vorwärts- und Rückwärtskompatibilität. Auch diese werden von BiM unterstützt. Ein Decoder für BiM kann problemlos Daten einer älteren Sprachversion verarbeiten. Auf der anderen Seite kann ein Encoder Daten derart kodieren, dass ein Decoder für ältere Sprachversionen unbekannte Teile des Streams überspringen kann. Diese Eigenschaft erlaubt auch das Einfügen von privaten Erweiterungen ohne die Interoperabilität zu zerstören. Diese Vorwärtskompatibilität bietende Kodierungsvariante führt einige Redundanzen in den Stream ein. Wird keine Vorwärtskompatibilität benötigt, so kann darauf verzichtet werden, und die resultierenden BiM-Dokumente sind entsprechend kompakter.

BiM-kodierte Dokumente können komplett in einem Stück oder fragmentweise übertragen werden. Abbildung 2.4 zeigt einige mögliche Übertragungsstrategien für ein und dieselbe XML-Datei.

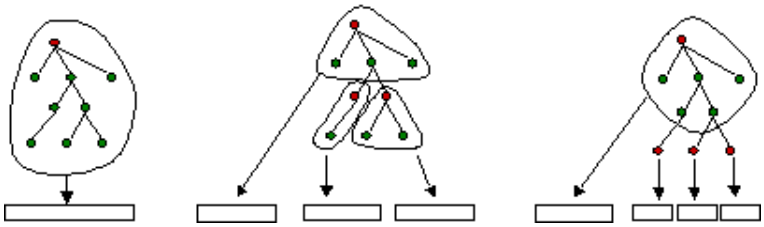

Abbildung 2.4: BiM Streamingstrategien

Der Decoder kann die einzelnen Fragmente verarbeiten, ohne Kenntnis des gesamten Dokuments zu haben.

Eine weitere Anwendungsmöglichkeit, die hierdurch entsteht, ist das Update von Dokumenten. Hier muss nicht mehr das gesamte neue Dokument übertragen werden, sondern jeweils nur die veränderten Teile. Das ist insbesondere dann nützlich, wenn Daten nicht automatisch übertragen werden, sondern von einer Quelle abgefragt werden. Je nach Wichtigkeit eines Teildokuments für eine Anwendung können dann verschiedene Update-Intervalle vereinbart werden, in denen Änderungen abgefragt werden. Stellt man sich beispielsweise eine Art Programmübersicht mit Informationen zu allen Sendungen vor, so interessieren besonders die Programme, die zum Profil des Benutzers passen.

Die bisher beschriebenen Eigenschaften dienen besonders zur Reduzierung der benötigten Bandbreite, der Dokumentengröße an sich und des Speicherbedarfs auf der Empfängerseite. BiM kann aber auch extrem schnell auf dem binären Level verarbeitet werden. Der Geschwindigkeitsgewinn wird mit einer Steigerung mindestens um den Faktor 10 bis maximal Faktor 30 im Vergleich zu textuellem XML angegeben. Ein Punkt, der dazu beiträgt, ist, dass ganze Unterbäume übersprungen werden können, wie die Abbildung 2.5 verdeutlicht.

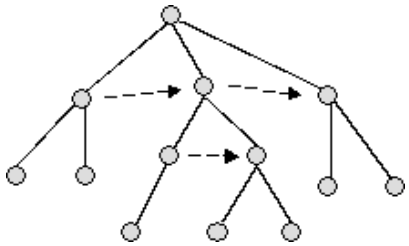

Abbildung 2.5: **FastSkip**: Schnelles Überspringen von Unterbäumen

Dieses Überspringen von Unterbäumen kann durch Elementnamen, Typen oder Attributwerte ausgelöst werden. Gerade bei großen XML-Dateien kann dadurch das Browsen und Suchen verbessert werden. Möchte man in einer Programmübersicht beispielsweise nach einem Film eines bestimmten Genres mit einem bestimmten Schauspieler suchen, so kann man Unterbäume überspringen, die Informationen zu Filmen anderer Kategorien bereitstellen.

Eine letzte Eigenschaft von BiM ist, dass es anpassbar auf verschiedene Sprachen und Anwendungsgebiete ist. Durch neue Codecs kann für bestimmte Datentypen (Inhalte) eine besondere Kompressionsmethode verwendet werden, es können Checksummen zur Fehlererkennung und Korrektur eingefügt werden, oder spezielle Ver- und Entschlüsselungsroutinen können implementiert werden.

2.3.2.3 Beispiel zur Strukturkompression bei BiM

Die Struktur des XML-Dokuments wird bei BiM mittels endlicher Automaten kodiert. Wie dieses Verfahren im Einzelnen funktioniert, soll in diesem Abschnitt an einem Beispiel verdeutlicht werden. Das Beispiel ist stark an ein Beispiel einer Präsentation von Robin Berjon [10] angelehnt.

BiM extrahiert die Struktur der zu (de-)kodierenden Daten aus der DTD bzw. der XML-Schema-Beschreibung. Listing 2.2 auf Seite 28 zeigt ein einfaches XML-Schema, aus welchem nun ein endlicher Automat konstruiert werden soll, mit dem man zugehörige Instanzen eindeutig kodieren und dekodieren kann.

Der Automat wird von “innen nach außen” aufgebaut, beginnend mit den einfachen Elementen des `<xs:choice>`-Elements. Der resultierende partielle Automat ist in Abbildung 2.6 zu sehen.

Als nächstes muss das `<xs:choice>`-Element eingefügt werden. Epsilon-Kanten werden hier der Einfachheit halber ohne Beschriftung notiert.

Wegen `minOccurs='0'` und `maxOccurs='unbounded'` können beliebig viele der inneren Elemente aufeinander folgen. Der Fall, dass kein einziges der inneren Elemente vorkommt, ist gesondert zu behandeln, und es muss eine Epsilon-Kante eingefügt

Listing 2.2: XML-Schema-Definition des `example` Elements

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

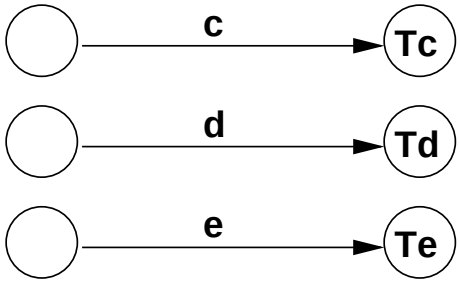

Abbildung 2.6: Schritt 1: Modellierung der inneren Elemente

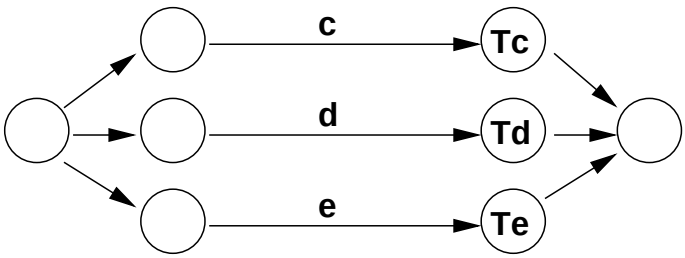

Abbildung 2.7: Schritt 2: Automat mit `choice`-Element

werden, mit welcher das `<xs:choice>`-Element übersprungen werden kann. Der resultierende Teilautomat ist in Abbildung 2.8 dargestellt.

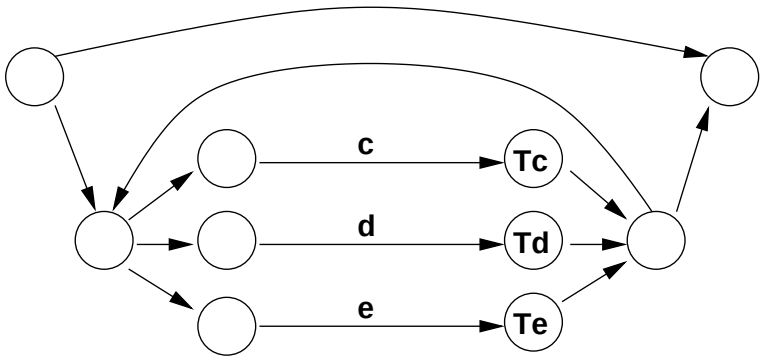

Abbildung 2.8: Schritt 3: Automat nach Modellierung der Kardinalitäten

Jetzt fehlen nur noch die Elemente vor dem <xs:choice>-Element. Der komplette endliche Automat ist dann in Abbildung 2.9 angegeben.

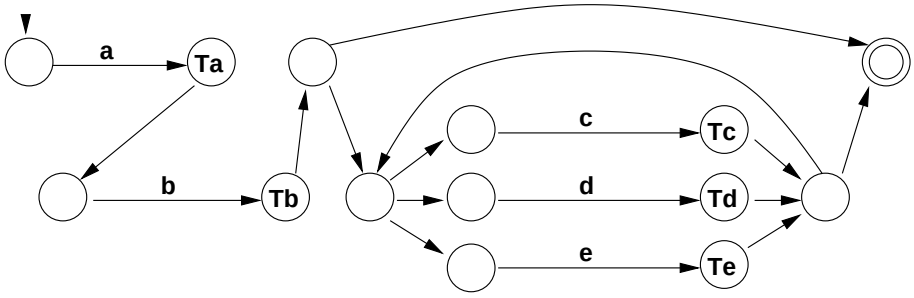

Abbildung 2.9: Schritt 4: Der fertige endliche Automat

An Verzweigungsstellen muss nun jeweils kodiert werden, welcher Kante zu folgen ist. Dies geschieht durch einfaches Durchnummerieren der alternativen Kanten (Abbildung 2.10).

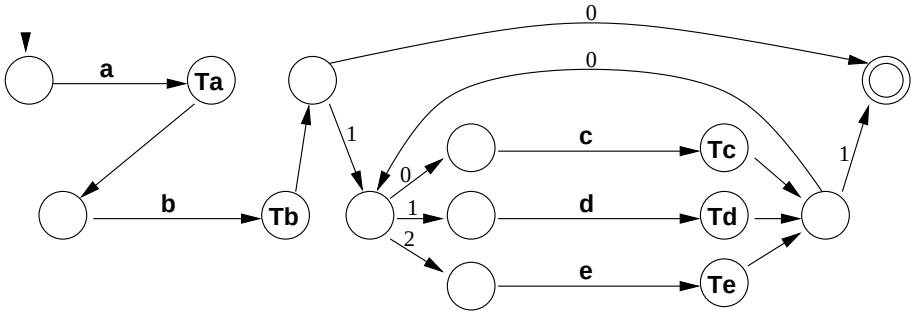

Abbildung 2.10: Schritt 5: Nummerierung der alternativen Kanten

Zuletzt kann der Automat noch ein wenig vereinfacht werden (Abbildung 2.11).

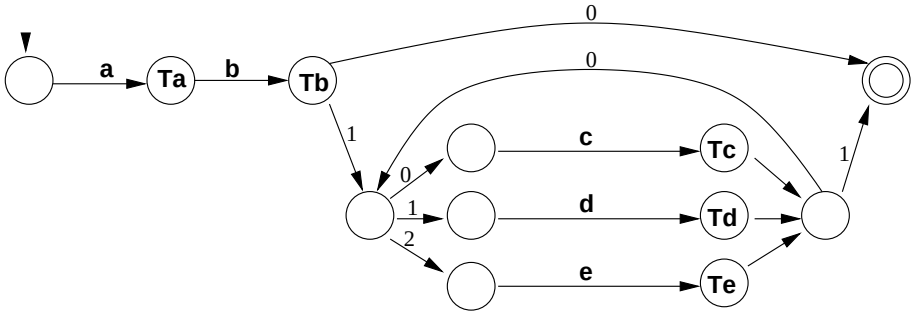

Abbildung 2.11: Schritt 6: Der finale endliche Automat

Im Folgenden soll nun anhand diese Automaten die Kodierung einer Instanz von example bestimmt werden. Die Instanz ist in Listing 2.3 auf Seite 30 angegeben.

Bei der Kodierung wird wie folgt vorgegangen. Da eine Instanz eines example-Elements immer mit einem a-Element gefolgt von einem b-Element beginnt, werden für diese keine Strukturdaten benötigt, sondern die Elemente können direkt kodiert werden. Danach befindet sich der Automat im Zustand T_b. Da kein schließendes example-Tag folgt, wird eine 1 eingefügt. Diese Kante führt in den Startzustand des Choice-Teilautomaten. Eine 0 zeigt an, dass ein c-Element folgt, nach welchem sich der Auto-

Listing 2.3: Die zu kodierende Instanz

```
<example>
  <a/>
  <b/>
  <c/>
  <c/>
  <e/>
  <d/>
  <c/>
</example>
```

mat im Zustand T_c befindet. Da weitere Elemente folgen, wird über die `0`-Kante zurück gesprungen. Für das zweite `c`-Element werden wieder die selben Strukturdaten generiert. Die restlichen Elemente werden analog kodiert, und zuletzt wird über die `1`-Kante in den Finalzustand gesprungen.

Damit wird die Struktur der Beispielinstanz kodiert zu:

```
(a b) 1 0 (c) 0 0 (c) 0 2 (e) 0 1 (d) 0 0 (c) 1
```

Besonders zu beachten ist, dass mit diesem einfachen Verfahren die korrekte Struktur des XML-Dokuments bereits bei der Kodierung validiert wird.

2.3.2.4 Fazit und Bewertung

Mit BiM scheint ein hervorragender Kandidat für einen einheitlichen binären XML-Standard gefunden worden zu sein. Einer der großen Vorteile von BiM ist, dass es sich für beliebige XML-Dialekte verwenden lässt, da das jeweilige Binärformat direkt aus dem XML-Schema oder der DTD abgeleitet werden kann.

Weiterhin bietet BiM eine sehr gute Kompression: Strukturdaten werden mittels eines einfachen aber effizienten Verfahrens, aufbauend auf endlichen Automaten, um durchschnittlich 98% komprimiert. Für die eigentlichen Inhaltsdaten können spezielle Codecs verwendet werden, die besonders für die jeweiligen Datentypen zugeschnitten sind. Denkbar sind hier neben Codecs zur optimalen Kompression auch Signatur- und Verschlüsselungs-Codecs. Da diese Codecs beliebig austauschbar sind, setzt BiM auch Maßstäbe in puncto Erweiterbarkeit und Flexibilität. Damit werden die XBC-Eigenschaften Allgemeingültigkeit, Erweiterungspunkte, Kompaktheit, Signierbarkeit, spezialisierte Codecs, Unterstützung von Fehlerkorrektur und Verschlüsselbarkeit erfüllt.

Neben der guten Kompression und der universellen Anwendbarkeit erlaubt BiM das Übertragen von Dokumenten nach verschiedenen Strategien, das Überspringen ganzer Unterbäume, und es ist außerdem ein pre-validated/pre-parsed Format, d.h. die Validierung erfolgt bereits bei der Kodierung und es werden Typinformationen und Defaultwerte eingefügt. Weitere erfüllte Eigenschaften sind beschleunigter sequentieller Zugriff, Fragmentierbarkeit, inkrementelle Verarbeitbarkeit und Übertragungsunabhängigkeit.

Erfüllt werden ebenfalls Vorwärts- und Rückwärtskompatibilität, so dass BiM nahezu alle Eigenschaften der XBC unterstützt. Der einzige Nachteil ist, dass bislang noch keine freie Implementierung des BiM-Standards existiert, was sich jedoch vermutlich mittelfristig ändern wird, falls BiM sich so stark durchsetzen kann, wie es seine Eigenschaften und Vorteile vermuten lassen.

Im folgenden Kapitel werden die Voraussetzungen für den praktischen Teil dieser Studienarbeit getroffen, indem die *Abstract Syntax Notation One* (ASN.1) vorgestellt und erläutert wird. Sie wird dann im vierten Kapitel zur Implementierung der binären Kodierung des XML-Dialekts GXL verwendet.

Kapitel 3

Binäre Kodierung mittels ASN.1 und DER

Um binär kodierbare Datentypen zu modellieren, bietet sich die *Abstract Syntax Notation One* (**ASN.1**) an. Sie ist eine von der ISO (*International Organization for Standardization*[1]) und der ITU (*International Telecommunication Union*[2]) standardisierte Notation zur Beschreibung von Datenstrukturen, welche zwischen Computersystemen ausgetauscht werden (s. [11] [12] [13] [14]). ASN.1 stellt dazu eine Reihe von Basistypen zur Verfügung, auf die später noch näher eingegangen wird. Wichtig hierbei ist, dass die Beschreibung unabhängig von Hersteller, Plattform oder Programmiersprache ist.

3.1 Der Entwicklungsablauf mit ASN.1

Entwickelt man eine Anwendung, die nach einem bestimmten Protokoll Daten mit anderen Systemen austauschen soll, so muss man die Struktur dieser Daten beschreiben. Dazu kann man unter anderem ASN.1 verwenden.

Eine solche ASN.1-Spezifikation nennt man auch *abstrakte Syntax*, denn sie ist unabhängig von Plattform und Programmiersprache und genügt den Regeln der Sprache ASN.1.

Im nächsten Schritt wird die abstrakte Syntax mittels eines ASN.1-Compilers in die sogenannte *konkrete Syntax* übersetzt. Diese ist eine Umsetzung der abstrakten Syntax in einer Programmiersprache wie C, C++ oder Java. Zusätzlich zu den Datentypen liefert der ASN.1-Compiler Funktionen zur Kodierung und Dekodierung derselben.

Diese Kodier-/Dekodierfunktionen überführen schlussendlich Instanzen der Datentypen der konkreten Syntax in die sogenannte *Transfersyntax*. Die Kodierung/ Dekodierung wird anhand fest vorgegebener *Encoding Rules* vorgenommen, die beschreiben, wie ein Datum der abstrakten Syntax als Bytefolge kodiert bzw. dekodiert werden muss. Die Abbildung 3.1 auf Seite 34 verdeutlicht den ASN.1-Workflow in graphisch aufbereiteter Weise.

Der ASN.1-Compiler **asn1c**, der im Rahmen dieser Studienarbeit benutzt wurde (s. Anhang A.2 auf Seite 76 und [5]), übersetzt die Typen der BinaryGXL-Spezifikation (ab-

[1] `http://www.iso.org`, 05.07.2006
[2] `http://www.itu.int`, 05.07.2006

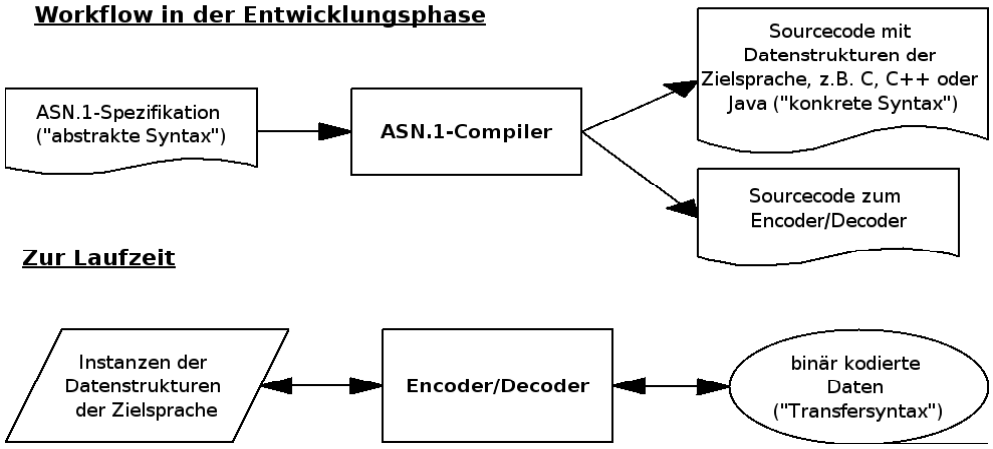

Abbildung 3.1: Der ASN.1-Workflow

strakte Syntax) beispielsweise in Strukturen der Sprache C (konkrete Syntax), also in eine Reihe von `structs` und `unions`. Genauso gut könnte man aber einen ASN.1-Compiler benutzen, der stattdessen als konkrete Syntax die Sprache Java verwendet. Dann würden die Typen der abstrakten Syntax als Klassen übersetzt. Der Bytestream (Transfersyntax), der bei der Kodierung erzeugt wird, ist aber, unabhängig von der verwendeten konkreten Syntax, immer inhaltsgleich.

Eine Auflistung und Beschreibung der standardisierten *Encoding Rules* findet sich im Abschnitt 3.4 auf Seite 40.

3.2 Beispiel einer ASN.1-Spezifikation

Eine ASN.1-Spezifikation kann aus beliebig vielen Modulen bestehen. Ein Modul hat folgende Grundform:

```
1 <ModulName> DEFINITIONS ::=
2 BEGIN
3
4 {<TypDefinition>}
5
6 END
```

Die Moduldeklaration kann noch deutlich komplexer als obige Grundform werden, so können z.B. mit `IMPORT <type>;` oder `EXPORT <type>;` angegeben werden, welche Typen schon in anderen Modulen deklariert wurden und hier benutzt werden bzw. welche Typen des aktuellen Moduls nach außen sichtbar sein sollen. Modul- und Typnamen beginnen typischerweise mit einem Großbuchstaben. In der ASN.1-Spezifikation von BinaryGXL wird für beides CamelCase verwendet.

Listing 3.1 zeigt die Spezifikation eines Moduls `Adressbuch`, in dem die Datentypen `AdressBuch` und `Kontakt` definiert sind.

Ein `AdressBuch` enthält eine Liste von Kontakten (Zeile 14). Der ASN.1-Typ `SEQUENCE OF` implementiert eine geordnete Liste.

Listing 3.1: ASN.1-Spezifikation eines einfachen Adressbuches

```
1  Adressbuch DEFINITIONS ::=
2  BEGIN
3
4  Kontakt ::= SEQUENCE {
5    name       BMPString,
6    vorname    BMPString,
7    strasse    BMPString,
8    hausnummer INTEGER,
9    plz        INTEGER,
10   stadt      BMPString,
11   stichworte SEQUENCE OF BMPString OPTIONAL
12 }
13
14 AdressBuch ::= SEQUENCE OF Kontakt
15
16 END
```

Jeder `Kontakt` hat wiederum einen Namen, einen Vornamen und eine Adresse (Zeile 4 bis 10). Der ASN.1-Typ `BMPString` implementiert eine unicodefähige Zeichenkette, `INTEGER` eine Ganzzahl.

Im Beispieladressbuch ist mit jedem Kontakt eine Stichwortliste assoziiert (Zeile 11). Das ASN.1-Schlüsselwort `OPTIONAL` deutet jedoch darauf hin, dass dieser Eintrag optional ist.

Der eingesetzte ASN.1-Compiler *asn1c* setzt diese Spezifikation in folgende konkrete Syntax (Programmiersprache C) um, welche aus Gründen der Übersichtlichkeit etwas vereinfacht wurde. In jedem der angegebenen `structs` gibt es noch Zeiger, die für die Kodierungsfunktionen wichtig sind, für den Entwickler allerdings keinerlei Bedeutung haben.

Listing 3.2: Datei: AdressBuch.h float

```
1 typedef struct AdressBuch {
2         A_SEQUENCE_OF(struct Kontakt) list;
3 } AdressBuch_t;
```

Listing 3.3: Datei: Kontakt.h float

```
1  typedef struct Kontakt {
2          BMPString_t      name;
3          BMPString_t      vorname;
4          BMPString_t      strasse;
5          INTEGER_t        hausnummer;
6          INTEGER_t        plz;
7          BMPString_t      stadt;
8          struct stichworte {
9                  A_SEQUENCE_OF(BMPString_t) list;
10         } *stichworte;
11 } Kontakt_t;
```

Neben den selbst definierten Typen `AdressBuch` und `Kontakt` erstellt der ANS.1-Compiler noch den Quellcode für den Encoder, den Decoder und die verwendeten ASN.1-Datentypen. Zusätzlich kann es noch Convenience-Funktionen für den Umgang mit ASN.1-Typen geben, wie Konvertierungsroutinen, die z.B. einen Standard-String der Programmiersprache (C++: `std::string`, Java: `java.lang.String`, C: `char*`) in einen BMPString und zurück umwandeln.

3.3 Grundzüge der Kodierung

Jeder mit ASN.1 spezifizierte Datentyp muss im allgemeinen als Oktettfolge der Form

Bezeichnerfeld	Längenfeld	Inhaltsfeld	[EOC-Oktette]

kodiert werden. Auf die EOC-Oktette wird im Abschnitt 3.3.2 noch genauer eingegangen. Die genaue Form der Kodierung beschreiben sogenannte *Encoding Rules*, auf die im Abschnitt 3.4 auf Seite 40 genauer eingegangen wird.

3.3.1 Das Bezeichnerfeld

Jeder ASN.1-Datentyp hat eine eindeutige *Tag*-Kennung (Markierung), welche im Bezeichnerfeld gespeichert wird und deren Länge ein oder mehrere Oktette beträgt. Darin kodiert sind die Tag-Klasse, ob es sich um einen einfachen oder zusammengesetzten Datentyp handelt und die eigentliche Tag-Nummer. Der `INTEGER`-Typ ist beispielsweise ein einfacher Typ der Klasse `UNIVERSAL` mit der Tag-Nummer 2. Die Tag-Klasse gibt den Bereich an, in dem die Tag-Nummer eindeutig sein muss. Zwei Markierungen werden als verschieden angesehen, wenn sie verschiedene Klassen angehören, oder wenn sie der gleichen Klasse angehören, aber unterschiedliche Nummern haben.

Es gibt vier verschiedene Tag-Klassen:

1. `UNIVERSAL`
2. *kontextspezifisch*
3. `APPLICATION`
4. `PRIVATE`

Diese haben folgende Anwendungsbereiche:

- `UNIVERSAL`: Diese Klasse ist für Typen des ASN.1-Standards reserviert und darf nicht für eigene Entwicklungen genutzt werden.

  Beispiel:

  ```
  BOOLEAN   ⟶  [UNIVERSAL 1]
  INTEGER   ⟶  [UNIVERSAL 2]
  SEQUENCE  ⟶  [UNIVERSAL 16]
  ```

- *kontextspezifisch*: Wird kein Klassenname angegeben, so gelten die Tags nur im aktuellen Kontext.

  Beispiel:

```
1 Attr ::= SEQUENCE {
2   id          INTEGER OPTIONAL,
3   name   [0]  INTEGER,
4   kind        INTEGER OPTIONAL,
5 }
```

  Ein `Attr` ist eine Folge, beginnend mit einer optionalen `INTEGER`-ID, einem Namen, der ebenfalls durch einen `INTEGER` angegeben wird, dann kommt eine optionale `INTEGER` Typangabe. Das ursprüngliche Tag `[UNIVERSAL 2]` des

name-Attributs wird im Kontext von Attr vom kontextspezifischen Tag [0] überschrieben.

Dekodiert man eine Instanz des Attr-Types, der mit zwei INTEGER-Werten beginnt, so könnte man ohne das kontextspezifische Tag [0] nicht wissen, um welche beiden Attribute es sich handelt (id und name oder name und kind), da alle vom gleichen Typ und mit [UNIVERSAL 2] markiert sind. Durch das kontextspezifische Tag [0] ist immer klar, um welches Attribut es sich handelt. Beim Dekodieren einer Attr-Instanz beginnend mit zwei INTEGER-Werten erhält man entweder die Tag-Folge [UNIVERSAL 2],[0] oder die Folge [0],[UNIVERSAL 2]. Bei der ersten Folge ist klar, dass das optionale kind-Attribut nicht angegeben wurde, bei der zweiten Folge ist klar, dass die id nicht angegeben wurde.

- APPLICATION: Bei Einführung der Klasse APPLICATION sollten Datentypen, die in einer Anwendung "weit verbreitet" sind, "an vielen Stellen" benutzt werden und im Sinne der abstrakten Syntax unterscheidbar sein müssen, mit Tags dieser Klasse markiert werden. Da Tags der Klasse APPLICATION per Definition verschieden von allen anderen Tags sind, wird hier im Gegensatz zu den kontextspezifischen Tags nicht das Tag des eigentlichen Typs überschrieben.

  Beispiel:

```
1 MyType ::= [APPLICATION 3] SEQUENCE {
2   version   INTEGER,
3   others    AnotherType
4 }
```

  MyType hat somit die Tags [APPLICATION 3][UNIVERSAL 16]. Ohne explizite Angabe besäße MyType nur das Tag von SEQUENCE, also [UNIVERSAL 16].

  Wegen einiger Ungenauigkeiten im Standard - z.B. ist nicht genau spezifiziert, in welchem Bereich APPLICATION-Tags eindeutig sein müssen und wie die Eindeutigkeit sichergestellt werden kann - wird das Markieren eigener Typen mit Tags dieser Klasse seit 1994 nicht mehr empfohlen.

- PRIVATE: Die Klasse PRIVATE wurde für Typen, die nur innerhalb einer Organisation oder eines Landes Anwendung finden und im Sinne ihrer abstrakten Syntax unterscheidbar sein müssen, eingeführt.

  Beispiel:

```
1 Animal ::= [PRIVATE 1] SEQUENCE {
2   id      INTEGER,
3   speed   INTEGER OPTIONAL,
4   name    BMPString
5 }
```

  Seit 1994 werden Markierungen dieser Klasse nicht mehr empfohlen.

Zusammenfassend kann also gesagt werden, dass man zum Markieren von Typen eigener Spezifikationen ausschließlich kontextspezifische Tags verwenden sollte.

Eine andere und heute ausdrücklich empfohlene Möglichkeit ist, ganz auf eigene Markierungen zu verzichten und einen der globalen Markierungsmodi zu benutzen. Diese werden im Abschnitt 3.3.1.2 auf Seite 38 erläutert.

3.3.1.1 Markierungs-Modi

Man kann zwei Markierungs-Modi unterscheiden: `IMPLICIT` und `EXPLICIT`. Mit ihnen kann man bestimmen, welche Markierungen kodiert werden und welche nicht.

- `EXPLICIT`: Dieser Modus ist standardmäßig aktiviert. Bei der Kodierung einer Instanz von

```
1 Afters ::= CHOICE {
2   -- die EXPLICIT-Marker könnten auch weggelassenen
3   -- werden
4             cheese    [0] EXPLICIT IA5String,
5             dessert   [1] EXPLICIT IA5String
6 }
```

  hat diese die Tags `[0][UNIVERSAL 22]` bzw. `[1][UNIVERSAL 22]`, je nachdem welche Alternative gewählt wurde (Informationen zum `CHOICE`-Typ sind im Abschnitt 3.5 auf Seite 43 zu finden.). Man sieht, dass die Markierung des `IA5String` unnötigerweise ebenfalls kodiert wurde, obwohl die kontextspezifischen Tags schon ausgereicht hätten, um Eindeutigkeit herzustellen.

- `IMPLICIT`: Typen, die implizit markiert wurden, überschreiben rekursiv das folgende Tag bis eine Markierung der Klasse `UNIVERSAL` (wird ebenfalls noch überschrieben) oder ein Auftreten des `EXPLICIT`-Markers erreicht wird.

```
1 T  ::= [1] IMPLICIT T1
2 T1 ::= [5] IMPLICIT T2
3 T2 ::= [APPLICATION 0] IMPLICIT INTEGER
```

  Für eine Instanz von `T` würde hier nur das kontextspezifische Tag `[1]` kodiert. `[1]` überschreibt `[5]`, welches wiederum `[APPLICATION 0]` überschreibt, das letztendlich das `[UNIVERSAL 2]`-Tag des `INTEGER`-Typs überschreibt.

  Der `IMPLICIT`-Marker kann allerdings nicht überall angewendet werden:

```
1 Afters ::= [0] IMPLICIT CHOICE {
2             cheese  [1] IA5String,
3             dessert [2] IA5String
4 }
```

  Dieses Konstrukt würde bei der Dekodierung Nichtdeterminismus erzeugen, denn `[0]` überschreibt die jeweiligen Tags der Alternativen. Der Dekoder kann nun nicht mehr zwischen `cheese` und `dessert` unterscheiden.

  Ein im `IMPLICIT`-Modus kodierter Typ kann nur dekodiert werden, wenn der Decoder ebenfalls die abstrakte Syntax kennt. Das ist immer der Fall, wenn sowohl Encoder als auch Decoder vom selben ASN.1-Modul generiert wurden.

Da es bei langen ASN.1-Spezifikationen sehr mühsam wäre, jedes Tag selbst zu definieren, wurden die sogenannten globalen Markierungs-Modi eingeführt, auf die im folgenden Abschnitt kurz eingegangen wird.

3.3.1.2 Globale Markierungs-Modi

Mit einem globalen Markierungsmodus kann modulweit bestimmt werden, wie das Tagging anzuwenden ist. Es gibt die folgenden drei Modi:

1. `EXPLICIT TAGS`
2. `IMPLICIT TAGS`
3. `AUTOMATIC TAGS`

Die ersten beiden Modi sind selbsterklärend. Ist ein Modul mit `IMPLICIT TAGS` gekennzeichnet, so werden alle Komponenten der darin auftretenden `SEQUENCE`-, `SET`- und `CHOICE`-Typen implizit markiert, mit Ausnahme der Komponenten, denen ein `EXPLICIT`-Marker oder ein `CHOICE`-Typ folgt.

Im Modus `EXPLICIT TAGS` werden alle Komponenten explizit markiert, es sei denn, sie sind in der Spezifikation mit `IMPLICIT` versehen.

Beide bisher betrachteten Modi machen nur Sinn, wenn auch wirklich Tags in der Spezifikation angegeben wurden.

Im `AUTOMATIC TAGS`-Modus werden alle Komponenten von `SEQUENCE`-, `SET`- und `CHOICE`-Typen **automatisch** vom ASN.1-Compiler markiert. Dieser beginnt mit `[0]` und erhöht den Wert dann schrittweise. Die Markierungen erfolgen standardmäßig im impliziten Modus, mit den oben schon genannten Ausnahmen. Der `AUTOMATIC TAGS`-Modus ist genauestens im ASN.1-Standard spezifiziert. Dadurch ist das Resultat nicht compilerabhängig.

Aus diesem Grund wird der `AUTOMATIC TAGS`-Modus für alle neuen Spezifikationen empfohlen und wurde auch für BinaryGXL verwendet.

3.3.2 Das Längenfeld

Das Längenfeld gibt die Länge des Inhaltsfeldes an. Es sind zwei unterschiedliche Ausprägungen definiert:

1. **bestimmte (*definite*) Form**: Gibt die Länge des Inhaltsfeldes in Oktetten an. Hier muss das Längenfeld natürlich auch eine variable Länge haben, so dass alle beliebigen Inhaltsfeldlängen angegeben werden können.
2. **unbestimmte (*indefinite*) Form**: Ein spezielles Oktett signalisiert, dass das Inhaltsfeld mit EOC-Oktetten (*End of Content*) abgeschlossen wird (2 Oktette mit jeweils nur Nullen). Das ermöglicht die Kodierung von Typen, deren Länge zu Beginn der Kodierung noch unbekannt ist.

3.3.3 Das Inhaltsfeld

Im Inhaltsfeld ist der eigentliche Inhalt kodiert. Der ASN.1-Standard definiert genau, wie Instanzen eines bestimmten Typs zu kodieren sind, wobei es je nach benutzten *Encoding Rules* Unterschiede geben kann. Auf die einzelnen Kodierungen der ASN.1-Typen wird allerdings nicht detaillierter eingegangen, da sie für das Thema dieser Studienarbeit unerheblich sind.

3.4 Die verschiedenen Encoding Rules

Der ASN.1-Standard definiert eine Reihe von sogenannten *Encoding Rules*, welche beschreiben, wie die verschiedenen ASN.1-Datentypen kodiert werden müssen. Dafür werden der abstrakten Syntax genaue Kodierungs- und Dekodierungsregeln zugeordnet. Bislang sind folgende *Encoding Rules* von der ISO bzw. der ITU normiert:

1. *Basic Encoding Rules (BER)*
2. *Canonical Encoding Rules (CER)*
3. *Distinguished Encoding Rules (DER)*
4. *Packed Encoding Rules (PER)*
5. *XML Encoding Rules (XER)*

Im Folgenden werde alle Encoding Rules kurz erläutert.

1. **Basic Encoding Rules**
   Die Basic Encoding Rules waren die ersten Kodierungsregeln von ASN.1. Sie lassen zuweilen einige Freiheiten bei der Kodierung, so darf z.B. das Längenfeld in einer bestimmten oder einer unbestimmten Form kodiert werden.

2. **Canonical Encoding Rules**
   Bei den Canonical Encoding Rules sind alle alternativen Kodierungsformen beseitigt worden. Damit ist die Kodierung bei Verwendung dieser Regeln injektiv, d.h. es gibt zu jedem ASN.1-Typ genau eine Möglichkeit der Kodierung.

3. **Distinguished Encoding Rules**
   Die Distinguished Encoding Rules sind genau wie die Canonical Encoding Rules eine injektive Kodierung. Eine genaue Unterscheidung von CER und DER würde den Rahmen dieser Studienarbeit sprengen, denn dazu müsste man sich wesentlich detaillierter mit der Kodierung beschäftigen. Sie ist aber bei [Dubu00], ab Seite 420 zu finden.

   BinaryGXL benutzt die Distinguished Encoding Rules. Damit ist sichergestellt, dass das Kodieren einer inhaltsgleichen GXL-Datei auf verschiedenen Systemen immer in identischen BinaryGXL-Dateien resultiert.

4. **Packed Encoding Rules**
   Die Packed Encoding Rules werden dort verwendet, wo es besonders auf die Kompaktheit der Daten ankommt. Die Werte werden hier immer auf die kürzeste Weise kodiert. Bei den anderen drei Encoding Rules wird jeder Datentyp mit einem Tripel (Bezeichner, Länge, Inhalt) kodiert. Bei PER sind alle drei Elemente des Tripels optional und werden nur kodiert, wenn es nötig ist. Wenn man in der ASN.1-Spezifikation zu allen Typen noch möglichst restriktive Contraints angibt, so können laut [Dubu00] S. 425, bis zu 60% Größenersparnis erzielt werden.

   In dieser Hinsicht wären die Packed Encoding Rules eigentlich noch besser als DER für BinaryGXL geeignet, jedoch werden sie vom für diese Studienarbeit verwendeten ASN.1-Compiler (siehe Anhang A.2 auf Seite 76) nicht unterstützt.

   Die Unterstützung ist aber in Planung. Eventuell kann eine zukünftige BinaryGXL-Version dann die Packed Encoding Rules verwenden, wobei dann natürlich keine Abwärtskompatibilität mehr gegeben wäre.

5. **XML Encoding Rules**
   Die XML Encoding Rules standardisieren Regeln zur Überführung von in ASN.1 spezifizierten Daten in XML und zurück.

   Beispiel:

   Eine Instanz des Typs

```
1 Dictionary ::= SEQUENCE {
2   germanwords   WordList,
3   englishwords  WordList
4 }
5 WordList ::= SEQUENCE OF BMPString
```

   wird beispielsweise folgendermaßen XER-kodiert:

```
1  <Dictionary>
2    <germanwords>
3      <BMPString>...</BMPString>
4      <BMPString>...</BMPString>
5      <BMPString>...</BMPString>
6    </germanwords>
7    <englishwords>
8      <BMPString>...</BMPString>
9      <BMPString>...</BMPString>
10     <BMPString>...</BMPString>
11   </englishwords>
12 </Dictionary>
```

In vielen Beispielen wurden bislang schon ASN.1-Datentypen verwendet. Jetzt endlich folgt die Erklärung zu allen in BinaryGXL benutzten Typen.

3.5 ASN.1-Typen in BinaryGXL

In diesem Abschnitt wird kurz auf alle in der BinaryGXL-Spezifikation benutzten ASN.1-Typen eingegangen. Die meisten Typen erlauben Erweiterungen, verschiedene Schreibweisen und Constraints, die jedoch nur dann erläutert werden, wenn sie in der BinaryGXL-Spezifikation vorkommen. Genauere Beschreibungen und konkrete Angaben zur Kodierung der einzelnen Typen sind in [Gora98], [Larm99], [Dubu00], dem ASN.1-Standard und den Standards der zugehörigen Encoding Rules zu finden.

- `BOOLEAN`:
  - Klasse/Tag: `UNIVERSAL 1`
  - Der `BOOLEAN`-Typ ist der einfachste ASN.1-Typ. Er kann nur die Werte `TRUE` oder `FALSE` annehmen.
- `INTEGER`:
  - Klasse/Tag: `UNIVERSAL 2`
  - Der `INTEGER`-Typ kann vorzeichenbehaftete Ganzzahlen praktisch beliebiger Länge annehmen. Theoretisch kann allerdings ein Maximal- bzw. Minimalwert berechnet werden, der aus Restriktionen der *Basic Encoding Rules* resultiert. Wenn man eine Übertragungsgeschwindigkeit von einem Terabit

pro Sekunde annimmt, so würde es ungefähr 100 Millionen Jahre dauern, diesen Maximalwert zu übertragen (vgl. [Larm99], Seite 81).

- `ENUMERATED`:
  - Klasse/Tag: `UNIVERSAL 10`
  - Der `ENUMERATED`-Typ implementiert einen Aufzählungstyp ähnlich wie die aus C/C++ bekannten `enum`-Typen. Zu jedem Element eines `ENUMERATED`-Typs kann explizit eine Nummer vergeben werden. Wird nichts angegeben, so hat das erste Element die Nummer 0, und folgende Elemente werden aufsteigend nummeriert.

Listing 3.4: Beispiel

```
1 ConnectionStatus ::= ENUMERATED {
2     connected,       -- 1 -> open
3     closed,          -- 2 -> closed
4     connecting,      -- 3 -> connecting
5     terminating(0)   -- 0 -> terminating
6 }
```

Im Beispiel aus Listing 3.4 wurde die Null explizit für das letzte Element vergeben (Zeile 5). Bereits vergebene Nummern werden bei der automatischen Durchnummerierung einfach übergangen, so dass sich die im Beispiel angegebene Nummerierung ergibt.

- `REAL`:
  - Klasse/Tag: `UNIVERSAL 9`
  - Der `REAL`-Typ implementiert Dezimalzahlen beliebiger aber endlicher Länge. Er wird durch ein Tripel $(Mantisse, Basis, Exponent)$ dargestellt: Durch $(5, 2, 0)$ und $(5, 10, 0)$ wird jeweils die Dezimalzahl fünf repräsentiert, jedoch sind die Darstellungen nicht semantisch äquivalent! Beispielsweise ist 0.2_{10} nicht auf endliche Weise als Binärzahl darstellbar, sondern es ergibt sich $0.\overline{0011}_2$. Deshalb kann ein und dieselbe Dezimalzahl auf unterschiedliche Weise kodiert werden, sogar bei Verwendung kanonischer *Encoding Rules* (DER/CER).

    Es gibt genau zwei besondere `REAL`-Werte: `PLUS-INFINITY` ($+\infty$) und `MINUS-INFINITY` ($-\infty$).

    Die ASN.1-Kodierung für den `REAL`-Typ basiert nicht auf dem *Standard for Binary Floating-Point Arithmetic* ([IEEE 754]), jedoch besagt der ASN.1-Standard, dass die Darstellung so gewählt wurde, dass sie leicht in andere Formate und zurück konvertiert werden kann.

- `IA5String`:
  - Klasse/Tag: `UNIVERSAL 22`
  - Eine Zeichenkette des Typs `IA5String` kann beliebig viele Zeichen der Menge $LateinischeBuchstaben \cup ControlCharacters \cup Space \cup Del$ enthalten. Eine Tabelle mit allen Zeichen ist bei [Dubu00], Seite 178 zu finden.

IA5 steht hierbei für “International Alphabet number 5” (= ASCII), welche eine 7-Bit-Kodierung der ISO und ITU-T[3] ist ([IA5]).

- `BMPString`:
  - Klasse/Tag: `UNIVERSAL 30`
  - Um alle möglichen Zeichen des Unicode-Standards ([UCS]) kodieren zu können, werden vier Byte benötigt. Dazu kann man den ASN.1-Typ `UniversalString` verwenden. Tatsächlich sind heute aber nur die ersten 65536 Zellen belegt (die sogenannte *Basic Multilingual Plane*). Zu ihrer Kodierung werden folglich nur 2 Byte benötigt. Diese Art der Kodierung wird *UCS-2* genannt und vom `BMPString`-Typ implementiert.
- `SEQUENCE`:
  - Klasse/Tag: `UNIVERSAL 16`
  - Mit dem ASN.1-`SEQUENCE`-Typ kann man zusammengesetzte Typen definieren. Eine `SEQUENCE` ist eine geordnete Liste und entspricht dem aus C/C++ bekannten `struct`. Jede `SEQUENCE` enthält eine beliebige aber feste Anzahl an Unterelementen, jeweils durch Bezeichner und Typ gegeben.

Listing 3.5: Beispiel

```
1 CompositeType ::= SEQUENCE {
2     identifier1   Type1,
3     identifier2   Type2
4 }
```

- `SEQUENCE OF`:
  - Klasse/Tag: `UNIVERSAL 16`
  - Eine `SEQUENCE OF` ist eine geordnete Liste von beliebig vielen Elementen des selben Typs.

Listing 3.6: Beispiel

```
1 List        ::= SEQUENCE OF Item
2 ListOfLists ::= SEQUENCE OF List
```

`SEQUENCE` und `SEQUENCE OF` haben aus historischen Gründen das gleiche Tag. Dies wirkt sich nachteilig auf die Kodierung aus, denn ein `SEQUENCE OF`-Element wird genau wie ein `SEQUENCE`-Element kodiert, was insbesondere heißt, dass zu jedem Element eines `SEQUENCE OF`-Containers das jeweilige Tag gespeichert wird. Da alle Elemente den gleichen Typ haben, ist das natürlich hochgradig redundant.

- `CHOICE`:
  - Klasse/Tag: hat keinen eigenen Tag, sondern immer den der gewählten Alternative

[3] International Telecommunication Union – Telecommunication Standardization Sector

– Der `CHOICE`-Typ wird zur Implementierung von Alternativen benutzt. Er ist äquivalent zu dem aus C/C++ bekannten Typ `union`.

Listing 3.7: Beispiel

```
1 Creature ::= CHOICE {
2     plant       Plant,
3     animal      Animal
4 }
```

Ein Lebewesen ist also entweder eine Pflanze oder ein Tier.

Bislang wurde in dieser Arbeit noch nichts Konkretes zu BinaryGXL gesagt, sondern nur die Voraussetzungen für ein Verständnis der Spezifikation erarbeitet. Im folgenden Kapitel geht es nun konkret um BinaryGXL, und alle bisher vorgestellten Konzepte finden ihre Anwendung.

Kapitel 4

Binäre Kodierung von GXL

4.1 Anforderungen an BinaryGXL

Das zu entwickelnde Kommandozeilentool soll als Parameter eine GXL-Datei erhalten und diese binär kodieren. Als Dateiendung für BinaryGXL-Dateien wird `.bgxl` empfohlen, jedoch wird jede andere Endung auch akzeptiert. Übergibt man dem Tool eine BinaryGXL-Datei, so soll diese wieder in die GXL-Textform umgewandelt werden.

Die Anforderungen an BinaryGXL sind:

1. Bei der Konvertierung von GXL zu BinaryGXL dürfen keine Informationen verloren gehen, die inhaltliche Bedeutung haben. Im Idealfall ist eine in BinaryGXL und wieder zurück kodierte GXL-Datei absolut inhaltsgleich mit der Ursprungs-GXL-Datei. Zumindest muss jedoch eine semantische Äquivalenz gewährleistet werden.
2. Die Dateigröße der BinaryGXL-Dateien soll deutlich unter jener der GXL-Datei liegen. Eine Verkleinerung um mindestens den Faktor 10 wird angestrebt.
3. Das Kodieren und Dekodieren soll in einer akzeptablen Zeit geschehen. Als akzeptabel ist hier eine Zeit in der Größenordnung der Pack-/Entpackzeit der gängigen Komprimierungstools anzusehen.

4.2 Einschränkungen von BinaryGXL

1. GXL unterstützt für die Elemente `gxl`, `graph`, `node`, `edge`, `rel`, `val` und `relend` sogenannte Extensions. Damit kann ein Element zusätzlich zu den in der GXL-DTD [2] angegeben Kindelementen weitere, selbst definierte Kindelemente enthalten. Mit sogenannten Attribute-Extensions können auf die gleiche Weise eigene Attribute zu oben genannten Elementen verwendet werden. Diese Extensions werden von BinaryGXL nicht unterstützt.
2. Kommentare in GXL-Dateien werden nicht in BinaryGXL gespeichert.
3. Eine weitere Einschränkung ist, dass BinaryGXL einen non-validating Parser zum Parsen der GXL-Dateien verwendet. So ist es möglich, ein nicht valides GXL-Dokument erfolgreich in BinaryGXL und zurück zu konvertieren.

- Laut Kommentar in der GXL-DTD darf z.B. der Container-Typ `seq` nur Werte *eines* Typs enthalten, wohingegen der BinaryGXL-Konverter auch Konstrukte wie

```
<seq>
  <float>3.1</float>
  <bool>true</bool>
</seq>
```

  akzeptiert und kodiert bzw. dekodiert.

- GXL verlangt außerdem die Vergabe von eindeutigen IDs innerhalb eines Dokuments und verbietet das Referenzieren von Graphelementen eines anderen, außerhalb des aktuellen Graphen liegenden Graphs. Auch das wird vom BinaryGXL-Konverter nicht überprüft.

- Ist der `edgemode` eines GXL-Graphen auf `directed` oder `undirected` gesetzt, so dürfen darin enthaltene Kanten und Relationen ihn nicht überschreiben. Überschreibt eine Kante trotzdem den ererbten Wert, so ist das GXL-Dokument ungültig. Auch hier würde der BinaryGXL-Konverter ohne eine Warnung das nicht valide GXL kodieren und wieder dekodieren.

Möchte man ein GXL-Dokument als BinaryGXL weitergeben, so sollte man also vor der Kodierung die Validität des GXL-Dokuments mit dem GXL-Validator [4] überprüfen.

Zusammenfassend kann gesagt werden, dass das Konvertierungstool ein XML-Dokument, das der GXL-DTD [2] genügt, “as-is” konvertiert. Alle Einschränkungen, die nur im GXL-Schema [3] oder den Kommentaren der DTD eingeführt werden, finden keine Beachtung.

4.3 Die ASN.1-Spezifikation zu BinaryGXL

Um die ASN.1-Spezifikation zu entwickeln, wurde die GXL-DTD [2] mit den ASN.1-Typen nachgebildet.

4.3.1 Allgemeine Vorgehensweise

Bei der Erstellung der ASN.1-Spezifikation wurden fast alle GXL-Elemente in gleicher Weise auf die ASN.1-Typen abgebildet. Da es allerdings auch Ausnahmen von der Regel und andersartige Designentscheidungen gibt, werden im Abschnitt 4.3.2 alle GXL-Elemente und ihre ASN.1-Gegenstücke einzeln besprochen.

4.3.1.1 Primitive Typen

In diesem Abschnitt wird darauf eingegangen, wie primitive GXL-Typen (Attribute und nicht-zusammengesetzte, attributlose Elemente) auf die ASN.1-Typen abgebildet werden.

GXL-Attribute: XML besitzt keine wirklichen Datentypen. Alle Attributwerte eines Elements sind als Zeichenketten, eingeschlossen durch doppelte Anführungsstriche, gegeben. Aus der kommentierten GXL-DTD lässt sich jedoch ein "semantisch" passender Typ herauslesen.

Kann ein Attribut nur die Werte `true` oder `false` annehmen, so hat das korrespondierende ASN.1-Attribut den Typ `BOOLEAN`.

Kann ein Attribut nur ganzzahlige Werte annehmen, so hat das entsprechende ASN.1-Attribut den Typ `INTEGER`.

Manche Attribute können einen von endlich vielen alternativen Werten annehmen, weshalb das korrespondierende ASN.1-Attribut vom Typ `ENUMERATED` ist.

Attribute, die beliebige Zeichenketten enthalten können, werden auf ASN.1-Attribute vom Typ `INTEGER` abgebildet. Dieser `INTEGER` ist eine Referenz (im Folgenden auch String-Referenz genannt) auf den entsprechenden String. Die Gründe und Details dieser Vorgehensweise sind im Abschnitt 4.3.3 auf Seite 57 beschrieben.

Nicht-zusammengesetzte, attributlose GXL-Elemente: Alle primitiven GXL-Elemente haben passende ASN.1-Typen. Dementsprechend werden die GXL-Elemente `bool`, `real` und `int` mit den ASN.1-Typen `BOOLEAN`, `REAL` und `INTEGER` implementiert.

Analog zu Attributen, die beliebige Zeichenketten enthalten dürfen, wird das GXL-`string`-Element mit einer String-Referenz vom ASN.1-`INTEGER`-Typ implementiert.

4.3.1.2 Komplexe GXL-Elemente

Ein GXL-Element wird hier als komplex betrachtet, wenn es zusammengesetzt ist, also Unterelemente enthalten darf, oder wenn es attributiert ist.

Im Allgemeinen wird jedes solche GXL-Element auf ein eigenes ASN.1-Element vom Typ `SEQUENCE` abgebildet. Diese `SEQUENCE` enthält zuerst die entsprechenden ASN.1-Elemente der GXL-Attribute, gefolgt von denen der GXL-Unterelemente.

4.3.1.3 Optionale ASN.1-Elemente

Viele der nun folgenden ASN.1-Elemente haben optionale Unterelemente. Diese werden in der ASN.1-Spezifikation durch das Schlüsselwort `OPTIONAL` gekennzeichnet. Da eine Anforderung an BinaryGXL ist, dass die hin und zurück kodierte GXL-Datei möglichst identisch mit der ursprünglichen GXL-Datei ist, werden diese optionalen Elemente genau dann gesetzt und gespeichert, wenn sie auch in der zu kodierenden GXL-Datei vorkommen. Ist ein optionales GXL-Element nicht im GXL-Dokument angegeben, so wird das entsprechende ASN.1-Attribut ein Null-Pointer sein.

4.3.2 Die GXL-Elemente und ihre BinaryGXL-Gegenstücke

Im Folgenden werden nun alle Elemente der GXL-DTD [2] in der Reihenfolge ihrer Beschreibung in der DTD aufgeführt und ihre ANS.1-Repräsentationen vorgestellt und erläutert.

4.3.2.1 Das `gxl`-Element

Listing 4.1: GXL-Element `gxl`

```
<!ELEMENT gxl (graph* %gxl-extension;) >
<!ATTLIST gxl
    xmlns:xlink CDATA \
        #FIXED "http://www.w3.org/1999/xlink"
    %gxl-attr-extension;
>
```

Das `gxl`-Element ist GXLs Top-Level-Element. Jedes GXL-Dokument besitzt genau ein `gxl`-Element, welches eine beliebige Anzahl von Graphen (`graph`) beinhalten kann. Damit ist es möglich sowohl Schema als auch Instanz, alle jeweils als Graph repräsentiert, in einem einzigen GXL-Dokument anzugeben.

Mittels XLink[1] kann GXL externe Objekte referenzieren. Dazu muss, wie oben angegeben, dessen Namensraum deklariert werden.

Listing 4.2: ASN.1-Repräsentation von `gxl`

```
Gxl ::= SEQUENCE {
  version       INTEGER, -- string reference
  graphs        Graphs
}

Graphs ::= SEQUENCE OF Graph
```

Das `gxl`-Element hat nur ein Attribut: `xmlns:xlink`. Dieses hat als Wert immer den String `"http://www.w3.org/1999/xlink"`, weshalb es bei BinaryGXL keine Beachtung findet, und nicht gespeichert wird.

Das `Gxl`-Element der ASN.1-Repräsentation hat ein zusätzliches Unterelement `version`, welches die GXL-Version der kodierten GXL-Datei angibt. Diese wird als String in einer der in Abschnitt 4.3.3 auf Seite 57 angegebenen `StringLists` gespeichert, und der hier angegebene `INTEGER` dient als Referenz darauf. Zur Zeit gibt es ausschließlich die Version 1.0 weshalb `version` immer auf `"GXL-1.0"` gesetzt wird.

Die beliebig vielen Graphen werden in einer Liste von Graphen gespeichert.

4.3.2.2 Das `type`-Element

Das `type`-Element referenziert mittels XLink eine interne oder externe Deklaration. Ist beispielsweise ein Schema-Element an einem gegebenen URI deklariert, so kann in einem GXL-Dokument auf diese Deklaration verwiesen werden.

[1] `http://www.w3.org/1999/xlink`, 05.07.2006

Listing 4.3: GXL-Element `type`

```
<!ELEMENT type EMPTY>
<!ATTLIST type
    xlink:type  (simple)    #FIXED  "simple"
    xlink:href  CDATA       #REQUIRED
>
```

Das GXL-Element `type` hat keine eigenständige ASN.1-Repräsentation. Typisierte Elemente speichern bei BinaryGXL eine String-Referenz auf den `xlink:href`-Identifier. Da `xlink:type` immer `"simple"` ist, wird es nicht gespeichert.

4.3.2.3 Das `graph`-Element

Listing 4.4: GXL-Element `graph`

```
<!ELEMENT graph (type?, attr*, (node | edge | rel)*
                 %graph-extension;) >
<!ATTLIST graph
    id          ID       #REQUIRED
    role        NMTOKEN  #IMPLIED
    edgeids     ( true | false ) "false"
    hypergraph  ( true | false ) "false"
    edgemode    ( directed | undirected |
                  defaultdirected |
                  defaultundirected) "directed"
    %graph-attr-extension;
>
```

Ein GXL-Graph wird immer von einem öffnenden und einem schließenden `graph`-Tag umschlossen.

Mittels dem optionalen Unterelement `type` kann ein Graph zu seinem Schema verweisen.

Zusätzlich besitzt ein Graph eine beliebige Anzahl von Attributen und Graphelementen (`node`, `edge` und `rel`).

Er ist gekennzeichnet durch eine eindeutige ID, welche immer anzugeben ist.

Wenn die im Graph enthaltenen Kanten (`rel`) und Relationen (`rel`) eigene IDs besitzen, so muss `edgeids` auf `true` gesetzt sein.

Ist `hypergraph` auf `true` gesetzt, so darf ein Graph sogenannte Hyperkanten besitzen, welche eine beliebige Anzahl von Elementen verbinden. Ansonsten verbinden Kanten immer genau zwei Elemente.

Das Attribut `edgemode` gibt an wie die (Hyper-)Kanten im jeweiligen Graphen interpretiert werden. Im Falle von `directed/undirected` können einzelne Kanten und Relationen dieses Graphs den hier angegebenen Defaultwert nicht überschreiben, im Falle von `defaultdirected/defaultundirected` ist dies möglich.

In der ASN.1-Repräsentation wird die eindeutige ID (`id`) als String-Referenz gespeichert.

Listing 4.5: ASN.1-Repräsentation von `graph`

```
Graph ::= SEQUENCE {
  id            INTEGER,
  role          INTEGER OPTIONAL, -- string reference,
                                  -- 0-ptr -> not defined
  edgeids       BOOLEAN OPTIONAL, -- 0-ptr -> not defined
  hypergraph    BOOLEAN OPTIONAL, -- 0-ptr -> not defined
  edgemode      ENUMERATED {
                  directed(0),      -- 0 -> directed,
                  undirected,       -- 1 -> undirected,
                  defaultdirected,  -- 2 -> defaultdirected,
                  defaultundirected -- 3 -> defaultundirected
                } OPTIONAL,         -- 0-ptr -> not defined
  type          INTEGER OPTIONAL,   -- string reference,
                                    -- 0-ptr -> not defined
  attrs             Attributes,
  gelems            GraphElements
}

Attributes ::= SEQUENCE OF Attr
GraphElements ::= SEQUENCE OF GraphElement

GraphElement ::= CHOICE {
  node          Node,
  edge          Edge,
  rel           Rel
}
```

Auf die gleiche Weise wird mit den Strings `role` und `type` verfahren, sie sind jedoch optional.

Die Attribute `egdeids` und `hypergraph` sind analog der DTD als `BOOLEAN` implementiert. Beide sind optional.

Das optionale Unterelement `edgemode` des ASN.1-Graphs ist mit dem Aufzählungstyp `ENUMERATED` implementiert, allerdings ohne Default-Wert.

Analog zum GXL-`graph`-Element hat ein ASN.1-Graph eine Liste von beliebig vielen Attributen (`attrs`).

Da `node`, `egde` und `rel` beliebig oft und in beliebiger Reihenfolge als Kindelemente eines GXL-Graphs vorkommen dürfen, sind sie in einem ASN.1-Element `GraphElement` zusammengefasst. Ein solches `GraphElement` ist entweder `Node`, `Edge` oder `Rel`.

4.3.2.4 Das node-Element

Listing 4.6: GXL-Element `node`

```
<!ELEMENT node (type?,attr*,graph* %node-extension;)>
<!ATTLIST node
    id          ID       #REQUIRED
    %node-attr-extension;
>
```

Ein Knoten (`node`) kann per XLink auf seinen Typ verweisen und enthält eine beliebige Anzahl von Attributen und Subgraphen.

Er ist mit einer in diesem GXL-Dokument eindeutigen ID gekennzeichnet.

Listing 4.7: ASN.1-Repräsentation von `node`

```
Node ::= SEQUENCE {
  id              INTEGER,           -- string reference
  type            INTEGER OPTIONAL,  -- string reference,
                                     -- 0-ptr -> not defined
  attrs           Attributes,
  graphs          Graphs
}
```

Die ID wird als String-Referenz gespeichert. Der optionale `type` ist ebenfalls eine String-Referenz.

Zusätzlich hat ein `Node` entsprechend der DTD eine Liste (`SEQUENCE OF`) von Attributen und Subgraphen.

4.3.2.5 Das `edge`-Element

Listing 4.8: GXL-Element `edge`

```
<!ELEMENT edge (type?, attr*, graph*
                %edge-extension;) >
<!ATTLIST edge
    id              ID        #IMPLIED
    from            IDREF     #REQUIRED
    to              IDREF     #REQUIRED
    fromorder       CDATA     #IMPLIED
    toorder         CDATA     #IMPLIED
    isdirected      ( true | false ) #IMPLIED
    %edge-attr-extension;
>
```

Kanten können mit dem optionalen Element `type` per XLink auf ihren extern definierten Typ verweisen.

Wie Knoten beinhalten sie eine beliebige Anzahl an Attributen und Graphen.

Eine Kante verbindet immer zwei Graphelemente (Knoten, Kanten, Relationen), deren IDs in den Attributen `from` und `to` stehen.

Eine Kante hat eine eigene, in diesem GXL-Dokument eindeutige ID, falls der umgebenden Graph `edgeids="true"` gesetzt hat.

In GXL ist es möglich eine Ordnung auf den einlaufenden und auslaufenden Kanten zu definieren. Dazu dienen die Attribute `fromorder` und `toorder`, welche jeweils ganzzahlige Werte annehmen können.

Falls der umgebende Graph `edgemode="defaultdirected"` oder `edgemode="defaultundirected"` gesetzt hat, so kann mit dem Attribut `isdirected` der für den Graphen definierte `edgemode` überschrieben werden. So

kann es z.B. in einem ansonsten gerichteten Graphen einige wenige ungerichtete Kanten geben. Ist der `edgemode` des umgebenden Graphen allerdings auf `directed` oder `undirected` gesetzt, so ist ein Überschreiben dieses Defaultwerts nicht gestattet.

Listing 4.9: ASN.1-Repräsentation von `edge`

```
Edge ::= SEQUENCE {
  id                INTEGER OPTIONAL, -- string reference,
                                      -- 0-ptr -> not defined
  fromid            INTEGER,          -- string reference
  toid              INTEGER,          -- string reference
  fromorder         INTEGER OPTIONAL, -- 0-ptr -> not defined
  toorder           INTEGER OPTIONAL, -- 0-ptr -> not defined
  edgeisdirected    BOOLEAN OPTIONAL, -- 0-ptr -> not defined
  type              INTEGER OPTIONAL, -- string reference,
                                      -- 0-ptr -> not defined
  attrs             Attributes,
  graphs            Graphs
}
```

Bei Kanten muss keine ID angegeben werden, weshalb das Element `id` von `Edge` eine optionale String-Referenz ist. Das Gleiche gilt für `type`.

`fromorder` und `toorder` sind sinngemäß als ”normale” `INTEGER` implementiert. Normal heißt hier, dass es keine String-Referenzen, sondern wirkliche Ganzzahlen sind.

Das Attribut `isdirected` ist in gleicher Weise durch das Unterelement `edgeisdirected` umgesetzt worden.

Genau wie `Node` hat auch eine Kante eine Liste von beliebig vielen Attributen (`attrs`) und Subgraphen (`graphs`).

4.3.2.6 Das `rel`-Element

Listing 4.10: GXL-Element `rel`

```
<!ELEMENT rel (type? , attr*, graph*, relend* \
               %rel-extension;) >
<!ATTLIST rel
    id          ID            #IMPLIED
    isdirected    ( true | false ) #IMPLIED
    %rel-attr-extension;
>
```

Eine Relation ist eine Beziehung zwischen einer beliebigen Anzahl von Graphelementen und wird deshalb auch als Hyperkante (*Hyperedge*) bezeichnet.

Auch hier kann mit dem Unterelement `type` auf den extern definierten Typ verwiesen werden, und eine Relation kann ebenfalls eine beliebige Anzahl von Attributen und Subgraphen enthalten.

Zusätzlich gehören zu einer Relation aber noch ihre Enden (`relend`). Modelliert eine Relation eine Beziehung zwischen 5 Graphelementen, so beinhaltet sie also fünf `relend`-Unterelemente.

Mit den Attributen `id` und `isdirected` verhält es sich analog zu den gleichnamigen Attributen von `edge`.

Listing 4.11: ASN.1-Repräsentation von `rel`

```
Rel ::= SEQUENCE {
  id                INTEGER OPTIONAL, -- string reference,
                                      -- 0-ptr -> not defined
  relisdirected     BOOLEAN OPTIONAL, -- 0-ptr -> not defined
  type              INTEGER OPTIONAL, -- string reference,
                                      -- 0-ptr -> not defined
  attrs               Attributes,
  graphs              Graphs,
  relends             RelEnds
}

RelEnds ::= SEQUENCE OF RelEnd
```

Wieder ist `id` als optionale String-Referenz umgesetzt.

Das Attribut `isdirected` ist als optionaler `BOOLEAN` `relisdirected` implementiert.

Auch hier ist `type` wieder eine optionale String-Referenz.

Die Liste der Relationsenden ist wieder mit dem ASN.1-Container `SEQUENCE OF` implementiert.

4.3.2.7 Das `relend`-Element

Listing 4.12: GXL-Element `relend`

```
<!ELEMENT relend (attr* %relend-extension;) >
<!ATTLIST relend
    target      IDREF                   #REQUIRED
    role        NMTOKEN                 #IMPLIED
    direction   ( in | out | none)      #IMPLIED
    startorder  CDATA                   #IMPLIED
    endorder    CDATA                   #IMPLIED
    %relend-attr-extension;
>
```

Wie oben schon beschrieben stellt eine Relation (`rel`) eine n-äre Beziehung zwischen Graphelementen dar. Zu jedem an der Beziehung beteiligten Graphelement gibt es ein Relationsende (`relend`).

Ein `relend` kann wie die Graphelemente (`node`, `edge`, `rel`) eine beliebige Anzahl Attribute beinhalten, allerdings keine Subgraphen.

Das Attribut `target` gibt die ID des Graphelements an, an welchem sich das Relationsende befindet.

Das Attribut `role` wird verwendet, um einem an der n-ären Beziehung teilnehmenden Graphelement einen Rollennamen zu geben. Der GXL-Standard weist es als optionales Element aus, allerdings bewertet der GXL-Validator das Fehlen des Attributs als Fehler.

Relationsenden können genau wie Kanten gerichtet sein - dies gibt das optionale Attribut `direction` an. Ist es auf `"in"` gesetzt, so ist die Richtung Graphelement ⟶ Relation, bei `"out"` ist sie Graphelement ⟵ Relation, bei `"none"` handelt es sich um ein ungerichtetes Relationsende. Die Richtung wird ignoriert, wenn bei der umgebenden Relation `isdirected="false"` gesetzt ist bzw. der die umgebende Relation umgebende Graph `edgemode="undirected"` gesetzt hat.

Ähnlich wie bei den Kanten, kann man auch bei den Relationsenden eine Ordnung definieren. Dabei gibt die `startorder` die Ordnungsnummer auf Relationsseite an, `endorder` bestimmt die Reihenfolge auf der Seite des Graphelements.

Listing 4.13: ASN.1-Repräsentation von `relend`

```
RelEnd ::= SEQUENCE {
  target         INTEGER,           -- string reference
  role           INTEGER OPTIONAL,  -- string reference,
                                    -- 0-ptr -> not defined
  direction      ENUMERATED {
                   none,            --  0 -> none,
                   in,              --  1 -> in,
                   out              --  2 -> out
                 } OPTIONAL,        -- 0-ptr -> not defined
  startorder     INTEGER OPTIONAL,  -- 0-ptr -> not defined
  endorder       INTEGER OPTIONAL,  -- 0-ptr -> not defined
  attrs          Attributes
}
```

Das Unterelement `target` ist eine String-Referenz, genau wie das optionale `role`.

Das Attribut `direction` wird als ASN.1-`ENUMERATED` umgesetzt. Da es bei GXL nicht immer angegeben sein muss, ist es optional.

Die Unterelemente `startorder` und `endorder` sind echte Integers.

Die Liste der Attribute ist wieder eine `SEQUENCE OF Attr`.

4.3.2.8 Das `attr`-Element und Attributwerte

Ein Attribut kann wiederum eine beliebige Anzahl von Attributen haben.

Zusätzlich beinhaltet jedes Attribut genau einen Wert, welcher einen der oben angegebenen Typen haben muss. Auf die einzelnen Wertetypen wird als nächstes eingegangen.

Einem Attribut kann eine in diesem Dokument eindeutige ID zugewiesen werden.

Das Attribut `name` gibt den Namen des Attributes an, `kind` kann genutzt werden, um verschiedene Attributtypen unterscheiden zu können.

Die Unterelemente `id`, `name` und `kind` sind wieder Referenzen auf Strings.

Daneben hat ein Attribut eine optionale Liste von untergeordneten Attributen und genau einen Wert `value`. In der ASN.1-Repräsentation wird zwischen einfachen (`boolVal`, `realVal`, `intVal` und `stringVal` (String-Referenz)) und zusammengesetzten Wertetypen (`CustomGxlValueType`) unterschieden. Die einfachen Typen sind mit den korrespondierenden ASN.1-Typen implementiert, Strings werden als String-Referenz gespeichert.

Listing 4.14: GXL-Element `attr`

```
<!ELEMENT attr (attr*, (%val;)) >
<!ATTLIST attr
    id     ID    #IMPLIED
    name   NMTOKEN  #REQUIRED
    kind   NMTOKEN  #IMPLIED
>

<!ENTITY % val "
           locator |
           bool    |
           int     |
           float   |
           string  |
           enum    |
           seq     |
           set     |
           bag     |
           tup
           %value-extension;">

<!ELEMENT bool    (#PCDATA) >
<!ELEMENT int     (#PCDATA) >
<!ELEMENT float   (#PCDATA) >
<!ELEMENT string  (#PCDATA) >
```

Listing 4.15: ASN.1-Repräsentation von `attr`

```
Attr ::= SEQUENCE {
  id          INTEGER OPTIONAL, -- string reference,
                                -- 0-ptr -> not defined
  name        INTEGER,          -- string reference
  kind        INTEGER OPTIONAL, -- string reference,
                                -- 0-ptr -> not defined
  attrs       Attributes,
  value       Value
}

Value ::= CHOICE {
  boolVal    BOOLEAN,
  realVal    REAL,
  intVal     INTEGER,
  stringVal  INTEGER,
  customVal  CustomGxlValueType
}

CustomGxlValueType ::= CHOICE {
  locatorCVal INTEGER, -- string reference
  enumCVal    INTEGER, -- string reference
  tupCVal     Tup,
  bagCVal     Bag,
  seqCVal     Seq,
  setCVal     Set
}
```

Auf die zusammengesetzten Value-Typen wird im folgenden Teil näher eingegangen.

4.3.2.9 Das `locator`-Element

Listing 4.16: GXL-Element `locator`

```
<!ELEMENT locator  EMPTY >
<!ATTLIST locator
    xlink:type  (simple)    #FIXED  "simple"
    xlink:href  CDATA       #REQUIRED
>
```

Ein `locator` verweist per XLink auf ein anderes Dokument oder oder ein Element eines anderen Dokuments.

Laut der DTD besitzt ein `locator` keine Unterelemente und zwei Attribute, wobei `xlink:type` immer `"simple"` ist, und somit bei BinaryGXL nicht gespeichert wird.

Ein Locator muss also nur den URI des `xlink:href`-Attributs speichern. Er wird folglich, wie in Listing 4.15 gezeigt, als normale String-Referenz implementiert. Ein eigenes ASN.1-Element ist nicht vonnöten.

4.3.2.10 Das `enum`-Element

Listing 4.17: GXL-Element `enum`

```
<!ELEMENT enum    (#PCDATA) >
```

Der Wert eines `enum`-Elements wird als einfache Zeichenkette zwischen zwei `enum`-Tags geschrieben. Wie beim Locator ist hier kein eigenes ASN.1-Element nötig, und der Wert wird als String-Referenz gespeichert (Listing 4.15).

4.3.2.11 Die Containertypen `seq`, `set`, `tup` und `bag`

Listing 4.18: GXL-Containertypen

```
<!ELEMENT seq     (%val;)* >
<!ELEMENT set     (%val;)* >
<!ELEMENT bag     (%val;)* >
<!ELEMENT tup     (%val;)* >
```

GXL stellt die vier Container `bag`, `seq`, `set` und `tup` bereit. Ein `tup` kann beliebig viele Elemente beliebigen Typs enthalten, und die anderen drei Container enthalten beliebig viele Elemente genau eines Typs.

Listing 4.19: ASN.1-Repräsentationen der GXL-Containertypen

```
Tup ::= SEQUENCE OF Value
Seq ::= SEQUENCE OF Value
Set ::= SEQUENCE OF Value
Bag ::= SEQUENCE OF Value
```

Die BinaryGXL-Typen `Bag`, `Seq`, `Set` und `Tup` enthalten eine beliebige Anzahl von `Value`-Elementen, wobei Value ein beliebiger Wertetyp ist.

Die Beschränkung, dass ein Container des Typs `bag`, `set` oder `seq` nur Elemente *eines* Typs enthalten darf, ist also nicht in BinaryGXL eingeflossen. Es ist durchaus möglich, ein (nicht valides) GXL-Dokument, welches beliebige Wertetypen in einem zusammengesetzten Wertetyp enthält, in BinaryGXL zu transformieren. (Siehe dazu auch Abschnitt 4.2 auf Seite 45.)

Alle vier Container sind mit dem `SEQUENCE OF`-Container implementiert.

4.3.3 Behandlung von Zeichenketten

In diesem Abschnitt wird die besondere Behandlung von Zeichenketten in BinaryGXL erläutert. Die dazu spezifizierten ASN.1-Typen haben kein Gegenstück in der GXL-DTD.

Listing 4.20: Das Top-Level-ASN.1-Element

```
1 BinaryGxl ::= SEQUENCE {
2   ustrings  UnicodeStringList,
3   astrings  AsciiStringList,
4   gxl       Gxl
5 }
6
7 UnicodeStringList ::= SEQUENCE OF BMPString
8 AsciiStringList ::= SEQUENCE OF IA5String
```

Das Top-Level-ASN.1-Element `BinaryGxl` enthält zwei Listen, die zum Speichern von Strings verwendet werden. Die Idee hierbei ist, dass man ein viel kompakteres Speicherformat schaffen kann, wenn man jeden String nur genau einmal speichert statt bei jeder Benutzung. Überall wo der String verwendet wird, speichert man nur eine `INTEGER`-Referenz, welche den Index in der Liste angibt.

Zusätzlich unterscheidet BinaryGXL noch nach dem Zeichensatz, den ein gegebener String zur Kodierung benötigt. Enthält ein String nur Zeichen aus dem ASCII-Zeichensatz, so wird er als `IA5String` in der `AsciiStringList` gespeichert, ansonsten als Unicode unterstützender `BMPString` in der `UnicodeStringList`. Da fast alle heute benutzten GXL-Dateien ohnehin fast ausschließlich ASCII-Zeichen benutzen, lässt sich hiermit eine große Platzersparnis erzielen, denn ein `IA5String` benötigt zur Speicherung nur 1 Byte pro Zeichen, der `BMPString` hingegen 2 Byte.

Um die String-Referenzen einfach unterscheiden zu können, wurde folgende Festlegung getroffen.

- $i < 0 \Rightarrow i$ referenziert den $-i$. `IA5String` aus der `AsciiStringList`
- $i > 0 \Rightarrow i$ referenziert den i. `BMPString` aus der `UnicodeStringList`

Die 0 ist also keine gültige String-Referenz und wird nicht verwendet.

Nachdem jetzt alle BinaryGXL-Typen vorgestellt wurden, wird im folgenden Kapitel genauer auf den Entwicklungsablauf und die Benutzung des Konvertierungstools `binarygxl` eingegangen.

Kapitel 5

Benutzung und Implementierung von `binarygxl`

In diesem Abschnitt wird zuerst eine Anleitung zur Bedienung von `binarygxl` gegeben. Danach wird die grobe Architektur beschrieben. Zuletzt wird eine Versionshistorie gegeben, in der auf die bei der Implementierung aufgetretenen Probleme eingegangen wird.

5.1 Benutzung des Tools `binarygxl`

In diesem Kapitel wird eine kurze aber vollständige Referenz zur Benutzung des Kommandozeilen-Konvertierungstools `binarygxl` gegeben. Mit der Option `-help` oder `-h` kann eine vollständige Liste der möglichen Argumente mit einer kurzen Beschreibung angezeigt werden. Diese Liste enthält auch alle Optionen, welche die *rgutils*, eine Bibliothek des IST Koblenz zur Behandlung von Kommandozeilenoptionen und zum Logging, betreffen und hier nicht beschrieben werden.

5.1.1 Kodierung einer GXL-Datei in BinaryGXL

Um eine GXL-Datei als BinaryGXL-Datei zu kodieren ist die Option `-e` (für *encode*) gefolgt von der GXL-Datei anzugeben.

Optional kann als zweites Argument der gewünschte BinaryGXL-Dateiname angegeben werden. Wird dies nicht getan, so werden die binär kodierten Daten auf `stdout` ausgegeben und können so mit einer Pipe an andere Tools weitergeleitet oder in eine Datei umgeleitet werden.

Mit dem optionalen Argument `-V` oder `-debug-encoding` wird vor dem Kodieren die erzeugte Struktur in einer textuellen Form auf `stderr` ausgegeben.

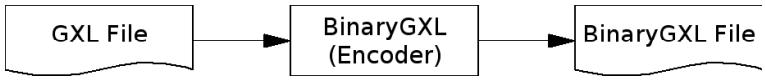

Abbildung 5.1: Kodierung einer GXL-Datei

Zur Veranschaulichung ein paar Beispiele:

- Die GXL-Datei `gxlfile.gxl` wird als BinaryGXL-Datei `binarygxlfile.bgxl` gespeichert. Vor dem Kodieren soll die erzeugte ASN.1-Struktur in einer textuellen Form ausgegeben werden.

```
$ binarygxl -e gxlfile.gxl binarygxlfile.bgxl -V
```

- Die GXL-Datei `gxlfile.gxl` wird nach BinaryGXL konvertiert, und die erzeugten Binärdaten werden über eine Pipe an bzip2 übergeben, welches diese komprimiert und in die Datei `binarygxlfile.bgxl.bz2` schreibt.

```
$ binarygxl -e gxlfile.gxl | bzip2 > \
              binarygxlfile.bgxl.bz2
```

- Die GXL-Datei `gxlfile.gxl` wird nach BinaryGXL konvertiert, und die erzeugten Binärdaten werden über eine Pipe an bzip2 übergeben, welches diese komprimiert und in die Datei `binarygxlfile.bgxl.bz2` schreibt. Zusätzlich soll die erzeugte ASN.1-Struktur in die Datei `encode.log` geschrieben werden.

```
$ binarygxl -e gxlfile.gxl -V 2> encode.log \
            | bzip2 > binarygxlfile.bgxl.bz2
```

5.1.2 Dekodierung einer BinaryGXL-Datei in GXL

Um eine BinaryGXL-Datei zu dekodieren ist die Option `-d` (für *decode*) anzugeben, gefolgt von der BinaryGXL-Datei.

Optional kann als zweites Argument der gewünschte GXL-Dateiname angegeben werden. Wird dies nicht getan, so werden die XML-Daten auf `stdout` ausgegeben und können so mit einer Pipe an andere Tools weitergeleitet oder in eine Datei umgeleitet werden.

Abbildung 5.2: Dekodierung einer BinaryGXL-Datei

Auch hier wieder einige Beispiele zur Veranschaulichung:

- Die BinaryGXL-Datei `binarygxlfile.bgxl` wird als `gxlfile.gxl` gespeichert.

```
$ binarygxl -d binarygxlfile.bgxl gxlfile.gxl
```

- Die BinaryGXL-Datei `binarygxlfile.bgxl` wird nach GXL dekodiert und der entstandene XML-Output wird mit `less` angezeigt.

```
$ binarygxl -d binarygxlfile.bgxl | less
```

- Die Knoten der in der BinaryGXL-Datei `binarygxlfile.bgxl` enthaltenen Graphen werden gezählt. Außerdem wird die `binarygxlfile.bgxl` als GXL-Datei `gxlfile.gxl` gespeichert.

```
$ binarygxl -d binarygxlfile.bgxl \
            | tee gxlfile.gxl \
```

```
| grep "<node" \
| wc -l
```

5.2 Implementierung des Tools `binarygxl`

Zuerst wird in diesem Abschnitt die Architektur vom Tool `binarygxl` erläutert. Danach wird in einer kurzen Versionshistorie beschrieben, wie die Entwicklung verlief und welche Probleme dabei auftraten.

5.2.1 Architekturbeschreibung

Die hier gegebene textuelle Architekturbeschreibung soll nur einen kurzen Überblick liefern. Details sind der Doxygen-Dokumentation zu entnehmen.

Der BinaryGXL-Konverter besteht aus dem Encoder, dem Decoder, einigen Klassen zur String-Behandlung und einer Reihe von vom ASN.1-Compiler automatisch generierten C-Dateien, welche die BinaryGXL-Typen definieren und Funktionen zur Kodierung und Dekodierung bereitstellen.

5.2.1.1 Der Encoder

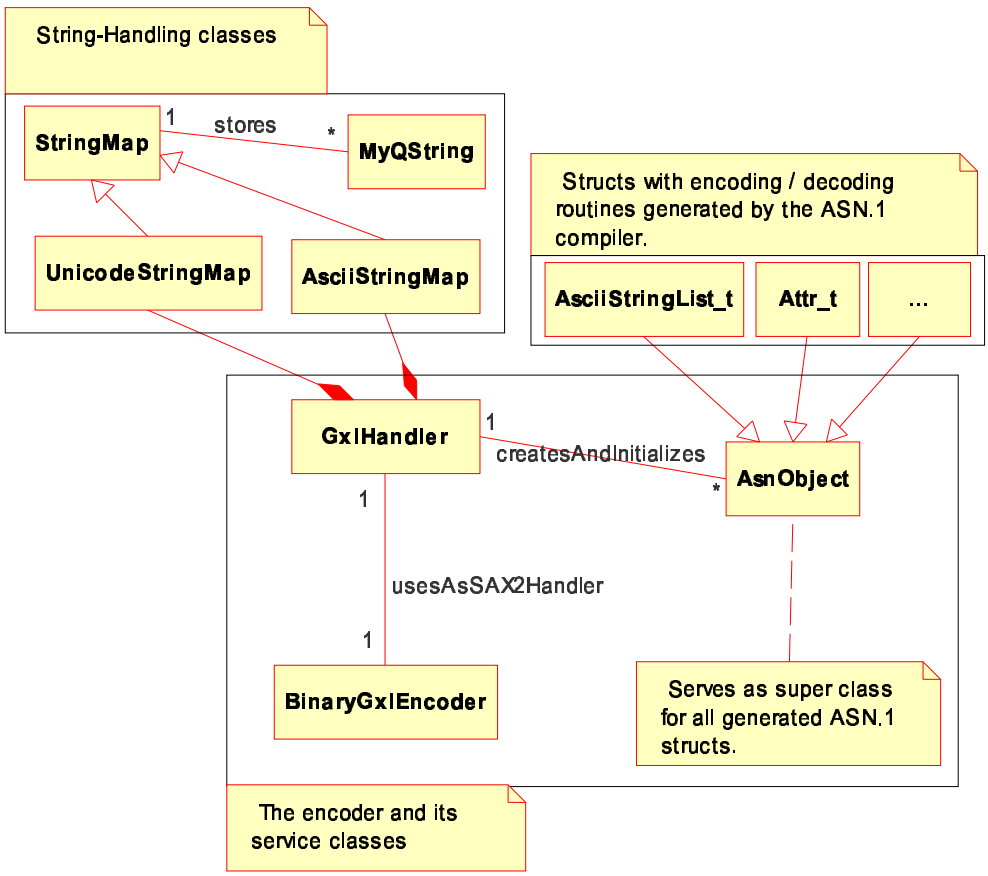

Abbildung 5.3: Grobarchitektur des BinaryGXL-Encoders

Der BinaryGxlEncoder öffnet die GXL-Datei und parst diese mit einem SAX2-XML-Parser[1] (QXmlSimpleReader). Als SAX2-Handler agiert eine Instanz von GxlHandler, welcher von QXmlDefaultHandler abgeleitet ist. Dieser hat das für SAX2 typische Interface und erzeugt beim Parsen der GXL-Datei Instanzen der entsprechenden BinaryGXL-Strukturen.

Dabei werden Zeichenketten, wie schon im Abschnitt 4.3.3 auf Seite 57 besprochen, gesondert behandelt, d.h. jede Zeichenkette wird nur ein einziges Mal gespeichert, und an Benutzungsstellen wird mit einer Integerreferenz darauf verwiesen. Zudem werden Strings, die nur Zeichen des ASCII-Alphabets beinhalten, als IA5String gespeichert, alle anderen als BMPString. Die Speicherung erfolgt in Instanzen der Klassen AsciiStringMap und UnicodeStringMap, welche Methoden zur einfachen Konvertierung zum entsprechenden ASN.1-Gegenstück (also AsciiStringList_t und UnicodeStringList_t) haben. Zur weiteren Vereinfachung dient die von QString abgeleitete Klasse MyQString. Diese bietet zusätzliche Convenience-Methoden zur Konvertierung von und in BMPString und IA5String. Weiterhin hat sie eine Methode isAscii(), welche testet, ob ein String ausschließlich ASCII-Zeichen enthält und eine Methode escape(), welche XML-Steuerzeichen ("'&<>) durch ihre Escape-Sequenzen ersetzt.

Um die Aggregationsstruktur beim Parsen der XML-Datei richtig zu behandeln, wird folgende, für alle baumartigen Strukturen anwendbare Technik benutzt:

Die Klasse AsnObject dient als eine Art Basisklasse für alle GXL-ASN.1-Typen, die Unterelemente haben können. Darunter fallen die Structs Gxl, Graph, Edge, Node, Rel, RelEnd, Attr, Seq, Set, Bag und Tup. Ein AsnObject kapselt genau eine Instanz einer solchen Struktur (struct der Sprache C), indem es einen void-Pointer auf diese und den Typ speichert. Über Methoden der Form toSomeType() kann wieder ein Pointer auf das struct erfragt werden.

Der GxlHandler verwaltet einen QStack<AsnObject*>. Erreicht der Parser ein öffnendes Tag einer der im vorigen Paragraphen aufgezählten Typen, so wird ein entsprechendes struct initialisiert, in einem AsnObject gekapselt und auf den Stack gelegt. Erreicht der Parser ein schließendes Tag eines einfachen, nichtaggregierten Typs, so kann das erzeugte Element einfach dem obersten Element des Stacks hinzugefügt werden. Wird ein schließendes Tag eines aggregierten Typs erreicht, so kann dieses vom Stack genommen werden.

Nach dem Parsen der GXL-Datei kann das Top-Level-BinaryGXL-Element (BinaryGxl) vom GxlHandler abgefragt werden. Läuft der Encoder im Debug-Modus (Option -V), so wird es in einer übersichtlichen Formatierung auf der Standardausgabe ausgegeben.

Danach wird noch überprüft, ob die in der ASN.1-Spezifikation angegebenen Constraints erfüllt werden. (Zur Zeit gibt es jedoch noch keine.) Falls dem so ist, wird die BinaryGXL-Datei geschrieben. Ansonsten wird mit einer Fehlermeldung abgebrochen.

5.2.1.2 Der Decoder

Im Gegensatz zum Encoder ist der BinaryGxlDecoder sehr simpel.

[1] http://www.saxproject.org, 05.07.2006

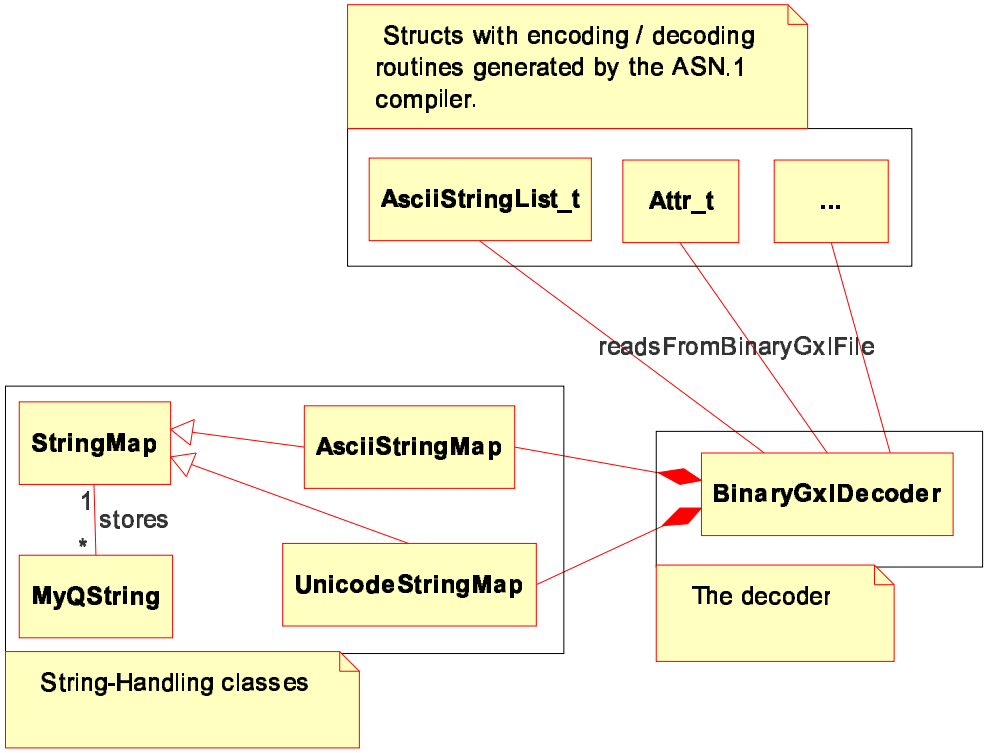

Abbildung 5.4: Grobarchitektur des BinaryGXL-Decoders

Der `BinaryGxlDecoder` öffnet die BinaryGXL-Datei und lässt vom ASN.1-Decoder das `BinaryGxl`-Element einlesen. Nun iteriert er rekursiv über alle Elemente und erzeugt direkt den übersichtlich formatierten GXL-Quelltext. Dabei werden Stringreferenzen wieder durch die zugehörigen Zeichenketten ersetzt, in welchen vorher XML-Steuerzeichen durch ihre entsprechenden Escape-Sequenzen ersetzt wurden.

Wenn in der Original-GXL-Datei nur ASCII-Zeichenketten verwendet wurden, so schreibt der Decoder ISO-8859-1-kodierte Daten, andernfalls ist die Zeichenkodierung UTF-8.

Wurde dem Decoder ein Dateiname für die GXL-Datei angegeben, so wird das erzeugte XML direkt in diese geschrieben, ansonsten erfolgt die Ausgabe auf der Standardausgabe.

5.2.2 Versionshistorie

5.2.2.1 Die erste Implementierung

Zum Parsen und erzeugen von GXL-Dateien gibt es eine vom IST Koblenz entwickelte API – die GXL Instance API. Diese nutzt den XERCES-C++-Parser[2] zur Validierung des GXL-Dokuments und um einen DOM-Baum zu erzeugen, auf den dann leicht mit spezialisierten Iteratoren zugegriffen werden kann. Die Erzeugung von GXL-Dokumenten wird ebenfalls unterstützt.

[2] `http://xml.apache.org/xerces-c/, 05.07.2006`

Damit schien die GXL Instance API das Mittel der Wahl für den BinaryGXL-Konverter zu sein. Der BinaryGxlEncoder war sehr schnell entwickelt, jedoch tauchten bei der Programmierung des Decoders unüberwindbare Probleme auf.

Ein Mangel an der GXL Instance API ist, dass die Reihenfolge, in der über die Elemente iteriert wird, nicht notwendigerweise der Reihenfolge des Erscheinens im GXL-Dokument entspricht.

Dies machte es unmöglich den Konverter so zu implementieren, dass eine hin und zurück kodierte GXL-Datei identisch mit der Originaldatei ist, was zunächst höchstens wie ein Schönheitsmakel erscheint.

Ein größeres Problem leitet sich aber direkt daraus ab. Die GXL Instance API benötigt zur Erzeugung einer Kante die beiden Graphelemente, die sie verbinden soll. Dadurch, dass die Element-Reihenfolge in der BinaryGXL-Datei nicht mehr die der Original-GXL-Datei ist, kann es passieren, dass eine Kante nicht direkt erzeugt werden kann, wenn in der BinaryGXL-Struktur über sie hinweg iteriert wird, da ihre Enden noch nicht erzeugt wurden. Möglicherweise könnte man durch intelligentere Algorithmen das Problem lösen. Allerdings erlaubt GXL auch Kanten, die sich auf sich selbst beziehen, und spätestens das ist mit der derzeitigen GXL Instance API unmöglich zu realisieren.

Damit konnte dieser Pfad nicht weiter verfolgt werden.

Immerhin lieferte diese erste Implementierung schon erste Ergebnisse, die allerdings enttäuschend waren. So hatte die BinaryGXL-Datei ungefähr die gleiche Größe wie die UTF-8-kodierte GXL-Datei.

Der Grund dafür war schnell gefunden. Die Zeichenketten des GXL-Dokuments wurden jeweils als BMPString an allen Benutzungsstellen in BinaryGXL gespeichert. Als Lösung wurde dann das Konzept der Stringreferenzen (s. Abschnitt 4.3.3 auf Seite 57 und Abschnitt 5.2.1 auf Seite 61) erarbeitet.

5.2.2.2 Die aktuelle Implementierung

Damit die Reihenfolge der Elemente beim Kodieren erhalten bleibt, wird in der aktuellen Implementierung das Parsen der GXL-Dokumente mit einem SAX2-Parser erledigt, was zusätzlich einen deutlichen Geschwindigkeitsgewinn mit sich bringt.

Bei der Wahl der Implementierung war besonders der Faktor der Portabilität wichtig. Der Quellcode soll sich auf möglichst allen gängigen Plattformen (Linux, Windows und Mac OS X) ohne Änderungen kompilieren lassen und natürlich quelloffen und kostenlos sein.

Bei der Wahl des Parsers fiel die Wahl auf die SAX-Implementierung des Qt Toolkits der norwegischen Firma Trolltech[3], welche sich gegen XERCES-C++ durchsetzen konnte. Das lag einerseits daran, dass die Weiterentwicklung von XERCES-C++ eingestellt wurde, aber andererseits auch an den vielen Vorteilen, die die Implementierung des Konverters mit dem Qt-Framework stark vereinfachten. Unter anderem zählen folgende Gründe dazu:

- sehr gute Dokumentation
- plattformunabhängiges Build-System *qmake*

[3] http://www.trolltech.com, 05.07.2006

- unicodefähige Stringklasse `QString`
- frei verfügbar für alle gängigen Plattformen (Linux, Windows und Mac OS X)

Die XERCES-API ist aber leistungsfähiger und kann z.B. XML-Dokumente validieren was mit der Qt-Implementierung derzeit nicht möglich ist. Allerdings gibt es ein eigenständiges Tool namens GXL-Validator [4], welches die Validierung von GXL-Dokumenten ermöglicht.

Kapitel 6

Vergleich von BinaryGXL und gepacktem GXL

Um die Güte der entwickelten Lösung BinaryGXL, bzw. dem Konvertierungstool `binarygxl`, genauer beurteilen zu können, wird diese im folgenden Abschnitt mit den gängigsten Kompressionsverfahren verglichen und zwar einerseits hinsichtlich der resultierenden Dateigröße als auch hinsichtlich der zur Kodierung/Dekodierung benötigten Zeit. Ein dritter, nicht unerheblicher Punkt ist, wie viel Speicher zur Kodierung/Dekodierung reserviert wird.

Folgende Komprimierungstools wurden getestet:

- **binarygxl**
- **gzip**
- **bzip2**
- **compress**
- **rar**
- **7-Zip**

Eine kurze Beschreibung jedes Tools wird im Anhang A.2 auf Seite 76 gegeben.

Alle Tests und Zeitmessungen wurden an den GXL-Dateien `stc-utf8.gxl` (ca. 35 MB) und `stc-utf16.gxl` (ca. 69 MB) vorgenommen. Beide Dateien befinden sich in BinaryGXLs Subversion-Verzeichnis unter `binarygxl/testit/gxl/`. Eine kurze Beschreibung des Testsystems ist unter A.3 auf Seite 78 zu finden.

Die Tests wurden automatisch durch das Ruby-Skript `compressor-comparison.rb` aus dem Verzeichnis `binarygxl/tools/` angestoßen und ausgewertet. Das Skript benutzt GNU time[1], um die Zeit, die ein Kommando benötigt (*user* und *system time*) und die durchschnittliche Prozessorauslastung zu messen. Daneben berechnet sie noch den Kompressionsfaktor, also welche prozentuale Größe die komprimierte Datei im Vergleich zu der Originaldatei hat. Zusätzlich kann die Anzahl der Durchläufe dem Skript als Kommandozeilenparameter mitgegeben werden. Der Durchschnitt aller Durchläufe bestimmt dann das Endresultat. Dieses Endresultat

[1] `http://www.gnu.org/directory/time.html`, 05.07.2006

wird dann in einer formatierten Form in eine ebenfalls als Parameter angegebenen Logdatei festgehalten.

Für die Vergleiche wurde jeweils die zu testende GXL-Datei zehn Mal in Folge mit jedem der genannten Tools komprimiert und wieder dekomprimiert. Die beiden daraus resultierenden Logdateien sind im Anhang C auf Seite 83 hinterlegt.

In den folgenden Abschnitten sollen die Ergebnisse nun aufgeschlüsselt und erörtert werden.

6.1 Komprimierte Dateigröße / Kompressionsfaktor

Als erstes wird die wichtigste Größe, der Kompressionsfaktor, beleuchtet. Die Ergebnisse der einzelnen Tools sind geordnet nach dem Kompressionsfaktor in der folgenden Tabelle zusammengefasst. Die Dateigröße ist jeweils in Byte angegeben, und der Kompressionsfaktor gibt den Prozentsatz der Größe der komprimierten Dateien gegenüber den Originaldateien an. Deren Größe ist der Übersicht halber noch einmal in der ersten Zeile aufgeführt.

	UTF-8		**UTF-16**	
Tool	**Größe**	**Faktor**	**Größe**	**Faktor**
-	35729694	-	71413392	-
7-Zip	142834	0.40%	177284	0.25%
RAR	270311	0.76%	951953	1.33%
BZip2	632425	1.77%	679271	0.95%
GNU Zip	1042317	2.92%	1430979	2.00%
Compress	2925926	8.19%	4933103	6.91%
BinaryGXL	5980013	16.74%	5980013	8.37%

Ganz klar am besten komprimiert 7-Zip, ganz egal wie die GXL-Testdatei kodiert ist. Im Schnitt ist die Kompression mehr als dreifach so gut wie die der nächstbesten Tools.

Bei der UTF-8-kodierten GXL-Datei komprimiert RAR diese auf unter ein Hundertstel der Originalgröße und liegt damit knapp vor BZip2. Dieses wiederum komprimiert die UTF-16-kodierte Datei etwas besser. Damit machen diese beiden Tools den zweiten Platz in dieser Kategorie unter sich aus.

GNU Zip liefert ebenfalls noch gute Ergebnisse.

Compress und BinaryGXL können hier nicht wirklich mithalten und liefern deutlich schlechtere Ergebnisse, wobei Compress bei der UTF-8-kodierten GXL-Datei noch mehr als doppelt so stark wie BinaryGXL komprimiert. Da BinaryGXL Zeichenketten intern immer ASCII- bzw. UCS2-kodiert speichert, entstehen bei der Kodierung der UTF-8-kodierten und der UTF-16-kodierten GXL-Datei identische BinaryGXL-Dateien. Deshalb ist Compress Vorsprung bei der UTF-16-kodierten GXL-Datei nicht mehr sonderlich gravierend.

Als Fazit kann man sagen, dass BinaryGXL in puncto Kompressionsfaktor keinesfalls mit aktuellen Kompressionstools mithalten kann, was im Grunde auch nicht weiter verwunderlich ist. Wirklich komprimiert wird hier ja nicht – es wird lediglich eine deutlich kompaktere Form der Strukturdatenrepäsentation genutzt, und bei den Zeichenketten werden Dopplungen vermieden und, falls möglich, wird ASCII als Kodierung verwendet.

6.2 Kompressionsgeschwindigkeit

Ein weiterer wichtiger Faktor zur Bewertung der Güte eines Kompressionstools ist die zur Kompression bzw. Dekompression benötigte Zeit. Die folgende Tabelle listet die Ergebnisse der Tests auf. Die angegebenen Zeiten sind jeweils die Summe aus *user* und *system time*.

	UTF-8		UTF-16	
Tool	**Kompression**	**Dekompr.**	**Kompression**	**Dekompr.**
Compress	1.08 sec	0.40 sec	2.01 sec	0.78 sec
GNU Zip	2.07 sec	0.24 sec	4.03 sec	0.46 sec
RAR	7.58 sec	0.51 sec	16.54 sec	0.93 sec
7-Zip	25.08 sec	0.55 sec	43.68 sec	1.07 sec
BinaryGXL	19.55 sec	12.38 sec	20.19 sec	12.32 sec
BZip2	29.62 sec	1.64 sec	84.63 sec	3.33 sec

Compress und GNU Zip liegen in etwa gleich auf mit ihren Ergebnissen. Compress ist bei der Kompression etwas schneller, wohingegen GNU Zip bei der Dekompression etwas bessere Werte liefert. RAR benötigt bei der Komprimierung deutlich länger, dekomprimiert aber ähnlich schnell. 7-Zip benötigt bei der Komprimierung länger als RAR, liegt bei der Dekomprimierung aber gleich auf. BZip2 verhält sich ähnlich, jedoch ist hier die Kompressionszeit insbesondere bei der UTF-16-kodierten GXL-Datei eindeutig am schlechtesten – in diesem Fall ist selbst BinaryGXL vier mal so schnell.

Im Gegensatz zu den fünf Allzweckkompressoren, bei denen immer die Kompression deutlich länger als die Dekompression dauert, ist bei BinaryGXL die Kodierung nicht wesentlich aufwändiger als die Dekodierung.

Resümierend lässt sich sagen, dass, was den Zeitfaktor angeht, GNU Zip und Compress die besten Ergebnisse liefern, wobei RAR nicht wesentlich schlechter ist. Die Geschwindigkeit von 7-Zip liegt eher im Mittelfeld. BinaryGXL liegt noch knapp vor BZip2, da dieses besonders bei UTF-16-kodierten GXL-Dateien sehr lange zur Komprimierung benötigt. Bei BinaryGXL fällt aber die im Verhältnis zu den Allzweckkompressoren deutlich zu hohe Dekodierungszeit negativ auf.

6.3 Speicherverbrauch

Als letzten wichtigen Faktor zur Beurteilung der Tools soll die Speichergröße, die durch das Tool während seiner Laufzeit reserviert wird, betrachtet werden. Da es ohne Instrumentierung der einzelnen Tools nicht möglich ist, auf genaue Werte zu kommen, sind die folgenden Werte nur Näherungswerte, die durch das Ruby-Skript `memory-logger.rb`[2] erzeugt wurden. Eine Instrumentierung kam schon deshalb nicht in Frage, da RAR ein Closed-Source-Tool ist und somit der Quellcode nicht einsehbar bzw. modifizierbar ist.

Das Skript `memory-logger.rb` wird mit dem auszuführenden Kommando als Parameter aufgerufen. Es führt dann dieses Kommando aus und überwacht den neuen Prozess

[2] `https://svn.uni-koblenz.de/gup/re-group/trunk/project/binarygxl/tools/memory-logger.rb`

mit `top`, wobei dessen Ausgaben in eine Datei umgeleitet werden, welche nach Beendigung des Kommando-Prozesses ausgewertet wird.

Testet man so alle Kompressoren, erhält man folgende Ergebnisse. Dabei ist der erste Wert jeweils die maximale und der zweite Wert die jeweils durchschnittliche Größe im Speicher.

	UTF-8 (max/avg)		**UTF-16** (max/avg)	
Tool	**Kompression**	**Dekompr.**	**Kompression**	**Dekompr.**
GNU Zip	648/648 KB	480/480 KB	656/655 KB	480/480 KB
Compress	1128/1124 KB	560/505 KB	1128/1118 KB	556/556 KB
BZip2	6.9/6.9 MB	4 / 4 MB	6.9/6.9 MB	4/4 MB
RAR	33/33 MB	7.1/7.1 MB	32/32 MB	6.8/6.6 MB
BinaryGXL	132/117 MB	69/64 MB	205/119 MB	69/64 MB
7-Zip	377/242 MB	35/35 MB	650/368 MB	65/65 MB

GNU Zip benötigt sowohl beim Komprimieren als auch beim Dekomprimieren am wenigsten Speicher. Compress beschlagnahmt beim Komprimieren knapp doppelt so viel Speicher und liegt beim Dekomprimieren nur knapp hinter GNU Zip. In beiden Fällen ist der Speicherverbrauch jedoch so gering, dass sie auch bei Low-End-Computern vernachlässigbar sind.

BZip2 verbraucht deutlich mehr Speicher, allerdings ist der Speicherverbrauch unabhängig von der Dateigröße. Laut der Manual-Seite kann er mit den Formel

$$\mathsf{MemSize} = 400\mathsf{k} + 8 \cdot \mathsf{BlockSize}$$

bei der Kompression und

$$\mathsf{MemSize} = 400\mathsf{k} + 4 \cdot \mathsf{BlockSize}$$

bei der Dekompression abgeschätzt werden. Da bei allen Tests die Option `--best` benutzt wurde, ist hier $\mathsf{BlockSize} = 900\mathsf{k}$, was mit den Beobachtungen von `memory-logger.rb` übereinstimmt.

RAR benötigt gerade bei der Komprimierung noch einmal deutlich mehr Speicher als BZip2, allerdings ist auch hier der Speicherverbrauch unabhängig vom zu komprimierenden Dokument.

Gerade das ist bei BinaryGXL nicht der Fall. Je größer das Dokument ist, desto mehr Speicher wird bei der Komprimierung verwendet, denn im Gegensatz zu den blockorientierten Kompressionstools, muss hier zunächst das gesamte GXL-Dokument geparst und die dadurch erzeugte ASN.1-Struktur komplett im Speicher gehalten werden. Erst dann kann die BinaryGXL-Datei geschrieben werden. Schon bei einer GXL-Dateigröße von 69 MB (die UTF-16-kodierte GXL-Datei) benötigt BinaryGXL in etwa 30 mal so viel Speicher wie BZip2, bzw. über 300 mal so viel Speicher wie GNU Zip.

Das gleiche Verhalten trifft in noch stärkerem Maße auf 7-Zip zu. Zur Komprimierung wird hier ungefähr ein Maximum der zehnfachen Größe der zu komprimierenden Datei benötigt. Zur Dekomprimierung wird Speicher in Größe der unkomprimierten Datei benutzt. Der Fairness halber muss hier gesagt werden, dass 7-Zip über eine große Anzahl an Optionen verfügt, mit denen unter anderem auch der Speicherverbrauch gesenkt werden kann, und hier wurden die Optionen so gesetzt, dass die Kompression am besten ist, was allerdings auch den Speicherverbrauch maximiert.

6.4 Fazit

Im Vergleich mit gängigen Kompressionstools zeigt BinaryGXL schnell seine Grenzen und Schwächen. BinaryGXL-Dateien sind deutlich größer als die mit einem herkömmlichen Kompressor gepackten GXL-Dateien und das teilweise um den Faktor 40 im Vergleich mit 7-Zip. Selbst das altgediente Compress kann bei UTF-8-kodierten Dateien mit doppelt so guter Kompression aufwarten.

Bei der Kompressionsgeschwindigkeit ist BinaryGXL immerhin schneller als BZip2, jedoch knapp zehnfach langsamer als GNU Zip. Bei der Dekomprimierung ist GNU Zip das Maß aller Dinge und ist teilweise um den Faktor 48 schneller. Auch alle anderen Tools entpacken deutlich schneller als BinaryGXL.

Die schlechtesten Ergebnisse liefert BinaryGXL aber beim Speicherverbrauch. Nur bei 7-Zip und BinaryGXL ist diese nämlich abhängig von der Dateigröße der GXL-Datei. Das liegt bei BinaryGXL daran, dass bei der Kodierung immer die gesamte GXL-Datei geparst werden muss. Dabei wird die ASN.1-Struktur erzeugt, welche im Speicher gehalten werden muss. Erst dann kann die BinaryGXL-Datei geschrieben werden. Bei der Dekodierung wird ungefähr so viel Speicher benötigt, wie die Dateigröße der UTF-16-kodierten GXL-Datei ist. Das ist durch die interne String-Repräsentation von BinaryGXL bedingt, denn diese werden mit jeweils 2 Byte pro Zeichen dargestellt. Hier sind die Kompressoren, die mit blockorientierten Algorithmen arbeiten, wesentlich speichereffizienter.

Ein weiterer Vorteil, der insbesondere den Einsatz von GNU Zip empfiehlt, ist, dass dieses für nahezu alle Architekturen und alle heute verwendeten Betriebssysteme verfügbar ist. Es ist Teil jeder Linux-Distributionen und immer schon vorinstalliert. Für fast alle anderen Plattformen werden auf `http://www.gzip.org` vorkompilierte Binaries zum Download angeboten.

Das gleiche Argument gilt ebenfalls, wenn auch etwas schwächer, für BZip2 und 7-Zip, welche ebenfalls für alle unixoiden Systeme und Windows verfügbar sind.

Im nun folgenden letzten Kapitel werden die Ergebnisse dieser Studienarbeit noch einmal zusammengefasst.

Kapitel 7

Zusammenfassung

Ein Ziel dieser Studienarbeit war die Beobachtung und Dokumentation der Bestrebungen zur Findung eines allgemein gültigen Standards zur binären Serialisierung von XML-Dokumenten. Da es unzählige verschiedene Anwendungsgebiete gibt, definierte die XML Binary Characterization Working Group zunächst ein Anforderungsdokument, welches alle benötigten Eigenschaften aufzählt und beschreibt. Da für ein universelles Binärformat alle davon wichtig sind, wurde ihnen der komplette Abschnitt 2.2.1 gewidmet.

Die XBC hat allerdings bislang kein allgemeingültiges Binary-XML-Format gefunden bzw. ein schon existierendes Format empfohlen, weshalb zwei alternative binäre XML-Formate untersucht wurden. Zunächst wurde im Abschnitt 2.3.1 das altgediente WBXML-Format untersucht. Zusätzlich wurde ein Patch für die *libwbxml*-Bibliothek entwickelt, welcher dieser GXL-Unterstützung hinzufügt. Da WBXML allerdings nur wenige der von der XBC geforderten Eigenschaften erfüllt, eignet es sich nicht als universelles Binary-XML-Format.

Als zweites Alternativformat wurde das Binary Format for MBEG-7, kurz BiM, untersucht. Dieses ermöglicht den Einsatz spezieller Codecs für die verschiedenen Inhaltsdaten. Mittels einer einfachen und robusten, auf endlichen Automaten basierenden Technik werden die Strukturdaten kodiert. Damit erreicht BiM sehr gute Kompressionsfaktoren. Auch die meisten anderen XBC-Eigenschaften werden erfüllt. So bietet BiM Möglichkeiten zur schnellen Suche im Dokument, es können Teilbäume übersprungen werden, es stehen verschiedene Strategien zur Übertragung eines Dokuments zur Verfügung, und die Validierung findet schon bei der Kodierung statt. Zudem war bei der Entwicklung von BiM die Erweiterbarkeit ein besonders wichtiger Faktor. Beispielsweise können Signatur- oder Kompressionscodecs integriert werden. Rückwärts- und Vorwärtskompatibilität spielen hier ebenfalls eine wichtige Rolle. Das alles macht BiM zu einem sehr guten Kandidaten für ein universell einsetzbares Binary-XML-Format.

Das zweite Ziel dieser Studienarbeit war die Entwicklung eines eigenen Binärformats für den XML-Dialekt GXL. Da der Ausgangspunkt eine rudimentäre ASN.1-Spezifikation war, wurde im Kapitel 3 zunächst die Beschreibungssprache ASN.1 und ihre verschiedenen Encoding Rules erläutert. Daran schließt sich das vierte Kapitel an, in dem die stark erweiterte, korrigierte und modifizierte ASN.1-Spezifikation der GXL-DTD gegenüber gestellt und erklärt wurde.

Im praktischen Teil dieser Studienarbeit wurde dann der Kommandozeilenkonverter `binarygxl` entwickelt, über dessen Benutzung, Architektur und Implementierung Kapitel 5 Auskunft gibt.

Im sechsten Kapitel ging es dann um den Vergleich der selbst entwickelten Lösung mit herkömmlichen Kompressionstools. Hier zeigte sich schnell, dass der bloße Übergang zu einem binären Speicherformat noch lange kein Garant für geringe Dateigröße ist, denn das beste Kompressionstool erzeugte vierzig mal kleinere Dateien als `binarygxl`. Auch was den Speicherbedarf bei der Kodierung und Dekodierung betrifft, konnte `binarygxl` nur einen Platz im hinteren Feld belegen. Hier haben die blockorientierten Packer wie GNU Zip und BZip2 deutliche Vorteile. Außerdem dauerte der Kodier- und insbesondere der Dekodiervorgang bei `binarygxl` deutlich zu lange, so dass man GXL-Dokumente lieber mit GNU Zip packen als in BinaryGXL konvertieren sollte, wenn es nur auf die resultierende Dateigröße ankommt.

Obwohl die Resultate nicht allzu erwähnenswert sind, so konnte der Autor im Rahmen dieser Studienarbeit trotzdem viele Erfahrungen sammeln und sein Wissen auf folgenden Gebieten vertiefen:

- Verfassen von wissenschaftlichen Arbeiten mit LaTeX
- Programmieren mit C/C++
- Programmieren mit Ruby
- XML, DTD, XML Schema
- XML-Verarbeitung mit SAX2 und DOM (GXL Instance API)
- Klassendokumentation mit Doxygen
- Unit-Testing mit cppunit
- Versionsverwaltung mit Subversion
- Beschreibung von Daten mittels ASN.1

Nicht zuletzt hat der Großteil der Arbeit sogar Spaß gemacht...

Anhang A

Tools, Bibliotheken und Systemumgebung

A.1 Bibliotheken

Der BinaryGXL-Kommandozeilenkonverter verwendet folgende Bibliotheken:

- **Qt**[1]

  Qt ist ein für alle gängigen Plattformen verfügbares C++ Entwicklungsframework der norwegischen Firma Trolltech. Einst als Bibliothek für GUI-Entwicklung konzipiert, ist es heute eine umfassende Sammlung von einzelnen Bibliotheken und Tools für nahezu alle wichtigen Anwendungsbereiche. So gibt es z.B. Module für Datenbankanbindungen, Netzwerkprogrammierung, Thread- und Prozesssteuerung, XML-Verarbeitung und natürlich GUI-Programmierung.

  Qt erfreut sich in der Open-Source-Community großer Beliebtheit. So setzt beispielsweise die Desktopumgebung KDE[2] auf Qt auf. Trolltech bietet Qt in zwei verschiedenen Versionen an. Entwickler freier Programme können Qt lizenziert unter der GPL[3] kostenlos herunterladen und benutzen. Für kommerzielle Anwendungen ist allerdings eine kommerzielle Lizenz erforderlich. Beide Versionen sind bis auf die Lizenz identisch.

  BinaryGXL nutzt zum Parsen der GXL-Dateien den SAX2-Parser der Qt-Bibliothek. Weiterhin werden programmintern `QString`s als unicodefähiger Stringtyp benutzt.

  BinaryGXL wurde mit der Version 4.0 entwickelt, es funktioniert aber auch problemlos mit der Version 4.1.

- **rgutils**

  Die *rgutils* sind eine Utility-Bibliothek vom IST Koblenz. BinaryGXL nutzt diese zum Loggen von Debug-Meldungen und zum Parsen der Kommandozeilenargumente.

[1] `http://www.trolltech.com`, 05.07.2006
[2] K Desktop Environment, `http://www.kde.org`, 05.07.2006
[3] GNU Public License, `http://www.gnu.org`, 05.07.2006

A.2 Tools

Zur Entwicklung und zum Testen von BinaryGXL wurden folgende Tools verwendet:

- **asn1c**

  Die ASN.1-Spezifikation von BinaryGXL wurden mit dem asn1c [5] in Strukturen der Sprache C übersetzt. Dazu wurde die Version 0.9.18 benutzt. Die dadurch entstandenen Header- und C-Dateien liegen dem restlichen Quellcode bei, so dass zur Kompilation von BinaryGXL keine Installation des asn1c nötig ist. Im Laufe der Studienarbeit erschien eine neue Version, die allerdings nicht kompatibel mit der benutzten Version und damit dem Quelltext ist.

- **GCC**[4]

  Der Quellcode wurde mit dem C++-Compiler der GCC übersetzt. Getestet wurden die Versionen 3.3.5, 3.4.4, 4.0.2, 4.1.0 und 4.1.1.

- **KDevelop**[5]

  Als integrierte Entwicklungsumgebung wurde KDevelop in den Versionen 3.2 und 3.3 verwendet.

- **teTeX**[6]

  Als LaTeX-Distribution wurde teTeX in der Version 3.0_p1 verwendet. Die Dokumentenklasse ist `report`, und folgende Pakete wurden benutzt:

  - **ngerman**
  - **times**
  - **listings**
  - **fancyhdr**
  - **parskip**
  - **hyperref**
  - **graphicx**

- **GNU Emacs**[7]

  Zum Schreiben der vorliegenden Ausarbeitung wurde GNU Emacs mit dem AUCTeX[8]-Erweiterungspaket genutzt.

- **7-Zip**[9]

  7-Zip, bzw. das 7z-Archivformat, ist ein neues Archivformat, welches besonders gute Kompression verspricht und Multithreading unterstützt. Es hat eine offene Architektur und kann damit immer neue Kompressionsmethoden unterstützen.

[4] Die GNU Compiler Collection, `http://gcc.gnu.org`, 05.07.2006
[5] `http://www.kdevelop.org`, 05.07.2006
[6] `http://www.tug.org/teTeX/`, 05.07.2006
[7] `www.gnu.org/software/emacs`, 05.07.2006
[8] `www.gnu.org/software/auctex`, 05.07.2006
[9] `http://www.7-zip.org/`, 05.07.2006

Die für die Test herangezogene Kompressionsmethode ist LZMA, welches eine optimierte und verbesserte Version des LZ77-Algorithmus ist, der z.B. auch von GNU Zip benutzt wird.

Für Linux/UNIX existiert eine Portierung[10], die versucht immer auf dem aktuellen Versionsstand der nativen Windowsversion zu sein. Diese Portierung wurde in Version 4.42 für die Tests verwendet.

Anbei sei noch vermerkt, dass das Kommandozeilentool `7z` auch die meisten anderen gängigen Archivformate unterstützt, wie z.B. GNU Zip oder BZip2 und außerdem quelloffen (LGPL-2.1) ist.

- **BinaryGXL**

  Das in dieser Studienarbeit entwickelte Konvertierungstool.

  Die zur Zeit aktuelle Version ist 1.1.1.

- **Compress**

  Compress ist das Standard-Kompressionstool der UNIX-Welt. Es verwendet zur Kompression eine Variante des LZW-Algorithmus[11]. Diese wurde 1983 für das Sperry Research Center patentiert, aus dem später UNISYS hervorging. Zu dieser Zeit erschien ebenfalls ein IEEE-Artikel über diesen Algorithmus, der jedoch dessen Patentiertheit verschwieg. Anhand dieses Artikels schrieb Spencer Thomas im Jahre 1984 das Tool *compress*. Da jedoch jeder, der *compress* benutzen wollte Patentgebühren an Sperry Research zahlen musste, wurden in der Open-Source-Szene die Alternativen *gzip* und *bzip2* geschaffen. Das Patent lief allerdings 2003 aus, so dass *compress* jetzt ein Public Domain Tool ist.

  Da es auch heute noch vielfach verwendet wird, soll es auch in diesem Vergleich berücksichtigt werden.

  Für die Tests wurde `ncompress-4.2.4` benutzt.

- **GNU Zip**[12]

  GNU Zip ist das Standardkompressionstool des GNU-Projektes[13]. Es wurde als Alternative zu *compress* von Jean-Loup Gailly und Mark Adler geschrieben, dessen Implementierung des LZW-Algorithmus zur damaligen Zeit von IBM und UNISYS patentiert war.

  Gzip ist eine dictionarybasierte Komprimierung (LZ77, Lempel-Ziv-Kodierung), was bedeutet, dass häufig vorkommende Zeichenketten durch den Integer-Index der Zeichenkette in diesem Wörterbuch ersetzt werden. Auch bei BinaryGXL werden gleiche Strings nur einmal gespeichert und Integer-Referenzen verweisen auf den richtigen Eintrag. Der Clou an Gzip ist aber, dass das Wörterbuch nicht in der komprimierten Datei gespeichert wird, sondern der Decoder es wieder automatisch aus der komprimierten Datei generieren kann.

  Für die Tests wurde GNU Zip in der Version 1.3.5 benutzt.

[10] `http://p7zip.sourceforge.net/`, 05.07.2006
[11] LZW steht für Lempel-Ziv-Welch
[12] `http://www.gnu.org/software/gzip/gzip.html`, 05.07.2006
[13] `http://www.gnu.org`, 05.07.2006

- **BZip2**[14]

  bzip2 ist ein unter einer BSD-artigen Lizenz vertriebenes Kompressionsprogramm, welches von Julian Seward entwickelt wird.

  Die Komprimierung erfolgt hier in zwei Schritten. Zuerst wird mittels des *Burrows-Wheeler Block-Sorting*-Algorithmus die zu komprimierende Datei in ein besser komprimierbares Format umgewandelt. Dazu werden die Originalzeichenketten so umgestellt, dass sie möglichst viele aufeinander folgende Zeichenwiederholungen enthalten. Diese sind gut komprimierbar.

  Danach folgt die eigentliche Kompression mittels *Huffman-Kodierung*.

  Im allgemeinen komprimiert Bzip2 besser als Gzip, ist aber langsamer.

  Die für die Tests benutzte Version ist 1.0.3.

- **rar**[15]

  RAR ist ein kommerzielles closed-source Archivdateiformat. Im Gegensatz zu Gzip und Bzip können also mehrere Dateien in einem komprimierten Archiv zusammengefasst werden.

  Welche Technik zur Komprimierung Verwendung findet, ist auf der Homepage jedoch nicht feststellbar. Features, die hier allerdings keine Berücksichtigung finden, sind beispielsweise die optionale Verschlüsselung der erstellten Archive oder die Fähigkeit zur Reparatur defekter Archive.

  Für die Tests wurde `rar-3.6.0` verwendet.

A.3 Systemumgebung

Der größte Teil der Entwicklung wurde in folgender Systemumgebung durchgeführt:

Betriebssystem	Gentoo GNU/Linux
System	Dell Inspiron 8200 (Laptop)
Prozessor	Intel®Pentium4 mobile™1.4 GHz
RAM	768 MB

Alle Tests und Messungen wurden in folgender Systemumgebung durchgeführt:

Betriebssystem	Gentoo GNU/Linux
System	Samsung R65 Charis (Laptop)
Prozessor	Intel®Core™Duo Processor T2300 1.66 GHz
RAM	1024 MB

[14] `http://www.bzip.org`, 05.07.2006
[15] `http://www.rarsoft.com`, 05.07.2006

Anhang B

Literaturverzeichnis

[Gora98] Walter Gora, *ASN.1 - Abstract Syntax Notation One*, FOSSIL-Verlag Köln, 1998

[Dubu00] Olivier Dubuisson, *ASN.1 - Communication between Heterogeneous Systems*, Morgan Kaufmann Publishers, 2000

[Larm99] John Larmouth, *ASN.1 Complete*, Morgan Kaufmann Publishers, 1999

[1] GXL Homepage: `http://www.gupro.de/GXL`, 05.07.2006

[2] GXL DTD: `http://www.gupro.de/GXL/dtd/dtd.html`, 05.07.2006

[3] GXL Schema: `http://www.gupro.de/GXL/xmlschema/xmlschema.html`, 05.07.2006

[4] GXL-Validator: `http://www.gupro.de`, 05.07.2006

[5] asn1c - The Open Source ASN.1 Compiler, developed by Lev Walkin, `http://lionet.info/asn1c/`, 05.07.2006

[6] Unicode Charts, `http://www.unicode.org/charts/`, 05.07.2006

[XBC] XML Binary Characterization Working Group: `http://www.w3.org/XML/Binary/`, 05.07.2006

[7] XML Binary Characterization Use Cases: `http://www.w3.org/TR/xbc-use-cases/`, 05.07.2006

[8] XML Binary Characterization Properties: `http://www.w3.org/TR/xbc-properties/`, 05.07.2006

[9] W3C Timed Text Working Group: `http://www.w3.org/AudioVideo/TT/`, 05.07.2006

[10] Robin Berjon (Expway), *Binary XML with BiM*, `http://www.mitre.org/news/events/xml4bin/pdf/thienot_binary.pdf`, 05.07.2006

[11] Standard: *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*, ISO/IEC 8824-1:2002 und ITU Recommendation X.680 (07/02)

[12] Standard: *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification*, ISO/IEC 8824-2:2002 und ITU Recommendation X.681 (07/02)

[13] Standard: *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification*, ISO/IEC 8824-3:2002 und ITU Recommendation X.682 (07/02)

[14] Standard: *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications*, ISO/IEC 8824-4:2002 und ITU Recommendation X.683 (07/02)

[15] Standard: *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*, ISO/IEC 8825-1:2002 und ITU Recommendation X.690 (07/02)

[16] Standard: *Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)*, ISO/IEC 8825-2:2002 und ITU Recommendation X.691 (07/02)

[17] Standard: *Information technology – ASN.1 encoding rules: XML Encoding Rules (XER)*, ISO/IEC 8825-4:2002 und ITU Recommendation X.693 (12/01)

[IA5] Standard: *Information Technology – International Reference Alphabet (IRA) (Formerly International Alphabet No. 5 or IA5) – 7-bit coded character set for information interchange*, ITU-T Recommendation T.50, (09/1992)

[UCS] Standard: *Information technology – Universal Multiple-Octet Coded Character Set (UCS)*, ISO/IEC 10646:2003

[IEEE 754] Standard: *Standard for Binary Floating-Point Arithmetic*, IEEE 754

[SGML] Standard: *Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, ISO 8879:1986

[XML] Standard: *Extensible Markup Language (XML) 1.0 (Third Edition)*, W3C Recommendation, `http://www.w3.org/TR/REC-xml/`, 05.07.2006

[18] Standard: *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)*, W3C Recommendation, `http://www.w3.org/TR/xhtml1/`, 05.07.2006
Standard: *XHTML 1.1 – Module-based XHTML*, W3C Recommendation, `http://www.w3.org/TR/xhtml11/`, 05.07.2006

[19] Standard: *Wireless Markup Language Version 2.0*, 11. September 2001, `http://www.openmobilealliance.org/tech/affiliates/wap/wap-238-wml-20010911-a.pdf`, 05.07.2006

[20] Standard: *Binary XML Content Format Specification*, Version 1.3, 25 July 2001, `http://www.openmobilealliance.org/tech/affiliates/wap/wap-192-wbxml-20010725-a.pdf`, 05.07.2006

[21] Standard: *The Extensible Messaging and Presence Protocol (XMPP)*, IETF RFC 3920, 3921, 3922 und 3923, http://www.xmpp.org/specs/, 05.07.2006

[22] Standard: *OpenDocument Format for Office Applications (OpenDocument) v1.0*, OASIS, http://www.oasis-open.org/specs/index.php, 05.07.2006

[23] Standard *Scalable Vector Graphics (SVG) 1.1 Specification*, W3C Recommendation, http://www.w3.org/TR/SVG11/, 05.07.2006

[24] Standard: *MPEG-7 – Multimedia Content Description Interface*, ISO/IEC 15938-6:2003, http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm, 05.07.2006

[25] Standard: *SOAP Version 1.2*, W3C Recommendations

- http://www.w3.org/TR/soap12-part0/, 05.07.2006
- http://www.w3.org/TR/soap12-part1/, 05.07.2006
- http://www.w3.org/TR/soap12-part2/, 05.07.2006
- http://www.w3.org/TR/soap12-testcollection/, 05.07.2006

[26] Standard: *XML Pointer Language (XPointer) Version 1.0*, W3C Recommendation, http://www.w3.org/TR/WD-xptr, 05.07.2006

[27] Standard: *Cascading Style Sheets, level 2 CSS2 Specification*, W3C Recommendation, http://www.w3.org/TR/REC-CSS2/, 05.07.2006

[28] Standard: *Extensible Stylesheet Language (XSL) Version 1.0*, W3C Recommendation, http://www.w3.org/TR/xsl/, 05.07.2006

[29] Standard: *XSL Transformations (XSLT) Version 2.0*, W3C Recommendation, http://www.w3.org/TR/xslt20/, 05.07.2006

[30] Standard: *XML Schema*, W3C Recommendations

- http://www.w3.org/TR/xmlschema-0/, 05.07.2006
- http://www.w3.org/TR/xmlschema-1/, 05.07.2006
- http://www.w3.org/TR/xmlschema-2/, 05.07.2006

[31] Standard: *Document Object Model (DOM)*, W3C Recommendations, http://www.w3.org/DOM/, 05.07.2006

[32] Standard: *Simple API for XML*, http://www.saxproject.org/, 11.09.2006

[33] Standard: *XQuery 1.0: An XML Query Language*, W3C Candidate Recommendation, http://www.w3.org/TR/xquery/, 05.07.2006

[34] Standard: *Resource Description Framework (RDF)*, W3C Recommendations, http://www.w3.org/RDF/, 05.07.2006

Anhang C

Ergebnisse des Kompressorenvergleichs

C.1 Log der UTF-8 kodierten GXL-Datei

```
1  ########################################
2  # Test Results of file 'stc-utf8.gxl' #
3  ########################################
4
5  Size of stc-utf8.gxl (Bytes): 35729694
6  Number of runs for each test: 10
7
8  Test: BinaryGXL
9  ===============
10
11 Encoding
12 --------
13 Average User CPU Time (secs):      19.549
14 Average System CPU Time (secs):    0.315
15 Average Total CPU Time (secs):     19.864
16 Average CPU Usage (%):             98.9
17 Size of Encoded File (Byte):       5980013
18 Compression Ratio (%):             16.7368156021711
19
20 Decoding
21 --------
22 Average User CPU Time (secs):      12.167
23 Average System CPU Time (secs):    0.215
24 Average Total CPU Time (secs):     12.382
25 Average CPU Usage (%):             96.4
26
27 ------------------------------------------------------
28
29 Test: GNU Zip
30 =============
31
32 Encoding
33 --------
34 Average User CPU Time (secs):      1.991
35 Average System CPU Time (secs):    0.077
36 Average Total CPU Time (secs):     2.068
37 Average CPU Usage (%):             99.0
38 Size of Encoded File (Byte):       1042317
39 Compression Ratio (%):             2.91722901405201
40
41 Decoding
42 --------
43 Average User CPU Time (secs):      0.201
```

```
Average System CPU Time (secs):    0.039
Average Total CPU Time (secs):     0.24
Average CPU Usage (%):             98.8

------------------------------------------------------

Test: BZip2
===========

Encoding
--------
Average User CPU Time (secs):      29.449
Average System CPU Time (secs):    0.172
Average Total CPU Time (secs):     29.621
Average CPU Usage (%):             99.0
Size of Encoded File (Byte):       632425
Compression Ratio (%):             1.77002635399005

Decoding
--------
Average User CPU Time (secs):      1.574
Average System CPU Time (secs):    0.062
Average Total CPU Time (secs):     1.636
Average CPU Usage (%):             99.0

------------------------------------------------------

Test: Compress
==============

Encoding
--------
Average User CPU Time (secs):      0.991
Average System CPU Time (secs):    0.085
Average Total CPU Time (secs):     1.076
Average CPU Usage (%):             98.7
Size of Encoded File (Byte):       2925926
Compression Ratio (%):             8.18905977756205

Decoding
--------
Average User CPU Time (secs):      0.331
Average System CPU Time (secs):    0.07
Average Total CPU Time (secs):     0.401
Average CPU Usage (%):             99.0

------------------------------------------------------

Test: RAR
=========

Encoding
--------
Average User CPU Time (secs):      7.432
Average System CPU Time (secs):    0.147
Average Total CPU Time (secs):     7.579
Average CPU Usage (%):             98.3
Size of Encoded File (Byte):       270311
Compression Ratio (%):             0.756544402535325

Decoding
--------
Average User CPU Time (secs):      0.423
Average System CPU Time (secs):    0.083
Average Total CPU Time (secs):     0.506
Average CPU Usage (%):             99.0

------------------------------------------------------

Test: 7-Zip
===========
```

```
115
116 Encoding
117 --------
118 Average User CPU Time (secs):      24.541
119 Average System CPU Time (secs):    0.537
120 Average Total CPU Time (secs):     25.078
121 Average CPU Usage (%):             99.9
122 Size of Encoded File (Byte):       142834
123 Compression Ratio (%):             0.399762729566058
124
125 Decoding
126 --------
127 Average User CPU Time (secs):      0.453
128 Average System CPU Time (secs):    0.095
129 Average Total CPU Time (secs):     0.548
130 Average CPU Usage (%):             98.8
131
132 ------------------------------------------------------
```

C.2 Log der UTF-16 kodierten GXL-Datei

```
1  ########################################
2  # Test Results of file 'stc-utf16.gxl' #
3  ########################################
4
5  Size of stc-utf16.gxl (Bytes): 71413392
6  Number of runs for each test: 10
7
8  Test: BinaryGXL
9  ===============
10
11 Encoding
12 --------
13 Average User CPU Time (secs):      20.188
14 Average System CPU Time (secs):    0.418
15 Average Total CPU Time (secs):     20.606
16 Average CPU Usage (%):             96.3
17 Size of Encoded File (Byte):       5980013
18 Compression Ratio (%):             8.37379773250373
19
20 Decoding
21 --------
22 Average User CPU Time (secs):      12.129
23 Average System CPU Time (secs):    0.186
24 Average Total CPU Time (secs):     12.315
25 Average CPU Usage (%):             99.0
26
27 ------------------------------------------------------
28
29 Test: GNU Zip
30 =============
31
32 Encoding
33 --------
34 Average User CPU Time (secs):      3.884
35 Average System CPU Time (secs):    0.145
36 Average Total CPU Time (secs):     4.029
37 Average CPU Usage (%):             99.0
38 Size of Encoded File (Byte):       1430979
39 Compression Ratio (%):             2.0037964307871
40
41 Decoding
42 --------
43 Average User CPU Time (secs):      0.376
44 Average System CPU Time (secs):    0.087
45 Average Total CPU Time (secs):     0.463
46 Average CPU Usage (%):             99.0
47
48 ------------------------------------------------------
```

```
49
50  Test: BZip2
51  ===========
52
53  Encoding
54  --------
55  Average User CPU Time (secs):      84.305
56  Average System CPU Time (secs):    0.322
57  Average Total CPU Time (secs):     84.627
58  Average CPU Usage (%):             99.0
59  Size of Encoded File (Byte):       679271
60  Compression Ratio (%):             0.951181537490895
61
62  Decoding
63  --------
64  Average User CPU Time (secs):      3.247
65  Average System CPU Time (secs):    0.084
66  Average Total CPU Time (secs):     3.331
67  Average CPU Usage (%):             99.0
68
69  ------------------------------------------------------
70
71  Test: Compress
72  ==============
73
74  Encoding
75  --------
76  Average User CPU Time (secs):      1.832
77  Average System CPU Time (secs):    0.18
78  Average Total CPU Time (secs):     2.012
79  Average CPU Usage (%):             99.0
80  Size of Encoded File (Byte):       4933103
81  Compression Ratio (%):             6.90781219298476
82
83  Decoding
84  --------
85  Average User CPU Time (secs):      0.64
86  Average System CPU Time (secs):    0.14
87  Average Total CPU Time (secs):     0.78
88  Average CPU Usage (%):             99.0
89
90  ------------------------------------------------------
91
92  Test: RAR
93  =========
94
95  Encoding
96  --------
97  Average User CPU Time (secs):      16.32
98  Average System CPU Time (secs):    0.216
99  Average Total CPU Time (secs):     16.536
100 Average CPU Usage (%):             99.0
101 Size of Encoded File (Byte):       951953
102 Compression Ratio (%):             1.33301748221118
103
104 Decoding
105 --------
106 Average User CPU Time (secs):      0.777
107 Average System CPU Time (secs):    0.153
108 Average Total CPU Time (secs):     0.93
109 Average CPU Usage (%):             99.0
110
111 ------------------------------------------------------
112
113 Test: 7-Zip
114 ===========
115
116 Encoding
117 --------
118 Average User CPU Time (secs):      42.757
119 Average System CPU Time (secs):    0.927
```

```
120 Average Total CPU Time (secs):     43.684
121 Average CPU Usage (%):             99.9
122 Size of Encoded File (Byte):       177284
123 Compression Ratio (%):             0.248250356179692
124
125 Decoding
126 --------
127 Average User CPU Time (secs):      0.875
128 Average System CPU Time (secs):    0.19
129 Average Total CPU Time (secs):     1.065
130 Average CPU Usage (%):             77.2
131
132 ------------------------------------------------------
```