Binärformatierung von XML-Parametrierdaten für Leittechnikkomponenten auf Basis des Systemstandards IEC61850

Christopher Pritchard[1]

Februar 2006

[1]Christopher.Pritchard@web.de

Inhaltsverzeichnis

1 Aufgabenstellung **2**

2 Einleitung **2**
2.1 XML als Gerätebeschreibung in der Stationsleittechnik 2
2.2 Konventionen zum Lesen der Arbeit 2
2.3 Aufbau der Arbeit 3

3 XML als Beschreibungssprache **4**
3.1 Definition einer Sprache 4
3.2 XML Überblick 5
3.2.1 Wichtige Begriffe 5
3.2.2 XML-Sprachelemente 6
3.3 Verarbeitung von XML 7

4 Der Systemstandard IEC61850 **10**
4.1 Das Objektmodell der SCL 10
4.2 Die Dateiformate 12
4.3 Die SCL-Syntaxelemente 14
4.3.1 Aufbau des Schemas 14
4.3.2 Besondere Merkmale und Bedeutungen des Schemas . . 18
4.4 Grade der Konformität 19

5 Das Binärformat **20**
5.1 Anforderungen an das Binärformat 20
5.2 Grundlegende Komprimierungsverfahren 22
5.3 Allgemeine Codierverfahren 22
5.3.1 LZ77-Algorithmus und GZIP 22
5.4 XML optimierte syntaxfreie Codierverfahren 24
5.4.1 XMLZIP 24
5.4.2 XMILL 24
5.4.3 WBXML 25
5.4.4 Millau 26
5.4.5 XMLPPM 27
5.5 Syntaxbasierte Codierverfahren 28
5.5.1 XML Xpress 28
5.5.2 ASN.1 28
5.5.3 BiM 30
5.6 Aktivitäten der W3C bezüglich eines Binärformats 32

6 Der BiM Algorithmus 34
6.1 Die Pfadcodierung 34
6.2 Die Payloadcodierung 38
6.3 Synthese der Pfad- und Payloadcodierung 40
6.4 Der Bitstrom 41
6.5 Der Bytecode 42
6.5.1 Die Zustände 44

7 Verwendung des BiM-Verfahrens zur Lösung von Problemstellungen 48
7.1 Namensraumbehandlung und Wildcards 48
7.1.1 Namensraumbehandlung 48
7.1.2 Wildcards 49
7.2 Codierung verteilt deklarierter Elemente 50
7.2.1 Beschreibung der Einschränkung von Erweiterungen . . 51
7.2.2 Spezielle Codierung der Erweiterungen 56
7.3 Abwärtskompatibilität trotz Schemaevolution 59
7.3.1 Mögliche Änderungen und ihre Auswirkung 60
7.3.2 Versionsanweisung 62
7.3.3 Versionsschemaaufteilung 64
7.3.4 Generische Vergleich 67
7.3.5 Codierung von versionierten Schemata im Bytecode . . 69
7.4 Codierung des Elements Private 70

8 Aktivitäten der W3C und IEC bezüglich der Schemaevolution 72

9 Prüfen der Tragfähigkeit einer Baugruppe unter Einsatz des BiM-Verfahrens 73
9.1 Struktur des zu untersuchenden Systems 73
9.1.1 Das Fragmentmanagement 75
9.1.2 Programmierschnittstelle innerhalb des Parameterservices 76
9.2 Anwendungsfälle für die Komponente Parameterservice 79
9.3 Beispielhafte Abläufe während eines Anwendungsfalls 80
9.3.1 Laden eines BiM-Fragments in den SCL-DOM 81
9.3.2 BiM-Fragment anlegen 83
9.3.3 Überführen eines DOM-Fragments in ein BiM-Fragment 84
9.3.4 Zugriff auf Konfigurationsdaten 86
9.3.5 Speichern von Änderungen an Konfigurationsdaten . . 87

9.3.6 Deltas und Fragmente zusammenfügen 89
9.3.7 Import einer CID-Datei 89
9.3.8 Export einer CID-Datei 93

10 Abschließend... 96
10.1 Rückblick 96
10.2 Weitere Punkte für die Zukunft 96

Erklärung gem. § 26 Abs. 1 ADPO

„Hiermit erkläre ich, dass die Diplomarbeit von mir selbstständig verfasst wurde und nur die angegebenen Quellen und Hilfsmittel benutzt wurden.“

____

(Christopher Pritchard)

Dortmund den 5. April 2006

1 Aufgabenstellung

Im Rahmen der Diplomarbeit soll ein geeignetes Verfahren zur Komprimierung der Gerätebeschreibung einer intelligente Baugruppe in der Stationsleittechnik nach IEC 61850 ausgewählt werden. Falls nötig, sollen erforderliche Anpassungen untersucht werden. Anschließend soll der Einsatz dieses Verfahrens in einer intelligenten Baugruppe auf seine Tragfähigkeit hin geprüft werden.

2 Einleitung

2.1 XML als Gerätebeschreibung in der Stationsleittechnik

In den letzten Jahren hat sich XML[1] als Standardbeschreibungssprache durchsetzen können. Die IEC61850, als neue Norm für die Kommunikation der Leit-, Feld- und Prozessebene, verwendet die XML-Syntax für die Beschreibungssprache SCL[2]. Mit dieser Sprache kann unter anderem die Konfigurierung von intelligenten Baugruppen, den so genannten IED's[3], beschrieben werden. Auf der Verarbeitung dieser Konfigurationsdaten liegt der Schwerpunkt dieser Arbeit. Die IEC61850 ist jedoch nicht der einzige Grund für die Stationsleittechnik sich mit XML zu befassen. Die Vorteile, die XML beim Austausch und der Verarbeitung von Daten bietet, soll auch in Bereichen, die nicht von der Norm spezifiziert sind, genutzt werden. Der Einsatz von XML-Dokumenten stellt besondere Anforderungen jedoch an die verarbeitenden Komponenten. Das hohe Datenvolumen eines XML-Dokuments im Verhältnis zu seinen Nutzdaten kann als eines der zentralen Probleme genannt werden. Da die meisten Baugruppen auf Prozess- und Feldebene nur einen begrenzten Speicher zur Verfügung stellen, kann hier ein XML-Format nur schlecht verarbeitet werden. Diese Problematik ist Grundlage für diese Arbeit.

2.2 Konventionen zum Lesen der Arbeit

Zur Kenntlichmachung von Quelltexten werden Auszüge, z.B. aus XML-Dokumenten, in `courier new` gesetzt. Die Bezeichnung der Absätze folgt folgender Konvention:

[1] eXtensible Markup Language
[2] substation configuration language
[3] intelligent electronical device

x Kapitel
x.1 Abschnitt
x.1.1 Unterabschnitt
Absatz
Unterabsatz

2.3 Aufbau der Arbeit

Im **Kapitel 3** werden die Vor- und Nachteile von XML und die sich daraus ergebenden Möglichkeiten zum Einsatz in der Stationsleittechnik untersucht.

Kapitel 4 befasst sich mit den für diese Arbeit wichtigen Teilen der IEC61850.

Im **Kapitel 5** wird auf die Vor- und Nachteile eines Binärformats eingegangen. Danach werden die Anforderungen an das Format formuliert. Anschließend werden verschiedene binäre Formate für textbasierte Informationen darauf geprüft, ob sie den Anforderungen entsprechen.

In **Kapitel 6** wird das für diese Arbeit ausgewählte Verfahren ausführlich beschrieben.

Kapitel 7 zeigt, wie mit Hilfe des Binärformats eine hohe Konformität zur IEC61850 erreicht werden kann.

In **Kapitel 8** wird das Thema der Schemaevolution und Versionierung noch mal unabhängig vom Einsatz eines Binärformats behandelt.

In **Kapitel 9** wird mit einer abstrakten Sichtweise der Einsatz des ausgewählten Binärformats auf einem IED auf seine Tragfähigkeit hin geprüft.

3 XML als Beschreibungssprache

XML gehört zu den so genannten Auszeichnungssprachen[4]. Wie alle anderen Auszeichnungssprachen erbt auch XML seinen Sprachschatz von SGML[5] und wurde im Februar 1998 von dem W3C[6] standardisiert. In diesem Kapitel wird zunächst ein allgemeiner Sprachgebrauch für Beschreibungssprachen eingeführt, um anschließend zu zeigen welche Komponenten des XML-Standards den Bestandteilen der Beschreibungssprache entsprechen. Danach werden häufig verwendete und für die technische Umsetzung später entscheidende Komponenten und Sprachelemente des XML-Standards beschrieben. Diese Beschreibung erhebt nicht den Anspruch einer vollständigen Beschreibung von XML, sondern dient lediglich als Basis für das Verständnis dieser Arbeit. Für eine genauere Beschreibung der Komponenten und Sprachelemente von XML wird auf weiterführende Literatur [XML01] verwiesen. Am Ende des Kapitels werden weit verbreitete Möglichkeiten für den Zugriff von Maschinen auf XML-Dokumente gezeigt.

3.1 Definition einer Sprache

Wie der Name es bereits sagt, handelt es sich bei einer Beschreibungssprache im Allgemeinen um eine Sprache. Im Rahmen der Definition einer Sprache wird zunächst das so genannte Vokabular definiert, welches eine Ansammlung von Wörtern darstellt. Anschließend müssen Regeln festgelegt werden, in welcher Art und Weise die Worte verwendet werden, was einer Art Grammatik entspricht. Danach wird den einzelnen Sprachbestandteilen eine Bedeutung zugewiesen. Mit der nun definierten Sprache können verschiedene Teilnehmer, denen die Definition der Sprache bekannt ist, miteinander kommunizieren. Bei den ausgetauschten Informationen spricht man von einer Instanz der Sprache. Bei der Betrachtung der Instanz wird zusätzlich zwischen Struktur und Inhalt unterscheiden. Die Struktur setzt den Inhalt in einen Kontext.

Die einzelnen Komponenten von XML können sehr genau den Bestandteilen einer Sprache zugeordnet werden. Einer der wichtigsten Bestandteile von XML stellt das Schema dar. In dem Schema wird sowohl das Vokabular als auch die Grammatik der Sprache festgelegt. Obwohl es verschiedene Möglichkeiten gibt ein Schema zu definieren, wird bei der IEC61850 ein XML-Schema als Schemasprache verwendet. Aus diesem Grund wird in dieser Arbeit nur das XML-Schema beschrieben. Andere bekannte Schemasprachen sind RelaxNG, Schematron und DTD. Die entscheidenden Vorteile

[4]markup languages
[5]Standard Generalized Markup Language
[6]World Wide Web Consortium

von XML-Schema sind die weite Verbreitung, eine große Mächtigkeit beim Deklarieren, eine große Ansammlung von Standard-Datentypen und die Verwendung der XML-Syntax. Die Instanz der Sprache entspricht dem eigentlichen XML-Dokument. Die hohe Flexibilität beim definieren jedes einzelnen Bestandteils einer Sprache ist der Grund, warum sich XML als Sprache für den Austausch von Daten immer mehr durchsetzt.

3.2 XML Überblick

In diesem Abschnitt werden die für diese Arbeit grundlegenden Komponenten des XML Standards beschrieben. Dabei wird noch mal speziell auf die Zusammenhänge zwischen XML und XML-Schema eingegangen.

3.2.1 Wichtige Begriffe

Dokument und Schema

Ein XML-Dokument kann prinzipiell ohne ein Schema bestehen. Wird das Dokument nach einem Schema erstellt, instanziiert das Dokument das Schema. Für ein XML-Schema gelten dieselben Syntaxregeln wie für ein XML-Dokument.

Infoset

Die W3C definiert für jedes wohlgeformte XML-Dokument ein so genanntes Informations Set oder kurz Infoset. Das Infoset ist eine abstrakte Beschreibung des Informationsgehalts eines XML-Dokuments. Ein Infoset besteht seinerseits wieder aus elf Informationseinheiten. Jede dieser Informationseinheiten besitzt wiederum eine bestimmte Anzahl von Eigenschaften. Viele XML-Technologien und Standards setzen auf dem Infoset auf, da dadurch ein Analysieren und Interpretieren des Dokuments überflüssig wird. Auch XML-Schema baut auf dem Infoset auf, jedoch definiert dieser Standard noch ein zusätzliches Informations Set, mit dem ein Schema-Prozessor ein Dokument validieren kann. Dieses Informations Set heißt „Post-Schema-Validation Infoset“. Das Infoset wird häufig als Grundlage für ein Binärformat für XML-Dokumente verwendet. Die Probleme, die sich daraus ergeben, werden in Abschnitt 5.6 beschrieben.

XML- und Schema-Prozessor

Ein Prozessor nach W3C kann sowohl eine Hardware als auch eine Software beschreiben,welche ein XML-Dokument bearbeitet. Ein Schema-Prozessor

kann nach der Verarbeitung eines XML-Dokuments eine Aussage machen, ob das Dokument gegen ein bestimmtes Schema validierbar ist.

3.2.2 XML-Sprachelemente

Die Beschreibung der Sprachelemente setzt bereits Grundkenntnisse in XML und XML-Schema voraus. Es soll lediglich ein einheitlicher Sprachgebrauch für die Syntaxelemente, die möglicherweise Auswirkungen auf die Codierung haben, eingeführt werden. Um diesen Abschnitt nicht mit zu Anfang unnötiger Information zu überfüllen, werden spezielle Syntaxelemente erst bei ihrer Anwendung beschrieben.

Eine der wichtigsten Eigenschaften von XML-Schema ist die Definition von Namensräumen. Sie ermöglichen eine funktionelle und logische Unterteilung von XML-Schemata. Dabei spezifiziert jeder Namensraum ein Schema.

Bei der Typdeklaration wird zwischen einfachen und komplexen Datentypen unterschieden. XML-Schema besitzt bereits eine große Anzahl an atomaren einfachen Datentypen. Viele dieser Typen lehnen sich an die aus der Programmierung bekannten Typen wie Integer und String an. Durch einschränkendes Erben dieser Standardtypen können eigene einfache Typen deklariert werden. Komplexe Typen können wiederum selbst Elemente und Attribute enthalten, von denen die Elemente ebenfalls eine komplexe Typdeklaration besitzen können. Komplexe Elemente dienen der Strukturierung. Werden mehrere komplexe Typen ineinander verschachtelt, spricht man von einer tiefen Hierarchie.

Werden in einem komplexen Typen Elemente deklariert, müssen sich diese innerhalb eines `sequence`-, `choice`- oder `all`-Strukturelements befinden. Bei der Sequenz `sequence` müssen die Elemente in der Reihenfolge in der sie deklariert werden, auch instanziiert werden. Bei einer Auswahl kann nur eines der im Syntaxelement `choice` eingeschlossenen Elemente instanziiert werden. Bei einem `all`-Syntaxelement können alle Elemente in einer beliebigen Reihenfolge instanziiert werden.

Über die Schlüsselworte `minOccurs` und `maxOccurs` kann die minimale und maximale Häufigkeit von Syntaxelementen festgelegt werden. Eine Häufigkeit kann für folgende Syntaxelemente definiert werden: `sequence`, `choice`, `all`, `simpleType`, `complexType` und `element`. Wird eine der Häufigkeiten nicht explizit angegeben, wird für diesen der Wert `"1"` angenommen.

Bei der Zuweisung eines Typen zu einem Element gibt es zwei Möglichkeiten der Deklaration. Bei der ersten wird der Typ global deklariert und dem Element über den Zuweisungsoperator `=` zugewiesen. Als zweite Möglichkeit kann der Typ direkt innerhalb der Elementdeklaration deklariert werden. Dabei spricht man von einem anonymen Typen. Eine globale Typdefinition

hat den Vorteil, dass von dem Typen geerbt und der Typ bei nur einmaliger Deklaration mehrfach verwendet werden kann.

Bei der Vererbung von Typen kann entweder erweiternd oder einschränkend vererbt werden. Bei der erweiternden Vererbung mit dem Schlüsselwort `extension` erbt ein komplexer Typ das komplette Inhaltsmodell des Basistypen. Dieser Typ kann nun beliebig erweitert werden. Die Instanz dieses Typen stellt sich als eine Sequenz des neuen und alten Inhaltsmodells dar. Bei der einschränkenden Vererbung mittels `restriction` können lediglich die Häufigkeiten des Basistyps eingeschränkt und Vorgabewerte vorgegeben werden. Eine allgemeine Bedingung an die Vererbung ist, dass Polymorphie möglich ist.

Die Syntaxelemente `any`, `anyType` und `anyAttributes` werden häufig Wildcards genannt. Sie erfüllen den Zweck von Platzhaltern. An ihrer Stelle kann beim Instanziieren ein beliebiges Element bzw. Attribut eingefügt werden. Bei der Deklaration einer Wildcard kann festgelegt werden, ob das instanziierte Syntaxelement im gleichen oder in einem anderen Namensraum wie das Schema selbst deklariert ist.

Durch das Setzen des Attributs `xsi:type` in der XML-Schemainstanz eines Elements kann die Polymorphie seines Typen genutzt werden. Das auch als typecast bekannte Verfahren ist nur auf Typen anwendbar, welche den Basistypen für den neuen Typen bilden, siehe auch Vererbung.

3.3 Verarbeitung von XML

Eines der wichtigsten Merkmale von XML ist die Verarbeitbarkeit durch Maschinen. Hier haben sich ein paar grundlegende Programmierschnittstellen für den Zugriff auf die Daten durchgesetzt. Alle hier beschriebenen Methoden nutzen das Infoset eines Dokuments für den Zugriff auf die Informationen. Eine Programmierschnittstelle wird im weiteren Verlauf der Arbeit auch API[7] genannt.

DOM

Der DOM[8] baut auf der Vorstellung auf, dass ein XML-Dokument eine Baumstruktur besitzt. Das Element, welches alle anderen Elemente umschließt, bildet die Wurzel des Baums. Jedes direkt unterhalb der Wurzel instanziierte Element oder Attribut bildet im Baum Knotenpunkte, welche durch einen Pfad bzw. Ast mit der Wurzel verbunden sind. In diesem Kontext sind die Knoten Kinder des Wurzelknotens bzw. des Vaterknotens. Handelt es

[7] API = **A**pplikation **P**rogramming **I**nterface

[8] Document Object Model

sich bei den Kindern um Elemente von einem komplexen Typen, können diese Knoten wieder Vaterknoten darstellen. Auf diese Weise baut sich ein beliebig verzweigter Baum auf, in dem die Elemente mit einfachen Typen und die Attribute die Blätter bilden. Bei dem DOM handelt es sich um eine Programmierschnittstelle für objektorientierte Programmiersprachen. Beim parsen des XML-Dokuments wird für jedes Element ein Objekt angelegt. Jedes Objekt hat vordefinierte Methoden, mit denen man im Baum navigieren kann. Bei Aufruf der Methode `getParent()` wird z.B. ein Verweis auf das Objekt, welches für den Vaterknoten steht, zurückgegeben. Obwohl die DOM-Schnittstelle von der W3C standardisiert wird, entwickeln viele Anwender eigene DOM-Implementierungen, die auf die eigenen Bedürfnisse besser zugeschnitten sind. Die Vorteile des DOM sind bei bekannter Baumstruktur der schnelle und komfortable Zugriff auf die Daten. Der Nachteil ist der hohe Speicherbedarf des DOM. Da die Objekte bereits beim ersten parsen des Dokuments erzeugt werden, wird eine große Menge Speicher belegt.

SAX

SAX[9] ist eine Event-orientierte Programmierschnittstelle. Um auf den Inhalt eines XML-Dokuments zuzugreifen, wird der API zunächst die Datei übergeben. Die SAX-API ließt das XML-Dokument sequenziell in einer Richtung durch. Trifft es dabei beispielsweise auf ein Attribut, wirft die API ein definiertes Event. Dies kann ein Aufruf einer Funktion der Anwendung sein, welcher der Attributname und Wert als Übergabeparameter gegeben wird. Weitere Funktionsaufrufe müssen z.B. für ein öffnendes bzw. schließendes Element und deren Inhalt vorgesehen werden. Nach dem Funktionsaufruf verwirft die SAX-API die geparsten Werte. Dadurch ist das SAX Verfahren ein sehr speichereffizienter Zugriff auf das XML-Dokument. Der große Nachteil ist jedoch der unflexible Zugriff auf den Inhalt eines XML-Dokuments. Je nachdem wie weit Hinten ein Element bzw. Attribut im Dokument instanziiert wird, ergeben sich unterschiedlich lange Zugriffszeiten.

Pull-API

Eine vereinfachte Schnittstelle gegenüber der SAX-Schnittstelle stellt die Pull-API dar. Im Gegensatz zu der SAX-API ruft nicht die Schnittstelle selbstständig eine Funktion auf, sondern der Anwender ruft eine Funktion auf, woraufhin die Pull-API erst das nächste Element parst. Dies erhöht zwar den Aufwand für die Anwendung, jedoch liegt auch der Kontrollfluss auf ih-

[9] Simple API for XML

rer Seite. Die Pull-API bietet Funktionen zum Abruf des nächsten Elements, Attributs und Inhalts.

4 Der Systemstandard IEC61850

Durch den Systemstandard IEC61850 „Kommunikationsnetze und -systeme in Stationen“ werden an die Stationsleitsysteme neue Anforderungen gestellt. So wird im Rahmen der IEC61850 auch die Konfiguration der intelligenten Baugruppen neu beschrieben. Da sich diese Diplomarbeit mit der Konfiguration von intelligenten Geräten befasst, wird in diesem Kapitel ein Überblick über die für die Diplomarbeit relevanten Teile der Norm gegeben. Gleichzeitig wird auf die sich aus der Norm ergebenden Problematiken eingegangen.

Ein grundlegendes Anliegen der Norm ist es, die Interoperabilität zwischen Geräten verschiedener Hersteller zu ermöglichen. Um nicht vollständige Gerätefunktionalitäten festzulegen, werden in der Norm logische Teilfunktionen spezifiziert. Diese logischen Funktionen stellen Objekte dar, welche über definierte logische Schnittstellen mit anderen Funktionen kommunizieren. Algorithmen dieser logischen Funktionen werden in der Norm nicht spezifiziert. Ein standardisierter Umfang von logischen Funktionen wird in Teil 7 der Norm beschrieben. Das Verhalten und die Schnittstellen der logischen Funktionen ist für diese Arbeit nicht von Interesse.

Teil 6 beschreibt die Konfiguration von Leitsystemkomponenten in Form von intelligenten Baugruppen (IEDs). Es wurde dazu mit den in Kapitel 3 beschriebenen Komponenten von XML eine eigene Sprache, die so genannte SCL, für die Konfigurierung entworfen. Teil 6 der IEC61850 beschreibt das dazu verwendetet Schema und weitere Regeln der Sprache, welche mit Hilfe eines XML-Schemas nicht beschrieben werden können. Zusätzlich zur Sprache wird auch der angestrebte Einsatz der SCL beschrieben.

4.1 Das Objektmodell der SCL

Zu Beginn des Teil 6 der Norm wird das Objektmodell beschrieben, dessen Objekte sich auf drei Objektgruppen aufteilen. Der Zweck dieses Modells ist es, der SCL eine Semantik zu geben. Um eine vollständige Abhängigkeit zwischen der Semantik und der Syntax zu vermeiden, beziehen sich die Objekte des Modells und ihre Beziehungen bewusst nicht auf die tatsächlichen Elemente und Attribute der SCL. Somit haben syntaktische Änderungen, wie z.B. ein zusätzliches Attribut, keine Auswirkung auf die grundlegende Semantik der SCL. Die drei Objektgruppen werden mit Hilfe der Abbildung 1 beschrieben und ihre Semantik in Abschnitt 4.3 den Syntaxelementen zugeordnet. Die Abbildung 1 zeigt ein UML[10] Klassendiagramm. Dieses Klassendiagramm entspricht weitestgehend der Abbildung des SCL-Objektmodells

[10]Unified Modeling Language

der IEC61850-6. Zum besseren Verständnis wurden folgende Änderungen vorgenommen:

- Die Objekte Router und Clock werden der Objektgruppe Kommunikation zugeteilt
- Das Objekt IED enthält nur das Objekt Access Point, dargestellt durch eine Agreggation
- Es werden keine Kurzformen für die Objektnamen verwendet

Bei den Objektnamen ist darauf zu achten, dass diese nicht auf tatsächliche SCL Elementnamen verweisen. Die Klassendiagramme, welche anhand des tatsächlichen SCL-Schemas erstellt wurden, unterscheiden sich von dem Klassendiagramm des Objektmodells aus Abbildung 1. Bestimmte Objekte

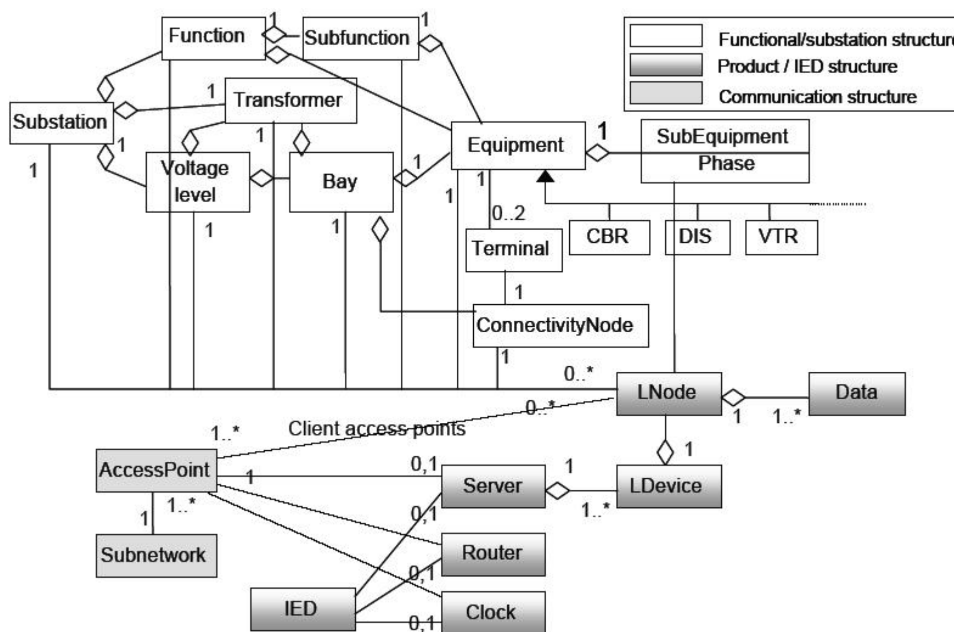

Abbildung 1: SCL Objektmodell

treten in mehr als einer der drei Objektgruppen auf, weswegen sie in diesem Abschnitt Verbindungsobjekte genannt werden. Die Verbindungsobjekte werden der Objektgruppe zugeordnet, in der sie die größte Funktionalität besitzen.

Mit der ersten Gruppe wird die Primärtechnik der Unterstation beschrieben. Die Gruppe enthält Objekte, die den realen Bauteilen einer Station entsprechen. Durch die Abbildung der realen Bauteile in der SCL ist es möglich, logische Funktionen den Bauteilen zuzuordnen. Die logische Funktion, die als logical Node bezeichnet wird, ist das Verbindungsobjekt zur zweiten Objektgruppe. Für ein IED spielen die Objekte der Unterstation eine untergeordnete Rolle.

Die zweite Objektgruppe beinhaltet Objekte zur Beschreibung der intelligenten Baugruppen. Da sich diese Arbeit mit der Binärformatierung der Konfigurationsdaten eines IEDs befasst, ist die Gruppe zur Beschreibung des IED Objektmodells von großem Interesse. Das oberste Objekt in der Hierarchie ist das IED selbst. Das IED enthält einen Access Point, welcher über den Server auf ein logical Device zugreift. Das logical Device enthält logical Nodes, die wiederum ihre Daten, dargestellt als DATA, enthalten. Ein logical Node ist die kleinste Einheit, welche noch Kommunikationsfähigkeiten besitzt. Es kann direkt über einen Access Point auf andere logische Funktionen zugreifen. Der Access Point ist das Verbindungsobjekt zur dritten Gruppe.

4.3. Die dritte Objektgruppe beschreibt die Kommunikation einer Station. Die Objektgruppe der Kommunikation ist im Gegensatz zu den anderen beiden Gruppen nicht hierarchisch strukturiert. Mit den Objekten der Kommunikation wird beschrieben, wie die verschiedenen Baugruppen über ein Netzwerk kommunizieren. Damit intelligente Baugruppen miteinander kommunizieren können, muss ihr Access Point demselben Subnetwork angehören.

4.2 Die Dateiformate

Es werden vier Dateiformate für die unterschiedlichen Einsatzzwecke der Beschreibungssprache spezifiziert. Mit Hilfe der Abbildung 2 werden die Dateiformate den von der Norm spezifizierten Softwarekomponenten zugewiesen. Abbildung 2 zeigt ein Referenzmodell aus der IEC61850-6 für den Informationsfluss bei der Konfigurierung.

ICD

Die *IED Capability Description* beschreibt Fähigkeiten, die eine intelligente Baugruppe besitzt. Dabei bestehen keine Verbindungen zwischen logischen Funktionen der Baugruppe und den Komponenten der Primärtechnik. Ausnahmen bilden gewisse Vorkonfigurierungen der intelligenten Baugruppen. Die ICD-Datei wird an den System-Konfigurator übergeben.

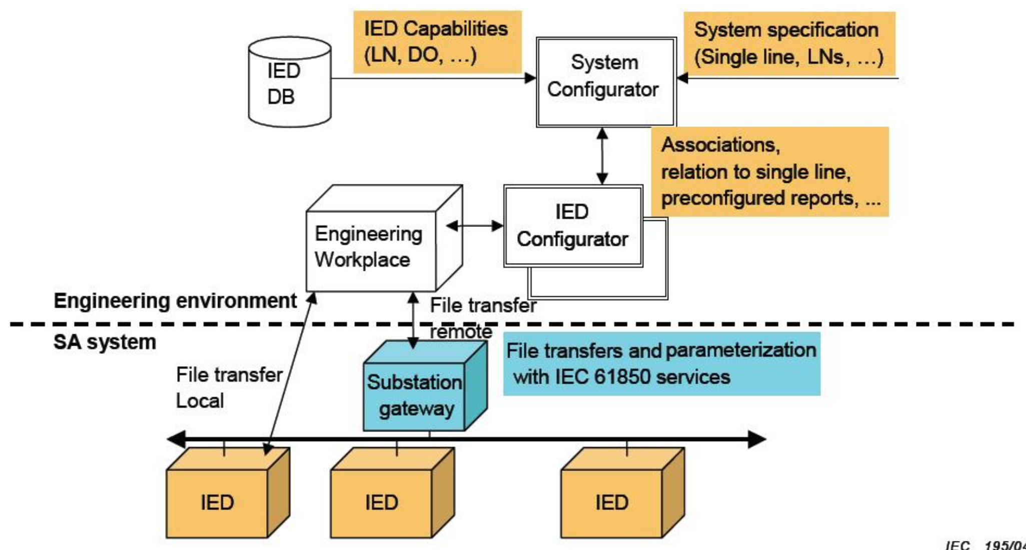

Abbildung 2: Referenzmodell für den Informationsfluss bei Konfigurierung

SSD

Die *System Specification Description* beschreibt die Unterstation textuell in einem single-line Diagramm[11]. In dieser Beschreibungsdatei werden bereits logische Funktionen der Sekundärtechnik den einzelnen Komponenten der Primärtechnik zugeordnet. Chronologisch steht dieses Dateiformat am Anfang der Planungsphase. Nachdem diese Datei beispielsweise von einem EVU[12] erstellt wurde, kann diese Datei zum Zweck der Realisierung, an die zuständige Firma übergeben werden. Die SSD-Datei wird an den System-Konfigurator übergeben.

SCD

Die *Substation Configuration Description* ist die vollständige Beschreibung einer konfigurierten Unterstation. Sie enthält die Beschreibung aller konfigurierten intelligenten Baugruppen, der Kommunikation und der Unterstation. Diese Datei wird vom System-Konfigurator erstellt. Dazu teilt der System-Konfigurator die logischen Funktionen aus der SSD-Datei mit Hilfe der ICD-Datei auf die Baugruppen auf. Zusätzlich werden noch Daten, z.B. für die Kommunikation, generiert. Anschließend wird die SCD-Datei an den IED-Konfigurator übergeben.

[11] einpolige Schaltplandarstellung
[12] Energieversorgungsunternehmen

CID

Die *Configured IED Description* wird aus der SCD-Datei extrahiert und beschreibt die endgültige Konfigurierung der intelligenten Baugruppe. Im Gegensatz zur ICD-Datei enthält die CID-Datei zusätzliche Informationen bezüglich der Kommunikation und der Position in der Anlage, in der sich die Baugruppe befindet. Die CID-Datei enthält alle Informationen, die eine intelligente Baugruppe zum Betrieb benötigt. Die Notwendigkeit einer resourcenschonenden Verarbeitung dieses Dateiformats ist der Grund, warum in diese Arbeit die Eignung einer Binärformatierung für Konfigurationsdaten untersucht wird. Natürlich kann die Binärformatierung auch für andere XML-Dokumente in der Leittechnik eingesetzt werden. Die IEC empfiehlt bei der Definition der CID-Datei, dass bei Einsatz eines Komprimierverfahrens für eine CID-Datei ein Verfahren gemäß RFC1952[13] verwendet werden sollte. Der Grund, warum sich die so genannte GZIP-Komprimierung nur schlecht eignet, wird in Kapitel 5 beschrieben. Das hier ausgewählte Verfahren verstößt jedoch nicht gegen die Intention dieser Empfehlung. In Abschnitt 5.1 wird beschrieben, dass die CID-Datei als XML-Dokument importiert und exportiert werden kann.

4.3 Die SCL-Syntaxelemente

Die Bedeutung der Syntaxelemente z.B. für die Applikation ist von Interesse, da ein Binärformat unter Umständen bestimmte Syntaxelemente bereits vorverarbeiten kann. Ebenso sind gewisse Regeln und Einschränkungen für die SCL nur schriftlich verfasst. In diesem Abschnitt werden ebenfalls nur die Syntaxelemente beschrieben, die für diese Arbeit von Interesse sind.

4.3.1 Aufbau des Schemas

Das SCL-Schema teilt sich auf acht Schemadateien auf, die in Tabelle 1 mit einer Kurzbeschreibung aufgelistet sind. Diese Schemadateien besitzen den gleichen Namensraum und binden sich gegenseitig über die Schemaanweisung `<include>` ein. Für einen W3C konformen XML-Prozessor stellen sich die acht Schemadateien als ein Schema dar. Die ersten drei Dateien beinhalten Typen, die an semantisch unterschiedlichen Stellen im Schema häufig wieder verwendet werden. Die folgenden drei Schemadateien beinhalten jeweils die Syntaxelemente zur Beschreibung der erwähnten Objektgruppen. Die Schemadatei SCL_DataTypeTemplates stellt Syntaxelelemente für eine vierte Objektgruppe zur Verfügung. Diese Gruppe wurde nicht im Objektmodell dar-

[13]Request for Comments

Dateiname	Bedeutung
SCL_Enums.xsd	Alle Enumerationen der SCL
SCL_BaseSimpleTypes.xsd	Alle einfachen Basistypen der SCL
SCL_BaseTypes.xsd	Alle komplexen Basistypen der SCL
SCL_Substation.xsd	Deklarationen zur Beschreibung Unterstation
SCL_Communication.xsd	Deklarationen zur Beschreibung der Kommunikation
SCL_IED.xsd	Deklaration zur Beschreibung des IEDs
SCL_DataTypeTemplates.xsd	Deklarationen zur Typisierung der IED-Beschreibung
SCL.xsd	Zusammenfügen der Schmadateien u. Deklaration des Elements `SCL`

Tabelle 1: Aufteilung des SCL-Schemas auf verschiedene Schemadateien und ihre Bedeutung

gestellt, da sie für die Verwendung der SCL nicht notwendig ist, sondern nur ihre Anwendbarkeit verbessert. In den folgenden Absätzen werden wichtige XML-Elemente den jeweiligen Objekten des Objektmodells zugeordnet. Einige Elemente werden im Hinblick auf die Repräsentation im Binärformat genauer beschrieben. Zusätzlich wird die Verwendung der XML-Elemente aus der Schemadatei SCL_DataTypeTemplates im vorletzten Absatz beschrieben.

SCL_Substation

Die Syntaxelemente in dieser Schemadatei werden in der SSD- und in der SCD-Datei instanziiert. Der Einstieg zur Beschreibung der Unterstation ist das Element `Substation`. Da jedoch die Beschreibung der Unterstation eine eher untergeordnete Rolle für dies Arbeit spielt, wird nur auf die Anbindung zur Beschreibung des IEDs eingegangen.

Das Element, welches den logical Node darstellt, hat den Namen `LNode`. Alle XML-Elemente, die primärtechnische Bauteile einer Unterstation darstellen, besitzen den Basistypen `tLNodeContainer`, der wiederum `LNode` in seinem Inhaltsmodell enthält. Daraus folgt, dass jedem Bauteil einer Unterstation ein oder mehrere `LNode` zugewiesen werden können. Das Element `LNode` stellt während der Planungsphase einer Anlage in der SSD-Datei zunächst einen Platzhalter dar. Bei der Realisierung wird das Element `LNode` in der SCD-Datei als Referenz auf das Element, welches dem logical Node entspricht, genutzt. Zur Referenzierung enthält das Element `LNode` sechs Attribute, mit denen beschrieben wird, in welcher intelligenten Baugruppe sich der logical Node befindet. Ein logical Node darf nur einmal durch ein Ele-

ment `LNode` referenziert werden. Eine solche Referenzierung sieht wie folgt aus:

```
<LNode lnInst="1" lnClass="CILO" ldInst="C1"
                                           iedName="D1Q1SB4"/>
```

In diesem Beispiel sind vier Attribute ausreichend. Die Identifikation eines referenzierten Elements erfolgt jeweils über seine Attribute. So muss der referenzierte logical Node sich innerhalb eines Elements `IED` befinden, welches in seiner Instanz ein Attribut `Name` mit dem Wert `"D1Q1SB4"` enthält. Das Element, welches den logical Device darstellt, muss in seiner Instanz ein Attribut `Inst` mit dem Wert `"C1"` besitzen. Das Attribut `lnClass="CILO"` sagt aus, dass das logical Node eine Verriegelungslogik darstellt. Der referenzierte logical Node besitzt das gleiche Attribut. Die Bedeutung dieser Kombination aus vier Buchstaben geht aus einem anderen Teil der Norm hervor. Da ein logical Node mit derselben `lnClass` innerhalb eines logical Device mehrfach auftreten kann, wird zusätzlich noch die Nummer der Instanz angegeben. Im Beispiel handelt es sich um die erste Instanz eines `"CILO"`. Alle Referenzierungen in der SCL nutzen diese Art der Referenzierung über Attribute.

SCL_Communication

Die Syntaxelemente dieser Schemadatei werden in der ICD-,SCD- und CID-Datei instanziiert. Das Element `Communication` ist der Einstiegspunkt für die Beschreibung der Kommunikation. Es enthält das Element `SubNetwork`, welches wiederum das Element `ConnectedAP` enthält. Das Element `ConnectedAP` stellt das Objekt Access Point dar. Es enthält eine Referenz auf das Element, welches den Access Point in der SCL_IED.xsd darstellt. Für die Referenz werden zwei Attribute deklariert. Die Bedeutung und Anwendung dieser Form der Referenzierung wurde bereits im vorhergehenden Absatz beschrieben. Zusätzlich beschreibt der `ConnectedAP` Eigenschaften, wie die Adresse des referenzierten Accesspoints aber auch die physikalische Steckverbindung der Datenleitung zum Gerät. Eine Besonderheit stellt das Element `P` der Kommunikation dar. Das Element `P` ist Teil des Elements `Adress` und ist speziell für einen typecast in verschiedene Darstellungsformen einer Adresse aus der Kommunikation ausgelegt, z.B. für eine IP-Adresse. Diese explizite Typzuweisung während der Instanziierung ermöglicht es einem XML-Prozessor, eine Adresse auf ihr eingeschränktes Format hin zu prüfen.

SCL_IED

Die Syntaxelemente dieser Schemadatei werden in der ICD-, SCD- und CID-Datei instanziiert. Das Element `IED` bildet den Einstiegspunkt für die Be-

schreibung einer intelligenten Baugruppe. Das Element `IED` bindet das Element `AccessPoint` ein, welches entweder direkt das Element `LN` oder das Element `Server` einbindet. Das Objekt logical Node, welches durch das Element `LN` dargestellt wird, weißt ebenfalls eine hierarchische Struktur auf. Ein Element `LN` kann ein `DOI` und `SDI` enthalten, welche wiederum das Element `DAI` enthalten. Während die Elemente `DOI` und `SDI` hauptsächlich der Strukturierung dienen, stellt eine „Data Attribute Instance“, kurz `DAI`, einen tatsächlichen Wert. Zur Darstellung eines Wertes enthält das Element `DAI` Attribute zur Beschreibung des Wertes und ein Element `Val`, in dem der Wert abgelegt wird. Für eine intelligente Baugruppe, die ihre Systemarchitektur an der IEC61850 anlehnt, stellt die Instanz des Elements `LN` die Beschreibung einer bestimmten Funktionalität dar. Da die Binärformatierung für Konfigurationsdateien unter anderem den Zugriff auf Daten optimieren soll, kann es sinnvoll sein, das Binärformat an die Elemente bzw. Funktionalitäten anzupassen. Zusätzlich beinhaltet das Element Attribute, mit denen es auf das Element referenziert, welche den logical Node in der SCL_DataTypeTemplates.xsd darstellen.

SCL_DataTypeTemplates

Die Syntaxelemente in dieser Datei dienen der Typisierung des Elements `LN` und damit auch dessen Kindelementen. Das Element `LNodeType` stellt eine Art Schablone für den Aufbau des tatsächlich verwendeten Elements `LN` dar. Innerhalb dieser Schemadatei sind die Elemente, welche die Schablonen für ein Kindelement des Elements `LN` darstellen, ebenfalls ausgelagert und zu Typen zusammengefasst. Dies hat den Vorteil, dass diese Elemente ebenfalls typisiert werden können und über Referenzen eingebunden werden. Durch die Typisierung wird die Dateigröße verringert, Redundanzen vermieden und eine bessere Datenkonsistenz erreicht. Hierdurch entsteht eine große Anzahl an Referenzen. Die folgenden Auszüge eines SCL-Dokuments verdeutlichen die Funktionsweise der Typisierung.

```
<LNodeType id="CILOa" lnClass="CILO">
  <DO name="Mod" type="myHealth"/>
  <DO name="Beh" type="myBeh"/>
  <DO name="Health" type="myINS"/>
  <DO name="EnaOpen" type="mySPS"/>
  <DO name="EnaClose" type="mySPS"/>
</LNodeType>

<DOType id="myINS" cdc="INS">
  <DA name="stVal" fc="ST" bType="INT32" dchg="true"/>
```

```
</DOType>
```

Als Beispiel wurde ein Template für den logical Nodes `lnClass="CILO"` verwendet. Dieses Template enthält die Elemente `DO`, welche den `DOI` im `LN` beschreibt. Das Element `DO` mit dem Namen `"Health"` referenziert das Element `DOType` mit dem Attribut `id="myINS"`. Das Element `DOType` beinhaltet ein Element `DA`, welches den `DAI` beschreibt. Zusätzlich kann auch der Wert eines `DAI` typisiert werden, indem innerhalb des Elements `DA` das Element `Val` instanziiert wird. Durch die Typisierung eines Wertes kann es jedoch vorkommen, dass zwei unterschiedliche Schemata nach der IEC61850 zwar semantisch gleich sind, jedoch sich nach XML-Gesichtspunkten syntaktisch unterscheiden. Dies tritt auf, wenn in einem Schema ein `LN` direkt nur unterhalb des Elements `IED` instanziiert wird, in dem anderen Schema der gleiche `LN` unterhalb des Elements `IED` nur sein Template referenziert. Da nach XML-Gesichtspunkten keine Verbindung zwischen `LN` und `LNodeTyp` besteht, handelt es sich für einen XML-Prozessor um zwei unterschiedliche Instanzen.

SCL

In dieser Schemadatei wird das Wurzel-Element `SCL` deklariert. Dieses Element bildet das Wurzel-Element für alle SCL-Dateien. Im Element SCL werden die vier zuvor beschriebenen Gruppen eingebunden. Zusätzlich wird das Element `Header` deklariert, welches der Identifikation der Kofigurationsdatei dient.

4.3.2 Besondere Merkmale und Bedeutungen des Schemas

Typhierarchie

Das SCL-Schema besitzt eine sehr tiefe Typhierarchie. Dies zeigt sich unter anderem daran, dass nahezu alle Element vom Typen `tBaseElement` erben. Dieser Typ legt die grundlegende Eigenschaft aller erbenden Elemente fest, private Erweiterungen zuzulassen. Die privaten Erweiterungen werden im folgenden Absatz beschrieben.

Private Erweiterungen

Die IEC61850 bietet zwei Möglichkeiten einer SCL-Instanz zusätzliche private Information hinzuzufügen. Die dafür verwendeten Wildcards sind im Typ `tBaseElement` und im Element `Private`, welches wiederum im Typ `tBaseElement` deklariert wird, deklariert. Zur Klärung der Begrifflichkeiten:

Der Begriff „private Erweiterung“ beschreibt nicht ausschließlich Erweiterungen mit dem Element `Private`, sondern beschreibt allgemein Erweiterungen der Anwender der SCL. Ist das Element in der SCL gemeint wird immer `new Courier` als Schrift verwendet.

Durch die erste Wildcard ist es möglich, private Elemente und Attribute am Anfang jedes von `tBaseElement` erbenden SCL-Elements einzufügen. Dabei müssen die privaten Erweiterungen einem anderen Namensraum angehören als die SCL-Elemente. Der Namensraum sollte zu Beginn des Elements `SCL` deklariert werden. Bei Verwendung eines Präfixes für den Namensraum soll dessen Name mit einem `e` anfangen. Bei diesen Erweiterungen kann nicht davon ausgegangen werden, dass bei der Verarbeitung durch Dritte die Erweiterungen in ihrer ursprünglichen Form bestehen bleiben.

Bei Erweiterungen innerhalb des `Private` Elements ist das Verändern, also auch Löschen, von Erweiterungen anderer Hersteller verboten. Das `Private` Element kann in jedem von `tBaseElement` erbenden SCL-Element instanziiert werden. Es kann Text, XML-Elemente oder beides enthalten. Beim Einfügen von Elementen bestehen die gleichen Einschränkungen wie beim Verfahren zuvor. Das `Private` Element besitzt zwei Attribute. Mit dem Attribut `type` wird unter anderem gekennzeichnet, welcher Firma dieser private Bereich gehört, indem der Anfang der Zeichenkette beispielsweise den Namen der Firma trägt. Durch das zweite Attribut `source` kann über eine URL eine Erweiterungsdatei adressiert werden. Diese Erweiterungsdatei gilt dabei als nicht geschützt und muss von Fremdherstellern nicht übernommen werden. Das heißt, dass ein Hersteller, der für den Betrieb seiner Baugruppe eine Erweiterungsdatei benötigt, selber dafür sorgen muss, dass diese Datei auf der Baugruppe vorhanden ist.

4.4 Grade der Konformität

Die IEC definiert drei Grade der Konformität, SCL.1, SCL.2 und SCL.3. Bei einer theoretischen Prüfung auf Eignung einer Binärformatierung ist es sinnvoll, das Formatierungsverfahren auf den höchsten Grad an Konformität zu prüfen. Die für diese Arbeit entscheidende Anforderung ist in Teil 8-1 der Norm, in Anhang D, Tabelle D.1 zu finden. Für den Konformitätsgrad SCL.3 wird folgendes definiert: „Unterstützung der Implementierung einer SCL-Datei und Rekonfigurierung durch diese im Betrieb“. Um diese Forderung optimal zu erfüllen, muss die CID-Datei als XML-Dokument importiert und exportiert werden können, ohne den laufenden Betrieb zu stören.

5 Das Binärformat

Viele Nachteile eines XML-Dokuments können durch eine Binärformatierung kompensiert werden. Der Vorteil des dabei entstehenden Binärformats ist in erster Linie der kleinere Speicherbedarf der Daten. Ebenso kann ein Binärformat eine Reihe von zusätzlichen Funktionalitäten unterstützen. Als Nachteil einer Binärformatierung wird sicherlich häufig der Verlust der Lesbarkeit eines XML-Dokuments genannt. Dieses Argument wird sich im Laufe der Arbeit jedoch relativieren, da genau wie bei einem XML-Dokument der Zugriff auf das Binärformat mit einem Anwenderprogramm möglich sein wird. Der Unterschied in der Lesbarkeit ist lediglich die Möglichkeit, ein XML-Dokument bereits mit einem einfachen Text-Editor interpretieren zu können.

Im folgenden Abschnitt werden die Eigenschaften beschrieben, die eine Binärformatierung unterstützen sollte. Anschließend werden im darauf folgenden Abschnitt verschiedene, bereits bekannte Verfahren auf ihre Eignung geprüft.

5.1 Anforderungen an das Binärformat

Die sicherlich grundlegendste Anforderung an ein Binärformat ist ein geringer Speicherbedarf. Trotz steigender Speicherkapazitäten der Geräte ist diese Anforderung nicht zu vernachlässigen. Durch eine signifikante Speicherersparnis können Speicherbausteinen mit geringeren Kapazitäten verwendet werden, was eine Kostenersparnis im Einkauf bedeutet. Ein weiterer Aspekt ist eine optimierte Übertragung der Konfigurationsdaten. Obwohl heutzutage hohe Übertragungsraten zur Verfügung stehen, werden in der Leittechnik zugunsten einer höheren Sicherheit nur geringe Bandbreiten genutzt. Beispielsweise kann es sein, dass die Konfigurierung eines IED nicht vor Ort geschieht, sondern über einen schmalbandigen Übertragungskanal. Bei einem geringen Datenvolumen des Binärformats können demnach mehr Nutzdaten pro Zeit gesendet werden bzw. gleiche Nutzdaten in einer kürzeren Zeit.

Aufgrund der schmalen Bandbreite der Übertragungskanäle wird auch die Anforderung gestellt, das Binärformat in Fragmente aufteilen zu können. Dies hat den Vorteil, dass bei Änderungen innerhalb eines Fragments nicht die ganze Datei gesendet werden muss, sondern nur das aktualisierte Fragment. Zusätzlich sollten die Größe und die Grenzen eines Fragments frei wählbar sein, damit bei Änderungen nur ein kleinstmögliches Fragment übertragen werden muss. Die Nutzbarkeit und Anwendbarkeit der Fragmentierung wird im Kapitel 9 ausführlich beschrieben.

Da aus Speicherplatzgründen eine vollständige Decodierung des Binärformats auf dem Gerät nicht möglich ist, sollte es möglich sein direkt auf das

Binärformat zuzugreifen. Ein direkter Zugriff bedeutet, dass zunächst innerhalb des Binärformats an die gesuchte Stelle navigiert werden kann um anschließend auf den Inhalt zuzugreifen. Das Gegenteil bedeutet, dass ein Binärformat vom Anfang der Datei an solange decodiert werden muss, bis die gesuchte Information gefunden wird. Um in einem fragmentierten Binärformat navigieren zu können, muss es eine Möglichkeit geben, die Fragmente zu identifizieren. Damit ein solcher Zugriff auf die Daten möglichst einfach ist, sollte sich die Strukturierung der Daten an die Struktur eines XML-Dokuments anlehnen. Dies hätte den Vorteil, dass der Zugriff auf ein zusätzliches Format nicht extra dokumentiert werden müsste. Ist ein Zugriff auf das Binärformat nicht möglich, muss auf jeden Fall die Fragmentierung gewährleistet sein. Dadurch muss dann jeweils nur ein Fragment vollständig dekodiert werden.

Der De- und Encoder des Verfahrens soll möglichst wenig Rechenleistung benötigen. Da sowohl Decoder als auch Encoder auf den Geräten während der Laufzeit zum Einsatz kommen, darf das Codieren nicht andere Prozesse beeinträchtigen.

Da die IEC61850 einen Export der CID-Datei als XML-Dokument vorsieht, muss bei der Decodierung ein dem ursprünglichen XML-Dokument entsprechendes Dokument erzeugt werden können.

Ebenso soll es möglich sein eine CID-Datei zu importieren. Da die Handhabung von XML-Dokumenten auf den Baugruppen problematisch ist, sollte das Dokument beim Einlesen bzw. Rausschreiben direkt en- bzw. decodiert werden können. Der Fachbegriff hierfür ist „on-the-fly“-Kompression.

Bei der Auswahl des Binärformats muss abgewogen werden wie wichtig die einzelnen Anforderungen sind. Unter Umständen kann es vorkommen, dass auf die Erfüllung einer Anforderung zu Gunsten einer anderen verzichtet wird, z.B. könnten Einbußen bei der Kompression zu Gunsten der Fragmentierbarkeit eingegangen werden. Die Anforderungen werden noch mal zusammenfassend aufgelistet:

- Hohe Kompression für die Übertragung und Speicherung des Binärformats
- Aufteilung des Binärformats in Fragmente
  - Fragmentgrenzen frei wählbar
  - Identifikationsmöglichkeit für Fragmente
  - Fragmente einzeln übertragbar
- Zugriff auf das Binärformat in komprimierter Form

- Geringer Rechenaufwand beim En- und Decodieren
- Genau Wiedergabe des Ursprungsdokuments beim decodieren
- Komprimierung während der Übertragung (on-the-fly)

5.2 Grundlegende Komprimierungsverfahren

Bei der Komprimierung des XML-Dokuments gibt es drei grundlegende Codierverfahren. Häufig werden allgemeine Codierverfahren verwendet, die auch zur Komprimierung anderer Datenformate verwendet werden können. Eines der bekanntesten allgemeinen Codierverfahren ist GZIP, welches speziell für die Komprimierung von Textdateien, wie es auch XML-Dateien sind, geeignet ist.

Die anderen zwei Gruppen von Codierverfahren sind speziell auf XML abgestimmte Verfahren. Die Verfahren unterscheiden sich in der Tatsache, dass es sich um syntaxfreie und um syntaxbasierte Verfahren handelt. Im Fall von XML bedeutet syntaxbasiert, dass das zugrunde liegende Schema der Instanz zur Codierung verwendet wird. Hier drin liegt auch der Nachteil der syntaxbasierten Codierverfahren. Für eine korrekte Decodierung muss das gleiche Schema vorliegen wie bei der Encodierung. Syntaxfreie Codierverfahren hingegen benötigen für die Codierung mehr Zeit und erreichen nicht die Strukturkompression eines syntaxbasierten Verfahrens.

In den folgenden Abschnitten werden verschiedene Codierverfahren vorgestellt, und anhand der Anforderungen aus Abschnitt 5.1 auf ihre Eignung geprüft.

5.3 Allgemeine Codierverfahren

5.3.1 LZ77-Algorithmus und GZIP

Der LZ77-Algorithmus ist einer der am häufigsten eingesetzten Komprimierverfahren in der Datenverarbeitung. Das Verfahren lässt sich auf alle Datenformate anwenden, ist jedoch besonders für Texte geeignet. Im Laufe der Zeit haben sich viele verschiedene Komprimierverfahren durchgesetzt, die im Prinzip den LZ77-Algorithmus verwenden. Das bekannteste ist das GZIP-Verfahren nach RFC1952. Da die meisten Codierverfahren entweder auf einem GZIP-Verfahren aufbauen oder ihn zum Teil verwenden, wird der LZ77-Algorithmus ausführlich beschrieben. Anschließend wird auf das, von der IEC61850 zur Komprimierung von CID-Dateien empfohlene, GZIP-Verfahren zur Komprimierung von CID-Dateien eingegangen.

Das Ziel des LZ77-Algorithmus ist es durch zusammenfassen von Redundanzen innerhalb des Datensatzes eine hohe Komprimierung zu erreichen. Der Algorithmus sucht beim Parsen der zu codierenden Datei in Vorwärtsrichtung die längste Zeichenkette, die bis dahin schon einmal aufgetreten ist, und schreibt eine Referenz auf die vorhergegangene Zeichenkette. Anhand des folgenden Beispiels wird dies nochmal genauer beschrieben. Die folgende Zeichenkette soll codiert werden:

```
<Substation name="IEC61850-Station"></Substation>
```

Der Algorithmus erkennt, dass die Zeichenkette „Substation“ bereits 38 Zeichen zuvor (inkl. Leerzeichen) aufgetreten ist und codiert eine Referenz auf diese. Ob es sich hierbei um Worte oder um aneinander gereihte Zeichen handelt, spielt für den LZ77-Algorithmus keine Rolle. Die Referenz verweißt relativ von ihrer Position aus auf das Referenzziel und auf die Länge der redundanten Zeichenkette. Das Beispiel würde dem entsprechend so encodiert:

```
<Substation name="IEC61850-Station"></(38,10)>
```

Trifft der Algorithmus beim decodieren auf die Referenz, löst er sie auf und setzt die Zeichenkette an deren Stelle ein. Die Effektivität der Komprimierung steigt mit der Anzahl der Redundanzen. Damit die Komplexität beim Erkennen der Redundanzen nicht zu groß wird, wird häufig die maximale Grenze eingeschränkt, bis zu der eine Redundanz rückwärts gesucht wird.

Das GZIP Verfahren kombiniert den LZ77-Algoritmus und die Huffmanncodierung. Nachdem die Redundanzen mit Hilfe des LZ77-Algorithmus reduziert wurden, wird die Huffmanncodierung angewendet. Diese Kombination der Verfahren wird auch deflate[14]-Algorithmus genannt. Im Gegensatz zum WINZIP-Verfahren, werden beim GZIP keine Container mit beliebig vielen Dateien angelegt, sondern nur einzelne Dateien komprimiert. Die folgende Bewertung bezieht sich auf die Komprimierung von XML-Parametersätzen mit Hilfe des GZIP-Verfahrens.

Bewertung

Beim GZIP-Verfahren stehen Flexibilität und eine schnelle Codierung einer hohen Kompressionsrate gegenüber. Wird ein großes Suchfenster zum Finden der Redundanzen gewählt, steigt die Kompressionsrate. Auf der anderen Seite jedoch steigt die Komplexität beim Codieren und somit auch die Rechenzeit. Da die Struktur des XML-Dokuments beim Komprimieren verloren geht, ist kein einfacher Zugriff auf den Inhalt möglich. Das Binärformat

[14] reduzieren, Luft ablassen

muss für den Zugriff vollständig decodiert werden. Dies widerspricht der ursprünglichen Begründung für ein Binärformat, da ressourcenarme Baugruppen wieder ein vollständiges XML-Dokument verarbeiten müssten. Um das Binärformat in Fragmente unterteilen und übertragen zu können, muss das XML-Dokument in Fragmente zerlegt werden. Dabei sinkt jedoch die Anzahl der Redundanzen und somit die Kompressionsrate. Ein großer Vorteil des GZIP-Verfahrens hingegen ist seine universelle Einsetzbarkeit. Die Version des Schemas oder der Kontext, in dem das Dokument steht, spielen keine Rolle.

5.4 XML optimierte syntaxfreie Codierverfahren

5.4.1 XMLZIP

XMLZIP ist eines der ersten Komprimierverfahren, welches an den speziellen Aufbau von XML angepasst wurde. Das XML-Dokument wird, ausgehend von dem Wurzelelement, in Fragmente aufgeteilt. Die Größe des Fragments ist dabei variabel. Alle Kindelemente, die ausserhalb des Fragments liegen, werden wiederum zu Fragmenten zusammengefügt. Die einzelnen Fragmente werden anschließend mit einem ZIP-Verfahren komprimiert. Durch die Aufteilung des Dokuments in Fragmente werden jedoch die Redundanzen reduziert, was zu einer schlechteren Komprimierung durch den LZ-Algorithmus führt. XMLZIP stellt keine nutzbare Alternative dar.

5.4.2 XMILL

Bei XMILL wird die Struktur vom Inhalt des XML-Dokuments getrennt. Anschließend werden die Inhalte nach ihrer Bedeutung auf verschiedene Container aufgeteilt und die Container mit geeigneten Kompressoren komprimiert. So könnten z.B. Elementinhalte vom Schemastandardtyp `Integer` als ganze Zahl mit 32 Bit codiert werden. Ebenso könnten Container für bestimmte komplexe Typen angelegt werden. Die Vorgehensweise beim Komprimieren wird Anhand eines Beispiels gezeigt:

```
<DOI name="Mod" desc="Modus">
   <DAI name="ctlModel">
       <Val>status-only</Val>
    </DAI>
</DOI>
<DOI name="Beh" desc="Verhalten"/>
```

Die Struktur des Codes würde wie folgt aussehen:

```
#1#2 C1/#3 C1/#4#5 C1/#6 C2///#1#2 C1/#3 C1//
```

Durch ein Codebuch werden den Elementen und Attributen feste Codes zugeteilt, z.B. #1=DOI, #2=name(DOI), #3=desc, #4=DAI, #5=name(DAI), #6=Val. Um ein solches Codebuch zu erstellen kann das XML-Schema verwendet werden. Trotzdem gehört XMILL zu den syntaxfreien Codierverfahren, da es keinen Bezug auf die Struktur nimmt, die durch das Schema vorgegeben wird. Das Codebuch kann komprimiert an den Decoder gesendet werden. Durch das Zeichen „/“ wird angegeben, dass das aktuelle Element bzw. Attribut geschlossen wird. Für den Inhalt in diesem Beispiel wurden die zwei Datencontainer C1 und C2 angelegt. Die Aufteilung der Inhalte auf die Container kann sowohl von Hand als auch von einem Algorithmus übernommen werden. Im Beispiel wurden die Werte der Attribute und der Inhalt des Elements `<val>` in separate Container gepackt. Zwar enthalten beide Container Zeichenketten, doch durch bestimmte Erwartungswerte können Redundanzen für eine ZIP-Komprimierung erhöht werden. Die Container selber sind Stapelspeicher, die nach dem FIFO[15]-Prinzip arbeiten.

Bewertung

Der Vorteil von XMILL ist wie bei allen syntaxfreien Verfahren, dass zumindest auf der Decoderseite kein Schema benötigt wird. Nachteilig ist, dass nicht einzelne Fragmente interpretiert werden können. Da es sich bei den Containern um Stapelspeicher handelt, wird beim Decodieren immer nur das unterste Speicherelement vom Stapel entnommen. Die Reihenfolge im Stapel entspricht jedoch der Instanziierungsfolge des gesamten Dokuments und nicht der des Fragments. Aus diesem Grund ist es ebenfalls nicht möglich, wahlfrei auf das komprimierte Format zuzugreifen.

5.4.3 WBXML

Das WBXML[16]-Verfahren ist speziell für die Übertragung über schmalbandige Kanäle konzipiert und wird bereits bei der Übertragung von WAP[17]-Diensten eingesetzt. WBXML legt wie XMILL ein Codebuch mit 265 Seiten an. Auf jeder Seite befinden sich sogenannte Token-Tabellen. In diesen Tabellen wird den häufig verwendeten Symbolen aus dem XML-Dokument ein Token zugewiesen. Wird ein Dokument encodiert, wird z.B. anstelle eines Elements der Token aus der Tabelle plus zusätzlicher Bits eingesetzt. Die

[15] First In First Out = Das älteste Element wird als erstes entnommen
[16] WAP binary XML
[17] Wireless Application Protocol

zusätzlichen Bits geben beim Decodieren Auskunft darüber, ob z.B. ein Attribut oder Inhalt folgt. Der Inhalt wird nicht gesondert codiert. Das Codebuch wird in codierter Form vor dem Dokument übertragen.

Bewertung

Eine Übertragung von einzelnen Fragmenten ist nur bedingt möglich, da die Reihenfolge der codierten Elemente der Reihenfolge der Instanziierung im XML-Dokument entspricht. Obwohl WBXML, als eines der wenigen syntaxfreien Codierungsverfahren, den Zugriff auf das komprimierte Format ermöglicht, hat die Verarbeitung einen großen Nachteil. Damit ein Zugriff möglich ist, müssen die Token-Tabellen uncodiert zur Verfügung stehen. Da auch der Inhalt des Dokuments nicht gesondert komprimiert ist, wird die Speicherkapazität auf dem Gerät nicht mehr geschont.

5.4.4 Millau

Millau baut auf der Funktionsweise von WBXML auf. Es wurde für kleine XML Dateien entwickelt und erreicht bei diesen eine höhere Kompression als das GZIP-Verfahren.

Millau legt entgegen WBXML auch Token für Attribute an. Bei der Codierung wird zusätzlich mit einer gewissen Intelligenz gearbeitet. So kann auf ein Attribut-Token lediglich der Inhalt des Attributs folgen, nicht aber ein Element-Token. Durch mehrere solcher Logiken kommt Millau im direkten Vergleich zu WBXML mit einer geringeren Anzahl an Token aus. Der entscheidende Unterschied zu WBXML ist jedoch die Trennung von Struktur und Inhalt ähnlich wie bei XMill. Die Entwickler des Verfahrens sprechen von zwei getrennten Strömen. Im ersten Strom wird lediglich die Struktur des XML-Dokuments als Token-Strom gesendet, im zweiten Strom befinden sich die Inhalte als elementaren Datentypen codiert. Im Token-Strom wird an Stelle des Inhalts ein spezieller Token gesetzt der angibt, in welchem elementaren Datentypen der Inhalt im Inhalts-Strom codiert wurde. Momentan werden die vier Datentypen bool, byte, integer(4 byte) und float unterstützt, Strings werden direkt im Token-Strom codiert. Die Trennung in zwei Ströme hat den Vorteil, dass nur bei Bedarf auf den Strom mit dem Inhalt zugegriffen wird und so die Verarbeitungseffizienz steigt.

Um die Verarbeitbarkeit des Binärformats zu zeigen, wurden bei der Entwicklung von Millau verschiedene Programmierschnittstellen entwickelt, welche direkt auf das Binärformat zugreifen. Nach außen zeigen sich diese Programmierschnittstellen wie die von der XML-Verarbeitung bekannten Schnittstellen SAX und DOM. Dabei bieten die Schnittstellen des Millau-

Verfahrens zusätzlich die Möglichkeit, anstatt den Elementnamen als Zeichenkette ausgeben zu lassen, nur die Token durchzureichen. Dies ermöglicht eine schnellere Verarbeitung der Informationen, da der Anwender des Millau-Verfahrens keine Zeichenketten verarbeiten muss.

Bewertung

Millau wird als Verfahren bereits von IBM verwendet, was sicherlich als Vorteil anzusehen ist. Ebenfalls ist der schnelle Zugriff auf die Daten im Binärformat realisiert worden. Da Millau genau wie WBXML implizit von der Reihenfolge, in der die Elemente im XML-Dokument instanziiert werden, abhängt, ist eine Fragmentierung nur schwer möglich. Die Aufteilung in zwei Ströme macht die Fragmentierung sogar unmöglich, da beide Ströme für eine Decodierung inklusive Inhalt gleichzeitig in Reihenfolge des Dokuments gelesen werden müssen. Daraus folgt auch, das beim decodieren das Binärformat von Anfang an geparst werden muss.

5.4.5 XMLPPM

XMLPPM ist angelehnt an das PPM[18] Verfahren. Bei dem PPM-Verfahren wird, ausgehend von den vorangegangen Zeichen, eine Wahrscheinlichkeit bezüglich des folgenden Zeichens berechnet. Entspricht das Zeichen nicht dem wahrscheinlichen Zeichen wird ein Escape-Zeichen kodiert und geprüft, ob das Zeichen der nächst kleineren Wahrscheinlichkeit entspricht. Dies wird solange wiederholt bis das Zeichen gefunden wurde. Zur Codierung wird anschließend die Wahrscheinlichkeit des Zeichens arithmetisch codiert. Beim XMLPPM wird dieses Verfahren auf die spezielle Struktur eines XML-Dokuments angepasst. Da die Elemente, Attribute und der Inhalt sich syntaktisch unterscheiden, werden verschiedene Wahrscheinlichkeitsmodelle angelegt. Bei XMLPPM wird der Algorithmus des PPM-Verfahrens z.B. auf den Kontextpfad eines Inhalts angewendet. Zuerst wird der wahrscheinlichste Kontextpfad geprüft. Tritt dieser nicht auf, wird der nächst wahrscheinliche Kontext geprüft, solange bis der passende Kontext codiert ist.

Bewertung

Das XMLPPM-Verfahren zeichnet sich durch eine hohe Kompression bei großen Dokumenten aus. Je kleiner das XML-Dokument desto schwieriger wird es für das Verfahren verlässliche Wahrscheinlichkeiten zu berechnen. Da die statistische Wahrscheinlichkeit beim encodieren immer neu berechnet

[18] Prediction by Partial Match

wird, muss der Decodierer das komprimierte Dokument in exakter Reihenfolge decodiert werden, um die selbe Statistik ermitteln zu können. Dies verhindert eine beliebige Übertragung von Fragmenten und macht das Verfahren besonders fehleranfällig.

5.5 Syntaxbasierte Codierverfahren

5.5.1 XML Xpress

XML Xpress ist ein Komprimierungsverfahren der Firma ICT[19]. Da es sich bei dem Verfahren um ein kommerzielles Produkt handelt, können nur allgemeine Aussagen über die Funktionweise des Verfahrens gemacht werden. XML Xpress verwendet wahlweise eine DTD[20] oder ein XML-Schema zur Codierung. Um die Kompression zu erhöhen, kann zusätzlich eine Statistik, die aus Beispieldokumenten gewonnen wird, erzeugt und genutzt werden. Dafür werden die Beispieldokumente von der ICT in ein „Schema Model File“ (SMF) überführt. Da alle Aussagen über die Kompression nur bei Verwendung der SMF gemacht werden, wird das Verfahren nur unter Verwendung dieser geprüft. Das Binärformat kann in Fragmente unterteilt werden, jedoch müssen diese in der richtigen Reihenfolge decodiert werden.

Bewertung

Wie bereits erwähnt ist es nicht möglich einzelne Fragmente zu übertragen. Durch die Erstellung der Schema Model Files entsteht eine zusätzliche Abhängigkeit. Damit ein XML-Dokument korrekt decodiert wird muss auf En- und Decoderseite nicht nur das Schema sondern auch das SMF gleich sein. Da es sich bei ICT um eine kommerzielle Firma handelt und XML Xpress kein standardisiertes Verfahren ist, muss zuätzlich die Zukunftsfähigkeit in Frage gestellt werden.

5.5.2 ASN.1

Bei der Komprimierung von XML-Dokumenten mit Hilfe der „Abstract Syntax Notation One“ wird konzeptionell ein anderer Ansatz als bei den voherigen Verfahren gewählt. Die ASN.1 beschreibt dabei kein direktes Komprimierverfahren, sondern beschreibt den Datenaustausch zwischen den Anwendungen. ASN.1 ist von der ITU[21] standardisiert und ist ein weltweit anerkannter Standard für den Austausch von Informationen speziell für Kommu-

[19] Intelligent Kompression Technologies
[20] Document Type Definition
[21] International Telecommunication Union

nikationsanwendungen. Genau wie in XML-Schema besitzt die ASN.1 primitive Datentypen wie Zeichenketten, ganze Zahlen usw., ebenso können eigene Datentypen definiert werden. Ein Auszug aus einem XML-Schema könnte wie folgt in ASN.1 aussehen:

```
<xsd:complexType  name="LineItemPair">
    <xsd:sequence>
         <xsd:element
                    name="part-no"  type="xsd:number"/>
         <xsd:element
                    name="quantity" type="xsd:number"/>
     </xsd:sequence>
</xsd:complexType>
```

...und als ASN.1

```
Invoice ::= SEQUENCE {
                    number  INTEGER,
                    name            UTF8String,
                    details         SEQUENCE OF
                                      LineItemPair,
                    charge          REAL,
                    authenticator   BIT STRING}

LineItemPair ::= SEQUENCE {
                    part-no         INTEGER,
                    quantity        INTEGER }
```

Die ITU unternimmt momentan große Anstrengungen einen Brückenschlag zwischen XML und ASN.1 herzustellen. Die ASN.1 soll als eine weitere Schemasprache mit XML-Schema konkurrieren. Da bereits bei vielen Anwendungen ein XML-Schema definiert ist, wie z.B. auch die IEC61850, wurde ein Verfahren entwickelt, welches ein XML-Schema in ein ASN.1-Schema umwandelt. Dabei entsteht eine eins zu eins Beziehung, was bedeutet, dass ein ASN.1-Schema auch wieder in ein XML-Schema umgewandelt werden kann. Der Vorteil bei der Verwendung eines ASN.1-Schemas ist die Anwendung der bereits vorhanden „Encoding Rules", die ebenfalls von der ITU normiert sind. Diese Encodierungsregeln spezifizieren, wie abstrakte Werte die einem ASN.1 Modul entsprechen, in eine andere Darstellungsform überführt werden. Eigens für XML wurden die „XML Encoding Rules", kurz XER, spezifiziert. Die XER beschreiben, wie abstrakte Werte mit Hilfe des ASN.1-Schemas in eine XML-Instanz überführt werden. Die XML stellt für die Vertreter der

ASN.1 nur eine Form der Abstraktion von Daten dar, ebenso können die Daten durch eine andere Encodierungsregel in einem anderen Format dargestellt werden. Ein Beispiel für einen anderen normierten Satz von Regeln sind die BER[22], in denen die Datenelemente durch die Angabe der drei Parameter „Type, Length, Value“, kurz TLV, beschrieben werden. Dabei steht Type für den Datentypen, Length für die Länge des Wertes in Byte und Value für den eigentlichen Zahlenwert des Datenelements. Da durch die ASN.1 zwischen allen Darstellungsformen eine eins zu eins Beziehung besteht, kann so ein XML-Dokument ansatzlos in ein Binärformat umgesetzt werden. Um die Darstellung des Binärformats an die unterschiedlichen Anforderungen anpassen zu können, können mit der „Encoding control notation“ (ECN) auch eigene Encodierregeln beschrieben werden. Mit Hilfe ECN, die ebenfalls von der ITU standardisiert ist, versuchen verschiedene Verfahren ein Binärformat für XML-Dokumente zu entwickeln. Das ausgereifteste Verfahren, welches auch von der ITU unterstützt wird, wird im folgenden Absatz auf die Eignung geprüft.

Fast InfoSet

Bewertung

5.5.3 BiM

Bei dem BiM-Verfahren handelt es sich um das ausgewählte Verfahren, welches im weiteren Verlauf dieser Arbeit untersucht wird. Dafür wird es in einem extra Kapitel nochmal detailiert beschrieben. Dieser Unterabschnitt soll lediglich einen kurzen Überblick über das Verfahren geben und zeigen, warum sich für dieses Verfahren entschieden wurde. Es ist empfehlenswert, sich diesen Unterabschnitt vor Kapitel 6 durchzulesen, um ein grundlegendes Verständnis für dieses Verfahren zu entwickeln.

Die Abkürzung BiM steht für „Binary format for multimedia description streams“. Ursprünglich wurde das BiM-Verfahren für die Komprimierung der im MPEG-7 Standard definierten Metadaten entwickelt. Da diese Metadaten einem XML-Dokument entsprechen, lässt sich dieses Verfahren universell auf andere XML-Dokumente anwenden. Wie bereits erwähnt, besteht ein XML-Dokument zum Großteil aus Strukturinformation, der eigentliche Informationsgehalt des Dokuments ist relativ gering. Deshalb setzt der BiM-Algorithmus in erster Linie auf die Komprimierung der Strukturinformation. Er vereinigt zwei Komprimierverfahren, die jeweils eigene spezifische Vorteile bieten. Beide Verfahren verwenden das XML-Schema des XML-Dokuments,

[22]Basic Encoding Rules

um aus ihm Codeworte für das Binärformat zu bestimmen.

Das erste Verfahren nennt sich Pfadcodierung, da es den Kontextpfad eines jeden Elements und Attributs codiert. Der eigentliche Inhalt des Elements wird nicht codiert. Die Pfadcodierung wird ausschließlich zur Adressierung der Elemente und Attribute verwendet. Der Vorteil dieses Verfahrens ist, dass auf Syntaxelemente, deren Kontextpfad codiert wurde, direkt zugegriffen werden kann.

Bei dem zweiten Verfahren, der Payloadcodierung, wird aus dem XML-Schema ein Automatenmodell erzeugt. Die Automatenfragmente entsprechen z.B. einem Element. Beim parsen des Dokuments wird festgestellt, welche Elemente bzw. Automatenfragmente instanziiert wurden, um anschließend den Weg durch den Automaten zu codieren. Dieses Verfahren zeichnet sich durch eine höhere Kompression aus, zusätzlich wird der Inhalt typentsprechend codiert. Der Nachteil ist, dass für den Zugriff auf den Inhalt das Binärformat vom Anfang aus decodiert werden muss.

Bei der Kombination der Verfahren wird das XML-Dokument in Fragmente unterteilt. Diese werden mit der Payloadcodierung codiert, anschließend werden die Fragmente mit der Pfadcodierung adressiert.

Bewertung

Obwohl das BiM-Verfahren für den MPEG-7 Standard entwickelt wurde, gleichen die Anforderungen des MPEG-7 an ein Binärformat denen dieser Arbeit. Das BiM-Verfahren erfüllt nahezu alle Anforderungen:

Das Verfahren besitzt eine hohe Kompressionsrate. Das Binärformat ist in Fragmente unterteilbar, dabei lassen sich die Fragmentgrenzen frei wählen. Durch die Adressierung mit der Pfadcodierung können die Fragmente beliebig übertragen und einzeln für sich decodiert werden. Dadurch, dass die möglichen Automaten bzw. Kontextpfade schon vor der En- bzw. Decodierung durch das Schema bekannt sind, ist die benötigte Rechenleistung relativ gering. Die bereits bekannte Struktur macht es ebenfalls möglich ein XML-Dokument während der Übertragung zu encodieren. Kompromisse müssen jedoch bei dem freien Zugriff auf das Binärformat eingegangen werden. Das Fragment stellt quasi eine Grenze für den freien Zugriff dar. Da dessen Größe jedoch frei wählbar ist kann, durch in Kauf nehmen einer geringeren Kompression, die mögliche Tiefe des Zugriffs vergrößert werden. Mit dem BiM-Verfahren ist es ebenfalls möglich, ein komprimiertes XML-Dokument beim decodieren vollständig wiederherzustellen. Da der MPEG-7 Standard nicht alle Syntaxelemente des XML-Schemas nutzt, muss die Unterstützung bestimmter Syntaxelemente jedoch noch implementiert werden, was in Kapitel 7 beschrieben wird. Ein entscheidender Vorteil ist sicherlich, dass das

BiM-Verfahren bereits vom MPEG-7 Standard normiert ist und somit das Verfahren über längere Zeit noch unterstützt wird.

Als Nachteil ist sicherlich nochmal die Abhängigkeit der Codierung vom XML-Schema zu nennen. Um ein Dokument korrekt decodieren zu können, muss bei der Encodierung das selbe Schema wie bei der Decodierung vorliegen.

5.6 Aktivitäten der W3C bezüglich eines Binärformats

Die W3C hat Anfang 2004 damit begonnen, sich mit dem Thema binäres XML auseinanderzusetzen [W3C1]. Es wurde eine Arbeitsgruppe mit dem Namen „XML Binary Characterization Working Group" gebildet, die ihre Arbeit am 1. April 2005 eingestellt hat. Als Ergebnis dieser Arbeitsgruppe wurde eine „Working-Note" verfasst, welche nicht den Status einer Empfehlung der W3C besitzt. Aufgabe der Gruppe war es herauszufinden, ob Gründe für ein weiteres Vorgehen der W3C bezüglich eines binären XML-Standards bestehen. Im Abschluss der Working-Note wird ein weiteres Vorgehen empfohlen, zum Zeitpunkt dieser Arbeit (5. April 2006) wurden jedoch noch keine weiteren Arbeiten durchgeführt. Prinzipiell kann es nur von Vorteil sein, wenn das eingesetzte Binärformat für die Parameterübertragung einem späteren Standard der W3C entspricht, deshalb wird das BiM-Verfahren auf die möglichen Anforderungen eines solchen Standards geprüft. Es sei noch mal erwähnt, dass die Arbeitsgruppe nur Vorschläge für Anforderungen unterbreitet hat, trotzdem lohnt sich eine Prüfung gegen diese Anforderungen.

Zunächst wird die Vorgehensweise der Arbeitsgruppe beschrieben. Im ersten Schritt der Arbeitsgruppe wurden diverse Anwendungsfälle für ein binäres Format aufgezeigt. Dabei kam die Arbeitsgruppe bereits auf 18 Anwendungsfälle, was den Bedarf an einem Binärformat für XML deutlich macht. Kein Anwendungsfall vereinigt alle in Abschnitt 5.1 beschriebenen Anforderungen unter seiner Domaine, jedoch treten die Anforderungen verteilt in mehreren Anwendungsfällen auf. Nach der Aufstellung der Anwendungsfälle wurden von der Arbeitsgruppe die sich aus diesen Anwendungsfällen ergebenden Eigenschaften eines Binärformats rausgeschrieben. Zusätzlich wurden Eigenschaften, welche im Falle einer Normierung von der W3C gefordert werden, in die Auflistung der Eigenschaften aufgenommen. Zusammenfassend kam die Arbeitsgruppe auf 38 Eigenschaften. Die große Anzahl der Eigenschaften kann dabei eher als hinderlich angesehen werden, da ein Standard, der von keinem Verfahren erfüllt werden kann, nicht von Entwicklern bzw. Anwendern angenommen wird. Über diverse Verfahren hat die Arbeitsgruppe die Eigenschaften und Anforderungen auf 15 Eigenschaften, welche unterstützt werden müssen, reduziert. Weiterhin wurden sechs Ei-

genschaften beschrieben, welche durch den Einsatz des Binärformats nicht verhindert werden dürfen. Hierbei handelt es sich eher um Eigenschaften der Anwendung. Um die technische Machbarkeit dieser Anforderungen zu zeigen, wurden acht bereits existierende Binärformate, unter anderem auch das BiM-Format, auf ihre Eignung geprüft.Der Bericht der Arbeitsgruppe schließt ab mit einer Empfehlung für eine Standardisierung von binärem XML durch die W3C. Es bleibt jedoch offen, ob bestimmte Verfahren, Formate oder nur Empfehlungen für ein Binärformat formuliert werden. Ein Hindernis für die Standardisierung von einzelnen Verfahren ist die Forderung der W3C nach einem frei verfügbaren Verfahren. Auf viele der Verfahren zur Binärcodierung, unter anderem auch auf das BiM-Verfahren, bestehen jedoch Patentrechte, welche dies verhindern. Im Falle der ausschließlichen Standardisierung der Eigenschaften eines Binärformats ist die Wahrscheinlichkeit groß, dass das BiM-Verfahren die von der W3C geforderten Eigenschaften bereits besitzt.

6 Der BiM Algorithmus

In diesem Kapitel wird das Prinzip des BiM-Algorithmus für das Verständnis der Lösungsansätze in Kapitel 7 ausführlich erklärt. Es ist empfehlenswert, sich den Unterabschnitt 5.5.3 vor diesem Kapitel durchzulesen, um ein grundlegendes Verständnis für dieses Verfahren zu besitzen. In Unterabschnitt 5.5.3 wurden bereits die beiden Verfahren, welche beim BiM zum Einsatz kommen, erwähnt. Diese werden in den folgenden Abschnitten zunächst getrennt vorgestellt. In Abschnitt 6.3 wird dann die Kombination der Verfahren beschrieben. Anschließend wird auf den Aufbau des Bitstroms eingegangen. Ein wichtiger Teil des BiM-Algorithmus, der Bytecode, welcher noch nicht beschrieben wurde, wird in Unterabschnitt 6.5 behandelt. Eine ausführliche Beschreibung des Algorithmus findet man in [NIE03].

6.1 Die Pfadcodierung

Das Prinzip der Pfadcodierung ist es, die speicherintensive Struktur eines XML-Dokuments binär nachzubilden. Dabei wird das XML-Dokument als Baum betrachtet, in dem jedes Element und Attribut einen Knoten darstellt. Anhand der Typdeklaration eines Elements im XML-Schema kann man sehen, welche weiteren Elemente und Attribute im Baum vorkommen können. Für jedes Element wird eine Liste der möglichen folgenden Elemente und Attribute erstellt. Nachdem die Elemente und Attribute in der Liste nach einem festen Muster sortiert wurden, wird jedem ein Codewort zugeteilt. Die Sortierung ist notwendig, damit aus semantisch gleichen Schemata, die sich nur in der Reihenfolge ihrer Elemente unterscheiden, gleiche Codeworte entstehen. Für Attribute und Elemente mit einfachem Typ wird keine Liste angelegt, da sie keine weiteren Kinder besitzen können. Die beschriebene Liste wird im folgenden Text als *BVC*[23]*-Tabelle* und die Codeworte als *Strukturcodes* bezeichnet. Zusätzlich werden noch zwei weitere Strukturcodes in die BVC-Tabelle eingefügt. Am Ende der BVC-Tabelle wird ein Codewort spezifiziert, welches das Ende eines Pfades angibt. Das zweite Codewort verweist zurück auf den Vaterknoten. Dieses Codewort macht es möglich, sich vom zuletzt spezifizierten Knoten wieder zurück zu navigieren. Die BVC-Tabelle stellt das zentrale Prinzip der Pfadcodierung dar. Bevor die Pfadcodierung mit Hilfe der BVC-Tabelle an einem Beispiel erklärt werden kann, müssen noch weitere Codes beschrieben werden, welche die diversen Besonderheiten eines XML-Dokuments behandeln. Da diese Codes für das Verständnis der Arbeit nur bedingt nötig sind, werden sie nur kurz beschrieben.

[23]BVC = *B*aum*v*erzweigungs*c*ode

Der *Typecode* ermöglicht einen Typecast in XML zu codieren. In XML ist ein Typecast nur mit Typen möglich, die direkt oder indirekt vom ursprünglichen Typen des Elements abgeleitet sind. Für den ursprünglichen Typen kann also eine Art Vererbungsbaum angelegt werden, nach dem den möglichen Typen für einen Typecast ein Typecode zugeordnet wird. Über ein Flag[24] im Bitstrom wird angezeigt ob ein Typecast auftritt. Falls dies der Fall ist, folgt der Typecode im Bitstrom.

Der *Positionscode* wird eingesetzt damit eine frei wählbare Fragmentierung möglich ist. Da dies auch eine Anforderung an das Binärformat dieser Arbeit ist, wird er kurz erklärt. In XML ist es möglich, dass gleiche Elemente mehrfach auftreten. Ohne einen Positionscode müssen solche Elemente der Istanziierungsreihenfolge entsprechend codiert werden. Dies schließt die Möglichkeit aus, eines der Elemente nicht der Reihe nach wie sie im Dokument zuerst instanziiert wurden, als Fragment gesondert zu schicken. Um dies auszuschließen wird jedes Element mit einer Häufigkeit größer eins durch einen Positionscode ergänzt. Der große Nachteil ist, dass das Datenvolumen durch den Positionscode um bis zu 60% steigt.

Der *Ersetzungscode* dient zur Codierung von Ersetzungsgruppen in XML. Eine Ersetzungsgruppe hat dem Schema nach immer ein Head-Element und eine feste Anzahl an Mitgliedern. Der BiM-Algorithmus trägt alle Mitglieder alphabetisch geordnet in eine Tabelle ein und adressiert diese mit einem Ersetzungscode. Bei der Codierung eines Head-Elements wird dann über ein Flag signalisiert ob eine Ersetzung auftritt. Wenn ja, folgt der Ersetzungscode.

Soll ein Element durch einen Pfad adressiert werden, gibt es wahlweise die Möglichkeit, den Pfad absolut oder relativ aufzubauen. Abbildung 3 zeigt beide Möglichkeiten der Pfadcodierung. Beim absoluten Pfad wird ausgehend von der Wurzel das Element adressiert. Beim relativen Pfad bildet der zuletzt adressierte Knoten den Ausgangspunkt was bedeutet, dass dieser Knoten auf En- und Decoderseite bekannt sein muss. Für die Codierung eines gesamten Dokuments ist die relative Pfadcodierung kompakter. Für den in dieser Arbeit beschriebenen Anwendungsfall ist jedoch nur die absolute Pfadcodierung anwendbar, da bei fragmenteller Übertragung nicht davon ausgegangen werden kann, dass der zuletzt codierte Knoten bekannt ist.

Zur weiteren Optimierung des Verfahrens wird ein Pfad in zwei Teile aufgeteilt, die durch einen Termination-Code getrennt werden. Bis zum Vaterknoten des zu adressierenden Elements, dem so genannten Kontext-Knoten, ist es ausreichend, mit einer Kontext-BVC-Tabelle den Strukturcode zu ermitteln. Da es sich bei dem Kontext-Knoten zwangsläufig um einen komple-

[24]Flag = Zustandsbit

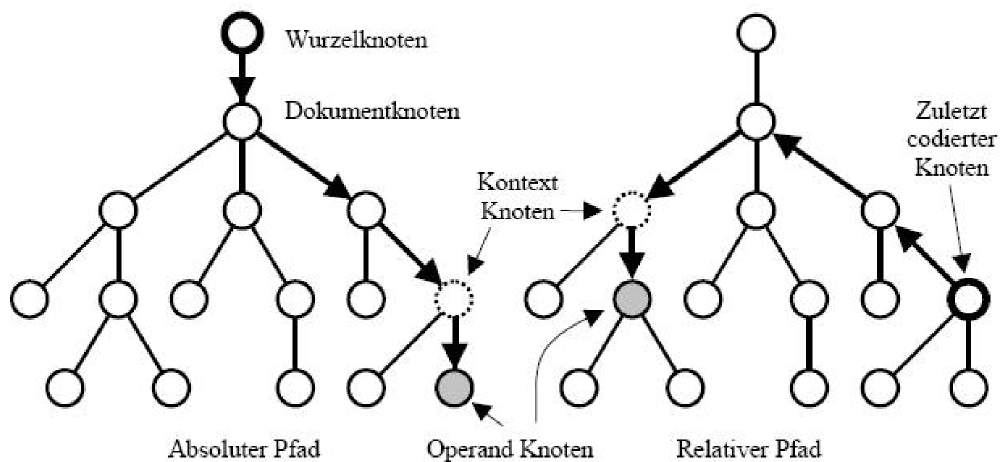

Abbildung 3: Absoluter und Relativer Pfad mit Kontext- und Operandencode

xen Typen handelt, werden in der Kontext-BVC-Tabelle nur Elemente mit komplexem Typ und der Termination-Code eingetragen. Der Strukturcode, der das Pfadfragment zwischen Kontext-Knoten zum adressierten Knoten spezifiziert, wird aus der Operanden-BVC-Tabelle ermittelt. In dieser sind nun alle Elemente und Attribute aufgelistet. Für den folgenden Typen bzw. Knoten wird jeweils eine Operanden-BVC- und eine Kontext-BVC-Tabelle erstellt.

```
<complexType name="tElement">
  <sequence>
    <element name="E_a" type="KomplexA"/>
    <element name="E_b" type="KomplexB"/>
  </sequence>
  <attribute name="A_a" type="string"/>
  <attribute name="A_b" type="string"/>
</complexType>
```

Strukturcode	Schemaelement
00	Vaterknoten
01	Element E_a
10	Element E_b
11	Termination

Tabelle 2: Kontext BVC-Tabelle

Strukturcode	Schemaelement
000	frei
001	Element E_a
010	Element E_b
011	Attribut A_a
100	Attribut A_b
101-111	frei

Tabelle 3: Operand BVC-Tabelle

Der Vorteil zeigt sich in der Ersparnis von Bits für den Strukturcode.

Ein kompletter Pfad setzt sich also wie folgt zusammen: Der Pfad besteht aus n Pfadfragmenten. Ein Pfadfragment besteht aus einem Strukturcode und gegebenfalls aus dem Ersetzungscode und dem Typecode. Bis zum Pfadfragment n-1 werden die Strukturcodes aus der Kontext-BVC-Tabelle erzeugt. Anschließend folgt der Termination-Code gefolgt von einem weiteren Pfadfragment, dessen Strukturcode aus der Operanden-BVC-Tabelle gewonnen werden. Als letztes folgen die Positionscodes aller Pfadfragmente, die einen solchen benötigen. Abbildung 4 zeigt den Aufbau eines solchen Pfades.

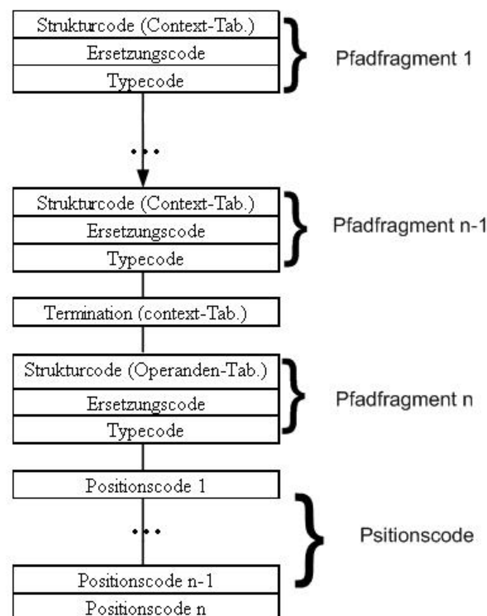

Abbildung 4: Aufbau eines Pfades

6.2 Die Payloadcodierung

Die Payloadcodierung verwendet genau wie die Pfadcodierung das XML-Schema zur Erstellung der Codeworte. Anders als bei der Pfadcodierung werden nicht einzelne Knoten identifiziert, sondern ganze Baumfragmente inklusive Nutzdaten codiert. Die Payloadcodierung basiert auf einem Zustandsautomatenmodell. Jeder komplexe Typ in XML besteht aus allgemeinen bekannten Syntaxkonstrukten. Jedes dieser Konstrukte entspricht einem bestimmten Muster. Das Syntaxkonstrukt choice z.B. stellt einen Knoten dar, der sich entsprechend der Anzahl der Möglichkeiten aufspaltet. Die entsprechenden Muster in einem Typen werden gerichtet miteinander verschaltet und ergeben somit einen Automaten. Beim Durchlaufen dieses Automaten wird jeder Möglichkeit einer Verzweigung ein Codewort zugewiesen. Dieser Automat steuert sowohl die En- als auch die Decodierung eines Typen. Trifft der Automat beim Decodieren auf eine Auswahl, wird der Bitstrom gelesen und das ausgewählte Element decodiert. Der Vorgang beim Encodieren ist der Gleiche. Das XML-Dokument wird gelesen. Trifft der Automat auf eine Auswahl, wird geprüft welches Element instanziiert wurde und anschließend das entsprechende Codewort in den Bitstrom geschrieben.

Dieser Vorgang soll anhand eines Beispiels gezeigt werden. Der folgende Ausschnitt aus einem XML-Schema wird in Abbildung 5 als Automatenmodell dargestellt.

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

Wird eine Instanz des XML-Schemas, in der zuerst das Element „a“ und anschließend das Element „d“ ohne Element „e“ instanziiert wird, mit dem Automaten codiert, würde folgender Bitstrom daraus resultieren:

```
00010 00 <Code für Typ "Ta"> 10 <Code für Typ "Td"> 0
```

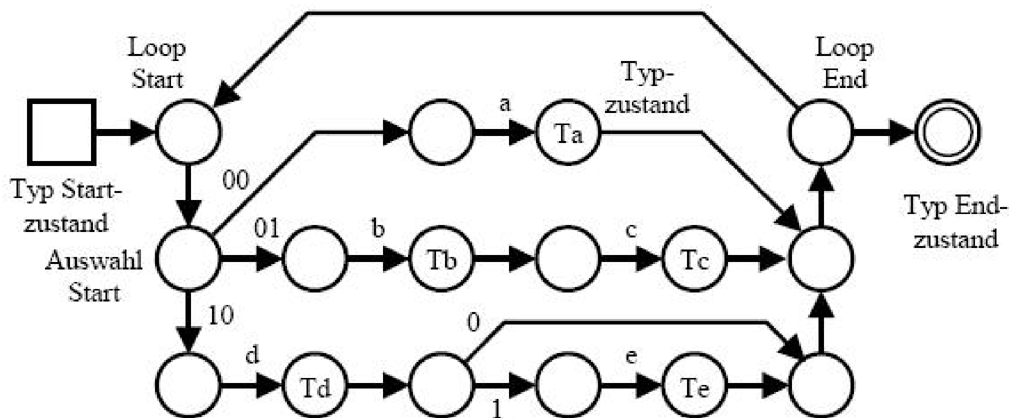

Abbildung 5: Automatenmodell zu einer Syntaxdefinition [NIE38]

Die ersten fünf Bits zeigen an, dass die Auswahl zweimal codiert wird, darauf folgt der Code für den Typen von Element „a“. Dieser Typ hat wiederum sein eigenes Automatenmodell, welches hier nicht gezeigt wird. Der Codec läuft weiter über „Loop End“ zurück auf „Loop Start“. Als nächstes wird der untere Zweig mit `01` codiert, anschließend folgt der Code für den Typen des Elements „d“. Mit der `0` am Ende wird die Häufigkeit von 0 des Elements „e“ codiert. Es fällt auf, dass für die Codierung der Struktur des XML-Dokuments im Fall der oben beschriebenen Instanz, nur zehn Bits benötigt werden. Schreibt man nur die Zeichen für die Struktur einer solchen Instanz heraus, kommt man auf folgende Zeichenfolge: `<a></a><d></d>`. Codiert man jedes einzelne Zeichen der Struktur mit acht Bit, was ohne Komprimierung der Fall ist, kommt man auf 14 * 8 = 112 Bits. Die Payloadcodierung erreicht in diesem Beispiel eine Kompression für die Struktur von ca. 9%. Bei einem Verfahren wie Millau wird zum Vergleich für jedes Element ein acht Bit Token gesetzt, was zu einer Strukturcodelänge von 16 Bits (ca.14%) führt. Keines der Verfahren aus Kapitel 5 erreicht eine höhere Kompression wie die Payloadcodierung, lediglich das GZIP-Verfahren erreicht bei großen Dateien ähnliche Werte.

Die Payloadcodierung ermöglicht ebenfalls eine optimierte Codierung von Inhalt. Anhand des Typen kann man für die Codierung des Inhalts speziell optimierte Codecs verwenden. Ist der Inhalt eines Elements z.B. vom Typ `xs:Integer` kann der Inhalt des Elements als ganze Zahl mit 32 Bit codiert werden. Spezielle Codecs für Typen lassen sich jedoch auch auf komplexe Typen anwenden, wie in Kapitel 7 gezeigt wird.

Codiert man ein XML-Dokument vollständig mit der Payloadcodierung

ohne die Verwendung der Pfadcodierung, können viele Anforderungen aus Abschnitt 5.1 nicht erfüllt werden. Zunächst ist das Binärformat eines mit Payloadcodierung codierten Dokuments nicht fragmentierbar, woraus folgt, dass Änderungen im Inhalt nicht separat gespeichert werden können. Ebenso wird ein direkter Zugriff auf den Inhalt des Binärformats nicht unterstützt, da der Codec immer am Anfang des Automatenmodells beginnt und die Längen der Codes von der Instanz abhängen. Will ein Anwender auf ein Element im Binärformat zugreifen, muss der Codec vom Anfang des Binärformats solange das Binärformat decodieren, bis er auf das gesuchte Element trifft. Wurde das gesuchte Element am Ende des ursprünglichen XML-Dokuments istanziiert, tritt es auch erst am Ende des Binärformats auf, was zu langen Zugriffszeit führt.

6.3 Synthese der Pfad- und Payloadcodierung

Beide Codierverfahren die hier beschrieben wurden haben sowohl Vor- als auch Nachteile. Um die Vorteile beider Verfahren zu nutzen, wurden sie miteinander kombiniert. Ausgehend von der Darstellung des XML-Dokuments als Baum, kann man den Baum in beliebig kleine Teilbäume zerlegen. Diese Teilbäume werden von der Wurzel aus durch einen Pfad spezifiziert. Das Fragment selbst wird mit der Payloadcodierung codiert. Abbildung 6 zeigt

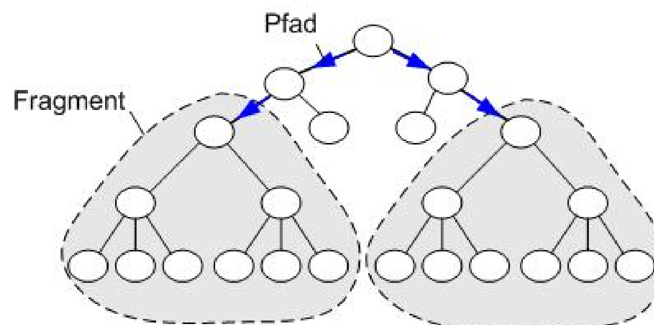

Abbildung 6: Kombination der Pfad- und Payloadcodierung

eine solche Kombination aus Pfad- und Payloadcodierung. Die Größe eines Fragmentes kann dabei frei gewählt werden. Durch die Kombination dieser beiden Verfahren werden alle Anforderungen aus Abschnitt 5.1 erfüllt. Die Fragmente werden mit der Payloadcodierung effizient codiert, der Baum ist beliebig fragmentierbar und die Fragmente können über den codierten Pfad identifiziert werden.

Bei näherer Betrachtung der Fragmentierung ergibt sich jedoch eine Besonderheit. Da die Pfadcodierung wegen ihrer fehlenden Codierung von In-

halt nur zur Adressierung der Fragmente dient, müssen auch die Knoten oberhalb der Fragmente aus Abbildung 6 mit der Payloadcodierung codiert werden. In Abbildung 7 werden die oberen Knoten zu einem Fragment zusammen gefasst, ebenso könnte jeder Knoten alleine ein Fragment darstellen. Es fällt auf, dass sich das obere Fragment jeweils mit den unteren Fragmen-

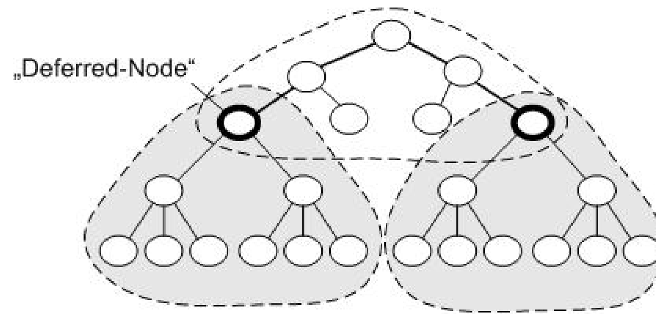

Abbildung 7: Deferred-Nodes in Fragmenten

ten überschneidet. Die Knoten, die jeweils in zwei Fragmenten auftauchen, sind dick umrandet. Zunächst wird nur das obere Fragment betrachtet. Die Payloadcodierung arbeitet sich vom Wurzel-Knoten bzw. Typen aus durch die Automaten. Trifft die Codierung auf den Knoten an der Fragmentgrenze, erwartet sie, dem Automatenmodell entsprechend, die Kindelemente bzw. Knoten. Diese sind jedoch nicht vorhanden, da sie in einem anderen Fragment codiert werden. Der Payloadcodierung muss nun mitgeteilt werden, dass sich an dieser Stelle eine Fragmentgrenze befindet, hierfür wurden im Binärformat sogenannte „Deferred-Nodes"[25] oder auch Platzhalterknoten eingeführt. Trifft der Decoder auf einen Platzhalterknoten weiß er, dass die Information in einem bestimmten Fragment codiert wurde. Die dick umrandeten Knoten in Abbildung 7 werden demnach im oberen Fragment als Platzhalter und erst im unteren Fragment vollständig codiert.

6.4 Der Bitstrom

Für das BiM-Verfahren wurde ebenfalls eine spezielle Organisation des Binärformats entwickelt, das im weiteren Verlauf als Bitstrom bezeichnet wird. Der Name Bitstrom zeigt, dass das BiM-Verfahren ursprünglich für „Streams"[26] entwickelt wurde. Die Abbildung 8 zeigt den Aufbau eines Bitstroms. Am Anfang jedes Bitstroms wird die „DecoderInit" gesendet. Sie dient der Initialisierung des Decoders. In ihr sind beispielsweise Informationen über den Na-

[25] aufgeschobene Knoten

[26] Streams sind eine ständige Abfolge von Datensätzen, deren Ende nicht bekannt ist

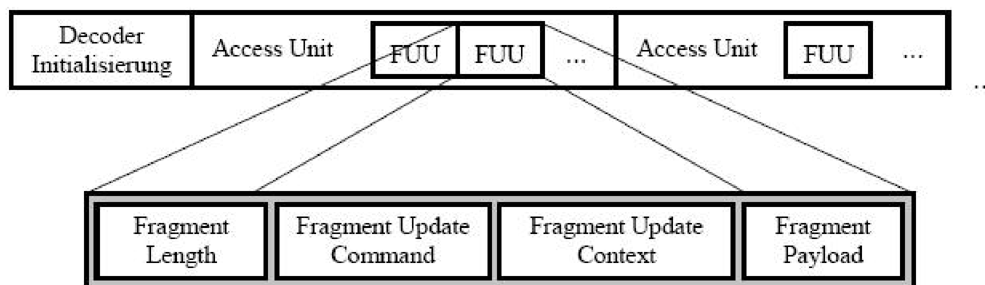

Abbildung 8: Aufbau eines Bitstroms [NIE45]

mensraum des Schemas gespeichert. Die DecoderInit kann beliebig erweitert werden. Danach teilt sich der Bitstrom in „Access Units“(AU) auf, die wiederum aus einem oder mehreren „Fragment Update Units“(FUU) bestehen. Abbildung 8 zeigt im unteren Teil den Aufbau einer FUU. Zuerst wird die Fragmentlänge („FragmentLength“) übertragen. Mit Hilfe der Länge können bestimmte Fragmente übersprungen werden, wie später noch erklärt wird. Im „Fragment Update Command“ werden Befehle wie „add“, “update“ und „delete“ codiert, hier zeigt sich auch die Philosophie des „streaming“. Die Information ist in einer Baumstruktur gespeichert und die Fragmente werden durch den „Stream“ ständig erneuert. Anschließend folgt der Kontextpfad („Fragment Update Context“) des Fragments, bevor dann das Fragment als Payload codiert folgt. Mit Hilfe des Kontextpfads kann das Fragment immer eindeutig identifiziert werden. Entspricht der Kontextpfad nicht dem des gesuchten Fragments, kann mit Hilfe der Fragmentlänge das Fragment übersprungen werden. Bei diesem Verfahren spricht man von filtern.

6.5 Der Bytecode

Obwohl beide Verfahren ihre Codeworte unterschiedlich erstellen, verwenden beide zur Erstellung der Codes das XML-Schema. Ohne weitere Entwicklungen müsste, bei einer Kombination der Verfahren, das Schema doppelt im Speicher gehalten werden. Der Zugriff auf die Schemainformationen würde über einen Text- Parser geschehen. Da es sich beim parsen von Texten um eine rechenintensive Operation handelt, welche den Aufwand der En- und Decodierung erhöht, wurde diese Operation auf einen Kompilationsprozess ausgelagert. Die kompilierte Datenstruktur wird im folgenden Text Bytecode genannt. Dieser Bytecode ermöglicht einen gleichzeitigen Zugriff beider Codierverfahren auf die Schemainformationen. Die Struktur des Bytecodes wurde so angelegt, dass ein Kompromiss zwischen schneller En- und Decodie-

rung und speichereffizienter Darstellung der Schemainformationen gefunden wurde. Da der Bytecode vorkompilierbar ist, ist er für den Einsatz in kleinen Baugruppen gut geeignet. Vorkompilierbar bedeutet, dass der auf dem PC kompilierte Bytecode direkt in das Gerät gespielt werden kann. Daraus ergibt sich der Vorteil, dass der Bytecode nicht auf den Baugruppen kompiliert werden und dazu das XML-Schema unkomprimiert gespeichert werden muss.

Der Bytecode stellt durch verschiedene Strukturen das Schema nach. Im Bytecode werden für bestimmte Bestandteile des Schemas, siehe Tabelle 4, so genannte Zustände angelegt. Die Zustände bestehen aus ganzen Vielfa-

Schemaelemente	Zustand
anonyme und nicht anonyme Typen	Kopfzustand
anonyme und nicht anonyme Typen	Endzustand
Elemente	Elementzustand
Attribute	Attributzustand
choice, all, Attributsequenz	Strukturzustand
min- und maxOccurs	Häufigkeitszustand

Tabelle 4: Schemabestandteile und die daraus erzeugten Zustände

chen eines Bytes. Das erste Byte eines jeden Zustands ist der sogenannte Header[27]. Die ersten drei Bits des Headers sagen aus, um welche Art von Zustand es sich handelt. Die restlichen Bits werden zustandsspezifisch als Flags eingesetzt. Jeder Zustand wird an einer bestimmten Adresse im Bytecodevektor abgelegt. Über diese Adresse verzweigen die Zustände in andere Zustände und bilden somit die Struktur des Schemas nach. Die Namen der einzelnen Elemente und Attribute werden, aufgrund ihrer unterschiedlichen Länge, in einem separaten Stringvektor abgelegt. Für den Vererbungsbaum wird eine separate Struktur angelegt, mit deren Hilfe ein Typecast codiert werden kann.

In der weiterführenden Literatur [NIE03] wird der Bytecode mehr als ein Konzept anstatt nur als ein vorkompiliertes Schema gesehen. So wird nach der Einführung des Bytecodes nicht mehr von Encodern und Decodern gesprochen, sondern nur von einem Bytecode-Interpreter. Der Grund dafür wird aus dem nächsten Unterabsatz klar. Die verzweigten einzelnen Zustände ergeben bereits nahezu vollständig das Automatenmodell für die Payload-Codierung und enthalten wichtige Daten für die Pfadcodierung, so dass Bytecode nur noch „interpretiert“ werden muss. In dieser Arbeit wird trotzdem weiterhin von En- bzw. Decodern gesprochen, was sich in Kapitel 9 auch als nützlich erweist.

[27] Dateikopf

6.5.1 Die Zustände

Für eine ausführliche Beschreibung der einzelnen Zustände wird auf die weitere Literatur verwiesen [NIE03]. Zur Veranschaulichung wird jedoch beispielhaft der Kopfzustand eines Typen beschrieben. Der Kopfzustand eines

Länge/Byte	Funktion
1	Header
2	Zeiger auf Name im Stringvektor
2	Zeiger auf Liste der Attribute
2	Zeiger auf Basistyp
2	Zeiger auf Vererbungsbaum
1	Länge des Strukturcodes für Context-BVC
1	Länge des Strukturcodes für Operand-BVC
2	Zeiger auf ersten Zustand des Typs

Tabelle 5: Kopfzustand eines Typs

Typen ist der Einstiegspunkt für die Codierung eines Elements von diesem Typen. Wie bereits erwähnt beginnt jeder Zustand mit einem Kopfelement. Mit den nächsten zwei Byte wird auf den Namen des Typs im Stringvektor verwiesen. Das nächste Feld zeigt auf einen Sequenz-Zustand, welcher eine Liste mit Zeigern auf die Attribute des Typs enthält. Falls der Typ von einem Basistypen erbt, wird ein Zeiger auf den Kopfzustand des Basistypen abgelegt. Unabhängig davon ist ein Zeiger auf die Position des Typen im Vererbungsbaum immer vorhanden. Mit Hilfe dieses Zeigers wird ein möglicher Typecast codiert. Für eine Pfadcodierung werden die Längen des Strukturcodes angegeben. Ist ein Element von diesem Typ beispielsweise ein Teil des Kontext-Pfades, weiß der Codec, wieviele Bits er braucht um die möglichen Kindelemente zu codieren. Mit dem letzten Datenfeld wird in den ersten Zustand verzweigt, mit dem die eigentliche Codierung beginnt.

Abbildung 9 zeigt wie der folgende Auszug aus einem XML-Schema im Bytecode dagestellt wird. Die einzelnen Blöcke stellen die verschiedenen Zustände dar, in der oberen Unterteilung steht jeweils der Name des Syntaxelements.

```
<complexType name="CT_B">
  <extension base="CT_A">
    <sequence>
      <element name= "E_d" type="Bit"/>
      <element name= "E_e" type="Bit" maxOccurs="4"/>
    </sequence>
```

```
    <attribute name="A_b" type="string"/>
    <attribute name="A_c" type="string"/>
  </extension>
</complexType>

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

Bei der Betrachtung fallen zwei Besonderheiten auf. Zum einen werden für Sequenzen von Elementen im Schema keine besonderen Zustände angelegt. Die Elemente in der `sequence` werden nur entsprechend ihrer Reihenfolge angeordnet und zeigen jeweils auf den Folgezustand. Die zweite Besonderheit ist, dass aus zwei Zuständen auf die Attributzustände („Attrib A_a, Attrib A_b und Attrib A_c") verwiesen wird. Dies liegt an der unterschiedlichen Behandlung von Attributen durch die Pfad- und Payloadcodierung. Für die Payloadcodierung wird im Kopfzustand auf einen Sequenzzustand(„Sequenz") verzweigt. Bei diesem Sequenzzustand handelt es sich nicht um einen Zustand für eine Sequenz im XML-Schema mittels `<sequence>`. Der Sequenzzustand enthält eine Liste von Zeigern auf Attributzustände. Diese Liste ist alphabetisch sortiert und enthält auch Zeiger auf die Attribute aller Basistypen des zu codierenden Typs. Der Grund hierfür ist folgender:

Bei der Payloadcodierung wird die XML-Instanz seriell in der Reihenfolge der Instanziierung codiert. Dabei trifft der Codec zuerst auf die Attribute des Typen und dessen Basistypen bevor er den Inhalt bzw. die Struktur codiert. Da, wie bereits erwähnt, der Bytecode das Automatenmodell darstellt, muss der Codec für die Payloadcodierung deshalb zuerst auf den Attribut-Automaten bzw. Attributzustände zugreifen können.

Für die Pfadcodierung wird am Ende des Endzustands („Type End") eine Liste angehängt mit Zeigern auf ausschließlich im Typen selbst deklarierte Attribute. Der Grund hierfür ist folgender:

Die Pfadcodierung stellt den Inhalt eines XML-Dokuments als Baum dar, in dem jeder Typ einen einzelnen Knoten darstellt. Besitzt ein Typ einen Basistypen, so stellt dieser Basistyp einen eigenen Knoten unterhalb des erbenden Typs dar. Da Attribute nur codiert werden, wenn ihr Vaterknoten bzw. der Typ in dem sie deklariert werden den Operanden-Knoten darstellt,

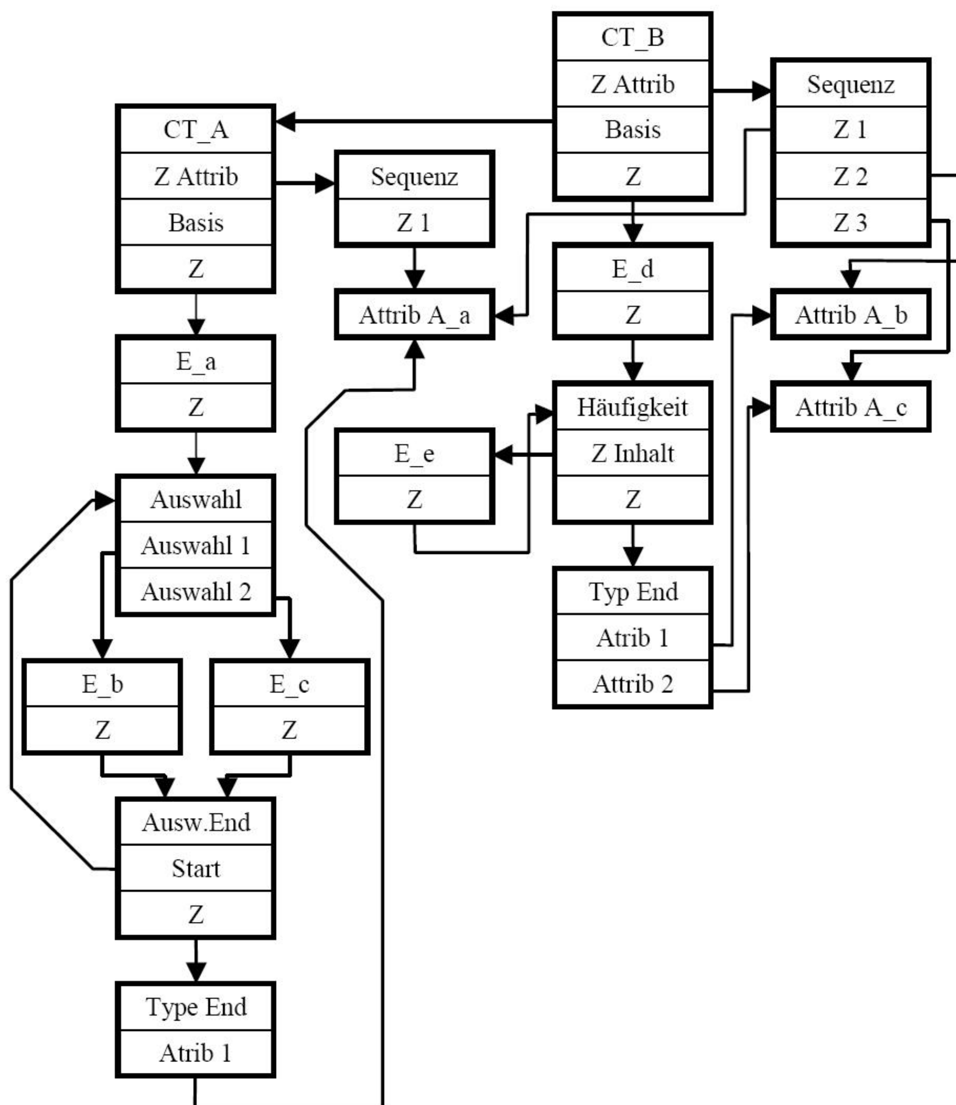

Abbildung 9: Umsetzung einer Syntaxdefinition im Bytecode[NIE83]

sind für die Pfadcodierung nur die im Typen direkt deklarierten Attribute relevant.

Weitere Strukturen wie der Vererbungsbaum werden hier nicht näher beschrieben.

7 Verwendung des BiM-Verfahrens zur Lösung von Problemstellungen

In diesem Kapitel wird die Nutzbarkeit des BiM-Verfahrens zur Lösung von sich aus der Norm ergebenden Problemen untersucht. Die dabei untersuchten Punkten dienen sowohl dem Erreichen einer hohen Konformität, siehe Abschnitt 4.4, als auch der optimierten Verarbeitung einer CID-Datei auf der Baugruppe.

7.1 Namensraumbehandlung und Wildcards

Namensräume und Wildcards sind grundlegende Syntaxelemente eines XML-Schemas, die auch ihre Anwendung in der SCL finden und somit vom Binärformat direkt unterstützt werden müssen. Da jedoch diese Syntaxelemente von den ersten BiM-Implementierungen nicht unterstützt wurden bzw. die Unterstützung nicht dokumentiert war, wurde die Behandlung von Namensräumen und Wildcards in die Anforderungen an den BiM zur Implementierung aufgenommen. Bei der zum Zeitpunkt dieser Arbeit aktuellen Version V2 [MPEG06] des BiM-Verfahrens werden bereits Namensräume und Wildcards unterstützt. Die Realisierung im BiM-Verfahren ist für die folgenden Abschnitte von Interesse, da Namensräume und Wildcards die Grundlage für Erweiterungen der Hersteller im SCL-Dokument bilden. Aus diesem Grund wird die Behandlung von Namensräumen und Wildcards in den folgenden Unterabschnitten beschrieben.

7.1.1 Namensraumbehandlung

Bei der Namensraumbehandlung handelt es sich im Prinzip um das Erkennen verschiedener Namensräume in einem XML-Dokument. Diese Eigenschaft ist wichtig, damit bei Dokumenten, die aus Elementen verschiedener Namensräume bestehen und somit in verschiedenen Schemata deklariert wurden, besonders behandelt werden können. Da der Namensraum eines Elements teil des XML-InfoSet ist, kann der Namensraum über einen normalen XML-Parser, wie ihn auch das BiM-Verfahren verwendet, abgerufen werden. Um den Namensraum-Namen in den Bitstrom mitzucodieren, gibt es verschiedene Möglichkeiten. Eine Möglichkeit ist, den Name eines Elements anstatt mit einem Präfix mit der vollständigen URI abzulegen und das Präfix zu verwerfen. Diese Vorgehensweise verlängert jedoch die Größe der zu verarbeitenden Element- und Attributnamen unnötig. In der aktuellen Version des BiM-Verfahrens werden die Namensraum-Namen in der DecoderInit des Bitstroms abgelegt, die Elemente und Attribute erhalten einen Verweis

auf ihren jeweiligen Namensraum, die Präfixe werden nicht mitcodiert. An der Schnittstelle zum Decoder, wie in Absatz 9.1.2 beschrieben, gibt es nun mehrere Möglichkeiten für den Rückgabewert eines Element- bzw. Attributnamens. Möglich sind eine getrennte Übergabe der Namensraum-URI und des Elementnamens, die Übergabe einer zusammenhängenden Zeichenkette und zuletzt die Übergabe des Elementnamens mit einem Präfix, welches vorher der URI erneut zugeteilt werden muss. Da sich die Form des Rückgabewertes leicht variieren lässt, kann diese Entscheidung bei der tatsächlichen Implementierung getroffen werden.

Mit der Erkennung eines fremden Namensraums eines Elements ist noch nicht die Codierung dieses Elements abgehandelt. Eine spezielle Codierung der verteilt deklarierten Elemente wird in Abschnitt 7.2 beschrieben.

7.1.2 Wildcards

In XML gibt es mehrere Möglichkeiten Wildcards zu deklarieren. Im SCL-Schema werden Widcards wie folgt deklariert:

```
<xs:any namespace="##other" processContents="lax" minOccurs="0"
maxOccurs="unbounded"/>
...
<xs:anyAttribute namespace="##other" processContents="lax"/>
```

Die Deklaration sagt aus, dass die Erweiterungen aus einem anderen Namensraum stammen müssen (`##other`) und der Prozessor die Erweiterung nur bei bekanntem Namensraum validieren muss (`lax`). Für die Codierung einer solchen Wildcard muss also eine Namensraumbehandlung vorhanden sein. Der Codec prüft beim decodieren, ob ihm das Schema als Bytecode vorliegt. Ist dies nicht der Fall verwirft der Codec des BiM-Verfahrens diese Erweiterung. Eine andere Möglichkeit der Codierung von Elementen und Attributen, deren Namensraum nicht bekannt ist, ist es, die Erweiterung vollständig als Zeichenkette zu encodieren. Dies wird vom BiM-Verfahren in der jetzigen Version V2 jedoch nicht unterstützt. Ist das Schema zu der Erweiterung bekannt, wird die Erweiterung wie folgt codiert:

Alle Elemente und Attribute des Erweiterungsschemas müssen global deklariert sein, mit anderen Worten, sie müssen direkt unter dem Schema-tag deklariert werden. Die Zustände aus dem Bytecode des Erweiterungsschemas und die Erweiterungsschemas selbst bekommen eine ID zugewiesen. Für ein besseres Verständnis kann man sich vorstellen, dass ein Erweiterungselement im Bytecode als ein Häufigkeitszustand gefolgt von einem Auswahlzustand dargestellt wird. Die oben beschriebene ID für das Element entspricht dem Code für das ausgewählte Element im Auswahlzustand. Alle

Elemente des Erweiterungsschemas gehören quasi zu einer Auswahlgruppe im Bytecode. Sind mehrere Erweiterungsschemata bekannt, wird jeweils die Auswahlgruppe des einzelnen Schemas in eine übergeordnete Auswahlgruppe der verschiedenen Schemas eingefügt. Trifft der Codec demzufolge beim decodieren auf ein Wildcard-Element müssen folgende Schritte durchgeführt werden: Zunächst wählt der Decoder anhand der Schema-ID die Auswahlgruppe der mögliche Elemente aus, anschließend wird über die Element-ID der Zustand für das zu decodierende Element ausgewählt und das Element im Bitstrom decodiert.

Bei der Codierung von Wildcard-Attributen muss anders vorgegangen werden. Wie in Abschnitt 6.5 beschrieben wird, wird sowohl vom Endzustand als auch vom Sequenzzustand auf Attributzustände verwiesen. Die Deklaration eines `<anyAttribute>` sagt aus, dass beliebig viele Wildcard-Attribute bei der Instanziierung gesetzt werden dürfen, für den Bytecode-Compiler ist die Anzahl somit unbekannt. Da es keine Häufigkeitszustände innerhalb der Endzustände und Sequenzzustände gibt und nicht beliebig viele Platzhalter für Attribute in die Listen eingetragen werden können, kann ein Wildcard-Attribut mit den bestehenden Zuständen zunächst nicht codiert werden. In [MPEG06] wird deshalb wieder mit zusätzlichen IDs oder Flags gearbeitet. Zuerst wird mit einem Flag angezeigt, dass ein Wildcard-Attribut beim codieren des Typs auftritt. Daraufhin wird, wie beim Wildcard-Element, über eine Schema-ID und eine Attribute-ID der passende Attributzustand ausgewählt und codiert. Anschließend folgt ein Flag welches angibt, ob ein weiteres Wildcard-Attribut instanziiert wurde.

Da es sich bei der Codierung einer Wildcard, deren Schema dem Codec bekannt ist, um Erweiterungen der Herstellerfirma handelt, hat diese Verfahrensweise jedoch viele Nachteile. Deshalb werden in Abschnitt 7.2 spezielle Methoden zur Codierung von verteilt deklarierten Elementen untersucht. In Abschnitt 7.2 werden auch die Nachteile der in diesem Abschnitt beschriebenen Methode erklärt.

7.2 Codierung verteilt deklarierter Elemente

In XML-Dokumenten ist es möglich, Elemente und Attribute aus einem anderen Namensraum als dem Zielnamensraum des Dokuments zu instanziieren. Dazu muss im XML-Schema des Dokuments eine Wildcard an der entsprechenden Stelle deklariert werden, welche Elemente bzw. Attribute aus anderen Namensräumen zulässt. Damit das Element bzw. Attribut aus dem fremden Namensraum vom BiM-Verfahren typentsprechend codiert werden kann, wird zusätzlich das Schema zu dem fremden Namensraum benötigt. Der beim Codieren der Wildcards vom BiM-Verfahren verwendete Algorithmus

wie er in Abschnitt 7.1.1 beschrieben wurde, ist für den effizienten Einsatz in einer intelligenten Baugruppe jedoch nur bedingt geeignet.

Im SCL-Schema werden Elemente und Attribute aus fremden Namensräumen für Erweiterungen genutzt, weswegen in diesem Abschnitt der Begriff Erweiterung parallel genutzt wird. Es ist davon auszugehen, dass bei Erweiterungen der Herstellerfirma ein XML-Schema für die Erweiterungen vorliegt, welches auch Erweiterungsschema genannt wird. Aus der Beschreibung des ursprünglichen Verfahrens zur Codierung der Wildcards geht hervor, dass Erweiterungen im Erweiterungsschema global deklariert werden müssen. Geht man von einer großen Zahl an verschiedenen Erweiterungen im Erweiterungsschema aus, werden bei dem ursprünglichen Verfahren zur Codierung von Wildcards alle deklarierten Erweiterungen als mögliche Instanzen im XML-Dokument angesehen. Werden beispielsweise 25 Erweiterungen im Erweiterungsschema deklariert, muss der Codec, wenn er auf eine dieser Erweiterung im XML-Dokument trifft, eine Auswahl mit 25 Möglichkeiten codieren. Geht man nun davon aus, dass an dieser Stelle im Dokument nur drei Erweiterungen sinnvoll sind, lässt diese große Auswahl das Binärformat unnötig anwachsen, zumal die Auswahl bei Häufigkeiten größer eins mehrmals codiert werden muss. Ebenfalls muss der Hersteller eine zusätzliche Kontrollinstanz einführen welche prüft, ob die eingefügte Erweiterung im XML-Dokument semantisch sinnvoll ist bzw. verarbeitet werden kann. Die Erweiterungen von Fremdfirmen deren Schema nicht vorliegt können, solange sie nicht im `Private` Element eingefügt sind, weiterhin verworfen werden. Das Verwerfen der Erweiterungen ist kein Verstoß gegen die IEC61850, siehe Unterabschnitt 4.3.2, und würde eine überflüssige Vergrößerung des Binärformats verhindern.

Um die Codierung von Erweiterungen der Herstellerfirma zu optimieren, muss dem Codec mitgeteilt werden, welche Erweiterung an welcher Stelle auftreten dürfen. Der Codec muss daraufhin nur die erlaubten Erweiterungen codieren. Aus dieser groben Umschreibung der Lösung ergeben sich zwei Punkte, die untersucht werden müssen. Als erstes muss eine geeignete Form gefunden werden, mit der dem Codec mitgeteilt werden kann, welche der Erweiterungen an welchen Stellen erlaubt sind. Der zweite Punkt, der untersucht werden muss, ist die technische Umsetzung im Codec und im Bytecode. Beide Punkte werden hier getrennt dargestellt.

7.2.1 Beschreibung der Einschränkung von Erweiterungen

In diesem Unterabschnitt wird beschrieben wie die Information über eine lokale Eingrenzung einer Erweiterung an den Codec weitergegeben werden kann. Dabei muss ein Kompromiss zwischen der Verarbeitbarkeit für das

BiM-Verfahren und der Anwendbarkeit für die Herstellerfirma gefunden werden. Für diesen Unterabschnitt wird von dem SCL-Schema und einem Erweiterungsschema ausgegangen. Da die Abhängigkeit zwischen dem Codec und der Anwendung möglichst gering sein soll, um eine Austauschbarkeit der Komponenten zu ermöglichen, ist eine properitäre Lösung nicht empfehlenswert. Eine properitäre Lösung ist beispielsweise, dass direkte einprogrammieren der lokalen Eingrenzung der Erweiterungen in den Codec. Die Änderung einer lokalen Eingrenzung hätte eine Neugenerierung des Codec zu Folge. Eine andere Möglichkeit ist es, über eine Initialisierung in einem neutralen Dateiformat, dem BiM-Codec bzw. Bytecodecompiler die lokale Einschränkung der Erweiterungen mitzuteilen. Anstatt jedoch eine weitere Datei zur Initialisierung zu erstellen und zu pflegen, kann die Information durch das Schema mitgeteilt werden. Alle hier vorgestellten Lösungsmöglichkeiten verwenden ein XML-Schema zur Beschreibung der lokalen Eingrenzung der Erweiterung. Eine gute Anwendbarkeit für die Herstellerfirma im Zusammenhang mit XML-Schema bedeutet, dass sich das Schema mit Standardwerkzeugen erstellen und prüfen lässt.

Zusammenfügen des SCL- und Erweiterungsschemas

Die vermeidlich einfachste Möglichkeit ist das Zusammenfügen des SCL-Schemas und des Erweiterungsschema. Die Erweiterungen werden dabei einfach direkt in der Elementdeklaration deklariert in deren Instanz diese Erweiterung auftreten darf. Der Mechanismus der Wildcard wird dabei vollständig umgangen und der Bytecode-Compiler kann direkt aus dem neuen Schema seinen Bytecode erstellen. Diese vermeidlich einfache Lösung hat jedoch Nachteile, welche sich nicht mit den Anforderungen an eine Baugruppe nach IEC61850 decken. Das Problem bei dieser Vorgehensweise ist, dass laut W3C-Norm ein Schema genau einem Namensraum angehört. Die Erweiterung müsste demnach im gleichen Namensraum wie das SCL-Schema liegen. Das dabei entstehende Schema wäre semantisch nicht mehr gleich dem SCL-Schema aus der IEC61850. Eine normgerechtes SCL-Dokument einer Fremdfirma, die keine Erweiterung des Erweiterungsschemas enthält, kann weder codiert noch validiert werden. Auf der anderen Seite kann ein SCL-Dokument, validiert nach dem neuen Schema der Herstellerfirma, nicht mehr mit einem normkonformen SCL-Schema validiert werden, da laut IEC61850-6 sich die Erweiterung in einem anderen Namensraum befinden muss. Zusätzlich würde ein firmenspezifisches Schema einen höheren Aufwand bezüglich der Erstellung und Pflege bedeuten. Als Schlussfolgerung wird eine Beeinflussung der grundlegenden Semantik des SCL-Schemas für die weiteren Lösungsansätze zunächst ausgeschlossen.

Verarbeitungsanweisung

Das grundlegende Prinzip aller Lösungsansätze ist, dass der Bytecode-Compiler einen Bytecode kompiliert, der sich nach außen für den Codec wie ein Schema darstellt. Dazu muss der Bytecode-Compiler wissen, bei welchen lokalen Elementen die Erweiterung erlaubt ist. Die Erweiterungen müssten demnach Bezug auf die Struktur des SCL-Schemas nehmen. XML-Schema bietet jedoch keine Möglichkeit, aus einem Schema bezug auf ein anderes Schema aus einem anderen Namensraum zu nehmen. Das Schemaelement `<import>`, welches verschiedene Schemainhalte aus verschiedenen Namensräumen kombiniert, wird bewusst erst in einem der folgenden Absätze beschrieben. Eine einfache Möglichkeit ist es, Information über die Struktur des SCL-Schemas direkt in die Deklaration der Erweiterung einzufügen. In XML-Schema werden für Verarbeitungsinformationen bestimmte XML-Schemaelemente oder bestimmte Darstellungsformen definiert. Der Vorteil bei der Verwendung dieser Schemaelemente und Darstellungsformen liegt bei der Verarbeitbarkeit für den Bytecode-Compiler. Der Bytecode-Compiler verwendet zum parsen des XML-Schemas einen Standard-Schemaprozessor bzw. Parser, welcher diese Schemaelemente weitergibt. Anstatt diese Schemaelemente wie bisher zu verwerfen, muss der Bytecode-Compiler diese entsprechend verarbeiten.

Die erste Möglichkeit ist das Einfügen einer `<annotation>`. Innerhalb einer `<annotation>` gibt es die Möglichkeit eine `<appinfo>` einzufügen. Im Schemaelement `<appinfo>` können beliebige Verarbeitungsanweisungen stehen. Der Bytecode-Compiler kann die Anweisung interpretieren, andere Schemaprozessoren ignorieren sie einfach. Jedes Element und Attribut in dem Erweiterungsschema muss, im Fall der Verwendung dieser Darstellungsform, diese Verarbeitungsanweisung enthalten. Das folgende Beispiel zeigt die Anwendbarkeit:

```
<xs:element name="ExtraText">
  <xs:annotation>
    <xs:appinfo>
      lokale Einschränkung: LN,DOI
    </xs:appinfo>
  </xs:annotation>
  <!--eigentliche Deklaration des Elements ExtraText -->
</xs:element>
```

Das Beispiel stellt einen Ausschnitt aus einem möglichen Erweiterungsschema dar. Das Element `"ExtraText"` soll nur in Elementen des SCL-Schemas mit den Namen LN und DOI vorkommen. Eine Auflistung der Elementna-

men stellt nur eine Möglichkeit zur Übergabe der Informationen dar. Innerhalb einer <appinfo> kann der Inhalt beliebig sein, es können sogar XML-SchemaelementE verwendet werden.

Eine andere Möglichkeit, dem Bytecode-Compiler Verarbeitungsanweisungen zu geben, ist die „Process-Information“ oder kurz PI. Diese sieht wie folgt aus: `<?ext [Anweisungen]?>`. Dabei dient das `ext` der Identifikation und kann beliebig umbenannt werden. Wie zuvor muss jedes Element und Attribut in dem Erweiterungsschema eine PI enthalten. Beide Methoden lassen sich mit dem gleichen Aufwand implementieren. Aufgrund des geringeren Schreibaufwandes, wurde sich im Rahmen dieser Arbeit für die PI entschieden. Die Schreibweise der Anweisung kann beliebig festgelegt werden. Die einfachste Möglichkeit eine lokale Einschränkung auszudrücken ist die, Elementnamen der Elemente in denen die Erweiterung auftreten darf, hintereinander durch ein Komma getrennt in die PI zu schreiben. Diese Form der Anweisung umfasst alle Anwendungsfälle und ist für jeden lesbar. Das folgende Beispiel hat die gleiche Aussage wie das Beispiel im Absatz zuvor:

```
<xs:element name="extraText">
<?ext LN,DOI ?>
<!--eigentliche Deklaration des Elements ExtraText -->
</xs:element>
```

Der Form halber wird hier noch mal ein Ausblick auf die mögliche Darstellungsform in der Zukunft gegeben. Der neue W3C-Standard XML1.1 ermöglicht es in XML-Schema eigene „Top-Level“ Anweisungen zu formulieren. Top-Level Anweisungen sind Anweisungen wie `<xs:element>`, `<xs:any>` und `<xs:Attribute>`. Für die Verarbeitungsanweisung können beispielsweise Top-Level Attribute in einem weiteren Schema deklariert werden. Eine Anweisung würde dann wie folgt aussehen:

```
<xs:element name="extraText" abc:Local="LN,DOI">
<!--eigentliche Deklaration des Elements ExtraText -->
</xs:element>
```

Das Präfix `abc:` deutet auf das neue Schema mit der Deklaration des Top-Level Attributs `Local`. Der Vorteil dieser Darstellungsform ist, dass durch das Schema des Attributs `local` die Eingabemöglichkeit z.B. nur auf Buchstaben, getrennt durch ein Komma, beschränkt werden kann. Der Nachteil ist, dass alle Komponenten, die mit dem Erweiterungsschema arbeiten, in der Lage sein müssen XML1.1 zu verarbeiten.

Import von Namensräumen

Die Möglichkeit, Elemente und Attribute aus einem anderen Namensraum in ein Schema mit dem Schemaelement `<import>` einzubinden, wird in diesem Absatz separat dargestellt. Zunächst wird der Mechanismus anhand eines Beispieles beschrieben:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.iec.ch/61850/2003/SCL"
    xmlns:ext="www.Herstellerfirma.org/Erweiterungsschema"
    targetNamespace="http://www.iec.ch/61850/2003/SCL">

<xs:import namespace="www.Hersteller.org/Erweiterungsschema"/>

<xs:element name="LN">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="ext:ExtraText"/>
        <!--eigentliche Deklaration des Elements LN -->
      </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
```

Die Deklaration des Elements `<LN>` weicht im Beispiel von der tatsächlichen Deklaration im SCL-Schema ab und dient nur der Beschreibung der `<import>`-Anweisung. Dem Namensraum in der `<import>` Anweisung wird im Kopf das Präfix `ext` zugewiesen. Im Element LN wird nun ein Element eingebunden, welches das Element `ExtraText` aus dem Erweiterungsschema referenziert. Das eingefügte Element übernimmt quasi die Deklaration der Erweiterung. An jeder Stelle im SCL-Schema, in der Erweiterungen vorkommen dürfen, können Erweiterungen über eine solche Referenzierung eingebunden werden. Die Instanz der Referenz übernimmt den Namensraum des Erweiterungsschemas. Bereits in Unterabschnitt 4.3.2 wurde jedoch darauf hingewiesen, dass die Wildcards eines jeden Elements in der SCL über das Element `tBaseElement` vererbt werden. Um nun in bestimmten Elementen nur bestimmte Erweiterungen einzubinden, müssen alle Vererbungen aufgelöst werden. In der sich dabei ergebenden flachen Hierarchie wird der Typ jedes Elements anonym deklariert, wodurch die Größe des Bytecodes anwächst. Ob bei der Nutzung des Schemaelements `<import>` das SCL-Schema semantisch grundlegend beeinflusst wird, kann nicht eindeutig gesagt werden. Ein SCL-Dokument inklusive der Erweiterungen ist sowohl

nach dem neu erzeugten Schema als auch nach dem original SCL-Schema validierbar. Ein SCL-Dokument einer Fremdfirma ohne die Erweiterungen, welche eine minimale Häufigkeit von größer gleich eins besitzen, ist jedoch mit dem erzeugten Schema nicht validierbar. Zwar eignet sich das Schemaelement `<import>` aus diesem Grund nicht für eine lokale Eingrenzung der Erweiterungen, jedoch kann die Darstellung der lokalen Eingrenzungen mit dem hier beschriebenen Verfahren zur Analyse verwendet werden. Nachdem ein Erweiterungsschema mit Eingrenzungen in Form einer PI erstellt wurde, kann beispielsweise durch ein selbst erstelltes Programm ein Schema mit dem beschriebenen `<import>` Mechanismus erzeugt werden. Anhand dieses erzeugten Schemas kann nun geprüft werden, ob die Erweiterungen an der richtigen Stelle eingefügt wurden. Ebenso kann ein solches Schema zu Kontrollzwecken aus dem Bytecode erzeugt werden. Dies würde zusätzlich der Überprüfung des Bytecode-Compilers dienen.

7.2.2 Spezielle Codierung der Erweiterungen

In Abschnitt 6.5 wurde beschrieben, dass die Zustände des Bytecodes mit der Codierung direkt verbunden sind. Die in diesem Abschnitt beschriebene Codierung der Erweiterung, beschreibt im Wesentlichen eine Anpassung des Bytecodes bzw. des Bytecode-Compilers. Als Darstellungsform der lokalen Einschränkung für das Erweiterungsschema wird von der „Process Information“ oder kurz PI ausgegangen. Die beschriebenen Algorithmen dienen nur der Prüfung der Machbarkeit, sie müssen nicht den tatsächlichen Algorithmen entsprechen.

Vollständige Hierarchie Auflösung

Eine bereits erwähnte Möglichkeit ist es, die Hierarchie auf eine Ebene zu verkleinern. Jeder Elementzustand hat eine eins zu eins Beziehung zu seinem Kopfzustand bzw. Typen. Anders formuliert, jedes Element hat einen eigenen Typen. Dadurch enthält jedes Element direkt eine Wildcarddeklaration. Der Bytecode-Compiler ließt die PI einer Erweiterung im Erweiterungsschema und erzeugt die Zustände für diese Erweiterung. Anschließend sucht der Bytecode-Compiler den Elementzustand mit dem Namen der in der PI steht. Von dort aus arbeitet sich der Compiler bis zu den Zuständen vor, welche der Wildcard entsprechen, und fügt die Zustände, die für die Erweiterung erstellt wurden, vor der Wildcard ein. Der Codec muss zusätzlich bezüglich der Codierung von Dokumenten ohne Erweiterung angepasst werden. Bei diesen Dokumenten ist davon auszugehen, dass es sich um SCL-Dokumente von Fremdfirmen handelt. Da der Bytecode statisch ist und sich nicht während

der Laufzeit ändert, muss der Codec bei der Codierung eines fremden SCL-Dokuments die Zustände für die Erweiterungen der Herstellerfirma überspringen. Dies ist notwendig, weil unter Umständen die Erweiterung eine minimale Häufigkeit von größer gleich eins besitzt. Um zu erkennen, dass der Elementzustand der Erweiterung entspricht, kann der Namensraum des Elementzustands geprüft werden. Die Überprüfung des Namensraums bei jedem Elementzustand verlangsamt jedoch die Codierung. Das zentrale Problem dieser Form der Codierung von Erweiterungen ist jedoch der erhöhte Speicherbedarf des Bytecodes durch die flache Typhierarchie. Die Zustände der Basistyen werden für jedes Element, dessen Basistypen sie darstellen, wiederholt erzeugt.

Minimierte Hierarchieauflösung

Für die folgenden Lösungsansätze wird noch mal auf die Instanziierungsreihenfolge bei Vererbungen eingegangen. Ein Basistyp wird vollständig in einer Sequenz vor der Deklaration des eigentlichen Typen eingefügt. Die Elemente und Attribute des Basistypen müssen also vor den restlichen Elementen und Attributen eines Elements instanziiert werden. Wie bereits in Abschnitt 6.5 beschrieben, besitzen alle Kopfzustände einen Zeiger auf ihren Basiszustand. Der Codec geht beim Codieren eines Typen bis zu dessen letztem Basistypen vor, der selbst keinen Basistypen mehr besitzt, und arbeitet sich rückwärts an den Basistypen entlang bis zu den Zuständen des Ausgangstypen. Bei der Instanziierung eines SCL-Elements müssen beispielsweise die Erweiterungen als erstes instanziiert werden, da die Wildcards aller SCL-Elemente über den Basistypen `tBaseElement` geerbt werden. Daraus kann man wiederum folgern, dass bei der Codierung eines SCL-Dokuments der Kopfzustand von `tBaseElement` bei jedem zu codierenden Element zuerst aufgerufen wird.

Auf dieser Überlegung baut der zweite Lösungsansatz zur Codierung von Erweiterungen auf. Die Intention dieses Ansatzes ist die Ausdehnung des Speicherbedarfs, verursacht durch die Verkleinerung der Typhierarchie, zu verhindern. Anstatt die gesamte Typhierarchie aufzulösen sollen nur die Zustände des Typen `tBaseElement` in die jeweiligen Elementtypen reingezogen werden. Dadurch enthält jeder Elementtyp eine Wildcard und es kann, wie bereits im oberen Absatz beschrieben, der Zustand für die jeweilige Erweiterung im Elementtypen direkt eingefügt werden. Technisch stellt sich dieser Lösungsansatz jedoch als äußerst kompliziert heraus. Beim codieren muss der Codec zuerst die Zustände der Erweiterung und die Zustände von `tBaseElement` abarbeiten, anschließend muss der Codec die Zustände der restlichen Basistypen ohne `tBaseElement` codieren, um anschließend wieder mit den Zuständen des Ausgangstypen den Typen vollständig zu codieren.

Eine Möglichkeit dies im Bytecode darzustellen könnte folgende sein:

Zunächst wird der Zeiger auf den Kopfzustand des Typs `tBaseElement` aus den Kopfzuständen der direkt erbenden Typen gelöscht. Anstatt einen Elementzustand auf seinen ursprünglichen Kopfzustand verweisen zu lassen, lässt man einen Elementzustand auf den Kopfzustand von `tBaseElement` zeigen. Der Typ von `tBaseElement` wird um die Erweiterungen erweitert. Am Endzustand angekommen springt der Codec wieder auf den Elementzustand. Dieser Elementzustand muss nun um ein weiteres Datenfeld erweitert werden, in dem er auf den eigentlichen Kopfzustand seines Typen verweißt.

Der Einsatz dieser technischen Umsetzung ist, aufgrund ihrer derzeitigen Komplexität, nicht empfehlenswert.

Bytecodefragmente

Das dritte Verfahren wählt einen völlig anderen Ansatz. Die Idee, die zu diesem Ansatz geführt hat, ist folgende: Anstatt aus dem SCL-Schema und dem Erweiterungsschema einen Bytecode zu erstellen, werden zwei getrennte Bytecodes erstellt, welche über eine Schnittstelle miteinander verbunden werden. Zwar kann ein Codec nur mit einem Bytecode arbeiten, jedoch wird die Grundidee, die Zustände für die Erweiterung von den anderen Zuständen zu trennen, für das in diesem Absatz beschriebene Verfahren verwendet.

Der Bytecode für das SCL-Schema wird zunächst vollständig erstellt. Beim Parsen des Erweiterungsschemas erstellt der Bytecode-Compiler eine Tabelle mit allen Elementnamen der SCL-Elemente, die in den „Process-Informations" jeder Erweiterung eingetragen sind. Jeder Elementname tritt dabei nur einmal auf. Anschließend werden alle Erweiterungen, die in dem jeweiligen SCL-Element verwendet werden dürfen, den Namen der SCL-Elemente in der Tabelle zugeordnet. Der Bytecode-Compiler erzeugt die Zustände für die einzelnen Erweiterungstypen. Dies kann bereits vor dem Anlegen der Tabelle passieren. Anhand der Tabelle erzeugt nun der Bytecode-Compiler zu jedem Elementnamen aus der SCL ein Bytecodefragment. Ein Bytecodefragment ist je nach Wunsch der Herstellerfirma eine Auswahl oder eine Sequenz der Erweiterungen, die in einem bestimmten SCL-Element vorkommen dürfen. Die eingetragenen Namen der Erweiterungen werden nun durch einen Zeiger auf das jeweilige Bytecodefragment ausgetauscht. Jedem Namen eines SCL-Elements kann nun, sofern es erweitert wird, ein Bytecodefragment zugeordnet werden. Beim codieren einer SCL-Instanz arbeitet der Codec mit dem aus dem SCL-Schema erstelltem Bytecode. Trifft der Codec beim codieren auf ein SCL-Element, dessen Typ von `tBaseElement` erbt, passiert folgendes:

Der Codec greift auf den Elementzustand des zu codierenden SCL-Ele-

ments zu, von dem er auf dessen Typen bzw. Kopfzustand verzweigt wird. Von dem Kopfzustand aus wird der Codec weiter an den jeweiligen Basistypen verzweigt, bis er auf den Typen `tBaseElement` trifft. Trifft der Codec auf den Wildcard-Zustand, wird ein spezieller Algorithmus abgearbeitet. Zunächst wird über den Interpreterstack, der in Abschnitt 6.5 beschrieben wurde, zurückverfolgt, welches SCL-Element Ausgangspunkt für Codierung des Typen war und die Wildcard enthält. Mit Hilfe des Namens des SCL-Elements kann nun in der Tabelle das passende Bytecodefragment gefunden werden. Der Codec arbeitet dieses Bytecodefragment durch, springt an die Stelle im Bytecode an dem der Wildcard-Zustand den Algorithmus angestoßen hat, und arbeitet von dort aus die Folgezustände ab.

Die Vorgehensweise um die Tabelle zu erstellen und der Algorithmus, der das Fragment einfügt, sind nur beispielhaft erklärt. Um das genaue Vorgehen zu beschreiben, muss der Quellcode des BiM-Verfahrens vorliegen. Für Erweiterungen in Form von Attributen sieht der Ablauf ähnlich aus. Da es für Attribute jedoch keine Auswahlgruppe oder ähnliches gibt, müssen die Attributzustände des Erweiterungsschema in einer bestimmten Struktur abgelegt werden, damit anhand des SCL-Elementnamens auf alle Attribute, welche für das SCL-Element erlaubt sind, gesammelt zugegriffen werden kann. Alternativ kann sich eine Herstellerfirma dazu entscheiden, für Erweiterungen nur Elemente und keine Attribute zu verwenden. Dies würde jedoch zu Lasten der Lesbarkeit gehen.

Um nicht in Konflikt mit dem Mechanismus der Codierung von Wildcard-Elementen, wie er in Unterabschnitt 7.1.2 beschrieben wird, zu kommen, sollte der gesamte Mechanismus durch das Verfahren der Bytecodefragmente ersetzt werden.

7.3 Abwärtskompatibilität trotz Schemaevolution

Eine der größten Schwachstellen der IEC61850 ist es das Verhalten bei Schemaevolution nicht vorzuschreiben. Der Begriff Schemaevolution beschreibt, dass sich ein Schema bezüglich seines Inhaltes im Laufe der Zeit, durch Änderung der Ansprüche an den Informationsgehalt einer XML-Datei, ändert. Die Schemaevolution kann sowohl beim SCL-Schema als auch beim firmeneigenen Erweiterungsschema auftreten. Das Problem was sich aus der Schemaevolutionen ergibt ist, dass es zwischen dem Gerät und der Konfigurationsdatei zu Inkompatibilitäten kommen kann. Da der BiM-Algorithmus anhand des zugrundelegenden Schemas eine XML-Datei encodiert, muss zur Decodierung das gleiche Schema bzw. der gleiche Bytecode vorliegen. Nimmt man an, dass der Bytecode eines IED, der ausschließlich aus einem neuen Schema compiliert wurde, welches sich im Inhaltsmodell von dem alten Schema

des IED-Configurators unterscheidet, so würde eine von dem Configurator encodierte Datei auf dem IED gar nicht oder falsch decodiert werden. Auch ohne den Einsatz der BiM-Codierung wäre ein bestimmter Zugriff auf die XML-Daten nicht möglich. Zwar kann über eine DOM-API strukturiert auf die Daten zugegriffen werden, jedoch fehlt der Anwendung die Information über den möglicherweise neuen Aufenthalt der Daten.

Eine Möglichkeit, um eine solche Fehlerquelle abzufangen, ist die Versionierung der XML-Dateien und Schemata. Das heißt, Schema und XML-Datei erhalten jeweils ein Versionsmerkmal, welches zu dem anderen in Relation steht. Eine einfache Möglichkeit ist ein Versionsattribut in der XML-Datei und dem SCL-Schema. Ist die Zahlenkombination des Attributwertes gleich der, die im Schema hinterlegt worden ist, so ist davon auszugehen, dass die XML-Datei nach dem gleichen Schema validiert worden ist. Bei der Versionierung sollte versucht werden, dass mehrere noch nicht umgesetzte Änderungen im Schema zu einer Version zusammengefasst werden, um eine übermäßige Versionsvielfalt zu verhindern.

Mit diesem Verfahren könnte das IED aus unserem Beispiel den alten Parametersatz einfach ablehnen. Anstatt jedoch den Fehler nur zu blockieren, ist das Sicherstellen der Abwärtskompatibilität eine Möglichkeit, den Fehler nicht entstehen zu lassen. Unter Abwärtskompatibilität versteht man in diesem Fall die Tatsache, dass neuere Versionen einer Baugruppe mit den Configurationsdateien vorheriger Versionen kompatibel sind. Um dies sicherzustellen muss zuerst die Übertragung der Konfigurarationsdaten Abwärtskompatibilität ermöglichen. Zur Erfüllung dieser Anforderung musste sowohl das BiM-Codierverfahren als auch das Schemadesign angepasst werden. Die Aufwärtskompatibilität wird in Kapitel 8 beschrieben. Bevor die Lösungsansätze für die Abwärtskompatibilität in den folgenden Unterabschnitten erklärt wird, müssen die möglichen Änderungen bei einer Schemaevolution und ihre Auswirkung beschrieben werden.

Es wird in diesem Abschnitt nicht davon ausgegangen, dass die IEC in naher Zukunft konkrete Lösungsverfahren für die Problematik der Schemaevolution anbietet. In Kapitel 8 werden mögliche Verhaltensweisen der Norm bezüglich der Schemaevolution aufgezeigt.

7.3.1 Mögliche Änderungen und ihre Auswirkung

Prinzipiell kann davon ausgegangen werden, dass sich alles in einem Schema ändern kann. Allerdings ist es sinnvoll, jede Änderung, die eine Auswirkung auf die Abwärtskompatibilität hat, einzeln zu beschreiben. Dies hat den Vorteil, dass die Lösungsansätze auf diese exakte Änderungsmöglichkeiten geprüft werden können. Ebenso kann geprüft werden, ob gewisse Änderungen

beim Design des neuen Schemas sinnvoll oder technisch umsetzbar sind. Die in diesem Kapitel behandelten Lösungsansätze werden jedoch auf alle Änderungsmöglichkeiten hin geprüft.

Das Hinzufügen eines neuen Elements ist sicherlich der nachvollziehbarste Grund für eine Schemaevolution. Technisch ist eine solche Schemaevolution, wie in den folgenden Abschnitten beschrieben wird, leicht umzusetzen. Diese Änderung muss auf jeden Fall unterstützt werden.

Bei der Beurteilung der Änderung des Elementtypen muss genau unterschieden werden, wie sich die Änderung darstellt. Eine häufig eingesetzte Form der Schemaevolutionierung ist das Erweitern oder Begrenzen von Typen. Hierfür wird der Vererbungsmechanismus von XML genutzt. Der ursprüngliche Typ dient als Basistyp, der über die Schlüsselworte `extension` oder `restriction` eingebunden wird. Bei der Erweiterung mittels `extension` können Elemente und Attribute hinzugefügt werden. Bei der Verarbeitung einer solchen Erweiterung werden die neuen Elemente in einer Sequenz hinter dem Basistypen instanziiert. Bei der Einschränkung mittels `restriction` kann der Ursprungstyp innerhalb gewisser Grenzen eingeschränkt werden. Beide Änderungen an einem Typ müssen einen typecast ermöglichen. Das heißt, die Instanz des neuen Typen muss den Anforderungen an den Informationsgehalt des Basistypen genügen. Die technische Umsetzung einer solchen Schemaevolution wird vor allem von der Versionsschemaaufteilung in Unterabschnitt 7.3.3 unterstützt. Die Vererbung kann nicht bei anonymen Typen verwendet werden. Anonyme Typen sind Typen, die innerhalb eines Elements deklariert werden. Beim Schemadesign ist deshalb darauf zu achten, dass Typen möglichst nicht anonym deklariert werden.

Änderungen an einem Elementtypen, welche diesen nicht, wie oben beschrieben, erweitern oder einschränken, entsprechen einem komplett neuen Typen. Obwohl dies sehr unwahrscheinlich klingt, treten solche Änderung sehr häufig auf. Ein Beispiel ist die Änderung eines einfachen Elementtypen z.B. von einem `string` zu einem `integer`. Genauso ist es möglich, dass ein Typ sowohl um ein Element erweitert wird, ein anderes Element jedoch beschränkt wird. Trotz allem werden solche Änderungen durch die XML-Schema Syntax nicht unterstützt. Dies macht die technische Umsetzung sehr schwierig. Beim Design einer Schemaevolution sollte eine komplette Änderung des Typen vermieden werden. Bei der Begutachtung des neuen SCL-Schemaentwurfs treten jedoch komplette Typänderungen auf, weshalb diese möglichen Änderungen technisch unterstützt werden müssen.

Das Löschen von Elementen ist eine weitere mögliche Änderung, die in einer Schemaevolution vorkommt. Dabei muss zwischen dem Löschen von globalen und lokalen Elementen unterschieden werden. Ein lokales Element wird innerhalb eines Wurzelelements deklariert und besitzt somit einen Va-

terknoten. Demnach entspricht das löschen eines lokalen Elements einer kompletten Änderung des Typen seines Vaterknotens. Das löschen von globalen Elementen muss technisch wiederum gesondert behandelt werden. Eigentlich ist es nicht üblich, das Element ohne einen gemeinsamen Wurzelknoten zu deklarieren. Im SCL-Schema sind alle Elemente innerhalb des Elements `<SCL>` deklariert. Im Erweiterungsschema sind jedoch alle Erweiterungen globale Deklarationen. Allgemein sollte man auf das Löschen von Elementen bei einer Schemaevolution verzichten. Technisch muss sie trotz allem umgesetzt werden können.

Die Änderungsmöglichkeiten bei Attributen werden zusammengefasst beschrieben. Das Hinzufügen von Attributen ist über eine Erweiterung mittels `extension` möglich. Alle anderen Änderungen, wie Löschen und Typänderung, haben eine komplette Änderung des Elementtypen, in dem sie deklariert sind, zur Folge.

7.3.2 Versionsanweisung

Diese Lösungsmöglichkeit wurde zur selben Zeit wie die Möglichkeit zur lokalen Eingrenzung von Erweiterungen entworfen. Beide Verfahren wurden beim Entwurf des Erweiterungsschemas erarbeitet. Aus diesem Grund lag bei der Erarbeitung das Hauptaugenmerk auf der Schemaevolution des Erweiterungsschemas. Das Verfahren wurde erst im späteren Verlauf auf seine Eignung für die Evolution eines SCL-Schemas geprüft. Wie in Abschnitt 7.2 bereits beschrieben, macht man sich die Tatsache zu nutze, dass der Bytecode-Compiler ein Schema vorverarbeiten kann. Der Begriff Versionsanweisung deutet bereits darauf hin, dass durch das Schema Verarbeitungsanweisungen an den Bytecode-Compiler gegeben werden. Um genau zu sein, können hier die gleichen Mechanismen zur Mitteilung von Verarbeitungsanweisungen im XML-Schema genutzt werden, wie bei der lokalen Eingrenzung von Erweiterungen. Damit der Zugriff auf die Verarbeitungsanweisung sich gleich darstellt, sollten die Anweisungen für die lokale Eingrenzung und die Versionsanweisung dieselbe Mitteilungsform verwenden. Deshalb wird auch in diesem Unterabschnitt die „process-information“, oder auch PI, zur Beschreibung des Lösungsansatzes verwendet. Das Grundkonzept der Versionsanweisung ist die Versionierung wie sie zu Anfang des Abschnittes beschrieben wurde. Anstatt jedoch dem gesamten Schema ein Versionskennzeichen zuzuordnen, bekommt jedes Element sein eigenes Versionskennzeichen. Zum besseren Verständnis wird das Beispiel aus Kapitel 7.2 um die mögliche Darstellung einer Versionsanweisung erweitert.

```
<xs:element name="extraText">
```

```
<?ext locationRestriction="LN,DOI" versionInstruction="v2-v3"?>
</xs:element>
```

Die Schreibweise lehnt sich, auf Grund der besseren Lesbarkeit, an die Attributinstanziierung an. Da der technische Aufwand beim Bearbeiten der PI in dieser Form hoch ist, ist diese Form der Darstellung nicht Empfehlenswert. Für das bessere Verständnis wird diese Form der Darstellung jedoch beibehalten. Der Begriff `versionInstruction` ist gleichbedeutend mit der hier beschriebenen Versionsanweisung. Der Ausdruck `"v2-v3"` sagt aus, dass das Element `<extraText>` von Version v2 bis Version v3 verwendet werden kann. Hieraus wird ersichtlich, dass es bei diesem Verfahren nur ein Erweiterungsschema gibt. Die Änderungen vom evolutionierten Schema zum alten Schema werden einfach mit der entsprechenden PI in das Erweiterungsschema geschrieben. Mit dem bis jetzt beschriebenen Stand der Versionsanweisung können die folgenden Änderungen aus Unterabschnitt 7.3.1 umgesetzt werden:

- Hinzufügen neuer Elemente. Das Element `<extraText>` wird im Beispiel ab Version v2 neu hinzugefügt.
- Löschen eines Elements. Das Element `<extraText>` ist nur bis Version v3 gültig. Für Instanzen mit einer höheren Versionsziffer ist das Element quasi gelöscht. Die Tatsache ob das Element global oder lokal deklariert ist, spielt keine Rolle

Die Änderung des Elementtypen wird anhand des folgenden Beispiels beschrieben.

```
<xs:element name="extraText">
<?ext locationRestriction="LN,DOI" versionInstruction="v2-v3"?>
<--!Inhalt von Typ x-->
...
</xs:element>

<xs:element name="extraText">
<?ext locationRestriction="LN,DOI" versionInstruction="v4-v5"?>
<--!Inhalt von Typ y-->
...
</xs:element>
```

Die Tatsache, dass das Element `<extraText>` zweimal deklariert wird, ist sehr problematisch, da die W3C-Norm vorschreibt, dass der Elementname innerhalb eines Namensraums eindeutig ist. Das Gleiche gilt auch für

nicht anonym deklarierte Typen. Demnach wäre das Erweiterungsschema kein normkonformes Schema mehr. Dies schließt jedoch die Verwendung der Versionsanweisung nicht aus. Durch eine Anpassung des Bytecode-Compilers würde dieser bei zwei gleichnamigen Elementen bzw. Typen ein versioniertes Element bzw. einen versionierten Typen anlegen. Die Beschreibung der möglichen technischen Umsetzung wird im letzten Absatz beschrieben.

Die größten Nachteile eines nicht normkonformen Schemas liegen beim Schemadesign. Bei dem Design des Erweiterungsschemas kann die Syntaxkontrolle von Standardschemaeditoren nicht mehr verwendet werden. Dieser Nachteil relativiert sich, da davon auszugehen ist, dass eine Herstellerfirma zur Erstellung des Schemas eigene Programme mit einer selbstdefinierten Syntaxkontrolle verwenden wird.

Die in Unterabschnitt 7.3.1 beschriebene Möglichkeit, den Elementtypen zu erweitern oder einzugrenzen, ist mit der Versionsanweisung nicht möglich. Bei einer Erweiterung zum Beispiel, müsste der gesamte Typ neu deklariert werden, was zu einem erhöhten Schreibaufwand führt. Aus Gründen der Lesbarkeit sollten die Elemente und Typen mit gleichem Namen untereinander geschrieben werden.

Die Verwendung der Versionsanweisung für die Schemaevolution des SCL-Schemas stellt sich im wesentlichen gleich dar. Im Falle einer Schemaevolution kommt es ebenfalls zu doppelten Deklarationen von Elementtypen. Das dabei entstehende Schema dient lediglich dem Zweck, dem Bytecode mitzuteilen, welche Typen zu versionieren sind. Durch die Verletzung der XML-Schemaregeln ist eine Validierung gegen das erstellte Schema nicht möglich. Zusätzlich stellt das extra erzeugte Schema einen höheren Aufwand bezüglich der Versionsverwaltung dar.

7.3.3 Versionsschemaaufteilung

Bei dem Verfahren der Versionsschemaaufteilung wurde, wie der Name schon sagt, Hauptaugenmerk auf die Anforderungen der Schemaevolution und Versionierung gelegt. Die Versionsschemaaufteilung macht sich dabei das Schemaelement `<redefine>` zu Nutze, weshalb zunächst der `<redefine>` Mechanismus beschrieben wird.

```
Version1.xsd:

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:complexType name="tUnterstation">
  <xs:sequence>
```

```
    <xs:element name="Sationsname"/>
    <xs:element name="Spannungsebene"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="Unterstation" type="tUnterstation"/>

</xs:schema>

Version2.xsd:

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:redefine schemaLocation="Version1.xsd">
  <xs:complexType name="tUnterstation">
     <xs:extension base="tUnterstation">
       <xs:sequence>
         <xs:element name="Feld"/>
       </xs:sequence>
     </xs:extension>
  </xs:complexType>
</xs:redefine>

<xs:element name="Versorgungsgebiet" type="xs:string"/>

</xs:schema>
```

Die beiden Schemaausschnitte im Beispiel zeigen eine typische Anwendung des Schemaelements `<redefine>`, welches das bereits bekannte Schemaelement `<include>` impliziert. Das SCL-Schema beispielsweise verwendet, wie in Unterabschnitt 4.3.1 beschrieben, `<include>` zum Einbinden der einzelnen Schemadateien in die SCL.xsd. Das Schemaelement `<redefine>` ermöglicht zusätzlich zu der Einbindung der Deklarationen aus einem bestimmten Schema eine neue Deklaration von Typen auf Basis einer Vererbung. Dazu muss genau wie bei `<include>` der Namensraum des Schemas der eingebundenen Datei gleich dem Namensraum des Schemas der einbindenden Datei sein. Das oben gezeigte Beispiel hat folgende Aussage:

In der Datei Version1.xsd wird der Typ `tUnterstation` und das Element `Unterstation` deklariert. Für die neue Version des Schemas soll der Typ `tUnterstation` um das Element `Feld` erweitert werden, zusätzlich soll ein Element `Versorgungsgebiet` deklariert werden. In der Datei Version2.xsd

werden diese Änderungen durchgeführt. Dazu wird über `<redefine>` die Datei Version1.xsd eingebunden, anschließen wird der Typ `tUnterstation` neu deklariert, indem der Typ von seinem Vorgänger in Version1.xsd erbt und um das Element `Feld` erweitert wird. Das Element `Versorgungsgebiet` wird in Version2.xsd wie gewohnt deklariert. Bei der Instanziierung eines Dokuments nach Version2.xsd besitzt das Element `Unterstation` das Element `Feld`.

Wendet man dieses Beispiel auf das SCL-Schema an, muss bei einer Schemaevolution die Schemadatei SCL.xsd in einer neuen Schemadatei mit `<redefine>` eingebunden werden. Als Name für die neue Schemadatei wird SCL_v2.xsd eingeführt. Bei jeder weiteren Schemaevolution muss die aktuellste Schemaversion eingebunden werden, zum Beispiel bindet die Datei SCL_v3.xsd die Datei SCL_v2.xsd ein. Bei der Validierung bzw. Codierung einer SCL-Instanz gegen eine bestimmte Version des SCL-Schemas, muss in der Instanz als `schemaLocation` die jeweilige Version der SCL.xsd angegeben werden. Soll eine SCL-Instanz beispielsweise nach Version v3 validiert bzw. codiert werden, wird als Ort des Schemas die Schemadatei SCL_v3.xsd angegeben.

Es ist sicherlich anzweifelbar, ob der Umgang mit Namensräumen und Schemadateien in diesem Beispiel der Intention des W3C entsprechen. Ein Schema gehört genau einem Namensraum an, über den auch ein Schema identifiziert werden kann. Da jedes Versionsschema dem selben Namensraum angehört, entspricht das Schema immer der aktuellsten Version. Bei der Versionsschemaaufteilung wird dies jedoch ignoriert. Da jedoch das W3C selber keinen Mechanismus zur Versionierung zur Verfügung stellt, ist das Vorgehen bei der Versionsschemaaufteilung durchaus legitim.

Mit dem bis jetzt beschriebenen Stand der Versionsanweisung können die folgenden Änderungen aus Unterabschnitt 7.3.1 umgesetzt werden:

- Hinzufügen neuer Elemente. Das Element `Versorgungsgebiet` aus dem obigen Beispiel zeigt wie Elemente hinzugefügt werden.

- Ändern des Elementtypen durch erweiternde oder einschränkende Vererbung.

XML-Schema erlaubt beim Einsatz des `<redefine>` Mechanismus keine vollständige Neudeklaration des Elementtypen. Bei der neuen Version des SCL-Schemas, die sich zum Zeitpunkt dieser Arbeit im FDIS[28]-Stand befindet, treten jedoch Änderungen von Elementtypen auf, die nicht mit Vererbung dargestellt werden können, weshalb eine vollständige Neudeklaration unterstützt werden muss. Für die weitere Beurteilung des Verfahrens wird diese Schemaregel ignoriert und eine Verletzung der W3C-Regeln in kauf genommen.

[28]Final Draft International Standard

Bei einer vollständigen Typänderungen wird demnach im Versionsschema der Typ vollständig neu deklariert. Das Löschen eines Typs wird im SCL-Schema durch eine neue Typdefinition des Vatertypens des zu löschenden Typs realisiert. Diese Vorgehensweise ist für das Erweiterungsschema jedoch ungeeignet, da alle Erweiterungen global definiert sind und keinen Vatertypen besitzen. Geht man von einer lokalen Einschränkung im Erweiterungsschema in Form einer PI aus, lässt sich dieses Problem jedoch umgehen, indem bei jeder Versionierung einer Erweiterung die PI ebenfalls neu geschrieben wird. Soll ein Element im Erweiterungsschema im Lauf einer Schemaevolution gelöscht werden, muss die Erweiterung mit einer leeren PI neu deklariert werden und somit die Erweiterung für alle Elemente aus der SCL ausgeschlossen werden. Mit den hier beschriebenen Abweichungen von der W3C-Norm können alle Änderungen aus Unterabschnitt 7.3.1 versioniert werden.

Für die Anwendung der Versionsschemaaufteilung im Fall des SCL-Schemas, ergeben sich noch einige Besonderheiten. Bisher ist davon auszugehen, dass die IEC61850 im Falle einer Schemaevolution ein vollständig neues Schema rausgibt. Dies bedeutet, dass die Änderungen zur Erstellung der Versionsschemata eigenständig herausgesucht werden müssen. Dabei muss zusätzlich geprüft werden, ob ein veränderter Elementtyp vom Informationsgehalt der ursprünglichen Deklaration genügt, und somit eine Vererbung ermöglicht. Da in der SCL.xsd keine Typen deklariert werden, werden nahezu alle versionsbedingten Änderungen in den anderen sieben Schemadateien vorkommen. Dadurch, dass sich die acht SCL-Schemadateien über `include` gegenseitig einbinden, können diese Änderungen ebenfalls im Versionsschema deklariert werden.

7.3.4 Generische Vergleich

Die bisher in diesem Abschnitt vorgestellten Verfahren dienen dem Zweck, dem Bytecode mitzuteilen, welche Elemente versioniert werden und dem Schemadesigner eine geeignete Ausdrucksform für eine Schemaevolution zu ermöglichen. Die geeignete Ausdrucksform für den Schemadesigner ist sicherlich die schwerer umzusetzende Anforderung, welche von keinem der bereits beschriebenen Verfahren zufrieden stellend erfüllt wurde, da jedes Verfahren die W3C-Norm an bestimmten Stellen verletzt. Bei beiden bisher beschriebenen Verfahren muss der Bytecode speziell angepasst werden. Aus dieser Erkenntnis heraus kann die Versionierung von Elementtypen auch vollständig dem Bytecode-Compiler überlassen werden. Dem Bytecode-Compiler werden zwei in sich vollständige Versionen eines Schemas übergeben, woraufhin dieser über einen Vergleichsalgorithmus die Versionsunterschiede der Elementtypen feststellt und anschließend die Unterschiede versioniert. Bei einer Schemae-

volution des SCL-Schemas fallen bei solch einem generischem Vergleich keine Zusatzarbeiten an, der Bytecode-Compiler bekommt lediglich eine neue Version des bisherigen Schemas übergeben. Es ist sinnvoll für das Erweiterungsschema, den gleichen Versionierungsmechanismus wie beim SCL-Schema zu verwenden. Für eine Evolution des Erweiterungsschemas muss demnach ein vollständig neues Erweiterungsschema erstellt werden. Dies bedeutet jedoch keinen großen Mehraufwand.

Es gibt zwei Möglichkeiten den Vergleich durchzuführen. Die erste Möglichkeit ist ein textueller Vergleich der zu versionierenden Schemata. Hierfür bestehen bereits einige Standardwerkeuge, jedoch bedarf das Ergebnis einer Interpretation des Bytecode-Compilers. Die zweite Möglichkeit einen Vergleich durchzuführen ist es, den Bytecode-Compiler die sich aus beiden Schemata ergebenden Bytecodes vergleichen zu lassen. Der Vergleich der Bytecodes kann mit weniger Aufwand für den Bytecode-Compiler durchgeführt werden.

Bei dem Vergleich der Bytecodes werden alle Typen und ihre Zustände dem Namen nach miteinander verglichen. Dabei ließt der Vergleichsalgorithmus zwei Typen mit gleichem Namen der Reihe nach durch. Trifft der Algorithmus bei der einen Version auf einen anderen Zustand als bei der anderen Version, versioniert er alle folgenden Zustände bis zum Endzustand. Dabei fällt auf, dass nicht der gesamte Elementtyp versioniert wird, sondern wenn möglich erst die erste Änderung innerhalb des Typen. Dadurch wird der Speicherbedarf des versionierten Bytecodes noch geringer. Es ist demnach vorteilhaft bei der Deklaration eines Elementtypen in einer Schemaevolution, welcher den Inhalt des Vorgängers nur erweitert, die Erweiterungen am Ende des Typen zu deklarieren. Ein Problem, welches sich aus der Versionierung innerhalb des Typen ergibt ist es, dass die Länge des Strukturcodes für die Pfadcodierung unter Umständen sich mit der Version ändert. Die Länge des Strukturcodes wird jedoch im Kopfzustand des Typen gespeichert. Um dieses Problem zu umgehen, gibt es zwei Möglichkeiten. Es kann auf die Versionierung innerhalb des Typen verzichtet werden und der gesamte Elementtyp versioniert werden, wodurch zwei Kopfzustände entstehen mit jeweils den richtigen Längen für die Strukurcodes. Die zweite Möglichkeit ist die Strukturcodelänge bei der Pfadcodierung jedes mal neu zu berechnen, was den Nachteil einer höheren Codierzeit hat.

Die mögliche Darstellung eines versionierten Zustandes bzw. Elementtypens im Bytecode wird in Unterabschnitt 7.3.5 beschrieben.

7.3.5 Codierung von versionierten Schemata im Bytecode

Da der Bytecode ursprünglich die Versionierung von Typen und Elementen nicht unterstützt, müssen neue Strukturen für diese angelegt werden. Dies hat die Anpassung aller Komponenten des BiM-Codierverfahrens zur Folge, so muss der Bytecode-Compiler die neue Struktur erzeugen und der De- bzw. Encoder die neue Struktur interpretieren.

Die Darstellung im Bytecode kann verschiedene Formen haben, hier werden zwei Möglichkeiten beschrieben. Die erste Möglichkeit ist eine Erweiterung der bestehenden Zustände. Damit nicht alle Zustände angepasst werden müssen, wird hier nur die Anpassung des Kopfzustands untersucht. Wie Tabelle 5 in Abschnitt 6.5 zeigt, verweist das letzte Datenfeld auf den Folgezustand bzw. auf den ersten Zustand des Typen. Ändern sich die Folgezustände eines bestimmten Elementtypen im Laufe einer Schemaevolution, legt der Bytecode-Compiler für diesen Typen ein vollständig neues Bytecodefragment an. Am Ende des Kopfzustands des Typen wird nun ein weiterer Zeiger auf die neue Version des Bytecodefragments angehängt. An den Kopfzustand wird quasi eine Liste mit den verschiedenen Versionen des Typinhaltes gehängt. Zusätzlich wird ein Feld vor der Liste eingefügt welches angibt, auf wie viele Versionen von Zuständen der Kopfzustand zeigt bzw. wie viele Einträge die Liste besitzt. Diese Darstellung im Bytecode hat jedoch einige Nachteile. Geht man von einem generischen Vergleich der verschiedenen Schemaversionen wie in Unterabschnitt 7.3.4 beschrieben aus, so wird die Versionierung der Zustände innerhalb des Typen nicht unterstützt. Zusätzlich müssten die Strukturcodes für die Pfadcodierung jedes Mal neu berechnet werden, da der Kopfzustand für jede Version der gleiche bleibt. Ein anderer Nachteil wird aus einem Beispiel ersichtlich:

Ein Schema besteht aus den zwei Typen „A“ und „B“. Bei der ersten Schemaevolution ändert sich nur Typ „A“, im Kopfzustand des Typen wird deshalb ein zusätzlicher Verweis auf den neuen Folgezustand angehängt. In der darauf folgenden Schemaevolution wird nur für Typ „B“ eine neue Version angelegt. Zusammenfassend ergeben sich drei verschiedene Versionen des Schemas und jeweils zwei Versionen jedes Typs. Soll nun eine Instanz der zweiten Version des Schemas codiert werden, wird zunächst das Element vom Typ „A“ codiert. Da dem Codec mitgeteilt wurde, dass nach der zweiten Version des Schema codiert werden soll und Typ „A“ versioniert ist, wird in den Folgezustand, auf den in der zweiten Zeile der Liste verwiesen wird, verzweigt. Geht der Codec beim Element vom Typ „B“ genauso vor, wird ebenfalls in die zweite Version des Typen „B“ verzweigt, dieser Folgezustand darf jedoch nur bei der dritten Version des Schemas verwendet werden.

Um diesen Fehler zu vermeiden, muss bei jeder Schemaevolution ein neuer

Eintrag in die Liste mit den Zeigern auf die versionierten Folgezustände angehängt werden. Ändert sich der Typ zwischen zwei Schemaversionen nicht, wird zweimal derselbe Zeiger auf den selben Folgezustand eingetragen. Dies führt zu einer zunächst nicht erkannten Speichervergrößerung.

Die zweite Möglichkeit einen Elementtypen zu versionieren, ist die Einführung eines neuen Zustands im Bytecode. Der neue Zustand entspricht einer Auswahl der Versionen und wird deshalb in dieser Arbeit „Versionszustand" genannt. Der Versionszustand enthält wie alle anderen Zustände eine Kopfzeile über die der Zustand identifiziert wird. Danach folgt ein Byte, welches die Anzahl der Zeiger auf verschiedene Versionen des Folgezustands angibt. Anschließend folgt die Liste mit den Zeigern auf die verschiedenen Versionen des Folgezustands. Bei dieser Methode muss, genau wie bei dem Beispiel zur Erweiterung des Kopfzustands, für jede Version des Schemas ein Versionseintrag für den Folgezustand erzeugt werden, unabhängig davon, ob der Folgezustand sich nur bei bestimmten Schemaversionen geändert hat. Bei einem generischen Vergleich der verschiedenen Schemaversionen, wie er in Unterabschnitt 7.3.4 beschrieben wurde, können mit dem Versionszustand die Zustände innerhalb eines Typen versioniert werden. Soll die Berechnung der Strukturcodelänge für die Pfadcodierung während der Codierung zur Zeitoptimierung vermieden werden, kann mit dem Versionszustand der gesamte Typ inklusive Kopfzustand versioniert werden. Zum Einsatz des BiM-Verfahrens zur Übertragung der Parameterdaten kann der Versionszustand in zwei unterschiedliche Darstellungsformen unterteilt werden. Dadurch, dass sich das SCL-Schema und das Erweiterungsschema getrennt von einander ändern können, kann es vorkommen, dass eine unterschiedliche Anzahl an Versionen bei beiden Schemata vorkommt. Aus diesem Grund empfiehlt es sich, für beide Schemata einen eigenen Versionszustand anzulegen. Beide Versionszustände haben den gleichen Aufbau, sie müssen sich lediglich in ihrer Kopfzeile unterscheiden.

Die Umsetzung von versionierten Elementen bzw. Typen im Bytecode stellt zusätzlich eine hohe Anforderung an den Bytecode-Compiler, da durch das Einfügen oder Erweitern von Zuständen sich viele Adresse im Bytecodevektor verändern und rückwirkend die Verweise auf die Zustände sich ändern müssen.

7.4 Codierung des Elements Private

Für das BiM-Verfahren sieht das `Private`-Element aus wie jedes andere Element. Durch die IEC61850 bekommt das `Private`-Element jedoch eine besondere Bedeutung, welche in Unterabschnitt 4.3.2 beschrieben wird. Die Tatsache, dass innerhalb des `Private`-Elements Wildcards instanziiert

werden können, welche auch bei unbekannten Namensraum nicht verworfen werden dürfen, ist der Grund, warum dieses Element besonders behandelt werden muss. Damit das BiM-Verfahren den Inhalt nicht verwirft, kann der gesamte Inhalt einfach als Zeichenkette codiert werden. Das schließt jedoch eine typentsprechende Codierung für Erweiterungen der Herstellerfirma aus. Dieser Nachteil verstärkt sich, wenn mehrfach instanziierte Erweiterungen der Herstellerfirma als zusammenhängende Zeichenkette codiert werden. Der Codec muss demnach zwischen `Private`-Elementen der Herstellerfirma und `Private`-Elementen der Fremdfirmen unterscheiden. Im XML-Dokument werden die `Private`-Elemente anhand des `type`-Attributes unterschieden. Die Codierung des BiM-Verfahren codiert jedoch nicht in Abhängigkeit des Inhaltes der Elemente bzw. Attribute, die es codiert. Um dies zu ändern kann für das `Private`-Element ein spezieller Typcodec verwendet werden. Bisher werden spezielle Typcodecs nur für die Codierung von einfachen Typen verwendet, z.B. um einen den Inhalt eines Elements vom Typ Integer als ganze Zahl mit 32-bit zu codieren. Spezielle Typcodecs sind jedoch nicht auf einfache Typen beschränkt, deshalb kann für den Typ `tPrivate` ein spezieller Codec implementiert werden.

Der Typcodec prüft beim Encodieren zunächst den Inhalt des Attributs `type`. Ist ihm der Inhalt unbekannt, codiert der Codec den gesamten Inhalt des Elements `Private` als eine zusammenhängende Zeichenkette. Idealer Weise sollten die Zeilenumbrüche des Inhalts mitcodiert werden, um später den Inhalt im gleiche Format wieder ausgeben zu können. Handelt es sich bei dem Inhalt des Attributs `type` um eine dem Codec bekannte Zeichenkette wird der Inhalt, wie in Abschnitt 7.2 beschrieben, codiert. Da ein `Private`-Element auch nur Inhalt ohne weitere XML-tags enthalten kann, kann eine typentsprechende Codierung auch hierfür angewendet werden. Erkennt der Typcodec für das `Private`-Element z.B. die Zeichenkette `"Herstellername-Integer"`, wird der Inhalt als Integer codiert, erkennt er die Zeichenkette `"Herstellername-Byte"`, codiert er ein Byte. Im Bitstrom wird für jede Möglichkeit ein bestimmter Code gesetzt, so dass beim Decodieren die folgenden Codes richtig interpretiert werden. Der Bytecode-Compiler schreibt für den Typen `tPrivate` einen Kopfzustand für einen einfachen Typen in den Bytecode, dieser Kopfzustand enthält anschließend einen Zeiger auf den Typcodec.

8 Aktivitäten der W3C und IEC bezüglich der Schemaevolution

Eine der größten Schwächen beim Verarbeiten von XML in Verbindung mit XML-Schema, ist die starre Festlegung auf eine XML-Schemaversion. Bisher gab es keine verbindlichen Vorgaben der W3C oder IEC61850, wie bei Schemaevolutionen vorzugehen ist. Die W3C hat jedoch in der XML-Schema Arbeitsgruppe begonnen, das Thema Versionierung [W3C2] anzugehen. Der folgende Absatz gibt einen Überblick über die Aktivitäten der W3C.

Die Versionierungsarbeitsgruppe der W3C hat auf ihrer Webseite [W3C2] einige Berichte veröffentlicht, anhand derer man sich einen Überblick über das Thema Versionierung von XML-Schema verschaffen kann. Die aktuellen Protokolle der Sitzungen zum Thema Versionierung von XML-Schema sind Mitgliedern der W3C vorbehalten. In den aktuellen veröffentlichten Berichten werden zunächst Anwendungsfälle beschrieben, in denen Versionskonflikte auftreten. Darauf folgt eine Festlegung von einheitlichen Begriffen für eine Diskussion der Versionierung. Anschließend werden 21 Versionierungsmechanismen kurz beschrieben, ohne eine Aussage über ihre Eignung zu machen. Weiterführende Berichte der Arbeitsgruppe liegen zum Zeitpunkt dieser Arbeit noch nicht vor.

Eine mögliche Behandlung der Schemaevolution durch die IEC61850 kann verschiedene Formen annehmen. Im Falle einer Behandlung müssten zunächst einige Begrifflichkeiten geklärt werden. Zu Beginn sollten die Begriffe Auf- und Abwärtskompatibilität im Kontext einer Unterstation nach IEC61850 beschrieben werden. Aufbauend auf diesen Begriffen kann die Norm Anforderungen an die Kompatibilität von Anlagenkomponenten mit unterschiedlichen SCL-Schemaversionen formulieren. Eine weitere hilfreiche Maßnahme ist die Spezifizierung eines Versionierungsprozesses. Die Gründe für eine Schemaevolution sind die sich ändernden Anforderung an den Informationsgehalt einer Schemainstanz. Ein Versionierungsprozess schreibt ein definiertes Vorgehen bei der Änderung dieser Anforderungen vor. In einem solchen Prozess können z.B. die Folgen der aus den geänderten Anforderungen entstehenden Schemaevolutionen anhand ihrer Folgen für die Auf- und Abwärtskompatibilität eingestuft werden. Eine solche Einstufung macht jedoch eine zur Kenntnisnahme von Versionierungsmechanismen notwendig. Ein Bezug auf solche Mechanismen in der IEC61850 ist durchaus denkbar, da ein Versionierungsmechanismus keine konkrete technische Umsetzung impliziert. Eine genauere Betrachtung der Versionierung von XML-Schema, unabhängig von der Binärformatierung, ist jedoch nicht mehr Teil der Aufgabenstellung dieser Arbeit.

9 Prüfen der Tragfähigkeit einer Baugruppe unter Einsatz des BiM-Verfahrens

Dieses Kapitel dient der Feststellung, ob der Einsatz des BiM-Verfahrens auf einem IED tragfähig ist. Dabei wird untersucht, welche Anpassungen nötig sind, um das BiM-Verfahren in einem IED zu verwenden, dessen Softwarearchitektur sich an dem Datenmodell der IEC61850 orientiert. Ein IED mit einer solchen Softwarearchitektur ordnet bestimmte logische Funktionen einer Softwarekomponente zu.

Das Kapitel geht bei der Untersuchung nach der folgenden Methodik vor:

1. Modellierung der Softwarestruktur der hier untersuchten Komponenten
2. Aufzeigen von Anwendungsfällen
3. Anschließende Verhaltensmodellierung der Anwendungsfälle mit Hilfe von Aktivitätsdiagramm nach UML1.4

9.1 Struktur des zu untersuchenden Systems

Dieser Abschnitt dient als Basis für die folgenden Abschnitte. Um das Software-System eines IEDs in seiner Komplexität erfassen zu können, ist die Aufteilung der Softwarearchitektur in Komponenten absolut notwendig. So wird auch in dieser Arbeit nur eine Komponente eines IEDs untersucht. Die Aufteilung eines Systems in Komponenten macht es möglich, eine Komponente in ein System einzubinden, ohne ein spezifisches Wissen über die restlichen Komponenten des Systems zu besitzen. Eine solche komponentenbasierte Aufteilung eines Systems wird auch in der IEC61850 angewendet, indem ein intelligentes Gerät in mehrere logische Funktionen unterteilt wird. Damit eine Komponente mit anderen Komponenten des Systems Informationen austauschen kann, werden so genannte Programmierschnittstellen definiert. Eine Programmierschnittstelle, auch API genannt, stellt dabei einen festen Umfang an Funktionen zur Verfügung, über die mit der Komponente kommuniziert werden kann. Die feste Definition der API ermöglicht es, beliebig Komponenten auszutauschen oder zu entwickeln, welche über die API mit der Komponente kommunizieren. Die in diesem Kapitel beschriebene Komponente, welche den BiM-Codec enthält, wird als Parameterservice bezeichnet. In Abbildung 10 wird der strukturelle Aufbau der Komponente Parameterservices mit Hilfe eines UML-Klassendiagramms dargestellt. In den folgenden Abschnitten wird auf die genaue Bedeutung der einzelnen Klassen in Abbildung 10 eingegangen. Alle Systeme, welche mit dem Parameterservice kommunizieren, werden im Klassendiagramm durch das IED-Framework

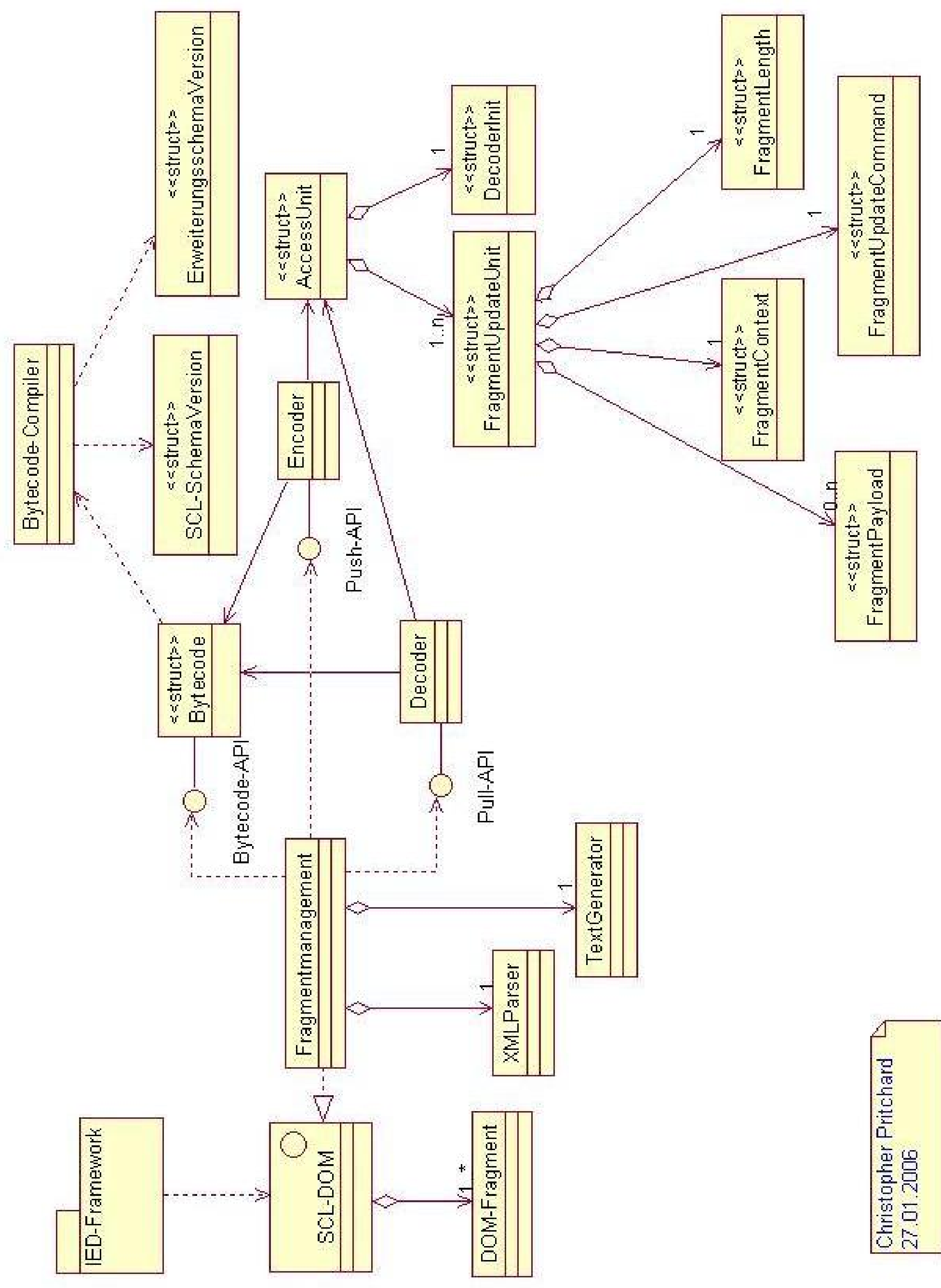

Abbildung 10: Struktureller Aufbau des Systems

vertreten. Ein Framework bildet einen Programmrahmen für Komponenten und stellt Funktionen und Strukturen zur Verfügung, welche für die Komponenten von Bedeutung sind. Das IED-Framework ist demnach eine Art Vermittler zwischen verschiedenen Komponenten bzw. Applikationen einer intelligenten Baugruppe. Aus dem Klassendiagramm wird ersichtlich, dass das IED-Framework nur über eine einzige Schnittstelle mit dem Parameterservice zusammenarbeitet, was eine minimale Abhängigkeit vom Parameterservice für das Framework bedeutet. Es wird nochmals darauf hingewiesen, dass die hier gemachten Annahmen zur Softwarearchitektur nur zur Überprüfung der Tragfähigkeit dienen und sich nicht auf ein reales System stützen.

Alle Klassen in Abbildung 10, die sich rechts von der Klasse Fragmentmanagement befinden, sind Teil des BiM-Algorithmus, der in Kapitel 6 beschriebenen wurde. Das Binärformat wird als ein Zusammenschluss verschiedener Strukturen (struct) dargestellt, welche Teil der Struktur Access Unit sind. Eine struct ist dabei eine Datenstruktur ohne eigene Funktionen. Die Klassen Decoder und Encoder bedienen sich bei der Codierung der Datenstruktur Bytecode.

Die bisher in dieser Arbeit nicht behandelten Klassen „Fragmentmanagement“ und Schnittstellen „Pull-API“, „Push-API“ und „Bytecode-API“, werden in diesem Abschnitt in gesonderten Unterabschnitten beschrieben.

9.1.1 Das Fragmentmanagement

Das Fragmentmanagement spielt eine zentrale Rolle für alle Themen, die sich mit der Verarbeitung der Konfigurationsdaten befassen. Deshalb realisiert das Fragmentmanagement auch die Schnittstelle zum IED-Framework. Der Name Fragmentmanagement wurde gewählt, da die Fragmentierung zwar vom BiM-Verfahren unterstützt wird, jedoch das Verfahren dem Anwender überlässt diese sinnvolle zu interpretieren und anzuwenden.

Bei der Schnittstelle handelt es sich um die Repräsentationsschicht der Konfigurationsdaten. Da die Konfigurationsdaten im Binärformat eine ähnlich hierarchische Datenstruktur wie ein XML-Dokument aufweisen, empfiehlt sich für die Darstellung der Daten eine ähnliche Repräsentation zu verwenden wie die in Abschnitt 3.3 beschriebenen XML-APIs. Um dem IED-Framework einen möglichst schnellen und komfortablen Zugriff auf die Daten zu ermöglichen, soll die Schnittstelle sich zum IED-Framework hin wie ein DOM verhalten. Da die Schnittstelle zur Repräsentation für SCL-Dokumente verwendet wird, wird die Schnittstelle im Klassendiagramm SCL-DOM genannt. Ohne dem folgenden Abschnitt zu weit vorzugreifen, ist der Hauptanwendungsfall für eine Komponente Parameterservice der Zugriff auf die Konfigurationsdaten während des Hochlaufs. Um dem in Abschnitt 3.3 be-

reits erwähnten hohen Speicherplatzverbrauch einer DOM-Repräsentation entgegen zu wirken, können die Daten nach dem Hochlauf geschlossenen wieder verworfen werden. Damit der Speicherplatzverbrauch auch während des Hochlaufs möglichst gering ist, können auch nur Fragmente des SCL-DOMs im Speicher aufgebaut werden. Im Klassendiagramm wird dies bereits durch die Klasse DOM-Fragment angedeutet. Die Abgrenzungen eines DOM-Fragments innerhalb des SCL-DOMs sollten dabei den Grenzen eines BiM-Fragments im Binärformat entsprechen. Wie sich später bei der Untersuchung des Verhaltens im Anwendungsfall „Auf Konfigurationsdaten zugreifen“ zeigt, ist es von Vorteil, wenn das IED-Framework Kenntnisse über die Fragmentgrenzen besitzt. Ein Beispiel für eine mögliche sinnvolle Aufteilung der DOM-Fragmente kann sein „ein Fragment pro LN“. Alle Elemente und Attribute in der Hierarchie oberhalb eines LN, würden dann ebenfalls zu einem Fragment zusammengefasst.

9.1.2 Programmierschnittstelle innerhalb des Parameterservices

In diesem Abschnitt wird die Bedeutung der drei Schnittstellen „Pull-API“, „Push-API“ und „Bytecode-API“ aus dem Klassendiagramm in Abbildung 10 beschrieben. Alle drei Schnittstelle werden von dem BiM-Verfahren realisiert und dem Fragmentmanagement zur Verfügung gestellt. Die Bytecode-API erhält eine Sonderstellung, da sie für die Codierung zunächst nicht benötigt wird. Die Funktionen zum Steuern des Codiervorgangs entfallen wie folgt auf die beiden anderen Schnittstellen:

- Decoderschnittstelle
  - Fragmente filtern
  - Fragmente decodieren
- Encoderschnittstelle
  - Access-Units erzeugen
  - Fragment-Kontextpfad codieren
  - Fragment-Payload codieren

Die Decoderschnittstelle

Dieser Absatz befasst sich mit der Schnittstelle, welche den Decodiervorgang steuert. Der Decoder realisiert hierfür eine „Pull-API“, deren grundlegende Funktionsweise bereits bei der Verarbeitung von XML in Abschnitt 3.3 kurz beschrieben wurde.

Die Entscheidung für eine Pull-API gegen eine der anderen Repräsentationsschnittstellen für XML, wird in den folgenden Sätzen kurz begründet. Für eine DOM-API muss das BiM-Verfahren die bereits zum Teil beschriebenen Eigenschaften des Fragmentmanagements besitzen, was eine sehr große Spezialisierung des BiM-Verfahrens bedeutet. Dies soll, wie bereits erwähnt, möglichst ausgeschlossen werden. Eine eventbasierte Schnittstelle, wie SAX beispielsweise, ist im Gegensatz sehr leicht zu implementieren, jedoch würde der Kontrollfluss beim decodieren auf Seite des Decoders liegen. Deshalb ist eine Pull-API die am besten geeignete Form für eine Schnittstelle.

Da die eigentliche Information in Payloadfragmenten encodiert ist, arbeitet die eigentliche Pull-API nur auf diesen Fragmenten. Über zusätzliche Funktionen der API muss dem Decoder mitgeteilt werden, welches Fragment er zu decodieren hat. Dazu gibt es einmal die Möglichkeiten, der Decoderschnittstelle einen Zeiger auf den Speicher zu übergeben, an dessen Stelle sich die „Fragment Update Unit“ befindet.

Die andere Möglichkeit ist es, der Decoderschnittstelle einen Fragment-Kontextpfad zu übergeben, anhand dessen der Decoder den Bitstrom filtert. In diesem Kapitel wird von einer absoluten Pfadcodierung ausgegangen. Anhand Abbildung 4 in Abschnitt 6.1 kann man sehen, welche Information die Schnittstelle benötigt um den Kontextpfad eindeutig bestimmen zu können. Für jedes Pfadfragment des Kontextpfades muss der Srukturcode und gegebenenfalls der Typecode und Ersetzungscode angegeben werden. Ist der Kontextpfad nicht eindeutig, was bei symmetrischer Verteilung der Fragmente die Regel ist, muss zusätzlich noch der Positionscode zu jedem Pfadfragment angegeben werden. Da dem Fragmentmanagement z.B. der Strukturcode eines Pfades zwischen zwei Elementen im SCL-DOM nicht bekannt ist, kann alternativ für den Strukturcode der Name des Elements, auf den das Pfadfragment verweist, angegeben werden. Anstelle des Typecode wird der Name des Typs, in dem die Umwandlung stattfindet, angegeben. Für den Positionscode wird einfach eine Zahl angegeben. Mit diesen Informationen kann der Decoder den Binärcode für den Kontextpfad erstellen, um anschließend die Fragment-Payload heraus zu filtern.

Die grundlegenden Funktionen der Pull-API sind das Ausgeben von Elementen, Attributen und Inhalt. Eine Pull-API bietet die Möglichkeit, sich den nächsten „tag“, den Inhalt eines Elements oder das nächste Attribut ausgeben zu lassen.

Wird über eine Funktion das nächste tag angefordert, muss diese Funktion im Rückgabewert die Information über den Namen, den Namespace und die Information, ob es sich um einen schließenden oder öffnenden tag handelt, enthalten.

Nach jedem öffnenden Element bzw. tag können die Attribute abgefragt

werden. Da ein Attribut immer einen einfachen Typen besitzt, kann zusätzlich zum Namen und Namensraum im Rückgabewert sofort der Wert des Inhalts enthalten sein. Der Wert kann dabei auf zwei verschiedene Arten zurückgegeben werden. Die erste Möglichkeit ist, den Wert als Zeichenkette ausgeben zu lassen. Dies ist von Vorteil beim Export einer CID-Datei, wie er in Unterabschnitt 9.3.8 beispielhaft beschrieben wird. Da das BiM-Verfahren die Werte typentsprechend im Bitstrom encodiert, kann als zweite Möglichkeit der Wert in codierter Form zurückgegeben werden. Aufgrund der verschiedenen Typcodierungen in einem Bitstrom, muss der Wert typunsicher übergeben werden, deswegen ist eine zusätzliche Typinformation notwendig. Über eine Typumwandlung (typecast) kann der Wert schließlich im Gerät verarbeitet werden. Da beide Möglichkeiten Vorteile besitzen, sollte die Pull-API zwei Funktionen mit jeweils einer Möglichkeit anbieten, um ein Attributwert ausgeben zu lassen.

Zur Abfrage des Inhaltes eines Elements werden ebenfalls zwei Funktion angeboten. Die erste Funktion gibt den Wert des Inhaltes als Zeichenkette aus. Die zweite Funktion übergibt den Wert Typunsicher mit einer zusätzlichen Typinformation.

Die Encoderschnittstelle

Der Encodiervorgang wird mit einer „Push-API“ gesteuert. Das Ergebnis einer Encodierung ist eine Fragment-Payload, weshalb der Encoder zuvor einen Kontextpfad für das Fragment erstellen muss. Wie bereits bei der Decoderschnittstelle erwähnt, kann das Fragmentmanagement einem Pfadfragment nicht direkt den Strukturcode des Bitstroms zuordnen. Deshalb wird, wie bei der Decoderschnittstelle, pro Pfadfragment der Elementname des Elements auf das verwiesen wird, dessen Positionscode und im Falle eines Typecast der Name des Typs in den umgewandelt wird, übergeben. Zusätzlich muss über die Encoderschnittstelle eine Access-Unit angelegt werden können, in der die „Fragment Update Units“ abgespeichert werden.

Die Verwendung der Funktionen, welche für die eigentliche Encodierung verwendet werden, kann man sich vorstellen wie das Schreiben einer XML-Datei. Es gibt jeweils eine Funktion für das Öffnen und Schließen eines „tags“. Bei der Funktion zum Öffnen des tags wird der Name des Elements übergeben.

Vor jedem öffnenden Element bzw. tag können die zugehörigen Attribute der Schnittstelle inklusive des Wertes übergeben werden. Auch die Push-API bietet hierfür zwei Funktionen an. Mit der ersten Funktion kann der Wert des Inhalts als Zeichenkette codiert übergeben werden. Diese Form der Übergabeparameter ist vorteilhaft beim Import einer CID-Datei, wie er in Unter-

abschnitt 9.3.7 beispielhaft beschrieben wird. Die zweite Möglichkeit ist, den Wert des Attributs typunsicher mit einer zusätzlichen Typinformation als Parameter zu übergeben. Auf diese Weise muss das Fragmentmanagement den Wert eines Attributs aus dem SCL-DOM, z.B. einen Zahlenwert, nicht erst in eine Zeichenkette umwandeln.

Der Inhalt eines Elements kann ebenfalls über zwei verschiedene Funktionen encodiert werden. Bei der ersten Möglichkeit wird der Funktion der Wert als Zeichenkette codiert übergeben. Die zweite Möglichkeit ist, den Wert des Elements typunsicher mit einer zusätzlichen Typinformation zu übergeben.

Bytecode-API

In erster Linie dient der Inhalt des Bytecodes nur der Codierung. Da zur Speicherersparnis nur der vorkompilierte Bytecode auf den intelligenten Baugruppen vorhanden ist, kann man auf die Schemainformation nur über eine Bytecode-API zugreifen. Obwohl die CID-Datei alle Informationen zur Konfigurierung der Baugruppe enthält, ist ein Zugriff auf die Schemainformationen nötig. Dies wird erst aus einem Anwendungsscenario ersichtlich:

Eine intelligente Baugruppe wird mit dem PC konfiguriert. Die CID-Datei wird in codierter Form an die Baugruppe geschickt und im SCL-DOM fragmentweise aufgebaut. Werden nun im Betrieb Konfigurationsänderungen vorgenommen, sollen diese im Binärformat wieder abgespeichert werden. Handelt es sich bei den Änderungen um nicht mit dem SCL-Schema validierbare Änderungen, wird der Encoder eine Fehlermeldung produzieren. Dem Encoder fehlt jedoch die Möglichkeit, den Fehler der Anwendung entsprechend zu interpretieren. Es empfiehlt sich, die Änderung schon während der Eingabe im SCL-DOM auf ihre Korrektheit zu prüfen. Die Regeln für eine solche Prüfung können mit Hilfe einer Bytecode-API ermittelt werden. Ein Beispiel für eine solche Änderung, welche mit einer Bytecode-API abgefangen werden kann, ist das Überschreiten einer Häufigkeit von einem Element.

Die genaue Darstellung einer solchen Bytecode-API ist noch nicht festgelegt. Der Form halber muss erwähnt werden, dass das Fragmentmanagement gegen Regeln prüfen muss, welche nicht mehr im Bytecode vorhanden sind, z.B. gegen Xpath-Anweisungen.

9.2 Anwendungsfälle für die Komponente Parameterservice

In diesem Abschnitt werden die Funktionalitäten der zu entwickelnden Softwarekomponente in Form von Anwendungsfällen aus Sicht des Benutzers beschrieben. Für diesen Zweck wurden so genannte Anwendungsfalldiagramme

in UML erstellt. Jeder Anwendungsfall abstrahiert dabei interne Funktionsabläufe des jeweils untersuchten Systems bzw. der jeweiligen Komponente. Aus diesem Grund eignen sich Anwendungsfalldiagramme gut als Einstiegspunkt für die in Abschnitt 9.3 folgende Beschreibung der Abläufen während bestimmter Anwendungsfälle. Die Abbildung 11 zeigt fünf grundlegende An-

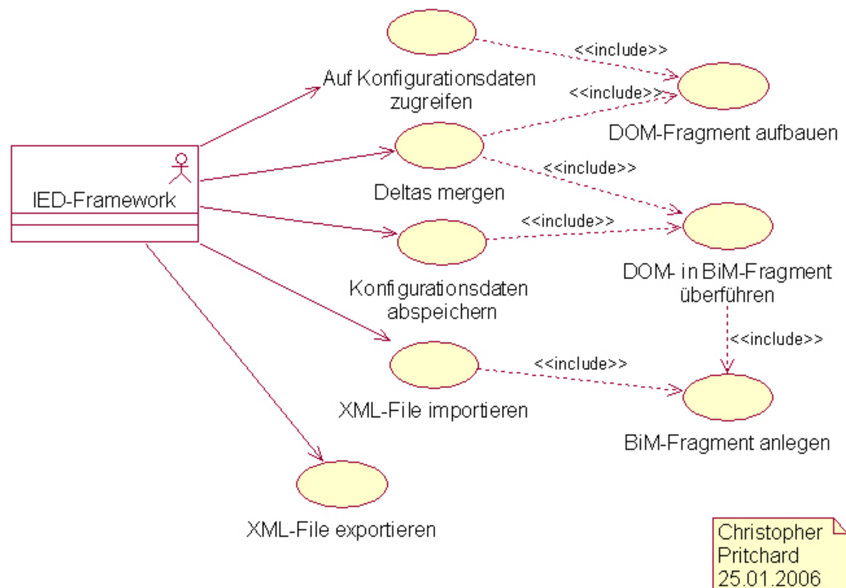

Abbildung 11: Direkte Anwendungsfälle für den Parameterservice

wendungsfälle für die Komponente Parameterservice. Da alle Anfragen über das IED-Framework mit dem Parameterservice kommuniziert werden, ist nur das IED-Framework als Aktor dargestellt. Jeder dieser fünf Anwendungsfälle wird im Laufe dieses Kapitels in einem eigenen Abschnitt beschrieben. Die drei Anwendungsfälle in Abbildung 11, welche über eine include-Beziehung eingebunden werden, haben sich bei der Verhaltensmodellierung der fünf Hauptanwendungsfälle als mehrmals wiederkehrend herausgestellt und werden deswegen noch vor den Hauptanwendungsfällen in einem extra Abschnitt beschrieben.

9.3 Beispielhafte Abläufe während eines Anwendungsfalls

Zweck dieses Abschnittes ist es, anhand von Aktivitätsdiagrammen einen Überblick über die mögliche Komplexität der Komponente Parameterservice

zu bekommen. Es handelt sich hierbei um eine Analyse eines fiktiven Systems zur Überprüfung der Tragfähigkeit eines Einsatzes des BiM-Verfahrens. Ein Aktivitätsdiagramm zeigt dabei den Kontroll- und Datenfluss zwischen einzelnen Arbeitsschritten, die zur Realisierung der aus dem Anwendungsfall heraus geforderten Funktionalität notwendig sind. Deshalb wird zu jedem Anwendungsfall in Abbildung 11 ein eigenes Aktivitätsdiagramm erstellt.

9.3.1 Laden eines BiM-Fragments in den SCL-DOM

Das Laden eines BiM-Fragments in den SCL-DOM stellt einen elementaren Ablauf von Arbeitsschritten dar, welcher sowohl beim Zugriff auf die Konfigurationsdaten als auch beim Zusammenfügen von Deltas und Fragmenten seine Anwendung findet. Deshalb wird das Laden von BiM-Fragmenten, obwohl es keinen direkten Anwendungsfall für das IED-Framework darstellt, separat beschrieben. Abbildung 12 zeigt das Aktivitätsdiagramm zum Anwendungsfall „DOM-Fragment aufbauen". Als erstes wird ein Decoderobjekt instanziiert und sofort im Anschluss der Kontextpfad oder die Speicheradresse des Fragments an den instanziierten Decoder übergeben. Daraufhin lädt der Decoder das Fragment und übergibt das „Fragment Update Command" (FUC) an das Fragmentmanagement. Das Fragmentmanagement interpretiert das FUC. Stellt es fest, dass es sich nicht um ein „delete" Kommando handelt, beginnt das Fragmentmanagement über die Pull-API wahlweise Element, Attribute oder Inhalt (kurz E/A/I) anzufordern. Der Decoder gibt das angeforderte Syntaxelement aus, woraufhin das Fragmentmanagement das Element bzw. Attribut dem FUC entsprechend in den Baum einfügt. Als nächster Schritt wird geprüft, ob das BIM-Fragment bereits vollständig decodiert bzw. das DOM-Fragment vollständig aufgebaut ist. Eine einfache Möglichkeit dies zu prüfen ist festzustellen, wann der tag des ersten Elements bzw. der Wurzel des Fragments wieder geschlossen wird. Trifft der Decoder auf das Fragmentende, wird er wahrscheinlich einen definierten Rückgabewert ausgeben, der ebenfalls kontrolliert werden kann. Solange das Fragmentende noch nicht erreicht ist, wiederholen sich die Schritte des Anforderns, Ausgebens und Einfügens in den Baum. Ist das Fragmentende erreicht, wird das Decoderobjekt zerstört um Speicher wieder freizugeben. Als nächstes wird geprüft ob Deltas für das soeben aufgebaute Fragment vorhanden sind. Um zu erklären was Deltas sind, muss an dieser Stelle den folgenden Abschnitten vorgegriffen werden.

Ein Delta ist ein Stereotyp eines BiM-Fragments, in dem Änderungen an dem ursprünglichen Dokument gespeichert werden. Wird an einem Element oder Attribut im SCL-DOM eine Änderung vorgenommen, wird dieses Element bzw. Attribut in einem Fragment, oder auch Delta genannt, abgelegt.

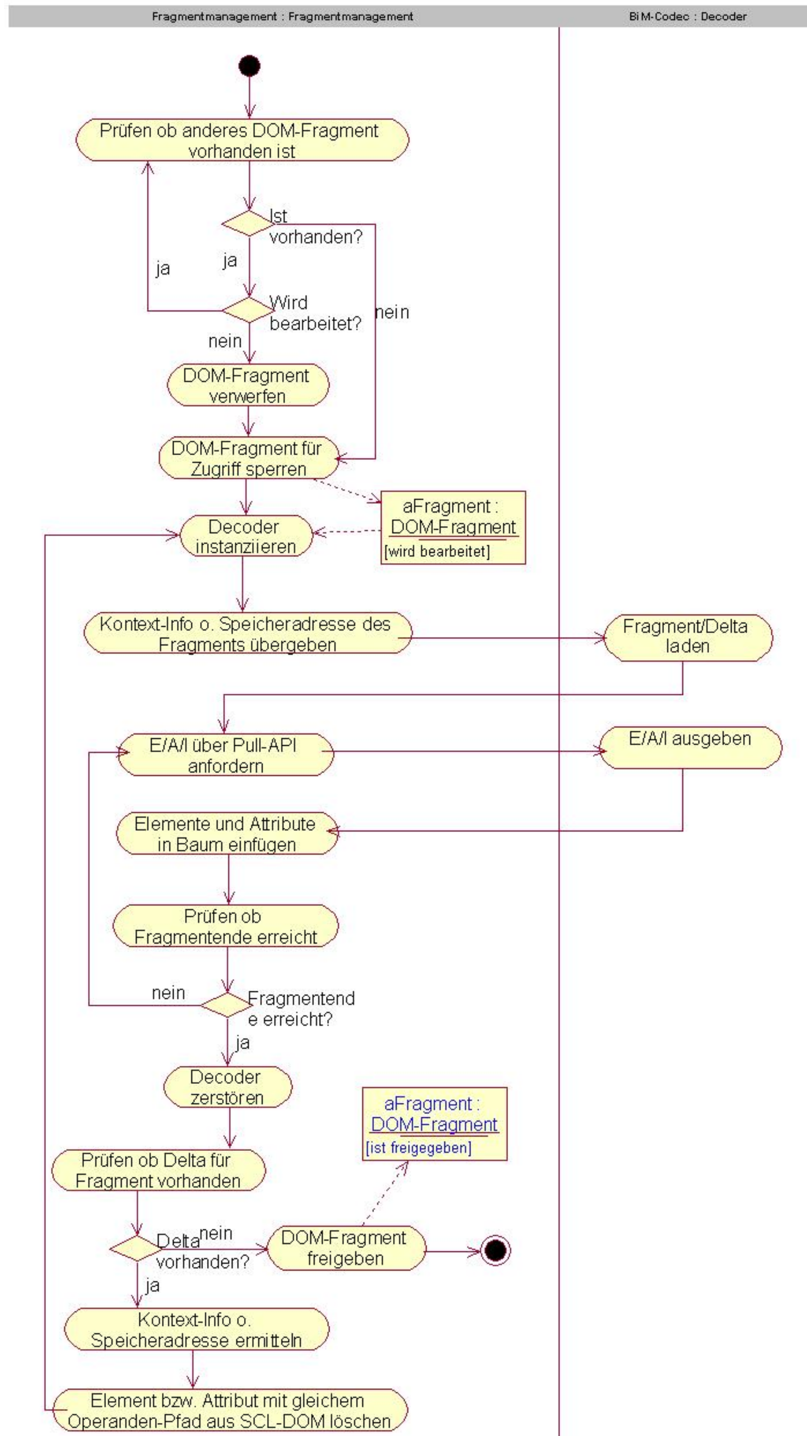

Abbildung 12: Möglicher Ablauf beim Laden eines BiM-Fragments in den SCL-DOM

Gegenüber dem Vorgang die Änderungen direkt im Dokument vorzunehmen, hat das Anlegen von Deltas einige Vorteile, welche in Abschnitt 9.3.5 genauer beschrieben werden.

Wird beim Laden eines BiM-Fragments in den SCL-DOM festgestellt, dass Deltas vorhanden sind, wird zunächst der Kontextpfad oder die Speicheradresse des Deltas ermittelt. Bereits beim Anlegen eines Deltas kann das Fragmentmanagement dessen Kontextpfad oder Speicheradresse in einer Liste speichern. Nachdem die Information ermittelt wurde, wird wieder ein Decoder instanziiert und das Delta abgearbeitet. Enthält ein Fragment das FUC „delete“, enthält es kein Fragment-Payload. Das Fragmentmanagement überspringt deshalb die Abfrage des Inhalts und löscht den über den Kontextpfad angesprochenen Knoten inkl. seiner Kinder. Ist kein Delta mehr für das DOM-Fragment vorhanden, wird der Vorgang abgeschlossen.

9.3.2 BiM-Fragment anlegen

Das Anlegen eines BiM-Fragments gehört ebenfalls zu den elementaren Abläufen, welche in verschiedenen Anwendungsfällen zum Einsatz kommen. Das Anlegen eines BiM-Fragments wird beim Import einer CID-Datei benötigt, beim Zusammenfügen von Deltas und Fragmenten sowie beim Abspeichern von geänderten Konfigurationsdaten. Mit dem Anlegen eines BiM-Fragments ist nicht das encodieren einer Fragment-Payload gemeint, sondern das Vorbereiten aller notwendigen Schritte für eine Encodierung. Abbildung 13 zeigt die Abläufe während des Anlegens eines Fragments. Der Vorgang wird vom Fragmentmanagement angestoßen, welches zunächst entscheidet, ob eine neue Access-Unit (AU) angelegt wird oder eine bestehende Access-Unit verwendet wird. Bei Verwendung einer bestehenden AU muss diese zugewiesen werden, woraufhin diese geladen wird. Soll eine neue AU angelegt werden, wird die Initialisierung vom Fragmentmanagement angestoßen. Der Encoder, welcher in den jeweiligen Anwendungsfällen bereits instanziiert wurde, erzeugt daraufhin eine Access-Unit. Ein Grund für das Anlegen einer neuen AU kann das separate Zusammenfassen von Deltas sein. Eine Decoder Initialisierung (DecoderInit) wird erst beim Übertragen einer Access Unit hinzugefügt, was nicht in dieser Arbeit behandelt wird. Nachdem dem Encoder seine AU bekannt ist, wird vom Fragmentmanagement das Fragment Update Command (FUC) an die Encoderschnittstelle übergeben. Der Encoder legt eine Fragment Update Unit mit einem FUC an. Anschließend wird der Kontextpfad übergeben und vom Decoder eingefügt. Das Fragmentmanagement kann nun mit der Encodierung beginnen.

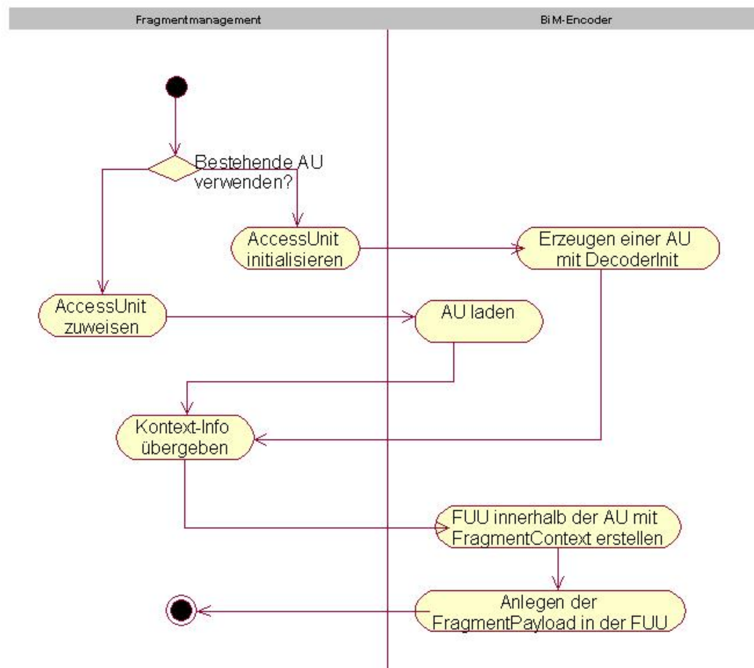

Abbildung 13: Möglicher Ablauf während des Anlegens eins Fragments

9.3.3 Überführen eines DOM-Fragments in ein BiM-Fragment

DOM-Fragmente in ein BiM-Fragment zu überführen ist ein elementarer Anwendungsfall, welcher beim Zusammenfügen von Deltas und Fragmenten und beim Abspeichern von geänderten Konfigurationsdaten auftritt. Der Anwendungsfall beschreibt, wie der Inhalt eines vollständigen DOM-Fragments aus dem SCL-DOM in ein BiM-Fragment encodiert wird. Abbildung 14 zeigt dabei die Aktivitäten während der Überführung eines DOM-Fragments in ein BiM-Fragment. Zunächst wird vom Fragmentmanagement der Encoder instanziiert und anschließend ein BiM-Fragment angelegt. Die Aktivität „BiM-Fragment anlegen“ wird im Abschnitt 9.3.2 zuvor mit einem eigenen Aktivitätsdiagramm beschrieben. Durch das Starten der Encodierung wird dem Encoderobjekt mitgeteilt, dass mit den folgenden Funktionsaufrufen die Fragment-Payload erzeugt wird. Anschließend beginnt das Fragmentmanagement wahlweise Elemente, Attribute oder Inhalt an die Push-API zu übergeben, woraufhin der Encoder das jeweilige Syntaxelement encodiert. Nach jedem encodierten Element wird vom Fragmentmanagement geprüft, ob das Fragment vollständig encodiert ist. Falls nicht, wird weiter encodiert. Ist das Fragment vollständig encodiert, wird der Encodiervorgang beendet. Der En-

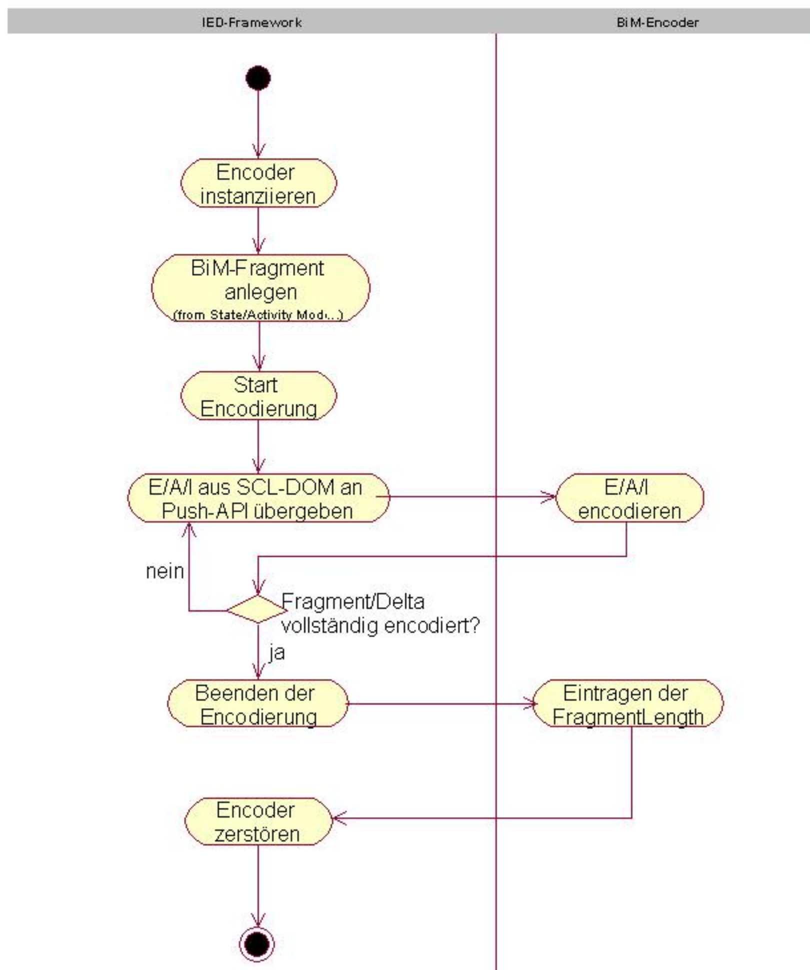

Abbildung 14: Möglicher Ablauf während der Überführung eines DOM-Fragments in ein BiM-Fragment

coder trägt daraufhin die Fragmentlänge in die Fragment Update Unit ein, und das Encoderobjekt wird wieder zerstört.

9.3.4 Zugriff auf Konfigurationsdaten

Der Zugriff auf die Konfigurationsdaten des SCL-DOMs ist ein direkter Anwendungsfall, welcher durch eine Applikation innerhalb des IED-Frameworks ausgelöst wird. Abbildung 15 zeigt den Ablauf beim Zugriff auf die Konfigurationsdaten.

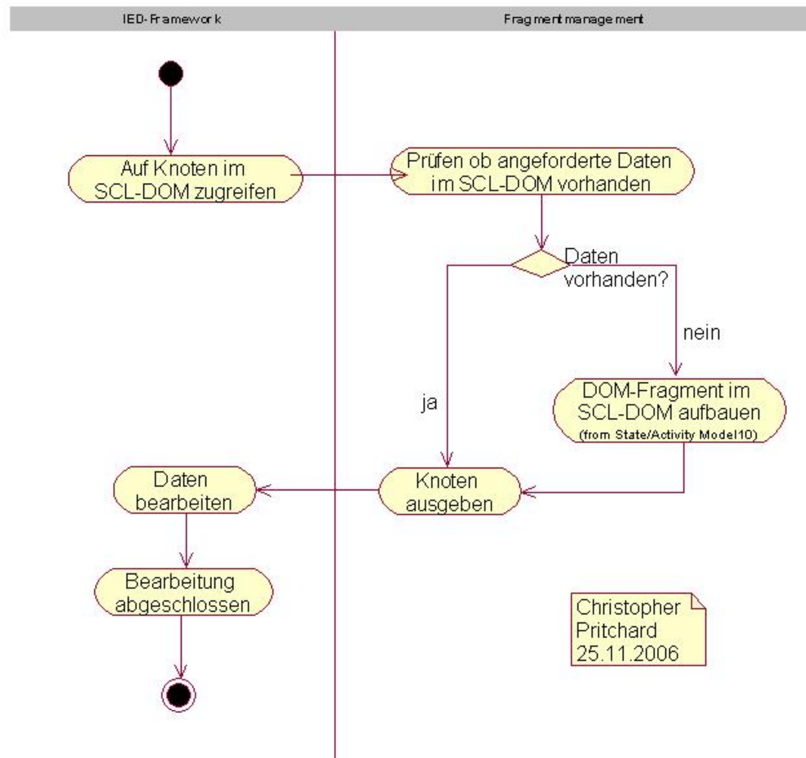

Abbildung 15: Möglicher Ablauf beim Zugriff auf Konfigurationsdaten

Ein typisches Beispiel für einen solchen Zugriff ist das konfigurieren der Applikationen durch das IED-Framework während des Hochlaufs der Baugruppen. Das Fragmentmanagement prüft zunächst, ob der angeforderte Knoten im SCL-DOM bereits aufgebaut ist. Ist dies der Fall, wird der Knoten ausgegeben. Wird auf einen Knoten im SCL-DOM zugegriffen, dessen DOM-Fragment nicht aufgebaut ist, wird, für den Anwender des SCL-DOM nicht sichtbar, das zugehörige BiM-Fragments in den SCL-DOM geladen. Anschließend wird der Knoten, wie zuvor beschrieben, ausgegeben. Die Aktivität „DOM-Fragment im SCL-DOM aufbauen“ wird im Abschnitt 9.3.1 mit einem eigenen Aktivitätsdiagramm beschrieben.

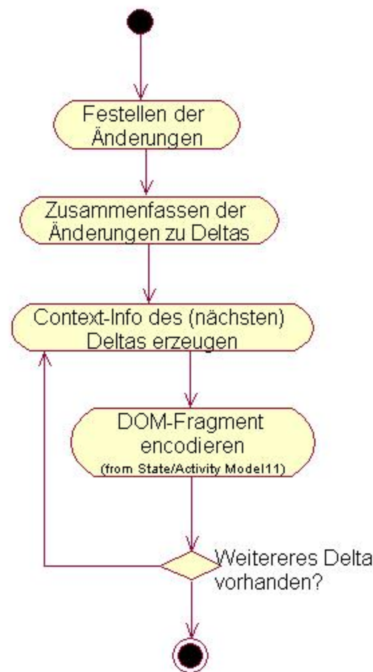

Abbildung 16: Möglicher Ablauf beim Speichern von Änderungen an den Konfigurationsdaten

9.3.5 Speichern von Änderungen an Konfigurationsdaten

Änderungen an Konfigurationsdaten können durch viele Einflüsse entstehen, z.B. durch einen Anwender, der die Konfiguration im Betrieb über das HMI[29] des Gerätes vornimmt. In dem Fall stellt das HMI eine Komponente dar, welche über das IED-Framework Daten innerhalb des SCL-DOMs vornimmt. Das IED-Framework nimmt die Änderungen zunächst im SCL-DOM vor, genauer im DOM-Fragment. Beim Beenden der Verarbeitung des DOM-Fragments bzw. durch einen gezielten Funktionsaufruf, beginnt das Fragmentmanagement mit dem folgenden, in Abbildung 16 dargestellten, Ablauf. Beim Speichern der Änderungen wird bereits davon ausgegangen, dass die Änderungen SCL-konform sind. Es empfiehlt sich, bei jedem Eintrag einer Änderung, diese auf Korrektheit zu prüfen. Das Prüfen der Änderungen ist ein Anwendungsfall, welcher in Abbildung 11 nicht dargestellt ist. Bei dem Beispielhaften Ablauf des Speicherns von Änderungen an den Konfigurationsdaten, beginnt das Fragmentmanagement damit festzustellen, welche Änderungen vorgenommen wurden. Dies kann bereits beim Ändern der Konfigura-

[29]Human Mashine Interface; Bedienerschnittstelle an einer Baugruppe

tionsdaten passieren. Danach werden alle Änderungen zu Deltas, einem speziellen Stereotyp eines Fragments, zusammengefasst. Beim Zusammenfassen wird darauf geachtet, dass Deltas mit einer sinnvollen Größe entstehen, dies ist jedoch nicht trivial. Werden z.B. zwei Attribute eines Elements geändert, kann es sinnvoll sein, ein Delta des Elements abzulegen. Besitzt das Element eine große Anzahl von Kind-Elementen kann es stattdessen sinnvoll sein, für jedes Attribut ein einzelnes Delta zu erzeugen. Die Vorteile des Abspeicherns von Änderungen in Deltas wird am Ende dieses Abschnittes beschrieben. Nachdem die Deltas feststehen, wird der Kontextpfad zum ersten Delta ermittelt und anschließend das Delta encodiert. Der Vorgang beim Encodieren eines Fragments bzw. Deltas aus dem SCL-DOM in ein BiM-Fragment wird in Abschnitt 9.3.3 mit einem eigenen Aktivitätsdiagramm beschrieben. Am Ende des Encodierens wird geprüft, ob noch weitere Deltas vorhanden sind. Falls ja, wird wieder der Kontextpfad ermittelt und das jeweiligen Delta encodiert. Dieser Vorgang wiederholt sich, bis alle Deltas encodiert sind.

Ablegen von Änderungen in Deltas

Wie bereits erwähnt ist ein Delta ein Stereotyp eines Fragments. Das heißt, ein Delta ist eine erweiterte Anwendungsmöglichkeit eines Fragments. Alle Änderungen an den ursprünglichen Konfigurationsdaten werden in einem Delta gespeichert.

Grundlage für eine sinnvolle Nutzung von Deltas ist, dass die zuletzt an das IED gesendeten Konfigurationsdaten von der Konfigurationssoftware auf dem PC gespeichert werden. Diese Konfigurationsdaten stellen die für das Gerät ursprünglichen Konfigurationsdaten dar. Das erste Beispiel für die sinnvolle Anwendung von Deltas geht von dem Fall aus, dass die Konfigurationsdaten auf dem IED während des Betriebs geändert wurden. Soll nun im Laufe einer Neukonfigurierung des IEDs mit der Konfigurationssoftware die aktuelle Konfiguration des Gerätes festgestellt werden, brauchen nur die Deltas übertragen werden. Andersherum kann, immer noch unter der Annahme der gleichen Konfigurationsdaten auf PC- und Geräteseite, das IED neue Konfigurationsdaten in Form von Deltas empfangen.

Der Vorteil bei der ausschließlichen Übertragung von Änderungen ist eine schnelle Übertragung trotz schmaler Bandbreiten. Aus der Beschreibung ergeben sich jedoch neue Anforderungen für die Komponente Parameterservice, die im Folgenden beschrieben werden. Die Annahme, dass die gleichen ursprünglichen Konfigurationsdaten auf PC- und Geräteseite bestehen, muss bei jeder Übertragung geprüft werden. Im Falle einer Delta-Übertragung von dem IED zum PC muss das Fragmentmanagement die Deltas an das IED-Framework übergeben. Andersherum muss das Fragmentmanagement emp-

fangene Deltas den ursprünglichen Fragmenten zuordnen.

9.3.6 Deltas und Fragmente zusammenfügen

Dadurch, dass Deltas immer innerhalb der Grenzen eines BiM-Fragments der ursprünglichen Konfigurationsdatei liegen, können die Deltas und das ursprüngliche Fragment zu einem neuen Fragment zusammen gefügt werden. Es gibt zwei Gründe für das Zusammenfügen von Fragmenten und den zugehörigen Deltas. Da die ursprünglichen Konfigurationsdaten im Speicher erhalten bleiben, erhöht sich mit jedem Delta der Speicherbedarf der Konfigurationsdaten. Wird eine bestimmte Speichergrenze überschritten, wird das Zusammenfügen der Fragmente und ihrer Deltas ausgelöst und anschließend die alten Fragmente und Deltas verworfen. Ein anderer Grund für das Zusammenfügen ist, wenn beim Herunterladen der Konfigurationsdaten die ursprüngliche Konfigurationsdatei im Konfigurationsprogramm nicht mehr vorliegt, und so die Deltas nicht mehr in einen Kontext gestellt werden können. Die Abbildung 17 zeigt das Aktivitätsdiagramm zu dem hier beschriebenen Anwendungsfall. Zunächst wird geprüft, ob das BiM-Fragment und seine Deltas bereits im SCL-DOM geladen sind. Ist dies nicht der Fall, wird das DOM-Fragment, welches dem BiM-Fragment und seinen Deltas entspricht, im SCL-DOM aufgebaut. Abbildung 12 zeigt das Aktivitätsdiagramm zu der Aktivität „DOM-Fragment im SCL-DOM aufbauen“. Ist das DOM-Fragment beim Start bereits vorhanden, wird dieser Schritt übersprungen. Nachdem nun das DOM-Fragment im SCL-DOM aufgebaut ist, wird es, wie bereits in Abschnitt 9.3.3 beschrieben, in ein neues BiM-Fragment encodiert. Dabei wird der Kontextpfad vom Ursprungsdokument übernommen. Das nun alte BiM-Fragment und seine Deltas werden gelöscht. Anschließend wird das DOM-Fragment wieder verworfen.

9.3.7 Import einer CID-Datei

Im Kapitel 4 „Der Systemstandard IEC61850“ wurden bereits in Abschnitt 4.4 die Grade der Konformität beschrieben. Dabei wurde festgestellt, dass eine CID-Datei im XML-Format sowohl importiert als auch exportiert werden können muss, um den höchsten Grad der Konformität zu erreichen. In diesem Abschnitt wird sich mit dem Import einer CID-Datei befasst. Wird der Begriff CID-Datei im weiteren Verlauf dieses Abschnittes verwendet, wird immer von einem XML-Dokument ausgegangen. Der Export einer CID-Datei wird im darauf folgenden Unterabschnitt beschrieben.

Das Speichern eines XML-Dokuments auf dem IED wurde bereits mehrfach aufgrund der geringen Speicherkapazität ausgeschlossen, weshalb auch

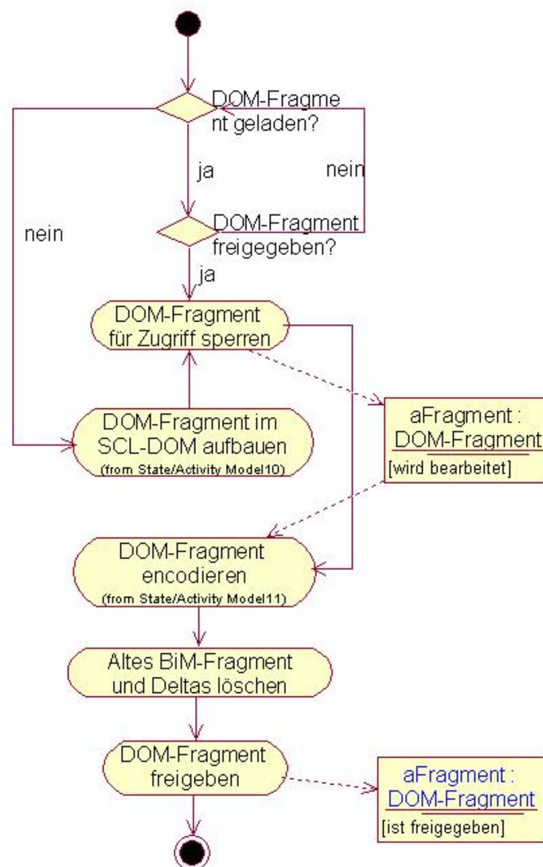

Abbildung 17: Möglicher Ablauf beim Zusammenfügen von Deltas und Fragmenten

die Anforderung an das Komprimierverfahren gestellt wurde, eine Komprimierung während der Übertragung zu ermöglichen („on-the-fly“). Das BiM-Verfahren ermöglicht dies zwar, jedoch bedeutet das Ablegen der CID-Datei in fragmentierter Form eine erhöhte Anforderung für das Fragmentmanagement. Dabei ist es absolut notwendig, dass eine importierte CID-Datei als Binärformat in derselben Form fragmentiert ist wie die ursprüngliche Konfigurationsdatei, z.B. pro LN ein Fragment. Der Grund hierfür ist, dass das IED-Framework für einen optimierten Zugriff auf Konfigurationsdaten Wissen über die Aufteilung der DOM-Fragmente benötigt, welche den BiM-Fragmenten entsprechen.

Damit der Parameterservice die CID-Datei encodieren kann, muss er zunächst auf den Inhalt der CID-Datei zugreifen können. Das Empfangen einer CID-Datei in Form eines Datenstroms ist in erster Linie Aufgabe der Kommunikation eines Gerätes und gehört nicht zum Thema dieser Arbeit. Für die Prüfung der Tragfähigkeit eines Gerätes unter Einsatz des BiM-Verfahrens ist nur die Form der Übergabe von Interesse. Für die hier betrachteten Vorgänge beim Importieren der CID-Datei wird von einer Art Puffer-Speicher ausgegangen, in dem immer der aktuell zu bearbeitende Teil der CID-Datei liegt. Die Kommunikation lädt eine feste Menge an Daten in den Puffer, und übergibt diesen an den Parameterservice zum encodieren. Ist der Inhalt des Puffer vollständig verarbeitet, wird über eine Rückmeldung des Parameterservices der Puffer erneut gefüllt. Dies wiederholt sich solange, bis die CID-Datei vollständig empfangen wurde. Abbildung 18 zeigt in einem Aktivitätsdiagramm einen möglichen Ablauf während des Imports einer CID-Datei. Da auch die Kommunikation in dieser Arbeit als eine Komponente angesehen wird, welche über das IED-Framework Informationen austauscht, wird in Abbildung 18 nur das IED-Framework dargestellt. Dieses startet den Vorgang und übergibt einen Zeiger auf den Puffer an das Fragmentmanagement. Das Fragmentmanagement instanziiert ein Encoderobjekt und lässt diesen ein BiM-Fragment anlegen. Die Aktivität zum Anlegen des BiM-Fragments wird in Abschnitt 9.3.2 beschrieben. Anschließend wird das Encoderobjekt in einen Stack eingefügt, dessen Bedeutung erst aus der weiteren Beschreibung des Vorgangs deutlich wird. Das Fragmentmanagement entnimmt das zuletzt eingefügte Encoderobjekt und startet den Encodiervorgang. Da noch kein XML-Inhalt geparst wurde, beginnt das Fragmentmanagement den XML-Inhalt im Puffer zu parsen. Ist ein Element, dessen Inhalt oder ein Attribut vollständig geparst, wird im Falle eines geparsten Elements dessen Name protokolliert, um später den Kontextpfad schneller erstellen zu können. Anschließend wird geprüft, ob es sich im Falle eines Elements um die Wurzel eines Fragments handelt. Zunächst wird davon ausgegangen, dass dies nicht der Fall ist. Das Fragmentmanagement generiert aus dem geparsten

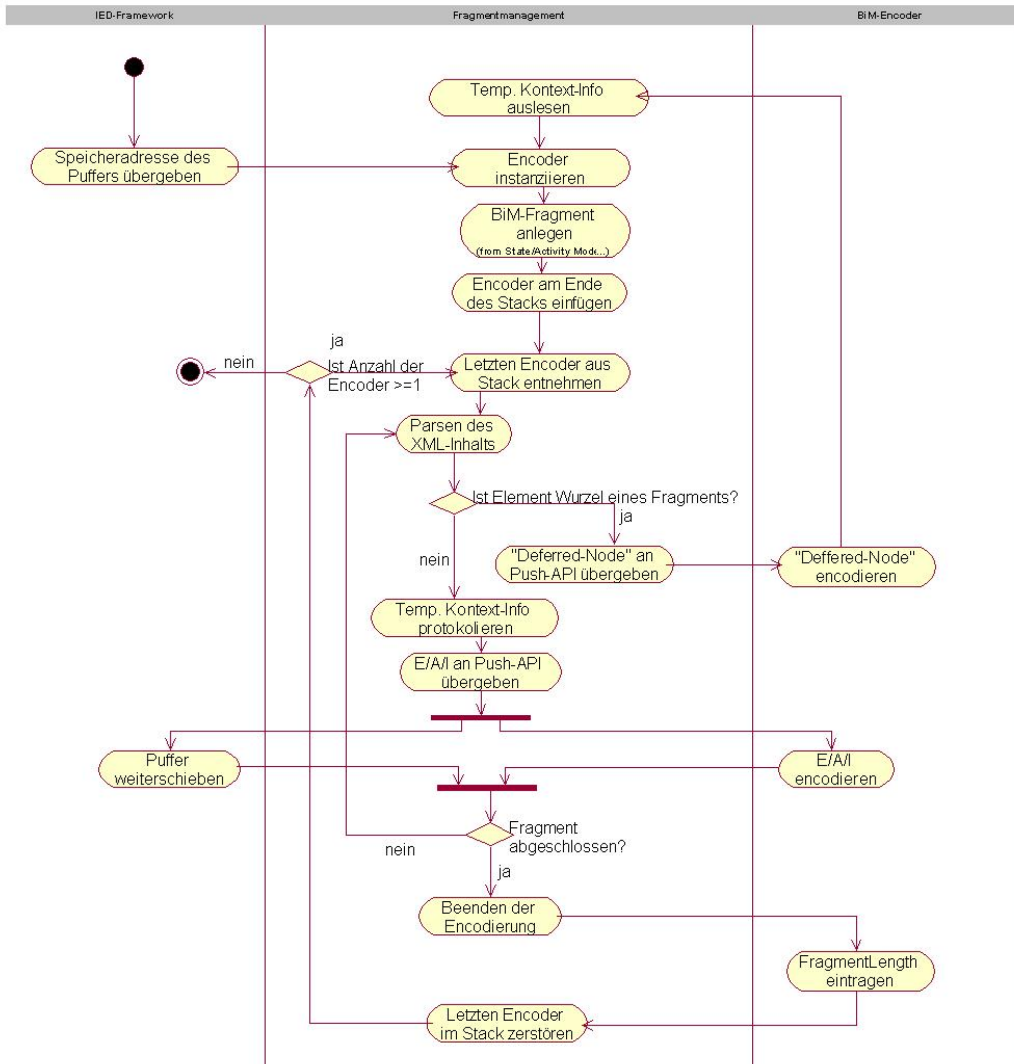

Abbildung 18: Möglicher Ablauf während des Imports ein CID-Datei

Inhalt des Puffers die Übergabeparameter und übergibt diese an die jeweilige Funktion der Push-API des Encoders, woraufhin der Encoder den Übergabewert encodiert. Ist das Fragment noch nicht vollständig encodiert, beginnt das Fragmentmanagement erneut mit dem parsen.

Trifft der Parser auf ein Element, welches die Wurzel des nächsten Fragments darstellt, wird nach dem protokollieren des Kontextpfades ein Platzhalter, auch „Deferred-Node“ in das aktuelle Fragment codiert. Als nächstes wird der Kontextpfad, welcher mitprotokolliert wurde, ausgelesen, ein neues Encoderobjekt instanziiert und ein BiM-Fragment angelegt. Für das Anlegen des BiM-Fragments wird der ausgelesene Kontextpfad verwendet. Das neue Encoderobjekt wird an das Ende des Stacks eingefügt, in dem sich bereits das zuvor instanziierte Encoderobjekt befindet. Das neue Encoderobjekt wird nun für die Encodierung der folgenden Elemente und Attribute verwendet. Bei der ersten Encodierung kann das Parsen übergangen werden, da dieses Element bereits zum Feststellen des Platzhalters geparst wurde.

Kann im Laufe des Parsens ein Syntaxelement nicht mehr vollständig geparst werden, da das Ende des Puffers erreicht wird, wird zunächst dieses unvollständige Syntaxelement gespeichert. Anschließend bekommt das IED-Framework eine Rückmeldung, dass der Parserinhalt vollständig bearbeitet wurde, woraufhin die Kommunikation den Puffer erneut füllt und über das Framework an den Parameterservice übergibt. Der Pufferspeicher wird nun soweit geparst, bis das gespeicherte unvollständige Syntaxelement vervollständigt wurde, anschließend wird die Encodierung fortgesetzt.

Ist die Codierung des aktuellen Fragmentes abgeschlossen, wird zunächst die Encodierung beendet, woraufhin der Encoder die Fragmentlänge in die Fragment Update Unit einträgt. Das Fragmentmanagement zerstört das letzte Encoderobjekte im Stack, welches zugleich den zuletzt verwendete Encoder darstellt. Als nächstes wird festgestellt ob ein weiterer Encoder im Stack besteht. Da noch das zuerst instanziierte Encoderobjekt im Stack besteht, wird mit diesem nun das zuerst begonnene Fragment weiter encodiert. Jedem Encoderobjekt wird also ein BiM-Fragment zugeordnet. Ist das zuerst begonnene Fragment ebenfalls vollständig encodiert, wird der zuerst instanziierte Encoder ebenfalls aus dem Stack gelöscht. Da keine weiteren Encoderobjekte vorhanden sind, wird das encodieren einer CID-Datei beendet.

9.3.8 Export einer CID-Datei

Genau wie der Import stellt auch der Export einer CID-Datei im XML-Format erhöhte Anforderungen an das Fragmentmanagement. Dies liegt vor allem daran, dass die Reihenfolge der Konfigurationsdaten in den BiM-Fragmenten nicht der Reihenfolge entspricht, in der sie im XML-Dokument auf-

treten. In einer CID-Datei wird beispielsweise anstelle eines Platzhalters im BiM-Fragment das vollständige Element erwartet, welches sich jedoch in einem anderen BiM-Fragment befindet. Zugleich ist die Anzahl der im SCL-DOM aufgebauten DOM-Fragmente im Extremfall auf ein Fragment begrenzt.

Auch beim Export wird von einem Puffer-Speicher auf Seite des IED-Framework ausgegangen. Diesmal werden die SCL-Syntaxelemente in diesem Puffer abgelegt, danach ist es Aufgabe der Kommunikation, die CID-Datei zu erstellen und zu senden. Aufgrund der geringen Speicherkapazität wird auch beim Export die CID-Datei „on-the-fly“ gesendet. Abbildung 19

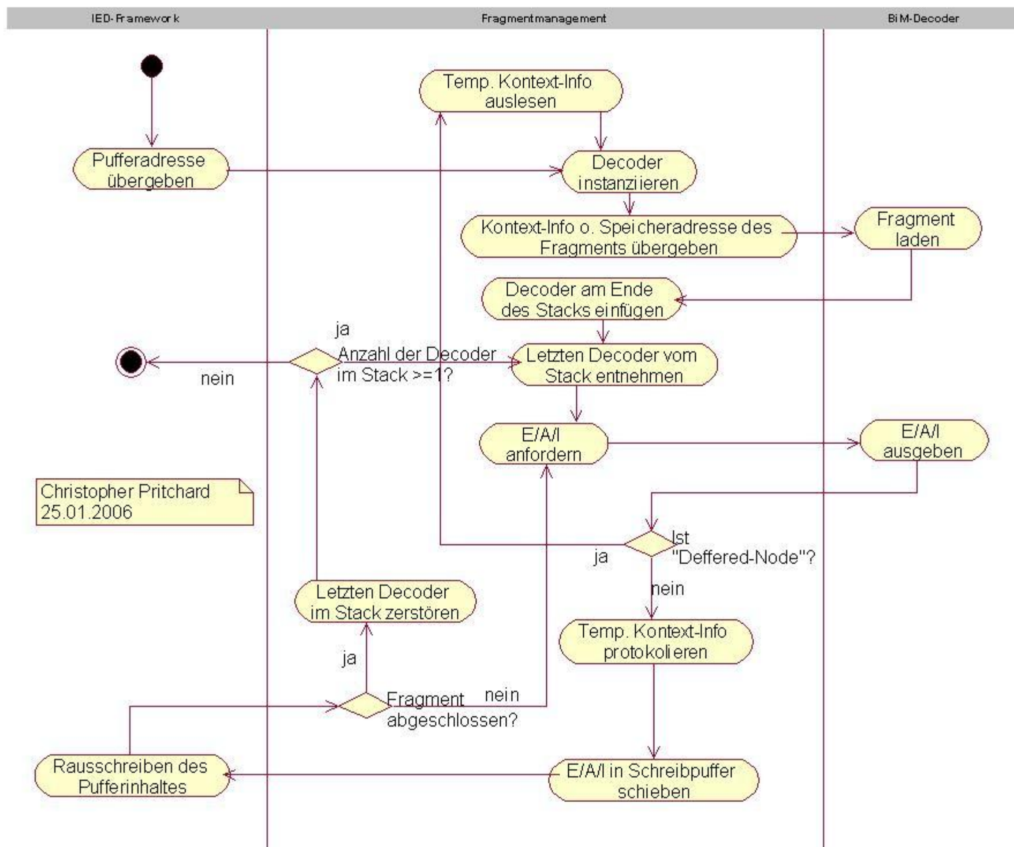

Abbildung 19: Möglicher Ablauf beim Export einer CID-Datei

zeigt den möglichen Ablauf beim Exportieren einer CID-Datei. Die Verbindung zur Komponente Kommunikation wird wieder über das IED-Framework hergestellt. Dem Parameterservice wird ein Verweis auf den Pufferspeicher übergeben, in dem das XML abgelegt werden soll. Als nächstes wird ein Decoderobjekt instanziiert. Diesem wird entweder der Kontextpfad oder die

direkte Speicheradresse des BiM-Fragments übergeben, welches der Decoder lädt. Das Decoderobjekt wird an das Ende eines Stacks eingefügt, von wo aus er sofort wieder entnommen wird. Über die Pull-API des Decoderobjekts wird nun wahlweise ein Element, dessen Inhalt oder ein Attribut angefordert. Nachdem der Decoder das Element bzw. Attribut ausgegeben hat, wird der Name des Elements oder Attributs für die Erstellung des Kontextpfads mitprotokolliert. Werden die BiM-Fragmente direkt über Speicheradressen zugewiesen, ist diese Aktivität jedoch überflüssig. Bei der Anforderung eines Elements wird zusätzlich geprüft, ob ein Platzhalter, oder auch „Deferred-Node" genannt, zurückgegeben wurde. Ist dies nicht der Fall, wird die Ausgabe des Decoders in XML in den Puffer geschrieben. Sobald die Speichergrenze des Pufferspeichers erreicht ist, wird der Puffer über das IED-Framework an die Kommunikation übergeben. Anschließend wird geprüft ob das Fragment abgeschlossen ist. Solange dies nicht der Fall ist, wird weiterhin das BiM-Fragment decodiert und in den Puffer geschrieben.

Wird beim Decodieren festgestellt, dass ein Platzhalter ausgegeben wurde, wird ein neues Decoderobjekt instanziiert und diesem Objekt ein Verweis auf das Fragment übergeben, welches die vollständige Instanz des Elements besitzt, das durch den Platzhalter vertreten wird. Dieses neue Decoderobjekt wird wieder an das Ende des Stacks eingefügt und sofort verwendet. Ist das BiM-Fragment des neuen Decoderobjekts vollständig decodiert, wird dieses Decoderobjekt zerstört. Da sich noch das erste Decoderobjekt im Stack befindet, wird dessen zugeteiltes Fragment weiter decodiert. Ist auch dieses Fragment abgeschlossen, wird das letzte Decoderobjekt zerstört und der Export der CID-Datei beendet.

10 Abschließend...

10.1 Rückblick

Zu Beginn der Arbeit wurden zunächst Grundlagen für das Verständnis der Aufgabenstellung und ihrer möglichen Lösungen geschaffen, bevor in Kapitel 5 mit der Untersuchung verschiedener Binärformate auf ihre Eignung begonnen wurde. Am Ende wurde sich für das Binärformat BiM entschieden, da es alle zu Beginn des Kapitels aufgestellten Anforderungen erfüllte. Im weiteren Verlauf der Arbeit wurde diese Entscheidung nochmals bestärkt, da das BiM-Verfahren sich, Aufgrund seiner Flexibilität, sehr gut an die Besonderheiten anpassen lässt. So ermöglichte das BiM-Verfahren Lösungsansätze für Probleme wie die „Abwärtskompatibilität trotz Schemaevolution" zu entwickeln, wie in Kapitel 7 beschrieben wurde. Das Kapitel 7 lässt dabei viele Probleme noch unbehandelt, für die in Zukunft noch Lösungsansätze entwickelt werden müssen. Speziell beim Thema „Schemaevolution und Versionierung" wurde auch über den Tellerrand geschaut, um mögliche Folgen für den Einsatz eines BiM-Verfahrens abschätzen zu können. Dieses Thema erwies sich jedoch als so komplex, dass sich eine weitere Untersuchung dieses Themas außerhalb dieser Arbeit empfiehlt.

Im letzten Kapitel wurde dann schließlich der Einsatz des BiM-Verfahrens auf einem IED auf seine Tragfähigkeit hin geprüft. Dabei zeigte sich, dass alle Anwendungsfälle mit einem überschaubaren Aufwand umsetzbar sind. Kommt das BiM-Verfahren in Zukunft auf einem IED zum Einsatz, sollten die Aktivitätsdiagramme jedoch nochmals konkretisiert werden. Dafür bedarf es jedoch einer genaueren Kenntnis der Softwarearchitektur eines Gerätes.

10.2 Weitere Punkte für die Zukunft

In diesem Abschnitt werden weitere Punkte angesprochen, welche aus Zeitgründen nicht mehr ausführlich beschrieben werden konnten.

XML-Schema für verschiedene Zwecke

Das BiM-Verfahren sollte auf seine Einsetzbarkeit für verschiedene XML-Schemata hin geprüft werden. Es ist durchaus denkbar, dass das XML-Format noch für weitere Zwecke außer der Konfigurierung eingesetzt wird.

Export begrenzt auf einzelne Namensräume

Zusätzlich zu dem geforderten Export einer CID-Datei, wie er in Unterabschnitt 9.3.8 beschrieben wird, können auch nur einzelne Namensräume ex-

portiert werden. Dies kann dem Schutz vor unerwünschten Rückschlüssen auf spezielle Funktionsweisen des Gerätes anhand der CID-Datei dienen.

Verschlüsselungen der Privaten Information

Eine Erweiterung des oberen Punktes ist die Verschlüsselung der privaten Informationen innerhalb der Konfigurationsdatei unter Verwendung eines speziellen Codecs. So ist es für die Herstellerfirma möglich, in einer CID-Datei, trotz Export durch einen Fremdhersteller, die privaten Informationen auszulesen.

Referenzen

In Abschnitt 4.3 wurde bereits darauf hingewiesen, dass in der SCL eine Vielzahl an Referenzen auftreten. Eine Behandlung von Referenzen könnte wie folgt aussehen: Wird im SCL-DOM ein Knoten aufgerufen, welcher eine Referenz auf einen anderen Knoten besitzt, kann über einen Verweis auf den referenzierten Knoten auf diesen sofort zugegriffen werden. Unter der Berücksichtigung der Fragmentierung des SCL-DOMs ist dies jedoch nur schwer umsetzbar.

Literatur

[NIE03] Ulrich Niedermeier, *Algorithmus und Softwarearchitektur für die binäre Codierung von Strukturierten Dokumenten*, Disertation, München 2003

[NIE35] Siehe [NIE03] **Seite 35**

[NIE45] Siehe [NIE03] **Seite 45**

[NIE38] Siehe [NIE03] **Seite 38**

[NIE83] Siehe [NIE03] **Seite 83**

[XML01] Stefan Minert (Hrsg.), *XML & Co*, Verlag: Addison-Wesley, ISBN:3-8273-1844-0

[XML02] Harald Schöning, *XML und Datenbanken*, Verlag: Hanser, ISBN:3-446-22008-9

[UML] Hitz, Kappel, Kapsmanner, Retschitzegger, *UML Work*, Verlag: dpunkt, ISBN:3-89864-261-5

[ETZ34] Karlheinz Schwarz u.a., *etz-Report 34* Offene Kommunikation nach IEC61850 für die Schutz- und Stationsleittechnik, Verlag: VDE Verlag, ISBN:3-8007-2788-9

[W3C1] World Wide Web Consortium, *XML Binary Characterization Working Group*, public Webpage, September 2005, http://www.w3.org/XML/Binary/

[W3C2] World Wide Web Consortium, *Versioning Resources*, public Webpage, Mai 2005, http://www.w3.org/2005/05/xsd-versioning-resources.html

[MPEG06] ISO/ IEC, *Binary MPEG format for XML*, ISO/ IEC 23001-1 Internatinal Standard FDIS, Januar 2006

[ASN1] International Telecommunication Union, *What ASN.1 can offer to XML*, public Webpage, Mai 2005, http://asn1.elibel.tm.fr/xml/

[FInfo] Sun Microsystems, *FastInfoset*, technical Article, Mai 2005, http://java.sun.com/developer/technicalArticles/xml/fastinfoset/

[XMill] *XMill*, public Webpage, Mai 2005, http://sourceforge.net/projects/xmill/

[dret] Erik Wilde, public Webpage, Dezember 2005, http://dret.net/netdret/

[bEss] Binary Essence, public Webpage, Mai 2005, http://www.binaryessence.de/dct/de000003.htm