OMDoc-Repräsentation von Programmen und Beweisen in √veriFun

Diplomarbeit

Normen Müller
20.09.2005

Fachgebiet Programmiermethodik
Fachbereich Informatik
Technische Universität Darmstadt

Eingereicht bei Prof. Dr. Christoph Walther

Betreut durch Prof. Dr. Michael Kohlhase
Dipl.-Inform. Andreas Schlosser
Dipl.-Inform. Stephan Schweitzer

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, September 2005

Normen Müller

Danksagung

An dieser Stelle möchte ich Herrn Prof. Dr. Christoph Walther für die Vergabe der Diplomarbeit im Fachgebiet Programmiermethodik der Technische Universität Darmstadt danken.

Weiteren Dank schulde ich den Mitarbeitern des Lehrstuhls. Ganz besonderer Dank gilt den Mitarbeitern Herrn Dipl.-Inform. Stephan Schweitzer, Herrn Dipl.-Inform. Andreas Schlosser und Herrn Dipl.-Inform. Markus Aderhold für die Bereitstellung des Themas sowie die intensive Betreuung. Sämtliche Mitarbeiter haben die gesamte Arbeit inhaltlich verfolgt, konstruktiv kommentiert und mir in den entscheidenden Phasen der Arbeit oft den Rücken freigehalten.

Prof. Dr. Michael Kohlhase möchte ich danken für die ständige Motivation und Unterstützung, insbesondere während der Implementierungsphase. Seine Bereitschaft, Modifikationen an der von ihm entwickelten Sprache OMDoc vorzunehmen, ermöglichte es mir erst, das Ziel dieser Arbeit zu erreichen.

Dipl. Phys. Immanuel Normann und Maren Gebhardt motivierten mich mit kritischen Bemerkungen an einigen Stellen zu der erforderlichen Gründlichkeit und gaben mir für diese Ausarbeitung nützliche orthographische Hinweise.

Meiner Freundin, Michaela Eckes, gebührt Dank dafür, mich immer aufgemuntert und unterstützt zu haben.

Zu guter Letzt möchte ich meinen Eltern, Ursel und Klaus Müller, danken, die mir das Studium der Informatik und damit auch diese Diplomarbeit ermöglicht haben.

Inhaltsverzeichnis

1 Motivation

Die vorliegende Diplomarbeit beschäftigt sich mit der Integration von OMDoc als neuartiges Speicherformat in ✓eriFun [Ver].

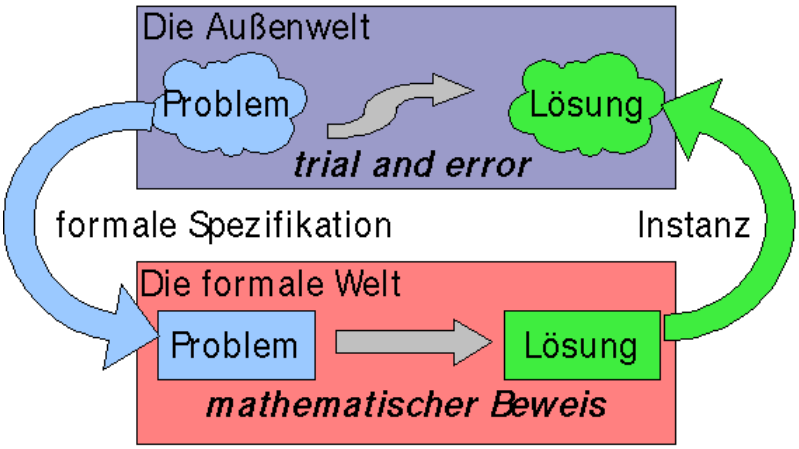


Abbildung 1.1: Die alltägliche Jagd nach Lösungen

Im Alltag ist man des öfteren mit Problemen konfrontiert, die nicht ohne Weiteres in wohldefinierte Teilprobleme untergliedern lassen. In solchen Fällen erweisen sich die erarbeiteten Problemlösungen als wenig zuverlässig (Abbildung 1.1). Ohne ein konkretes Rezept für die Lösungsfindung wird hier häufig der *trial-and-error*-Zyklus angewandt. Doch selbst mit einer bestimmten Vorgehensweise an der Hand kann für die gefundene Lösung nicht garantiert werden, dass diese ihre Gültigkeit für alle Fälle des Problembereichs beibehält. Wie kann man daher, zumindestens für eine Teilmenge aller alltäglichen Probleme, ein adäquates Rezept angeben? Die Antwort ist: *Formalisierung*. Insbesondere in der Welt der Informatik ist es üblich, ein gegebenes, jedoch noch nicht überschaubares Problem, in die formale Welt der Mathematik zu überführen, mit dem Ziel, eine analoge formale Spezifikation zu erhalten. Auf die Frage, inwieweit sichergestellt werden kann, dass die formale Spezifikation dem Problem aus der Außenwelt entspricht, ist in der Informatik letztendlich eine Glaubensfrage, der hier nicht weiter nachgegangen wird. Auf der Basis solch eines formalen Objekts lassen sich nun mathematische Algorithmen zur Lösungsfindung erstellen. Jedoch stellt sich auch hier wieder die Frage, wie man sicherstellen kann, dass solch ein gefundener Algorithmus korrekt für alle möglichen

Eingabewerte ist. Wie schon im Alltag, so wird man üblicherweise analog hier den sogenannten *development-test*-Zyklus, auch bekannt als *debugging* anwenden. Doch auch diese Vorgehensweise garantiert keine absolut sicheren Ergebnisse. Da man sich jedoch bereits in der formalen Welt befindet, sind die vorhanden Möglichkeiten noch nicht ausgeschöpft. Die Lösung auf unsere Frage hat in dieser Welt einen Namen: *Verifikation*. Darunter versteht man die wissenschaftstheoretische Forderung, nach der kein Satz als wahr anerkannt werden sollte, der nicht verifiziert, d. h. mit den spezifischen Methoden der betreffenden Wissenschaft aufgewiesen worden ist. In der formalen Welt der Mathematik bedeutet dies die Durchführung von mathematischen Beweisen für formal formulierte Aussagen. Eine solche formal formulierte Aussage ist zum Beispiel, dass der gefundene Algorithmus P als Lösung des Problems, für alle Eingabewerte d des Problembereiches D_σ terminiert: $P \in \mathtt{Prg}_{\sigma\to\iota}\text{terminiert} :\Leftrightarrow \ \forall d \in D_\sigma.\exists v \in D_\iota.P(d) \Downarrow v$. Während man somit mit Tests den Nachweis erbringen kann, dass nicht alle Fälle abgedeckt werden, also Fehler aufzeigen kann, ist es mit einer Verifikation möglich zu beweisen, dass ein Algorithmus für alle Fälle korrekt ist, also keine Fehler mehr existieren! Mit dem Werkzeug der Verifikation verliert das Debugging jedoch keineswegs seinen Stellenwert. Es empfiehlt sich vielmehr, immer erst einen Algorithmus zu testen, bevor man am Ende versucht, eine falsche Aussage zu verifizieren. Ist eine Verifikation einmal erfolgreich abgeschlossen, hat man sichergestellt, eine formale Lösung erhalten zu haben, inklusive der Garantie der Korrektheit für alle Werte des Problembereichs. Der Übergang zurück in den Alltag gestaltet sich dann relativ einfach, da es sich bei den alltäglichen Lösungen lediglich um Instanzen der formalen Lösungen handelt.

In der Informatik existieren eine Reihe von Untersuchungen, Verifikationen von komplexen Algorithmen durch Computer zu automatisieren. Durch diese Unterstützung kann man sich somit immer mehr auf die Erforschung neuer interessanter Aufgabenstellungen konzentrieren, statt zeitaufwendige Nachweise der Korrektheit einer Lösung zu erbringen. Im Fachgebiet Programmiermethodik des Lehrstuhls Informatik an der Technischen Universität Darmstadt ging das Verifikationswerkzeug ✓`eriFun` hervor (Verification of Functional programs). ✓`eriFun` ist ein Beweissystem für die Verifikation von Aussagen über Programme, geschrieben in der systemeigenen funktionalen Programmiersprache $\mathcal{FP}$. Der Anwender hat hier die Möglichkeit, *Programme* in Form von *Datenstrukturen* und *Funktionen* zu definieren. Desweiteren können über diese neu definierten Programmelemente Aussagen, *Lemmata*, erstellten werden. ✓`eriFun` enthält eine Vielzahl von vollautomatischen Routinen, sogenannte *Taktiken*, um den Anwender sowohl in der Durchführung der Beweise der einzelnen *Lemmata*, als auch in dem Nachweis der Terminierung der einzelnen *Funktionen* zu unterstützen. Da jedoch die meisten Probleme in der Programmverifikation nicht semi-entscheidbar sind, ist es unabdingbar, dass die eingebauten *Taktiken* nicht für jeden Beweis zum Erfolg führen werden. In diesem Fall bietet das System dem Anwender die Möglichkeit, in den jeweiligen Beweis einzugreifen. Er kann somit das System durch die Definition weitere *Lemmata* oder durch die explizite Anwendung einer *Taktik* in der weiteren Beweisführung unterstützen. Nach jeder Anwenderinteraktion innerhalb eines Beweises wird dann das System wieder selbstständig versuchen, den Beweis zu Ende zu führen. Um die Benutzerinteraktion so

einfach wie möglich zu gestalten, wurde in der Entwicklung von ✓eriFun sehr auf die Bedienungsfreundlichkeit der grafischen Benutzeroberfläche geachtet (Abbildung 1.2).

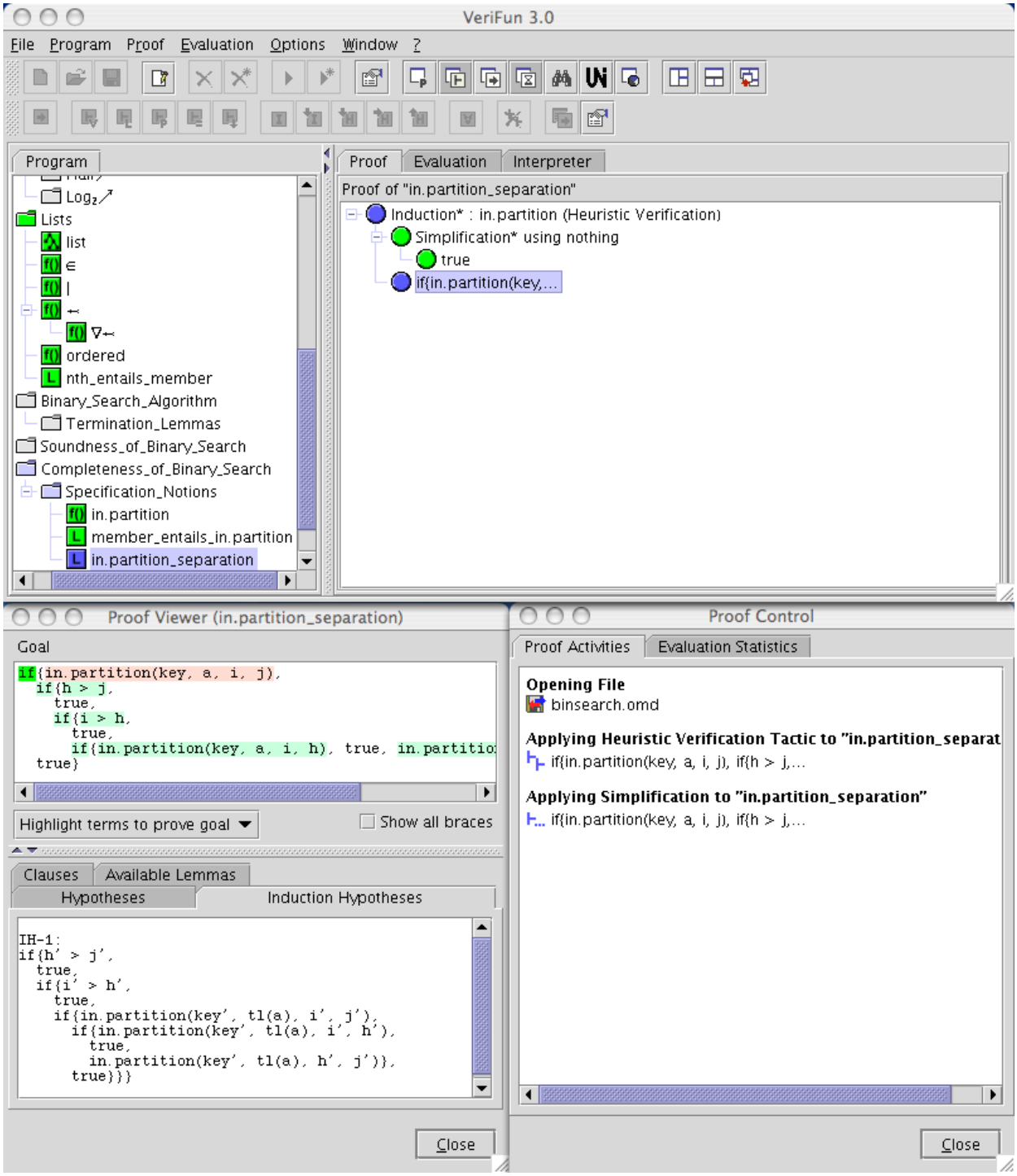


Abbildung 1.2: Eine ✓eriFun 3.0 Sitzung

Im Gegensatz zu den meisten anderen Theorembeweisern präsentiert sich ✓eriFun mit einer übersichtlichen und intuitiv bedienbaren Benutzungsoberfläche. Durch eine Reihe von grafischen Komponenten wird der Anwender mit übersichtlich formatierten und farblich hervorgehobenen Zusatzinformationen bedient, um diesem die bestmögliche Unterstützung während einer Verifikation zu bieten.

Zum Entwicklungsbeginn von ✓eriFun entschied man sich, das System auf der Plattform Java [Jav] zu implementieren. Java wurde von der Firma Sun [Sun] entwickelt und erstmals am 23. Mai 1995 als neue, objekt-orientierte, einfache und plattformunabhängige Programmiersprache vorgestellt. Der Name wurde nicht direkt von der indonesischen Insel Java übernommen, sondern von einer Bezeichnung für Kaffee, die bei amerikanischen Programmierern populär ist. Sun besitzt das Schutzrecht auf den Namen Java und stellt damit sicher, dass nur "100 % richtiges Java" diesen Namen tragen darf. Die Sprache ist für alle Computersysteme verfügbar. Java geht auf die Sprache Oak zurück, die 1991 von Bill Joy, James Gosling und Mike Sheridan im Green-Projekt entwickelt wurde. Sie verfolgte das Ziel, eine einfache und plattformunabhängige Programmiersprache zu schaffen, mit der nicht nur normale Computer wie Unix-Workstations, PCs und Apple programmiert werden können, sondern auch die in Haushalts- oder Industriegeräten eingebauten Micro-Computer, wie z.B. in Waschmaschinen und Videorekordern. Diese wichtigen Eigenschaften der plattformunabhängigen, objekt-orientierten und umfangreichen Java Klassenbibliotheken, spielten eine wesentliche Rolle in dem Entscheidungsprozeß für die ✓eriFun Entwicklungsplattform. Bei Java-Programmen muss zwischen zwei grundsätzlichen Arten unterschieden werden: Applikationen und Applets. Java-Applikationen sind Computer-Programme mit dem vollen Funktionsumfang, wie er auch bei anderen Programmiersprachen gegeben ist. Applikationen können als lokale Programme auf dem Rechner des Benutzers laufen. Java-Applets werden innerhalb einer Web-Page dargestellt und unter der Kontrolle eines Web-Browsers ausgeführt. ✓eriFun ist eine Java-Applikation.

Neben den benutzerfreundlichen, grafischen Komponenten, bietet ✓eriFun dem Anwender auch die Möglichkeit, seine erstellten Programme inklusive aller zusätzlichen Informationen zu speichern. Zur Speicherung eines Programms, mit all den darin enthaltenen *Datenstrukturen*, *Funktionen*, *Lemmata* und Beweisen, verwendet ✓eriFun den Java–Serialisierungsmechanismus[1]. Hier werden von Java Klassen für Dateioperationen zur Verfügung gestellt, mit deren Hilfe es möglich ist, beliebige Objekte (Quelle) des laufenden Programms in eine binäre Datei (Senke) zu schreiben und umgekehrt. Dieser einfache Mechanismus erlaubt es, Daten schnell persistent zu machen. Auch lassen sich in der Regel solche binären Dateien schneller laden und benötigen weniger Speicherplatz auf dem verwendeten Datenträger als textbasierte Dateiformate. Ferner werden Binärformate beim Austausch über verschiedene Plattformen hinweg (beispielsweise Windows, Macintosh, Linux) nicht beschädigt, da die jeweiligen Softwarekomponenten nicht versuchen, die Dateien für die Zielplattform zu konvertieren.

[1] Serialisierung: Enkodierung eines zur Laufzeit existierenden Java-Objektes in einem Bytestrom.

Allerdings lassen sich die Informationen in den binären Dateien jedoch aus beschädigten oder versions-inkompatiblen Dateien nicht wieder restaurieren. Werden so zum Beispiel von einer ✓eriFun-Version zur nächsten Attribute und/oder Methoden innerhalb der JAVA-Klassen des Systems modifiziert, bedeutet dies gleichzeitig, dass eine gespeicherte Objektserialisierung nicht mehr gültig ist. Sämtliche serialisierten Informationen eines ✓eriFun-Programms, wie z.B. *Lemmata* und deren Beweise, gehen somit verloren. Zusätzlich zu dieser Gefahr des Informationverlustes kommt die Notwendigkeit spezieller Editoren zum Lesen, Bearbeiten und Speichern solcher binären Datenformate, womit man sich oft in die Abhängigkeit eines Softwareherstellers begibt. In diesem Fall ist somit eine Weiterverarbeitung durch Nicht-JAVA-Applikationen bzw. vielmehr eine Weiterverarbeitung ohne ✓eriFun oder nachträgliche Änderungen per Hand, ohne Einlesen und Wiederaufbauen der Objektverbunde durch ✓eriFun, gänzlich unmöglich. Wünschenswert ist daher eine von der aktuellen ✓eriFun-Version unabhängige, nicht proprietäre und plattformübergreifende Repräsentation von ✓eriFun-Programmen. Auf diese Weise würde man die Zugänglichkeit und Dauerhaftigkeit der Daten gewährleisten, eine vollständige Transparenz der Daten erhalten und die Vervielfältigung und Kompatibilität fördern.

Die einfachste und wohl bekannteste Repräsentation, die diese Merkmale bis zu einem gewissen Grade erfüllt, ist das ASCII Datenformat (AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE). Hier werden die Informationen durch einen standardisierten Zeichensatz zur Textdarstellung für Computer und anderen Kommunikationseinrichtungen repräsentiert. Ein großer Vorzug von Textrepräsentation als Dateiformat ist offensichtlich: Der Inhalt von den Dateien ist prinzipiell für den menschlichen Betrachter lesbar. So können Dateien, die z.B. mit der dafür vorgesehenen Anwendung nicht mehr bearbeitet werden können[2], notfalls mit einem simplen Text-Editor modifiziert werden. Die Lesbarkeit von Dateien für Menschen kann somit im Einzelfall einen entscheidenden Beitrag zur Rekonstruktion von Daten und damit letztlich zur Robustheit von Anwendungssystemen leisten. Im Fall von ✓eriFun bedeutet dies, dass die erstellten Programme nicht mehr an die dafür verwendete ✓eriFun-Version gebunden sind. Systemveränderungen der Implementierung oder semantische Modifikationen, z.B. an den eingebauten *Taktiken*, sind nunmehr hinsichtlich der Wiederverwendbarkeit unkritisch. Die Zugänglichkeit, Dauerhaftigkeit und vollständige Transparenz der Daten wäre somit gewährleistet. Um jedoch die Vervielfältigung und die Kompatibilität zu fördern, muss die neue Repräsentation von ✓eriFun-Programmen desweiteren auf einem Standard basieren, der einen Austausch der Daten mit anderen Softwaresystemen, wie zum Beispiel dem Textsetzungssystem LaTeX oder anderen Theorembeweisern, ermöglicht. Eine gelungene Erweiterung des ASCII Datenformates sind die sogenannten *Auszeichnungssprachen* (engl. Markup Language, Abk. ML). Bei einer Auszeichnungssprache werden Eigenschaften, Zugehörigkeiten und Verfahren von bestimmten Wörtern, Sätzen und Abschnitten eines Textes beschrieben bzw. zugeteilt, meist in dem sie mit so genannten

[2]Die Gründe dafür können unterschiedlich sein. Beispielsweise könnte die Datei beschädigt worden sein, und die Anwendung könnte sich daraufhin weigern, die Datei zu öffnen.

TAGs[3] markiert werden. Die Quelltexte werden weiterhin in dem ASCII Datenformat verfasst[4].

Mit diesen Kriterien, Wiederverwendbarkeit, Maschinen-Verstehbarkeit von Inhalten, Interoperabilität zwischen Anwendungen, Lesbarkeit des Dateiformats für den Menschen und die Auszeichnung von Textelementen, beschäftigen sich auch die weltweiten Forschungen zum SEMANTIC WEB. Die Motivation in diesem Forschungsgebiet liegt in der täglich drastisch ansteigenden Menge verfügbarer Daten im Internet. Um so wichtiger werden Technologien, die es ermöglichen, aus den verfügbaren Daten auszuwählen und Informationen zu selektieren bzw. zu verarbeiten. Zur Zeit stehen im wesentlichen Suchmaschinen zur Verfügung, mit denen das Internet nach Informationen durchsucht werden kann. Zugriffsmuster werden herangezogen, um Relevanz von Daten zu bestimmen. Wichtige Charakteristika bei bisherigen Anwendungen für das Internet sind, dass die Auswahl der präsentierten Informationen und deren Einordnung durch den Menschen geschieht. Als Beispiel für das erste Kriterium stelle man sich vor, es soll ein Programm entwickelt werden, das alle relevanten Buchhändler-Seiten à la Amazon nach einem bestimmten Buch durchsucht und automatisch das günstigste Angebot ordert. Wo in der Seite der Preis versteckt ist, ob es sich um Euro oder Pfund handelt und der Versandkostenanteil inbegriffen ist, kann ein menschlicher Betrachter der Seite relativ leicht erkennen, ein Stück Software stolpert allerdings meist schon beim ersten Teil einer solchen Aufgabe. Ein Beispiel für die Einordung der präsentierten Informationen wäre die Anforderung, alle Nachrichten eines Web-Nachrichtendienstes automatisch nach Inhalt zu sortieren und in sinnvolle Kategorien zu unterteilen. Das Problem, welches hier zu Tage tritt ist, dass Computerprogramme die *Bedeutung* von Ressourcen nicht verstehen können. Genau hier setzt das SEMANTIC WEB ein: Ressourcen im Internet werden mit einer maschinenlesbaren Bedeutung versehen, so dass Computerprogramme durch die zusätzlichen Informationen über die Semantik bessere Unterstützung beim Umgang mit Informationen aus dem Internet bieten können. Es handelt sich somit hierbei um eine Erweiterung des WORLD WIDE WEB um maschinenlesbare Daten, welche die Semantik der Inhalte formal festlegen. Durch die zusätzlichen Annotationen wird zum einen eine bessere Kategorisierung der Daten und zum anderen Schlußfolgerungen aus diesen Daten ermöglicht [BLHL01]. Um diese Grundidee, die Kommunikation als Basis von Kooperation und Integration über Applikations- und Organisationsgrenzen hinweg, umzusetzen, werden die Erfahrungen und Techniken des *D*okumenten-Markups (Kapitel 2) eingesetzt. Die Untersuchungen des Dokumenten-Markups führten von *prozeduralem Markup* (Abschnitt 2.1) über *deskriptiven Markup* (Abschnitt 2.1) bis hin zu dem für diese Arbeit interessantem SEMANTIC XML (Abschnitt 2.5). Während *prozedurale Markup*sprachen

[3] In der Datenverarbeitung und Informatik steht das englische Wort *tag* (Etikett, Anhänger, Aufkleber, Marke) für die Auszeichnung eines Datenbestandes mit zusätzlichen Informationen.

[4] Ein weiteres heutzutage gängiges Textformat ist der UNICODE. Es handelt sich auch hierbei um eine Erweiterung des ASCII Datenformates jedoch nicht bis hin zur Ebene der Auszeichungssprachen. UNICODE ist ein internationaler Standard, in dem langfristig für jedes sinntragende Zeichen bzw. Textelement aller bekannten Schriftkulturen und Zeichensysteme ein digitaler Code festgelegt wird. Mit dem UNICODE-Format will man das Problem der verschiedenen inkompatiblen Codierungen in den unterschiedlichen Ländern beseitigen.

(z. B. HTML) dafür gedacht sind, Inhalte mit Formatierungsinformationen zu annotieren, lassen sich in *deskriptiven Markup*sprachen (z. B. XHTML als XML-Applikation) Strukturelemente definieren. Dokumente, die in einer *deskriptiven Markup*sprache erstellte sind, können somit auf der ihr unterliegenden Struktur (Grammatik) typisiert werden. SEMANTIC XML ermöglicht darüber hinaus, Eigenschaften dieser Strukturelemente und Relationen zwischen ihnen zu modellieren.

Die Entscheidung über die externe Repräsentation von ✓eriFun-Programmen fällt somit in die Menge der *deskriptiven* Markupsprachen, bzw. vielmehr in die Teilmenge des SEMANTIC XML.

> *Das Ziel dieser Arbeit ist es, eine semantische deskriptive Markupsprache mit wohlgeformter Grammatik in ✓eriFun zu integrieren und somit das System an dem komplexen Netzwerk der mathematischen Welt teilhaben zulassen.*

2 Dokumenten-Markup

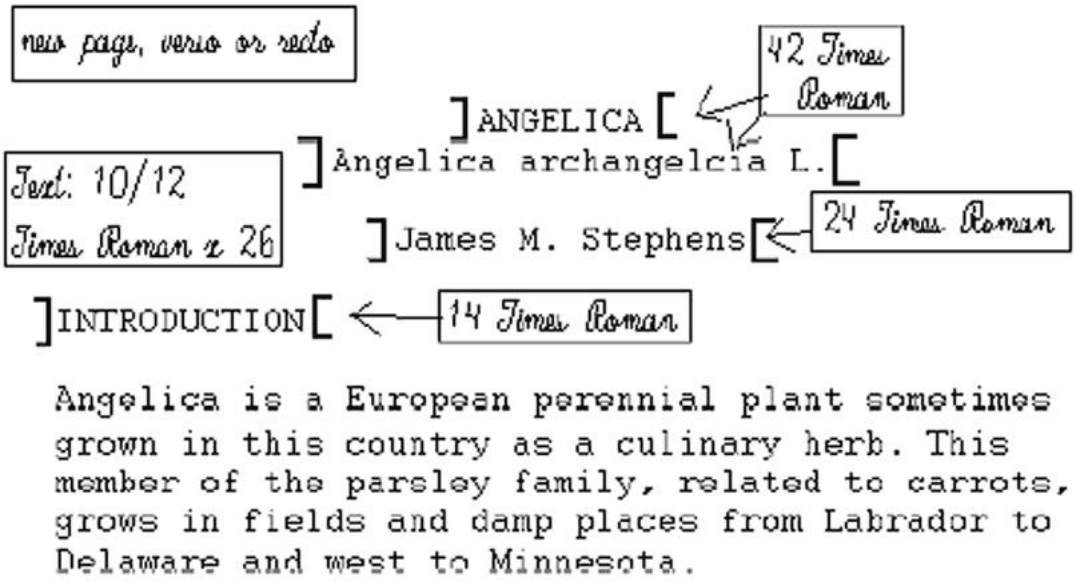


Abbildung 2.1: Die Anfänge von Dokumenten-Markup [Wat02].

Markup ist der Prozeß des Hinzufügens von Annotationen zu einem Dokument, um dessen Struktur und/oder Format zu kennzeichnen [Wat02]. *Dokumenten-Markup* als Unterstützung für eine bessere Kommunikation hat seit vielen Jahren Bestand. Bis zur Automatisierung der Druckindustrie wurden Auszeichnungssprachen hauptsächlich von Herausgebern von Manuskripten verwendet, um den Schriftsetzern Formatierungsinstruktionen mitzuteilen (Abbildung 2.1). Diese Instruktionen stellen die Anfänge des *prozeduralen Markups* (Abschnitt 2.1) dar. Mit der Zeit wurden eine Vielzahl von solchen Standardsymbolen entwickelt und benutzt. Es war daher im Computerzeitalter nur allzu offensichtlich, diesen Prozess zu digitalisieren, um eine höhere Effizienz in der Bearbeitung der Dokumente zu erlangen. Die verschiedensten Textformatierungssprachen entstanden, um das Format von Dokumenten beschreiben zu können. Ein Problem jedoch, welches durch die Vielzahl dieser individuellen Auszeichnungssprachen entstand, war der Austausch der Dokumente untereinander. Es konnte nicht sichergestellt werden, dass eine vollständige Konvertierung von der einen Sprache in eine andere korrekt vollzogen werden konnte. Die diversen Softwaresysteme, mittels derer Textformatierung vorgenommen werden konnten, entwickelten vielmehr zunehmend proprietäre Dateiformate, um ihre

system-spezifischen Formatierungsinformationen zusammen mit dem eigentlichen Text zu speichern. Die wohl bekanntesten Vertreter an dieser Stelle sind PDF und MS WORD DOC. Die Interoperabilität der Textformatierungsprogramme wurde mehr und mehr eingeschränkt, wenn nicht gar vollkommen unmöglich. Um, unter anderem, dieser Entwicklung entgegenzusteuern, besann man sich *deskriptiver Markup*sprachen (Abschnitt 2.1). Die Mutter aller *deskriptiver Markup*sprachen GML (Abschnitt 2.2) etablierte sich später in dem bekannten Standard SGML (Abschnitt 2.3). Mit diesen Meta-Markupsprachen[1] war der Weg für die Definition neuer Markups, wie XML (Abschnitt 2.4, Kapitel 3) und das in dieser Arbeit verwendete SEMANTIC XML (Abschnitt 2.5) geebnet.

2.1 Prozeduraler und deskriptiver Markup

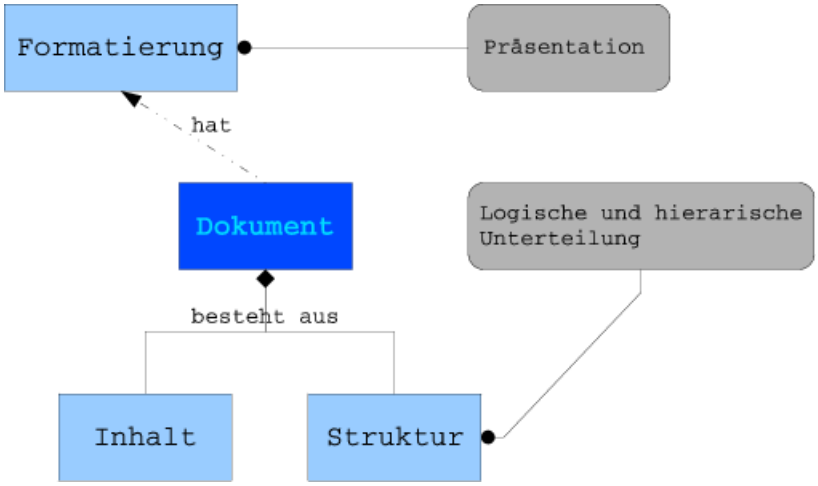


Abbildung 2.2: Abstrakte Bestandteile eines Dokuments.

Jedes Dokument kann verstanden werden als die Zusammenführung von Inhalt (*content*) und Auszeichnungen (*markup*) (Abbildung 2.2). Den Inhalt kann man sehr einfach erklären: Dort werden die einzelnen Zeichen des Textes, die Bilder und was sonst noch zu dem Informationsgehalt des Dokuments beiträgt, zusammengefügt. Zu beachten ist hierbei jedoch, dass es mit einfachem Zusammenfügen nicht getan ist, das Dokument wäre dann nicht lesbar. Man muss die einzelnen Textpassagen auszeichnen (*mark it up*). Darunter versteht man also, neue Informationen dem Dokument hinzuzufügen, die nicht automatisch aus dem Inhalt des Dokuments abgeleitet werden können. Somit enthält auch ein einfacher ASCII-Text, oft referenziert als ein einfacher, purer Text ohne jegliche Auszeichnungen, ein gewisses Maß an Formatierungsanweisungen. Man bedenke nur den Markup, der es dem Leser gestattet zu erkennen, wann ein neuer Paragraf beginnt,

[1] Meta-Sprache: Eine Sprachebene, auf der über eine vorgegebene Sprache Aussagen gemacht werden, um diese gegebenenfalls zu beschreiben. Die beschriebene Sprache wird als *Objektsprache* bezeichnet.

oder wie breit die einzelnen Textspalten sind. Das zur Verfügung stehende Inventar solcher Auszeichnungsinstruktionen (TAG) hängt somit von dem verwendeten Werkzeug ab. Welche weiteren Informationen sollten neben dem Inhalt eines Dokuments noch beigefügt werden?

Diese Informationen lassen sich in zwei Rubriken unterteilen: *Struktur* und *Formatierung*. Die Struktur kennzeichnet dem Leser zum einen die logischen Unterteilungen des Dokuments, beispielsweise in Paragrafen, Sektionen und Kapitel und zum anderen, wie diese Unterteilungen hierarchisch, von dem Dokument als Ganzes bis hin zu den Atomen des Inhaltes, angeordnet werden. Formatierung bestimmt die Präsentation des Dokuments, zum Beispiel die Schriftart oder an welcher Stelle Seitenumbrüche in einem Ausdruck vorgenommen werden sollen. Genau diese Unterteilung leitete die kontroverse "Markup-Diskussion" ein, die eine der treibenden Kräfte für die Entwicklung von GML (Abschnitt 2.2) bzw. später SGML (Abschnitt 2.3) war. Im Kern dieser Diskussion ging es um die Frage, welche der beiden Rubriken wichtiger sein: *Struktur* oder *Formatierung*? Zu jener Zeit wurden eine Vielzahl von Textverarbeitungssystemen entwickelt, die zum einen auf proprietären Dateiformaten basierten und zum anderen die Konzentration des Markups auf die Formatierung des Textes legten. Diese Art von Auszeichnungen wird als *prozeduraler Markup* (oder auch *presentational Markup*) bezeichnet (Abbildung 2.3).
Der Grund für diese Entwicklungsrichtung war offensichtlich: Der gemeine Anwender

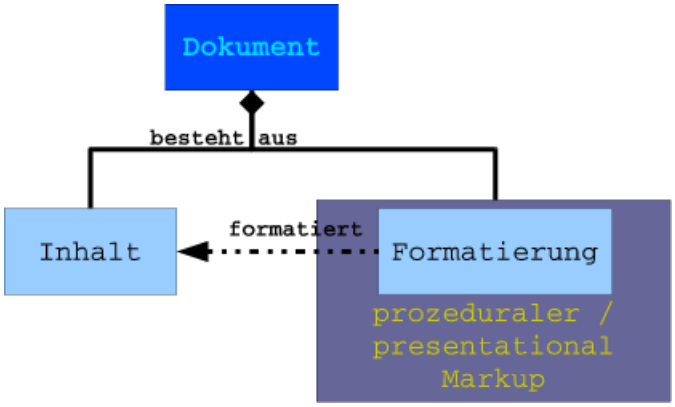


Abbildung 2.3: Prozeduraler Markup.

war mehr an der Formatierung seines Dokuments interessiert, statt an einem strukturellen Diagramm seiner Aufzeichnungen. Die bittere Wahrheit war jedoch, dass *prozeduraler Markup* die Verwendung und Wiederverwendung der Dokumente stark behinderte, wenn dieser nicht durch eine Struktur begleitet, ja nicht sogar angeführt, wurde. Formen von Markups, die dieses zusätzliche Kriterium erfüllen, bezeichnet man als *deskriptives Markup*. Um dies zu verdeutlichen, hier ein Beispiel: Wenn in einem Dokument Instruktionen enthalten sind, um eine Textzeile auf die Schriftart "Times Roman", Schriftgröße 12 und linksbündig zu setzten, aber an keiner Stelle des Dokuments erwähnt wird, dass es sich dabei um eine Überschrift handelt, dann ist der verwendete Markup sehr eingeschränkt. Man könnte auch nicht die Formatierung aller Überschriften des Dokuments

auf einmal verändern. Die Konvertierung solch eines Dokuments in ein anderes Format gestaltet sich auch sehr schwer, da zum Beispiel das Zielformat eventuell ganz andere Formatierungsanweisungen für Überschriften verwendet. Eine automatische Weiterverarbeitung, wie beispielsweise die Erstellung eines Inhaltsverzeichnisses, ist zudem auch nur eingeschränkt möglich. Um es kurz zufassen: Solange man nicht weiß, *was* bestimmte Textpassagen darstellen sollen, stellen die Anweisungen, *wie* die Darstellungsweise erfolgen soll, nur einen sehr begrenzten Informationsgehalt dar. Auf der anderen Seite ist es ein Leichtnis, nachdem man dem Dokument Informationen über die logischen Rollen der einzelnen Textbausteine hinzugefügt hat, *prozeduralen Markup* an diese logischen TAGs zu ettiketieren, statt an die Textbausteine direkt. Nun muss für eine Überschrift keine Schriftart oder Schriftgröße angegeben werden, sondern ein Textbaustein wird nur noch als solcher gekennzeichnet und der Rest funktioniert automatisch (unter der Voraussetzung, dass einer Überschrift schon Formatierungsanweisungen zugewiesen wurden). Dieses Konzept der Trennung von Präsentation und Inhalt, ist der große Vorteil sämtlicher Systeme, die einem *deskriptiven Markup* Vorrang geben. Ist die Trennung einmal vollzogen, können der Inhalt und die Präsentation von unabhängigen Parteien bearbeitet werden. *Deskriptiver Markup* dient somit als Schnittstelle zwischen dem Inhalt und der Formatierung eines Dokuments (Abbildung 2.4).

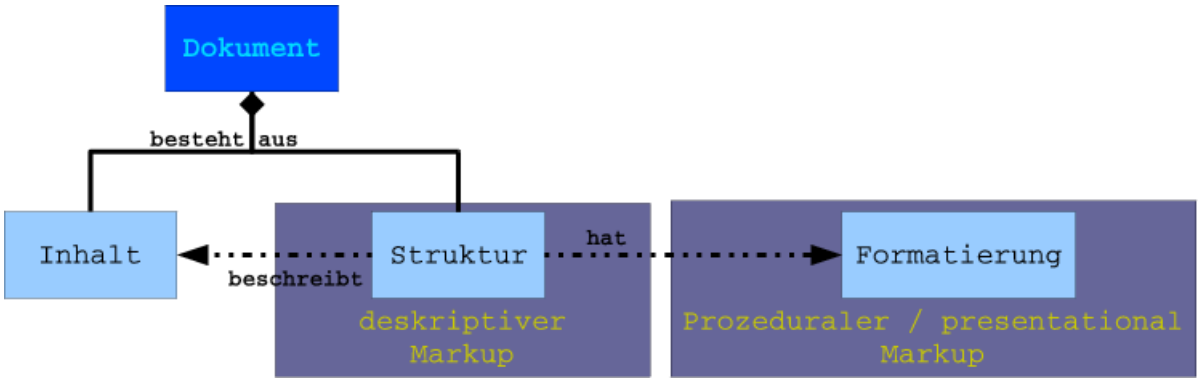


Abbildung 2.4: Deskriptiver Markup.

An dieser Stelle sei erwähnt, dass heutzutage gängige Textverarbeitungssystem, wie zum Beispiel MS WORD, diesen Ansatz nicht komplett ignorieren. Auch dort ist es möglich, wenn auch nur begrenzt, Textpassagen eines Dokuments einen bestimmten "Style" zuzuweisen. Diesem "Style" können dann wiederum Formatierungsanweisungen angehängt werden. Es handelt sich jedoch nur um eine partielle Lösung, einen *teilweise-deskriptiven Markup*, denn mit diesen "Style"-Anweisungen ist es in keiner Weise möglich, dem Dokument eine logische Struktur zu verleihen. Sinnlose Zuweisungen der Art, einer Fußnote den "Style" Überschrift zu erteilen, sind vollkommen legal. Solche "Styles" sollten also vielmehr als Container für Formatierungsanweisungen betrachtet werden.

Ein weiterer Vorteil von *deskriptivem Markup* stellt sich in der Weiterverarbeitung heraus. Durch die Verwendung einer wohldefinierten Menge von hierarchisch deskriptiver

TAGs können die Dokumente automatisiert nach Informationen durchsucht oder in andere Formate überführt werden. Zusammenfassend kann man sagen, um den Inhalt vor die Präsentation eines Dokuments zu stellen, benötigt man eine Markupsprache, die den Fokus auf die Struktur des Dokuments statt auf die Formatierung legt. Diese Sprache sollte es ermöglichen, eine Hierarchie über deskriptive TAGs aufbauen zu können, so dass damit nicht nur das Dokument in unterschiedliche und separierte Textpassagen aufgeteilt, sondern auch formal dessen Struktur beschrieben werden kann. Desweiteren sollte diese Sprache erweiterbar und modifizierbar sein. Idealerweise sollte der Anwender der Sprache eigene deskriptive TAGs definieren können. Letztendlich kommt hinzu, dass dieses System keinen proprietären Charakter erhalten darf, sondern, ganz im Gegenteil, die Interoperabilität zwischen Softwareprogrammen fördern soll.

Die folgenden Abschnitte geben einen chronologischen Überblick über die Entwicklung der einzelnen Markupsprachen. Dort soll unter anderem der Entscheidungsfindungsprozeß des Autors, über GML (Abschnitt 2.2), SGML (Abschnitt 2.3), XML (Abschnitt 2.4, Kapitel 3) und hin zu SEMANTIC XML (Abschnitt 2.5), verdeutlicht werden.

2.2 GML

C. F. Goldfarb gilt als Initiator der Standardisierung von Markup Sprachen, deren Geburtsstunde im Folgenden skizziert wird: C. F. Goldfarb übte 1966 sein Tätigkeit als Rechtsanwalt in Boston aus. Zu diesem Zeitpunkt hatte er, wie wohl in dieser Branche üblich, keinen direkten Kontakt zu der Welt des Computers. Doch auch ohne die Kenntnisse über die Fähigkeiten solcher digitaler Wunderwerke, machte er sich Gedanken über die Verwaltung von Dokumenten. Er wollte und konnte es nicht einsehen, dass die ihm bekannten Methoden zur Auszeichnung, Bearbeitung und Suche in/nach Dokumenten nicht verbessert werden konnten. Schließlich war es sein Hobby, die Auto-Rally, welches ihm auf den richtigen Weg zur Lösung seines Problems verhalf. Er war dort dafür zuständig, für den Fahrer einen Plan der abzufahrenden Route zu erstellen. Im Gegensatz zu allen anderen "Routenplanern" ähnelten seine Beschreibungen der jeweiligen Strecken sehr einem Computerprogramm (Listing 2.5).

Listing 2.5: Die Anfänge von GML

```
1 26. Left at light onto Jones Rd.
  27. (Repeat instructions 20 − 26, substituting "left" for "right".)
  28. Second right.
```

Auf diesen Sachverhalt macht ihn ein Freund aufmerksam, da er, wie gesagt, nicht mal im entferntesten eine Ahnung von einem Computerprogramm hatte.

1967 war es die Firma IBM, die ihn dazu ermutigte, herauszufinden welchen Profit man mit der Erstellung solcher "Routenplaner" gewinnen könnte. Einmal in dieser Firma Fuß gefasst, kamen weitere Projekte auf ihn zu, so zum Beispiel die Erstellung eines

Textformatierungssystems für eine lokale Zeitung. Dieses Projekt ermöglichte ihm seine schon oben erwähnten Überlegungen hinsichtlich der Verwaltung von Dokumenten, zu erweitern. Den Durchbruch bewirkte bei C. F. Goldfarb aber ein anderes Projekt. 1969 sollte er eine Textverarbeitungssystem entwickeln, mit welchem, neben anderen Funktionalitäten, nach Informationen in den erstellten Dokumenten gesucht, Dokumentseiten verglichen und ausgetauscht werden könnten. Desweiteren sollte mit diesem Textverarbeitungssystem Präsentationauszeichnungen des Dokumentformats beschrieben werden können. Jedoch direkt zu Beginn dieses Projektes, sah sich C.F. Goldfarb mit einer Menge von Problemen konfrontiert: Die Computer, auf denen das zu erstellende Textverarbeitungsprogramm laufen sollte, waren in keiner Weise in der Lage, miteinander zu kommunizieren. Hinzu kam die Plattformabhängigkeit bestehender zu integrierender Programme wie Datenbanken zur Archivierung der Dokumente, so dass auch diese nicht einfach auf anderen Computersystemen genutzt werden konnten. Der Autor möchte an dieser Stelle darauf hinweisen, dass in seinen Augen diese Probleme als Vorteile, im folgenden Sinne, zu deuten sind:

> *Auch eine durch Liebe hervorkommende Geburt ist schmerzhaft, anstrengend und auf den ersten Blick wohl kaum zu bewältigen, aber das Ergebnis ist einfach liebenswert!*
>
> Normen Müller

In welche Richtung hätten sich die Markupsprachen entwickelt, wäre C. F. Goldfarb nicht mit solchen Problemen konfrontiert worden?

Zurück zu dem Textverarbeitungsprogramm. Als ersten Schritt schafften es C.F Goldfarb und seine Mitarbeiter mittels eines Hypervisors[2], dass sich die verschieden Softwareprogramme nun untereinander unterhalten konnten. Damit jedoch die unterschiedlichen Programme die auszutauschenden Dateien interpretieren konnten, war anfangs *interpreted Markup* von Nöten. Unter *interpreted Markup* verstehen wir *prozeduralen Markup* angereichert durch spezifische Programminstruktionen. Die Gruppe um C. F. Goldfarb erkannte schnell, dass es sich hierbei um einen falschen Ansatz handeln würde. Der *interpreted Markup* musste aus den Dateien verschwinden und durch einen "nichtprogrammspezifischen" Markup ersetzt werden. Das Ergebnis war 1971 die Geburtsstunde der Generalized Markup Language (Gml). Diese neue *deskriptive Markup*sprache war nicht als eine Alternative zu *interpreted Markup* zu verstehen, sondern ermöglichte erstmals, die strukturelle Repräsentation der Daten zu beschreiben. Durch die Menschenlesbarkeit und die Möglichkeit, eine Menge von deskriptiven Tags zu definieren, waren Autoren ermächtigt, ihr Dokument in unterschiedliche und separierte Textpassagen aufzuteilen. Die Präsentationsinformationen konnten dann programmspezifisch den Tags zugewiesen werden. Durch die Offenheit dieser Markupsprache bezüglich der Definition der Auszeichnungselemente, bezeichnet man Gml auch als Meta-Markupsprache (siehe hierzu auch Abschnitt 2.5). C.F. Goldfarb stellte jedoch fest, dass diese Errungenschaften noch nicht zum eigentlichen Ziel geführt haben. Man

[2]System, welches es erlaubt, mehrere Betriebssysteme unverändert, auf einem Computer gleichzeitig laufen zu lassen.

benötigt zusätzlich einen Mechanismus, mittels dessen man die Struktur eines Dokuments und dessen Typ restriktiver festlegen kann. Man benötigt die Möglichkeit, Grammatiken definieren zu können, um auf diese Weise Typklassifikationen über Dokumente vornehmen und deren grammatikalische Struktur strikt definieren zu können (Abschnitt 2.3). Als sich 1973 die Entwicklung des Textverarbeitungssystem dem Ende zuneigte, erblickte GML das erste mal das Licht der Welt. Das anfänglich noch sehr primitive Textverarbeitungssystem ADVANCED TEXT MANAGEMENT SYSTEM (ATMS) verwendet zur Speicherung und Verwaltung der Dokumente diese neue, offene und menschenlesbare Markupsprache.

Die Analyse des Dokumenten-Makups durch C.F. Goldfarb hat ihn zu folgenden weitreichenden Ergebnissen gebracht: Es wurde eine nicht programmspezifische, sondern beschreibende (deskriptive), offene und menschlesbare Markupsprache entwickelt, so dass durch diese die Interoperabilität zwischen verschieden Softwareprogrammen gefördert wurde. GML stellte über die Menge der deskriptiven TAGs sicher, dass gleich markierte Teile eines Dokuments über Programmgrenzen hinaus immer dieselbe Behandlung erfahren. Die Interpretation der jeweiligen Markierungen bleibt zwar weiterhin programmspezifisch, doch dort immer die gleiche! Desweiteren unterstützt diese Markupsprache auch die Robustheit der Daten. Mit der Robustheit wird an dieser Stelle die Menschenlesbarkeit, sprich die textuelle Repräsentation der Dokumente und den darin enthaltenen Annotationen, in den Vordergrund gehoben. Autoren könnten beispielsweise beschädigte Dokumente bearbeiten und gegebenenfalls rekonstruieren.

2.3 SGML

Motiviert durch die GML-Errungenschaften und die Erkenntnisse eines noch fehlenden Mechanismus zur Definition einer Grammatik, begann 1978 C.F. Goldfarb zusammen mit der GRAPHIC COMMUNICATIONS ASSOCIATION (GCA) basierend auf GML (Abschnitt 2.2) die Entwicklung der STANDARD GENERALIZED MARKUP LANGUAGE (SGML).

SGML ist eine rein *deskriptive Markup*sprache, die keine TAGs hinsichtlich *prozeduralem Markup* enthält. Die große Stärke verbirgt sich hinter der Möglichkeit, die Sprache nach Belieben anzupassen. Wie schon bei GML spricht man auch hier von einer Meta-Markupsprache (siehe hierzu auch Abschnitt 2.5). In Erweiterung zu GML ist es jedoch hier nicht nur möglich, neue Sprachen, sondern zugleich neue *Dokumenttypen* zu definieren. Auch aus der Sicht von SGML sind Dokumente hierarchische Strukturen ausgezeichnet durch deskriptive TAGs. Die formalen Informationen über die TAGs, die mit SGML kommuniziert werden können, betreffen ausschließlich den Kontext und die Ebene innerhalb einer Dokumenthierarchie, in dem die jeweiligen TAGs eingesetzt werden können. Es wird keine Semantik der TAGs gefordert (siehe hierzu Abschnitt 2.5), sondern vielmehr die Maxime von Wittgenstein propagiert:

The meaning of a word is its use!

Diese formale Zerlegung eines Dokuments ist aber dennoch nicht zu unterschätzen, denn alle Dokumente die mit der gleichen Hierarchie über einer Menge von deskriptive TAGs versehen werden können, gehören ein- und demselben Dokumenttyp an! Man konnte nunmehr in SGML nicht nur die Struktur von einem Dokument, sondern die Struktur von Dokument-Familien beschreiben und somit Typen von Dokumenten definieren. Zur Definition von Dokumenttypen bedient sich SGML der von Ed Mosher schon 1971 entwickelten Sprache der DOCUMENT TYPE DEFINITION (DTD). Mit dieser Sprache ist es zum einen möglich, die Menge der für einen Dokumenttyp zur Verfügung stehenden deskriptiven TAGs zu spezifizieren und zum anderen Regeln auf dieser Menge zu definieren, nach denen die TAGs zusammengesetzt werden müssen (Abschnitt 3.4). So wird dort unter anderem definiert, welche TAGs in diesem Dokumenttyp optional oder zwingend notwendig sind, wie die TAGs, wenn sie verwendet werden, verschachtelt werden müssen, welche Attribute ein TAG enthalten kann und von welcher Art die Werte dieser Attribute sein müssen. Aus diesen Gründen kann eine DTD als Grammatik einer Auszeichnungssprache bezeichnet werden. Konkret mit einer solchen Grammatik ausgezeichnete Dokumente werden als *Dokumenttypinstanz*en oder kurz *Dokumentinstanz*en bezeichnet (Abbildung 2.8). Dementsprechend muss jede Dokumentinstanz mit der Deklaration[3] einer DTD beginnen. Diese Typisierung von Dokumenten stellt im Zusammenhang der Standardisierung der Repräsentation von Informationen ein mächtiges Werkzeug dar, denn sie entscheidet über die Weiterverarbeitung des Inhaltes der Dokumente durch Programme. Für Dokumente, die den gleichen Regeln unterliegen, können Anwendungen geschaffen werden, die die Informationen aus diesen auslesen und automatisch weiterverarbeiten. Deswegen ist es sehr wichtig, genau zu spezifizieren, welche DTD man benutzt, wenn man über SGML-Applikationen spricht. Eine DTD wird auch das Herz einer SGML-Applikation genannt.

1986 wurde diese neue Meta-Markupsprache erstmals veröffentlicht. Man repräsentierte der digitalen Welt *die* Metasprache für die Definition von Markupsprachen. Darin erstellte Dokumente werden beschrieben als reine Textdokumente kombiniert mit Auszeichnungs-Elementen. Diese, für die Strukturauszeichnung des Dokuments herangezogenen TAGs, müssen einschließlich der Grammatik, zuvor in einer entsprechenden DTD beschrieben werden.

Eine der wohl bekanntestes SGML-Applikation entstand im Jahre 1989. Der Forscher Tim Berners-Lee schlug damals vor, dass Informationen innerhalb der CERN EUROPEAN NUCLEAR RESEARCH FACILITY unter Verwendung von Hyperlink-Textdokumenten gemeinsam genutzt werden könnten. Ein Kollege, der früher am SGML-Standard arbeitete, schlug ihm zur Repräsentation der Dokumente vor, eine ähnliche Syntax zu SGML zu benutzen. Somit begannen sie bei der Entwicklung mit einem einfachen Musterdokumenttyp im SGML-Standard und entwickelten eine Hypertextversion. Diese nannten sie HYPERTEXT MARKUP LANGUAGE (HTML). Verglichen mit dem 20-jährigen Entstehen

[3] Lokale Deklarationen einer DTD können durch Referenzen auf externe DTDs ersetzt bzw. ergänzt werden. Siehe hierzu auch Abschnitt 3.4

von SGML wurde HTML geradezu hastig entwickelt, aber es funktionierte. Der Forscher Tim Berners-Lee nannte sein Hypertextsystem das WORLD WIDE WEB, welches heute allen nur all zu gut bekannt ist. Als wichtiger Teil des Erfolges des WORLD WIDE WEB wird weitestgehend seine Einfachheit angesehen. HTML ist ein speziell für das Internet entwickelte Dokumentenbeschreibungssprache. Auf Grund der Syntax, die durch eine SGML-Deklaration und eine DTD festgelegt wird, nennt man diese Markupsprache eine SGML-Applikation. An dieser Stelle sei somit darauf hingewiesen, dass mit HTML ursprünglich ein Dokument mit einer Struktur versehen werden konnte! Es war anfänglich daran gedacht worden, um den verschiedenen Browser-Typen zu ermöglichen ein Dokument sinnvoll darzustellen, zwischen Inhalt, Struktur und Darstellung eines Dokuments klar zu unterscheiden. Dieses Konzept der Trennung von Struktur und Präsentation war in der HTML-2.0-Spezifikation sogar noch streng realisiert. Durch die Einführung neuer firmenspezifischer Elemente zur Präsentations-Auszeichnung wurde dieses Prinzip entscheidend aufgeweicht und nicht jedes Dokument konnte auf allen Browsern angemessen dargestellt werden. Mit HTML 3.2, in das zahlreiche proprietäre Elemente aufgenommen wurden, wurde das Problem der Vermischung von Struktur und Präsentation noch verschärft. HTML entsprach damit nicht mehr der ursprünglichen SGML-Philosophie.

SGML hat aber auch Nachteile. Aufgrund der großen Ausdrucksstärke[4] von SGML ist es praktisch unmöglich, die Sprache effizient einzusetzen und erlernen zu können. Lediglich "Global Players" können die nötigen Ressourcen hierfür aufbringen. Um diesen Mistand auszugleichen, ohne die Vorteile von SGML zu verlieren, entwickelte das WORLD WIDE WEB CONSORTIUM (W3C) 1998 eine überschaubare und leicht anwendbare Untermenge von SGML, die EXTENDED MARKUP LANGUAGE (XML)[5] (Abschnitt 2.4, Kapitel 3). Im Grunde ist jedes XML-Dokument (Abschnitt 3.1) auch ein SGML-Dokument, mit dem Vorteil, dass XML mit seiner Einfachheit es jedem gestattet XML-Applikationen zu entwickeln. Als weiterer wesentlicher Nachteil von SGML sei noch erwähnt, dass SGML-Applikationen, durch den hohen Grad ihrer Komplexität, schnell an Größe gewinnen und somit praktisch untauglich für verteilte Netzwerkanwendungen werden.

2.4 XML

Trotz des enormen Schubs für das WORLD WIDE WEB und der breiten Akzeptanz weltweit, kristallisierten sich in HTML, wie in dem vorangegangenem Abschnitt bereits erwähnt, immer mehr Nachteile heraus, die nicht den Prinzipien der generischen Kodierung entsprachen. So verwendete man beispielsweise für sämtliche Zwecke nur noch einen einzigen Dokumenttyp, was teilweise eine starke Überbeanspruchung der TAGs zur Folge hatte. Ein zweiter Nachteil war, dass viele TAGs ausschließlich Formatierungszwecken dienten. Wegen dieser Missachtung der Prinzipien der generischen Kodierung gab es Bemühungen, zu eben dieser generischen Kodierung zurückzukehren, indem man

[4] Die SGML-Standardreferenz umfaßt knapp 600 Seiten
[5] Die XML-Spezifikation umfaßt nur 26 Seiten

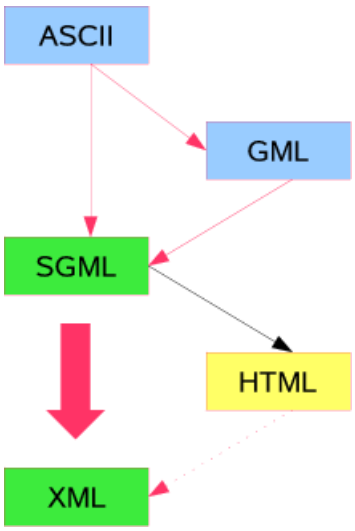


Abbildung 2.6: Entwicklung von XML

versuchte, SGML (Abschnitt 2.3) an das WORLD WIDE WEB anzupassen und sogar umgekehrt das WORLD WIDE WEB an SGML anzupassen. Beide Versuche schlugen fehl, da SGML viel zu komplex war, um beispielsweise in einen kleinen Webbrowser zu passen. Diese Unzufriedenheit mit den existierenden Standards war der erste Ansatzpunkt des W3C Mitte der 90er Jahre für der Entwicklung einer erweiterbaren (Auszeichnungs-) Sprache, die die Flexibilität und Mächtigkeit von SGML sowie die Einfachheit und Akzeptanz von HTML in sich vereinen sollte (Abbildung 2.6). Der zweite Ausgangspunkt zur Entwicklung einer neuen Sprache waren neue Anforderungen an Auszeichnungssprachen und die Grenzen von HTML. Die Notwendigkeit einer neuen, flexibleren und vor allem erweiterbaren Sprache ergibt sich damit aus folgenden drei Punkten:

1. Zunehmende Übertragung von Daten zur Anzeige *und* Weiterverarbeitung.
2. Wunsch nach einer Möglichkeit, Daten problemspezifisch zu strukturieren.
3. In HTML sind problembezogene Auszeichnung von Dokumenten nicht möglich, sondern die TAGs sind vornehmlich an der Anzeige der Dokumentstruktur orientiert.

Die Bemühungen zur Schaffung einer solchen Sprache endeten dann schließlich in XML, der eXTENSIBLE MARKUP LANGUAGE.

Bei der Entwicklung von XML bezog man sich stark auf die SGML-Spezifikation und konzentrierte sich darauf, das funktionierende Konzept von SGML zu vereinfachen. Mit SGML war es ja bereits möglich, eigene Auszeichnungssprachen zu entwickeln, jedoch war diese Sprache, wie bereits erwähnt, viel zu komplex. Die neue Sprache sollte leichter anzuwenden sein als SGML und musste daher einfacher gestaltet werden. Das Erstellen und Gestalten von Dokumenten sollte mit einfachen und verbreiteten Tools möglich sein und auch die maschinelle Verarbeitung und Transformation von Dokumenten sollte

vereinfacht werden, da diese Sprache für die breite Masse und nicht nur für große Unternehmen gedacht war. XML wurde schließlich als Untermenge von SGML spezifiziert und blieb somit aufwärts kompatibel. Unternehmen, die schon länger oder immernoch mit SGML-Umgebungen und Tools arbeiteten, konnten auch neuere XML-Dokumente verarbeiten. 1998 erklärte das W3C XML 1.0 zu einer Empfehlung[6].

Der Sinn und Zweck von XML bestand in der Hauptsache aus den folgenden Punkten [HCD04]:

- Anwendungsspezifische Auszeichnungen
- Eindeutige Strukturen
- Trennung von Inhalt und Präsentation
- Einfachheit
- Bestmögliche Fehlerüberprüfung

Anwendungsspezifische Auszeichnungen XML schreibt keinen festen Satz von Auszeichnungselementen vor, wie etwa HTML. So wie HTML durch SGML definiert ist, so kann man mit XML seine eigene Auszeichnungssprache mit beliebigen TAGs definieren. Man braucht nicht mehr auf Allzweck-TAGs (z.B. für einen Absatz) zurückgreifen und hoffen, dass diese immer und überall passen, sondern man hat es selbst in der Hand, welche TAGs man verwenden möchte und wie diese formatiert werden sollen. Auszeichnungselemente wie `<GegliederteListe>`, `<RechnungsPosition>` oder `<Kapitel>` sind nun kein Problem mehr. Mit XML können Auszeichnungssprachen für spezielle Situationen, z.B. den Datenaustausch zwischen zwei Unternehmen, aber auch für allgemein relevante, internet-weite Anwendungsgebiete festgelegt werden. Diesem Umstand verdankt XML auch das "eXtensible" im Namen.

Eindeutige Strukturen Wenn es um die Struktur von Dokumenten geht, verfolgt XML eine strikte Linie. Dokumente sollten immer so ausgezeichnet werden, dass sie, in Bezug auf Namen, Reihenfolge und Hierarchie der Elemente, immer auf die gleiche Weise interpretiert werden können. Dies reduziert die Komplexität des Codes und die Fehleranfälligkeit erheblich. Programme zur Verarbeitung von XML brauchen nun keine Syntaxfehler mehr zu korrigieren, wie es etwa HTML-Browser tun, sondern geben aussagekräftige Fehlermeldungen aus und ermahnen den Entwickler geradezu, diese strikten Strukturen und Syntaxregeln einzuhalten. Dadurch werden auch Inkonsistenzen verhindert, wenn beispielsweise Dokumente mit unterschiedlichen Programmen interpretiert oder verarbeitet werden. Entweder liefern alle Programme dasselbe Ergebnis oder sie produzieren

[6] Eine W3C Empfehlung ist ein formaler Text des W3C, der die Syntax und Semantik oder die Implementierung einer Web-Technologie beschreibt. Er entspricht einer offiziellen Norm eines Standardisierungsgremiums. Einige dieser Empfehlungen sind allgemein anerkannt und haben quasi den Status einer Norm oder stellen die Grundlage einer solchen dar.

entsprechenden Fehlermeldungen, bis das Ergebnis stimmt. Diese strengen Strukturen erschweren sicherlich das Auszeichnen von Dokumenten, da diese hinterher mit einem Überprüfungsprogramm, einem XML-Parser (Abschnitt 3.7), kontrolliert werden müssen. Es ist jedoch aus obigen Gründen angebracht, diese Strukturen beim Erstellen solcher Dokumente einzuhalten und ein paar Minuten mehr Zeit zu investieren.

Trennung von Inhalt und Präsentation Damit Dokumente flexibel bleiben und für verschiedene Anwendungsgebiete optisch aufbereitet werden können, ist die Trennung von Inhalt und Präsentation einer der Kernpunkte bei XML. Es ist aus vielen Gründen sinnvoll, eine Koppelung zwischen Inhalt und Präsentation, wie es bei HTML oft der Fall ist, aufzubrechen und die Formatierungsanweisungen aus dem Dokument fernzuhalten und in externe Dateien, sogenannte *Stylesheet*s auszugliedern (Unterabschnitt 3.8.2).

- Stylesheets können für unterschiedliche Dokumente wiederverwendet werden.
- Änderungen von Formatierungen können zentral im Stylesheet an einer einzigen Stelle vorgenommen werden.
- Unterschiedliche Stylesheets sind für unterschiedliche Ausgabemedien möglich (Druck, Web, Monitor).
- Der Austausch von Stylesheets für ein Dokument ist problemlos "on the fly" möglich.
- Inhalt und Struktur eines Dokuments bleiben erhalten, egal wie man an die Präsentation verändert.
- Dokumente sind leicht zu lesen und nicht mit Formatierungsvokabular überladen.
- Ohne Formatangaben kann man passendere Namen für Auszeichnungen wählen.

Einfachheit Die Einfachheit von Dokumenten ist für die Anwendungsentwicklung essentiell, da der Öffentlichkeit umso mehr Programme zur Verfügung gestellt werden können, je einfacher es ist, Software zur Be- und Verarbeitung von XML zu entwickeln. Diese Einfachheit ergibt sich aus der Tatsache, dass XML eine schlanke Untermenge von SGML ist und einiges der Komplexität von SGML weggelassen wurde. Durch diese Einfachheit können jedoch nicht nur mehr Programme zur Verfügung gestellt werden, sondern auch mehr Funktionalität innerhalb der Programme. Damit ist gemeint, dass in XML-Werkzeugen, um dem Anwender den größtmöglich Nutzen zu bieten, nicht noch zusätzlich die Sprache eines Cascading Style Sheets (CSS) (Unterabschnitt 3.8.2) oder die einer DTD (Abschnitt 3.4) implementiert werden muss. CSS können nun mehr durch die Sprache eXentsible Stylesheet Language (XSL) (Abschnitt 3.8) und Document Type Definitions durch XMLSchemata (Abschnitt 3.6) abgelöst werden. In beiden Fällen handelt es sich um XML-Applikationen, so dass XML-Werkzeuge solche Dateien wie jede andere XML-Datei einlesen können. Mit einem Werkzeug können somit unterschiedliche Technologien verarbeiten werden.

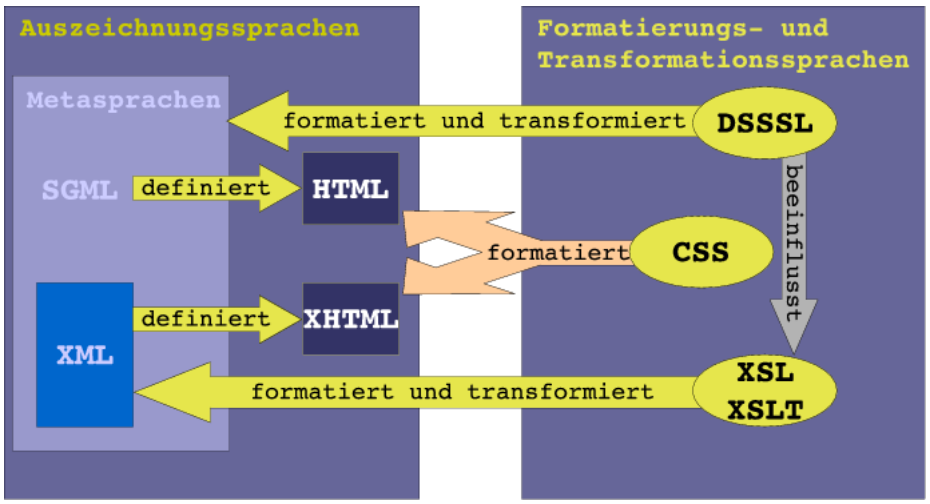


Abbildung 2.7: Einordnung von XML

Bestmögliche Fehlerüberprüfung Auszeichnungssprachen wie HTML gehen recht nachsichtig mit Syntax-Fehlern von Entwicklern um, was zu Fehlern bei der Darstellung der Dokumente, falschem Verhalten von Programmen und Verlust von Informationen führen kann. Solche Fehlerfälle werden beim Überprüfen von XML-Dokumenten durch bereits erwähnte XML-Parser (Abschnitt 3.7) abgefangen. Ein XML-Parser überprüft die *Wohlgeformtheit* (Unterabschnitt 3.1.1) von XML-Dokumenten, die nur gegeben ist, wenn die Dokumente den Auszeichnungsregeln von XML (Abschnitt 3.10) entsprechen. Zusätzlich kann die *Gültigkeit* (Unterabschnitt 3.1.2) bezüglich einer DTD (Abschnitt 3.4) oder eines XMLSchemas (Abschnitt 3.6) überprüft werden. Diese Gültigkeit ist gegeben, wenn das Dokument nicht gegen die in der verwendeten DTD oder dem verwendeten XMLSchema formulierten Regeln verstößt. Man mag anfangs ein wenig über die Strenge von XML verärgert sein, denn sie kostet Entwicklungszeit und sicher auch Nerven, aber die Dauerhaftigkeit und Nützlichkeit von Dokumenten sprechen für sich. XML ist somit für alle Anwendungen geeignet, für die der Einsatz von SGML zu komplex und teuer ist, für die aber HTML zu eingeschrängt ist. In einigen Fällen wird XML SGML vollständig ersetzen. Dies gilt für die klassischen Einsatzgebiete von SGML (Manuals, Kataloge, Database Publishing), als auch vor allem für Internet- und Intranet-Anwendungen. Gerade weil XML primär für Onlineanwendungen entwickelt wurde, besteht endlich die Möglichkeit, die Vorteile von SGML mit geringem Aufwand mit der Anwendung im Netz zu kombinieren. Bisher waren dazu spezielle Erweiterungen der Browser nötig. In Zukunft wird jeder Standardbrowser die XML-Darstellung — in Verbindung mit XSL — beherrschen. Somit ist zu erwarten, dass mehr und mehr Firmen von SGML auf XML umstellen. Allerdings kann XML, da es eine Untermenge von SGML ist, nicht alle Anwendungsbereiche abdecken und so kann eine Transformation alter SGML-Dokumente nach XML unter Umständen mit Problemen verbunden, oder unmöglich sein. Auch wird HTML nicht durch XML abgelöst werden, sondern vielmehr wird HTML statt SGML, in Zukunft XML als Grundlage haben (XHTML) (Abbildung 2.7). Aus der Sicht des Webautors ändert sich

sehr wenig. Ein grundlegender Wandel zeichnet sich ab: (X)HTML wird dann ein Datenformat unter vielen anderen XML-Datenformaten sein. Man wird HTML noch lange Zeit für die Präsentation von Daten verwenden, doch da XML im Gegensatz zu HTML beliebig erweiterbar ist, wird das Internet in Zukunft um viele neue kreative Technologien bereichert werden. Ein konkretes Beispiel dafür gibt es bereits für den Informationsaustausch im Bereich der Mathematik (Kapitel 4). Der Vollständigkeit sei noch auf die ISO-Norm DSSSL (sprich: Dißl) — die Document Style Semantics and Specification Language — hingewiesen. Sie erlaubt die Formulierung von Transformationsregeln (CSS, XSL und XSLT (Unterabschnitt 3.8.2)), die beliebige SGML-Dokumente in ein präsentationsfähiges Format überführen. Hierzu ist ein generischer Prozessor notwendig, der als Eingabe das umzuformende SGML-Dokument und die Transformationsregeln akzeptiert.

Es stellt sich nun die Frage: Wie kann XML bei der Kodierung von Informationen auf dem semantischen Niveau helfen? Tatsächlich ist XML eine Syntax, entwickelt zur Enkodierung von Datenbeständen und somit sehr limitiert was die semantische Modellierung von komplexen Objekten betrifft, wobei hier "Semantik" als Netz von Relationen zwischen Objekten und deren Eigenschaften zu verstehen ist. Kurz gesagt: XML im Allgemeinen ist eine schlechte Sprache für die Modellierung von Daten, mit dem Ziel, Informationen in gewissen Problembereichen ("domains") darzustellen, so dass diese transparent der Betrachtungsweise des aktuellen Benutzers entsprechen. Eine vollständig kompatible Lösung zu diesem Problem ist eine allgemeingültige Menge von semantischen Objekten (Vokabular), mit deren Hilfe es ermöglicht wird, Semantik auszudrücken und zu evaluieren.

Es muss sichergestellt werden, dass Informationsübermittlungen sinnvoll und korrekt sind, in dem Sinne, dass alle Teilhaber das gleiche meinen. Diesem Problem, insbesondere hinsichtlich mathematischem Informationsaustausch, widmet sich die SEMANTIC XML-Applikation OMDOC (Kapitel 4).

2.5 Semantic Xml

Alle Welt redet von Semantik, semantischem XML, semantischem Markup und verbannt visuelles bzw. *prozedurales Markup* in die Steinzeit des World Wide Web. Gerne werden auch die Begriffe *deskriptives* bzw. *generisches Markup* anstelle von *semantischem Markup* verwendet. An dieser Stelle wird der Autor eine Abgrenzung dieser Begrifflichkeiten vornehmen (Abbildung 2.9).

Generisches Markup bezeichnet die Familie der Meta-Markupsprachen. Durch *generisches Markup* wird lediglich eine Basissyntax festgelegt, mit der logische Elemente und Syntax-Regeln (Muster nach denen Elemente zu größeren funktionalen Einheiten zusammengestellt werden können) vereinbart werden. XML ist dafür das prominenteste Beispiele. Man unterscheidet bei XML zwischen *Dokumenttypen*, *Dokumenttyp-Definitionen*

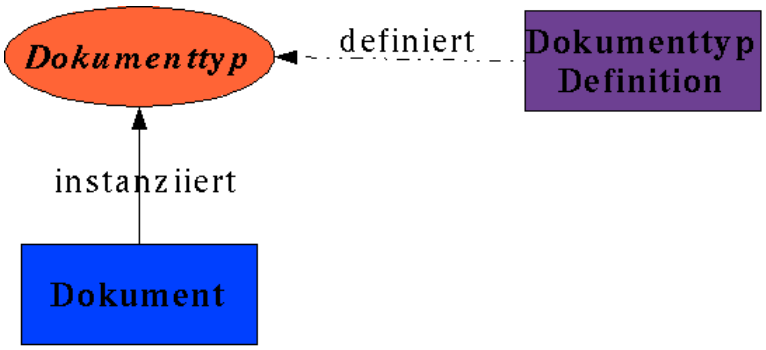


Abbildung 2.8: Dokumenttyp, Dokumenttyp-Definition und Dokumenttyp-Instanzen

und *Instanzen* von Dokumenttypen (Abbildung 2.8). Ein Dokumenttyp wird durch jeweils eine Dokumenttyp-Definition (Dtd) (Abschnitt 3.4) spezifiziert und das Dokument selbst ist dann eine (gültige) Instanz dieses speziellen Dokumenttyps. Im Sinne von "generisch" kann man 1.) die Dokumenttyp-Definition als ein Modell verstehen, das es zuläßt verschiedene weitere spezialisierte Instanzen zu beschreiben und 2.) den Ansatz dieser Modellbildung, mit allgemeingültiger Syntax und Topologie, selbst als generisch auffassen. Instanzen *generischen Markups* können sowohl *prozeduraler* als auch *deskriptiver Markup* sein. Zur Vervollständigung hier noch mal eine kurze Erklärung zu *prozeduralem* und *deskriptiven Markup*. Durch *prozeduralen Markup* kann ein Text zusätzlich mit Formatierungsanweisungen ergänzt werden, die lediglich Auswirkung auf die visuelle Repräsentation des Dokuments haben. Diese Formatierungsanweisungen haben in keiner Weise einen informativen Gehalt über den Inhalt des Textes bzw. über dessen Bedeutung. Primär visuell annotierte Dokumente können später nur mit großen Aufwand in andere Präsentationsformen/Medien überführt werden und sind nur mit großem Aufwand überhaupt maschinenlesbar bzw. maschinenverstehbar. Somit wurden *deskriptive Markups* entwickelt, die ein Dokument ausschließlich mit einer logischen und hierarchischen Struktur versehen und somit den Inhalt von der Präsentation entkoppelt. Um der durch *deskriptiven Markup* entstandenen Struktur eines Dokuments auch eine Bedeutung zu geben, ging man über zu Semantic Xml. Doch was verbirgt sich hinter dem Begriff Semantic? Semantik ist die (linguistische) Lehre der Bedeutung. Man unterscheidet allerdings zwischen Sinn und Bedeutung, wobei sich der Sinn aus den Relationen der sprachlichen Einheiten innerhalb der Sprache ergibt und sich die Bedeutung erst in der Relation zwischen Zeichen und Welt ausdrückt. Mit Hilfe von Markupsprachen können semantische Einheiten in einem Dokument annotiert werden. Wird hierbei z.B. in Html das `<i>` Tag als nicht semantische Einheit verstanden? `<i>...</i>` zeichnet einen String als kursiv aus und fällt laut SelfHtml [Sel] in die Gattung der Html Elemente zur physischen Auszeichnung. Physische Auszeichnung bedeutet eine Annotation, die sich primär auf die Formatierung (prozedural) auswirkt aber nichts über die Bedeutung aussagt. Das `<i>` Tag würde demnach also nicht zum Inventar des Semantic Xml gehören. Des Autors persönliche Meinung jedoch ist, dass diese Abgrenzung etwas unzureichend ist, da ein Autor der einen Text als kursiv auszeichnet sehr wohl damit

etwas über seine Bedeutung ausdrücken möchte und dies auch auf (menschlich) pragmatischer Ebene ohne Probleme verstanden wird. SEMANTIC XML ist somit *deskriptives Markup* inklusive einer Grammatik, die der zusätzlichen Doktrin unterliegt, die Menge der deskriptiven TAGs auf einem wohldefinierten Vokabular zu gründen, so dass die Semantik aus der Verwendung der intuitiven (selbsterklärenden) Begriffe für die einzelnen Vokabeln (z. B. `<important>` statt `<i>`) und deren sinnstiftenden grammatikalischen Anordnungen hervorgeht, wie zum Beispiel:

```
<advice>
 <important>
   I have never met a man so ignorant
   that I couldn't learn something from him.
 </important>
</advice>
```

Zum Abschluss dieses Abschnittes sei nun noch der letzte, bzgl. Abbildung 2.9, ausstehende Begriff *Semantischer Markup* erläutert. *Semantischer Markup* (in der Mathema-

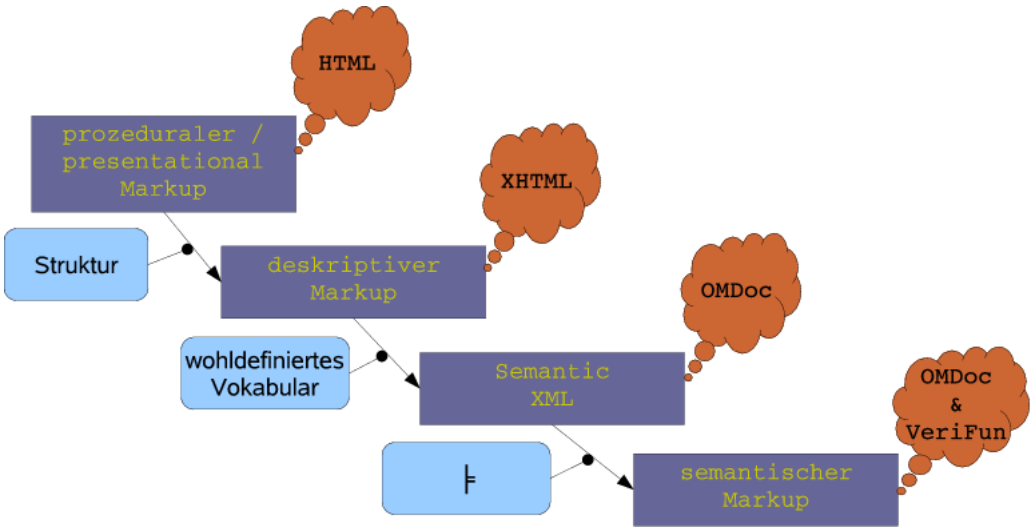


Abbildung 2.9: Abgrenzung der diversen Markups

tik auch bekannt unter der Bezeichnung: *Formales System*) ist ein Repräsentationssystem mit einer unterliegenden Semantik, wie zum Beispiel Systeme in denen spezifiziert wird, wann eine Formel aus einer anderen folgt. Viele solcher Systeme modellieren solch einen Folgerungsbegriff in Form eines Kalküls, der es erlaubt, Beweise zu erstellen und zu überprüfen. Es muss somit eine spezielle Logik zu Grunde liegen, um einen wohldefinierten Folgerungsbegriff modellieren zu können. Allgemeiner formuliert versteht man unter *semantischem Markup* eine SEMANTIC XML-Applikation gepaart mit einer Interpretationsfunktion, die die einzelnen Elemente des syntaktischen Bereichs in eine wohldefinierte,

standardisierte Menge von Begriffen mit Kontextbedingungen[7] überführt. Desweiteren existiert ein Operator auf der Menge der Begriffe, mit dessen Hilfe man aus bestehenden Begriffen neue ableiten kann. Durch die Integration der Semantic Xml-Applikation OMDoc (Kapitel 4) in ✓`eriFun` mit der darin enthaltenen Logik, wird diese Kooperation somit zu einem *semantischen Markup*.

[7]Kontext ist hier im Sinne der mathematischen Logik zu verstehen.

3 Einführung in XML

XML ist eine offene textbasierte Sprache, die sich für die Beschreibung strukturierter Informationen eignet. Wie HTML ist auch XML eine Markierungs- oder Auszeichnungssprache. Während erstere eine Anwendung der Meta-Markupsprache SGML ist, bildet letztere eine Teilmenge davon. Somit ist XML auch eine Meta-Markupsprache zur Instanziierung von neuen XML-Dokumenttypen. In dem Standard-Regelsatz (Abschnitt 3.10) wird die grundlegende Grammatik beschrieben, die von einer XML-Dokumentinstanz einzuhalten ist. Durch die Einfachheit dieser Grammatik eignet sich XML besonders gut für die Erzeugung abgeleiteter Sprachen, die als Sprachanwendungen von XML (XML-Applikationen) für bestimmte, meist spezifische Anwendungsbereiche mit ihren jeweils speziellen Bedürfnissen entwickelt werden. Im Unterschied zu HTML sind in XML nicht die Sprachelemente vordefiniert, sondern lediglich die Regeln für deren Definition. Eine XML-Applikation wird entwickelt, indem zu der Standard XML Grammatik (Abschnitt 3.10) in einer zusätzlichen DTD (Abschnitt 3.4) oder einem XMLSchema (Abschnitt 3.6)) die zulässigen TAGs des neuen Dokumenttyps und deren zulässige hierarchische Schachtelung determiniert werden. Diese zusätzliche DTD definiert somit die Grammatik, nach der ein XML-Dokument dieses neuen Typs aufgebaut werden muss. XML-Dokumente werden daher nicht wie HTML-Dokumente aus einem festen Stamm von Elementen aufgebaut, sondern aus den in einer DTD definierten Elementen und sie enthalten ebenfalls im Unterschied zu HTML keinerlei Information, mit der die Darstellung des Dokuments im Browser beschrieben wird. Gemeinsam ist beiden Sprachen grundsätzlich die Art der Auszeichnung von Inhalten, also die Markierung von Elementen mittels TAGs, sowie das Prinzip der Verschachtelung von Elementen.

In den folgenden Abschnitten wird eine Einführung in XML gegeben, die für das Verständnis von OMDoc (Kapitel 4) und die Integration in ✓eriFun (Kapitel 5) relevant ist. Die gesamte formale Grammatik von XML ist in Abschnitt 3.10 unter Verwendung der Extended Backus-Naur-Form (EBNF) notiert.

3.1 XML-Dokument

Ein Datenobjekt ist ein XML-Dokument, wenn es im Sinne der XML-Spezifikation des W3C wohlgeformt (Unterabschnitt 3.1.1) ist. Ein wohlgeformtes XML-Dokument kann darüber hinaus gültig (Unterabschnitt 3.1.2) sein, sofern es bestimmten weiteren Einschränkungen genügt [XMLa].

Jedes XML-Dokument hat sowohl eine *logische* als auch eine *physikalische* Struktur. Die logische Struktur eines XML-Dokuments ist gekennzeichnet durch die strukturelle Abfolge der Elemente (Abschnitt 3.2) in dem XML-Dokument. Die Informationen werden hierarchisch angeordnet und durch die Elemente werden Informationseinheiten geschaffen. Eine XML-Dokument ist somit beispielsweise vergleichbar mit der Unterteilung eines Buches in Kapitel, und diese in Überschriften, Absätze und Abbildungen. Die logische Struktur eines XML-Dokuments wird durch Deklarationen, Elemente, Kommentare, Zeichen-Referenzen oder/und Prozessor-Instruktionen bestimmt. Zur Veranschaulichung stelle man sich ein XML-Dokument als Baum-Struktur vor, das aus einem Wurzelelement besteht, von dem beliebig viele Zweige (enthalten Unterelemente) oder Blätter (enthalten keine Unterelemente) abgehen, die wiederrum aus Elementen bestehen. Die physikalische Struktur eines XML-Dokuments ist durch Speicherungseinheiten, die sogenannten Entitäten (Entity), gegeben. Dabei kann ein XML-Dokument aus mehreren Entitäten bestehen, die in verschiedenen Dateien auf verschiedenen Servern liegen können. Ein XML-Parser (Abschnitt 3.7) fügt die einzelnen Entitäten zu einer linearen Struktur zusammen. Wenn so zum Beispiel der XML-Parser während der Verarbeitung eines XML-Dokuments auf eine Uniform Ressource Identifier (URI) stößt, die auf Bruchstücke des XML-Dokuments auf einem anderen Servern verweist, so müssen diese geladen und ebenfalls rekursiv verarbeitet werden. Erst wenn jeder Verweis erfolgreich aufgelöst und selbst erfolgreich verarbeitet wurde, ist das Einlesen des XML-Dokuments abgeschlossen. Man unterscheidet hierbei zwischen analysierte (parsed) und nicht-analysierte (unparsed) Entitäten. Der Inhalt einer analysierten Entität wird als deren Ersetzungstext bezeichnet. XML-Dokumente können aber auch multimediale Inhalte einbinden (d.h. nicht-XML-Inhalte). Sie werden über externe Entitäten deklariert. Diese nicht-analysierten Entitäten sind zum Beispiel Grafiken, Videos oder einfacher Text und enthalten kein XML-Markup. Um wohlgeformte XML-Dokumente zu erhalten, müssen logische und physikalische Struktur vollständig und korrekt ineinander verschachtelt sein.

3.1.1 Wohlgeformtes XML-Dokument

Ein XML-Dokument ist wohlgeformt, wenn zum einen dessen Syntax der in Abschnitt 3.10 angegebenen Produktions-Regeln entspricht und zum anderen die logische und physikalische Struktur korrekt ist. Insbesondere gibt es einige markante Unterschiede zu der HTML-Syntax:

- XML unterscheidet zwischen Groß - und Kleinschreibung.
  `<name>` ist nicht gleich `<Name>`.
- Jedes Element muss auch geschlossen werden.
  `<p>` alleine ist nicht erlaubt. Es muss immer heißen: `<p>Text</p>`.
  Leere Elemente können entweder durch einen öffnenden und einen gleich darauffolgenden schließenden TAG (`<br></br>`) oder durch einen abschliessenden "`/`" vor der schliessenden Klammer angegeben werden (`<br/>`).

- Attribute müssen immer einen Wert haben.
  `<td nowrap>` ist nicht erlaubt. Es muss heissen: `<td nowrap="yes">`. Darüber hinaus müssen alle Attributwerte in (doppelten) Anführungszeichen stehen.

- Elemente in einem XML-Dokument müssen immer korrekt verschachtelt sein.
  `<b>fett<i>fett kursiv</b>kursiv</i>` ist illegal. Es muss z.B. heißen:
  `<b>fett<i>fett kursiv</i></b><i>kursiv</i>`.

- Kein Attributname darf mehr als einmal in einem Element verwendet werden.

- Ein Attributwert darf keinen Verweis auf eine externe Entität enthalten.

- Ein Attributwert darf keine öffnenden spitze Klammer `<` enthalten.

Die Wohlgeformtheit eines XML-Dokuments wird mit einem XML-Parser überprüft (Abschnitt 3.7). Neben dieser Überprüfung dienen XML-Parser hauptsächlich dazu, XML-Dokumente einzulesen und zur Weiterverarbeitung für andere Programme vorzubereiten. Einen Überblick über die verfügbaren XML-Parser findet man unter `www.xml.com/pub/rg/XML_Parsers` und `www.xmlsoftware.com/parsers.html`. Die bekannteren sind XP von James Clark, einer der ersten XML-Parser, EXPAT, dessen erste Versionen ebenfalls von James Clark entwickelt wurden, und XERCES aus dem Apache XML Projekt [Apa]. Letzterer ist der in dieser Arbeit verwendete XML-Parser.

3.1.2 Gültiges XML-Dokument

Listing 3.1: XML-Prolog

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE name externer.zeiger [interne.untermenge]>
```

Üblicherweise sollte ein XML-Dokument mit einem XML-Prolog beginnen (Listing 3.1), der aus zwei Deklarationen besteht. In der ersten Deklaration, der sogenannten XML-Deklaration, wird die XML-Version und die verwendete Zeichensatzkodierung angegeben. Als Wert des `version` Attributs ist zur Zeit nur 1.0 erlaubt. Wenn es später eine neue XML-Version geben sollte, können mittels dieses Attributs die einzelnen XML-Versionen unterschieden werden, wodurch die Abwärtskompatibilität gewährleistet bliebe. Die in dem XML-Dokument verwendete Zeichensatzkodierung wird in dem `encoding` Attribut festgelegt. XML erlaubt die Verwendung von UNICODE und verschiedener anderer Zeichensätze. Je nach Zeichensatz können verschiedene Zeichen in dem XML-Dokument verwendet werden. In der zweiten Deklaration eines XML-Prologs, der Dokumenttyp-Deklaration, kann in dem `DOCTYPE` Element die verwendete DTD referenziert werden (Abschnitt 3.4). Zu beachten ist, dass ein XML-Dokument bereits wohlgeformt sein kann, wenn kein XML-Prolog angegeben ist. Der XML-Prolog ist optional. Damit ein wohlgeformtes XML-Dokument jedoch auch das Gütesiegel "gültig" erhalten kann, muss ein

vollständiger XML-Prolog - XML-Deklaration und Dokumenttyp Deklaration - existieren. Denn erst wenn ein wohlgeformtes XML-Dokument auch der durch die referenzierte DTD beschriebenen Struktur genügt, handelt es sich um ein gültiges XML-Dokument.

3.2 Element

Listing 3.2: Syntax von Container-Elementen bzw. leeren Elementen

```
<name attribut1 attribut2 ... >
  content
</name>
<name attribut1 attribut2 ... />
```

Elemente zur Unterteilung eines Dokuments in hierarchisch angeordnete Sektionen stellen die eigentlichen Bauststeine von XML dar. Es wird grundsätzlich zwischen zwei Arten von Elementen unterschieden: *Container-Elemente*, die Text oder weitere Elemente enthalten können und *leere Elemente* als Auszeichnung zum Beispiel spezieller Instruktionen, wie den Import einer graphischen Annimation. In Listing 3.2 ist sowohl die Syntax von Container-Elementen als auch die Syntax von leeren Elementen dargestellt. Ein Container-Element beginnt mit einem Start-TAG, bestehend aus einer öffnenden Klammer `<` gefolgt von dem Elementnamen. In dem Start-TAG kann eine durch Leerzeichen getrennte Liste von Attributen deklariert werden. Das Start-TAG endet mit einer schließenden Klammer `>`. Nach dem Start-TAG folgt der Inhalt (content) des Elements und schließlich das End-TAG. Das End-TAG besteht aus einer öffnenden Klammer `<`, gefolgt von einem `/`, dem Elementnamen und der schliessenden Klammer `>`. Der Elementname des End-TAGs muss gleich dem des entsprechenden Start-TAGs sein. Ein leeres Element wird in analoger Weise aufgebaut. Es beginnt mit einer öffnenden Klammer `<`, dem Elementnamen und kann auch eine durch Leerzeichen getrennte Liste von Attributen enthalten. Wie der Name bereits suggeriert, enthält eine leeres Element keinen Inhalt (content), so dass keine Notwendigkeit für ein entsprechendes End-TAG besteht. Das Start-TAG eines leeren Elements wird folglich direkt mit `/>` geschlossen. In der durch Leerzeichen getrennten Liste von Attributen können Eigenschaften des jeweiligen Elements deklariert werden. Ein Attribut assoziiert dabei einen Namen mit einem Wert. Der Aufbau eines Attributes ist in Listing 3.3 dargestellt. Dort ist zu sehen, dass der Attributname von dem Attributwert mit einem Gleichheitszeichen (`=`) getrennt und der Wert des Attributes in doppelten Anführungszeichen eingeschlossen wird.

Listing 3.3: Syntax von Attributen

```
name = "value"
```

In der Deklaration von Elementen und Attribute ist zu beachten, dass die Namen den in Abschnitt 3.10 angegebenen Produktions-Regeln entsprechen müssen [HM02] [HCD04].

3.3 Zeichen, Namen und Zeichendaten

Der von XML verwendete Zeichensatz ist `ISO/IEC 10646`. Beherrschen die eingesetzen Programme diesen Zeichensatz, gibt es bezüglich der erlaubten Zeichen keine Probleme mehr. Insbesondere die westlichen Sprachen sind abgedeckt, was für deutsche Texte heißt, dass man alle Sonderzeichen direkt eingeben kann.

Es ist zwar schön, dass man alle Zeichen verwenden kann, die sich auf der Tastatur befinden, aber es gibt noch Zeichen, die auf einer gewöhnlichen Tastatur nicht vorhanden sind. Diese können in Form sogenannter *Zeichenreferenzen* geschrieben werden. Dahinter verbirgt sich einfach die Dezimalzahl des Zeichens im Zeichensatz. Diese Nummer wird zwischen `&#` und `;` eingeschlossen.

Ein weiterer Grund, Zeichen nicht direkt einzugeben, kann darin bestehen, dass diese Zeichen in XML eine besondere Bedeutung besitzen. Dies betrifft die spitze Klammer (`<`), das kaufmännische Und (`&`), Apostroph (`'`) sowie die Anführungszeichen (`"`). Alle genannten Zeichen sind Teil des Markups und müssen kodiert werden. Neben der genannten Zeichenreferenz gibt es auch noch Entity-Referenzen (Unterabschnitt 3.4.3). Diese sind eine Art *Abkürzungen* für beliebige andere Texte. Entity-Referenzen besitzen einen Namen, der zwischen `&` und `;` eingeschlossen wird. Die folgenden Zeilen zeigen die Entitäten, die immer zur Verfügung stehen:

$$
\begin{array}{rcl}
\& & \& & (et - Zeichen) \\
\< & < & (less\ than) \\
\> & > & (greater\ than) \\
\' & ' & (apostrophe) \\
\" & " & (quotation\ mark)
\end{array}
$$

Definitionen für eigene Entity-Referenzen können sowohl in einer DTD als auch in der Dokumenttyp-Deklaration untergebracht werden und genauso verwenden wie die vordefinierten. Falls man Text eingeben möchte, der wörtlich übernommen werden soll, also ohne Ersetzung von Entity-Referenzen oder ähnlichem, so steht dafür der `CDATA`-Abschnitt zur Verfügung (siehe hierzu auch Unterabschnitt 3.4.1). Als Zeichendaten sind hier alle Zeichen, also auch spitze Klammern usw. erlaubt, abgesehen natürlich von der abschließenden Kombination `]]>` [HM02].

3.4 DTD-Document Type Definition

Wenn man nicht nur mit einem (wohlgeformten) XML-Dokument arbeiten will, sondern mehrere ähnliche Dokumente hat wie beispielsweise mehrere mathematische Texte, dann ist es sinnvoll, einen eigenen XML-Dokumenttyp für diese Art von Dokumenten zu definieren. Für diesen Zweck wurde die Document Type Definition (DTD) entwickelt.

In Unterabschnitt 3.1.2 wurde bereits die Dokumenttyp-Deklaration angesprochen. Sie informiert den verwendeten XML-Parser (Abschnitt 3.7) darüber, welche Dokumenttyp-Definition für die einzulesende Instanz gültig ist.

Listing 3.4: Allgemeine XML Dokumenttyp-Deklaration

```
<!DOCTYPE name externer.zeiger [interne.untermenge]>
```

Innerhalb des `DOCTYPE` Elements wird als erstes der Name des Wurzelelements angegeben (Listing 3.4). In dem optionalen `externer.zeiger` Eintrag, der sogenannten *externen Untermenge*, kann eine Dokumenttyp-Definition in Form eines Dateipfades in dem System oder in Form einer URL zu einer Datei im Internet deklariert werden. Der letzte in Klammern eingeschlossene Eintrag ist ebenfalls optional. Dort können weitere Deklarationen von Entitäten vorgenommen werden. Diese sogenannte *interne Untermenge*, bildet zusammen mit der externen Untermenge die gesamte für den XML-Parser zur Validierung notwendige Menge aller Deklarationen.

Listing 3.5: Konkrete XML Dokumenttyp-Deklaration

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE omdoc SYSTEM "omdoc-vf.dtd">
```

Zur Veranschaulichung wird in Listing 3.5 die konkrete Dokumenttyp Deklaration, wie sie in dieser Arbeit verwendet wird, dargestellt. Das Wurzelelement des XML-Dokuments wird hier mit `omdoc` angegeben und gegen die externe DTD `omdoc-vf` validiert. Mit dem `SYSTEM` Eintrag wird spezifiziert, dass es sich um einen Dateipfad in dem System handelt[1]. Die interne Untermenge dieser Dokumenttyp-Definition enthält keine weiteren Einträge. Eine `omdoc` Dokument-Instanz darf nun alle in der DTD spezifizierten Elemente, Attribute, Entitäten und Zeichenfolgen unter Beachtung der dort festgelegten grammatikalischen Regeln verwenden.

Um ein besseres Verständnis über DTDs zu erlangen, wird die Syntax einer DTD im Folgenden näher beleuchtet [GP00] [HCD04] [MBK00]. In der Deklaration von Elementen (Unterabschnitt 3.4.1), Attributen (Unterabschnitt 3.4.2) und Entitäten (Unterabschnitt 3.4.3) wird sofort auffallen, dass die Syntax einer DTD offensichtlich keine XML-Syntax ist. Dieser Sachverhalt ist auch einer der großen Nachteile einer DTD. Denn XML-Werkzeuge, die auch DTDs verarbeiten sollen, müssen um die Möglichkeit, die Syntax einer DTD lesen zu können, erweitert werden. Der Vorteil einer DTD liegt somit einzig und allein in der Möglichkeit eine Grammatik für XML-Dokumente festlegen zu können. Zur Erinnerung: Wenn einem XML-Dokument eine DTD zugeordnet ist, kann nicht nur die Wohlgeformtheit überprüft werden, sondern auch ob das Dokument gültig im Sinne der DTD ist. Ein gültiges XML-Dokument entspricht dem Dokumenttyp, der in der DTD definiert ist. Man sagt dann auch: Das XML-Dokument ist eine gültige Dokumenttyp-Instanz, oder kurz Instanz, der DTD. XML-Parser, die auch diese Gültigkeit überprüfen, werden "Validating XML-Parser" genannt.

[1] Eine `URL` wird mit dem Schlüsselwort `PUBLIC` gekennzeichnet.

3.4.1 Elementdeklaration

Listing 3.6: Syntax der Elementdeklaration

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

Die Deklaration in Listing 3.6 sagt aus, dass ein Element mit dem Namen `autor` als Inhaltsmodell Daten des Typs `PCDATA`[2] enthalten kann. Das Zeichen `#` zeigt lediglich an, dass es sich bei `PCDATA` um ein vordefiniertes Schlüsselwort handelt. Weiter zur Verfügung stehende Datenypen sind `EMPTY`, `ANY` und `CDATA`. Durch `EMPTY` wird ein leeres Element deklariert. Der Datentyp `ANY` gibt an, dass das Element beliebig viele Kind-Elemente jeglicher Art in einer nicht weiter spezifizierten Reihenfolge und Anzahl beinhalten kann. `CDATA` bedeutet, dass der Inhalt des Elements nicht durch den XML-Parser überprüft werden soll. Der Inhalt eines Elements mit dem Typ `CDATA` sieht wie folgt aus: `<![CDATA[ Inhalt ]]>` Innerhalb eines solchen `CDATA`-Abschnitts können Markup-Zeichen (`<`, `>` und `&`) enthalten sein. Diese werden von dem XML-Parser ignoriert. Allerdings darf die Zeichenfolge `]]>` innerhalb von `Inhalt` nicht erscheinen, denn durch diese wird das Ende des `CDATA`-Abschnitts gekennzeichnet.

3.4.1.1 Strukturmodellierung

XML hätte nicht die Möglichkeit einer direkten Strukturmodellierung, wäre nicht die Organisation der Reihenfolge von Elementen durch eine DTD gegeben. Im Prinzip legt das Inhaltsmodell fest, in welcher Weise in einem Container-Element andere Elemente zueinander in Beziehung gesetzt werden und welchen Status die einzelnen Elemente besitzen. Hierzu gibt es, angelehnt an Reguläre Ausdrücke (engl.: regular expressions), drei *Operatoren*:

- `A` — `A` muss genau einmal auftreten (obligatorisch).
- `A?` — `A` kann einmal auftreten oder aber auch ausgelassen werden (fakultativ).
- `A+` — `A` muss mindestens einmal, kann aber beliebig oft auftreten.
- `A*` — `A` kann einmal oder beliebig oft auftreten, kann aber auch ausgelassen werden.

Als Beispiel werden nun in Listing 3.7 vier Elemente (`titel, stadt, verlag, jahr`) definiert, die jedoch keinerlei Strukturdefinition beinhalten. Diese zusätzliche Information wird erst durch das Container-Elemente `buch` in Kombination mit den drei oben erwähnten Operatoren realisiert.

[2] `PCDATA` steht für "parsed-character data" und bedeutet, dass der Inhalt dieser Datenelemente kein weiteres Element sondern ausschließlich Text sein darf. Dies wird von Validating XML-Parsern überprüft.

Listing 3.7: Elementdeklaration und Strukturmodellierung mit Operatoren

```
<!ELEMENT titel (#PCDATA)>
<!ELEMENT stadt (#PCDATA)>
<!ELEMENT verlag (#PCDATA)>
<!ELEMENT jahr (#PCDATA)>
<!ELEMENT buch (autor+, titel , verlag?, jahr?)>
```

Der Typ des Container-Elementes `buch` spezifiziert, dass mindestens ein Autoren für ein Buch angegeben werden muss, der Titel obligatorisch und die Verlags- und Jahresangabe nicht zwingend ist, also auch ausgelassen werden darf.

Für Beziehungen unter Elementen innerhalb eines Container-Elementen stehen zwei *Konnektoren* zur Verfügung:

`A,B` `B` folgt auf `A`

`A|B` `A` oder `B`

Die zusätzliche Verwendung dieser Konnektoren ermöglicht es, komplexere Strukturen erzeugen zu können, wie zum Beispiel in Listing 3.8. Diese Deklaration bedeutet, dass das Container-Element `buch`, um der DTD zu genügen, seine Daten-Elemente (oder: Kind-Elemente) folgendermaßen anordnen muss: Zuerst kommen mehrere oder nur ein `autor`, dann der obligatorische `titel`. Die fakultativen Elemente `verlag` und `jahr` können in beliebiger Reihenfolge auftreten, also entweder `jahr` auf `verlag` oder `verlag` auf `jahr`.

Listing 3.8: Elementdeklaration und Strukturmodellierung mit Konnektoren

```
<!ELEMENT titel (#PCDATA)>
<!ELEMENT stadt (#PCDATA)>
<!ELEMENT verlag (#PCDATA)>
<!ELEMENT jahr (#PCDATA)>
<!ELEMENT buch (autor+, titel , (verlag? , jahr?)
                | (jahr?, verlag?))>
```

3.4.2 Attributdeklaration

Attributdeklarationen bestimmen, welches Element welche Attribute haben kann, welchen Datentyp diese Attribute besitzen, ob bzw. welche Vorgabewerte[3], falls das Attribut weggelassen wird, eingesetzt werden sollen und ob die Angabe eines Attributwertes obligatorisch, fakultativ oder statisch sein soll. Insgesamt sind drei Klassen von Attributtypen möglich:

[3] Ein Beispiel für einen vorgegebenen (default) Attributewert ist:
`<!ATTLIST assertion type (theorem|lemma|corollary|conjecture) "conjecture">`

1. `String`-Attribute,
   die aus beliebig vielen Zeichendaten bestehen. Ein String-Attribut könnte folgendermaßen deklariert werden: `<!ATTLIST buch isbn CDATA>`. Die Werte von String-Attributen sind Zeichenketten, wobei jedes Attribut, das in einem XML-Dokument ohne DTD vorkommt, automatisch als ein String-Attribut behandelt wird.

2. `Token`-Attribute,
   deren Wert aus ein oder mehreren für XML relevanten Tokens bestehen. Token-Attribute sind sehr mächtig, deshalb folgt eine übersichtlich Aufstellung aller ihr enthalten Datentypen:

   - `ID`
     Dieses Attribut dient als Identifikator für ein Element. Ein `ID`-Wert muss den Standardnamensregeln von XML entsprechen und eindeutig innerhalb eines Dokuments sein. Weiter ist die Angabe des Status mit `REQUIRED`, `IMPLIED` oder `FIXED` vorgeschrieben. Das Schlüsselwort `IMPLIED` bedeutet, dass der XML-Parser das Fehlen eines Attributwertes ignoriert und der jeweiligen Applikation das weitere Vorgehen überläßt. Hingegen ist bei `REQUIRED` die Angabe des Attributs verpflichtend, der XML-Parser würde das Fehlen also nicht einfach übergehen, sondern eine Fehlermeldung ausgeben. `FIXED` bedeutet, dass das Attribut immer einen festen Standardwert hat.

   - `IDREF`
     Das `IDREF`-Attribut ist ein Zeiger auf ein `ID`-Attribut eines anderen Elements. Werte vom Typ `IDREF` müssen als Wert den identisch geschriebenen Namen erhalten, der in einem anderen Element, auf das mit dem `IDREF`-Attribut Bezug genommen wird, mit dem Attribut `ID` vergeben wurde. Auch der Wert dieses Attributs muss den Standardnamensregeln entsprechen.

   - `IDREFS`
     Der Wert dieses Attributs besteht aus mehreren Referenzen auf `ID`-Attribut, die durch Leerzeichen voneinander getrennt sein müssen.

   - `ENTITY`
     Das `ENTITY`-Attribut ist ein Zeiger auf eine externe Entität, welche in einer DTD-Untermenge deklariert wurde.

   - `ENTITIES`
     Der Wert dieses Attributs besteht aus einem oder mehreren `ENTITY`-Attributwerten, die wieder durch Leerzeichen voneinander getrennt werden.

   - `NMTOKEN`
     Der Wert dieses Attributs ist ein sogenannter *Name-Token-String*, der aus einer beliebigen Kombination von Namenszeichen besteht.

- `NMTOKENS`
  Dieser Wert besteht aus einem oder mehreren durch Leerzeichen getrennten `NMTOKEN`-Attributwerten.

3. `Aufzählungs`-Attribute,
   deren Wert einer aus einer deklarierten Liste stammen muss (Listing 3.10). Die Werte müssen jeweils einen gültigen Namen-Token (`NMTOKEN`) darstellen. Auch hier können Vorgabewerte bestimmt werden.

Zur Veranschaulichung einer Attributdeklaration ist in Listing 3.9 die allgemeine Form angegeben.

Listing 3.9: Allgemeine Form einer Attributdeklaration

```
<!ATTLIST element.name (attribut.definition)>
```

Konkret könnte man somit beispielsweise das Geschlecht einer Person mittels der in Listing 3.10 dargestellten Attributdeklaration modellieren.

Listing 3.10: Beispiel einer konkreten Attributedeklaration

```
<!ELEMENT person (vorname+, nachname)>
<!ATTLIST person geschlecht (m | f) #REQUIRED>
```

3.4.3 Entity-Deklaration

Eine Entity ist eine Abkürzung für eine Zeichenkette oder ein externes Dokument, das innerhalb der DTD oder des XML-Dokuments, das diese DTD benutzt, verwendet werden kann. Eine Entity-Referenz der Form `&Name;` wird dabei durch den Inhalt der Entity ersetzt. Entities werden prinzipiell zur flexibleren Dokumentorganisation verwendet und können von einfachen Abkürzungen innerhalb eines Dokuments bis zur Einbindung ganzer XML-Dokumente in einem "XML-Haupt-Dokument" verwendet werden. Die einfachste Art von Entities sind interne Entities. Diese bestehen aus Zeichenketten, die auch selbst wieder Entity-Referenzen und wohlgeformtes XML enthalten dürfen. Externe Entities bestehen aus dem Inhalt einer Datei, auf die verwiesen wird. In Listing 3.11 sind beispielsweise zwei interne Entities (`nrm, web`) und ein externes Entity (`thesis`) angegeben.

Listing 3.11: Entity-Deklaration

```
<!ENTITY nrm "Normen R. M&uuml;ller ">
<!ENTITY web
  "<a href='http://kwarc.eecs.iu-bremen.de/nmueller/index.html'>&nrm;</a>">
<!ENTITY thesis SYSTEM "~/mykwarc/doc/thesis/diploma/dt.xml">
```

Möchte man diese Abkürzungen innerhalb eines Dokuments verwenden, setzt man an der gewünschten Stelle den einfach den Namen der Entity ein (Listing 3.12).

Listing 3.12: Anwendung von Entities

```
<?xml version="1.0"?>
<!DOCTYPE html SYSTEM xhtml1-strict.dtd"
 [
   <!ENTITY nrm "Normen R. M&uuml;ller ">
   <!ENTITY web
     "<a href='http://kwarc.eecs.iu-bremen.de/nmueller/index.html'>&nrm;</a>">
 ]>
<html>
 <head><title>simple document</title></head>
 <body>
   Auf der Homepage von &web;,dem Autor dieses Dokuments, sind
   weitere Informationen &uuml;ber dieser Arbeit zu finden.
 </body>
</html>
```

3.5 Namensräume

Xml-Namensräume wurden von dem W3c-Konsortium entwickelt, um die Verwendung mehrerer Markupsprachen (markup vocabulary) in einem Dokument zu ermöglichen. Der Anlass für diese Entwicklung war die Annahme, dass sich, insbesondere indem E-Commerce-Bereich, eine Reihe von Standard-Dtds etablieren werden, inklusive den zugehörigen Applikationen, die die Dokumente "ihrer" Dtd erkennen und verarbeiten. Für den Fall, dass nun verschiedene solcher standardisierter Dtds gleichnamige Elemente definieren, die möglicherweise sogar eine völlig andere Bedeutung haben, besteht das Problem, dass es zu Kollisionen kommt, wenn man diese gleichnamigen Elemente in einem Dokument verwenden möchte. Um diese Kollisionen zu vermeiden und die Verwendung von gleichnamigen Elementen aus verschiedenen Dtds in einem Dokument zu ermöglichen, wurden die Xml-Namensräume (name spaces) definiert. Damit wurde gleichzeitig auch die langfristige Wiederverwendbarkeit von Dtds und den zugehörigen Applikationen gewährleistet, weil die Applikationen anhand der Namensräume die für sie bestimmten Elemente identifizieren können. Der Leser mag sich nun fragen, wie es überhaupt zu Namenskollisionen kommen kann, weil ein Dokument ja höchstens eine Dtd haben kann. Tatsächlich ist das W3c bei der Entwicklung der Namensräume gegenüber der Xml-Spezifikation ein ganzes Stück vorgeprescht. Aus heutiger Sicht, mit dem aktuellen Xml Standard Version 1.0, sind Namensräume nur für wohlgeformte Dokumente relevant, die über keine Dtd-Deklaration verfügen. In Dokumenten, die eine Dtd haben, können in Xml 1.0 keine Namenskonflikte auftreten. Möglicherweise wird der Xml-Standard einmal so erweitert werden, dass ein Dokument mehrere Dtds gleichzeitig verwenden kann, wobei hier noch unklar ist, wie dann die Inhalts-Modelle der Elemente definiert werden sollen.

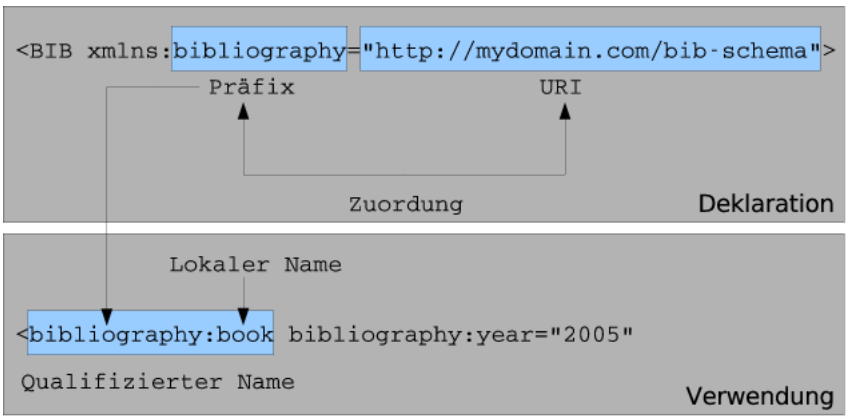


Abbildung 3.13: Deklaration / Verwendung von Namensräumen.

Die Spezifikation für XML-Namensräume sieht nun vor, dass man in einem XML-Dokument einem Element- oder Attributnamen einen Bezeichner für einen Namensraum voranstellen kann. Innerhalb eines Namensraums ist ein Elementname stets eindeutig, so dass auch beide Teile - Namensraum und Elementname - zusammen immer eindeutig im gesamten Dokument sind. Abbildung 3.13 veranschaulicht, wie ein Namensraum deklariert und danach verwendet werden kann [Nam][HCD04]. Ein in einem Namensraum benutzter Elementname `book` wird im Kontext seines Namensraums auch lokaler Elementname genannt. Lokal deshalb, weil er nur innerhalb seines Namensraums eindeutig ist. Präfix und lokaler Name bilden gemeinsam den qualifizierten Namen, der stets eindeutig im gesamten Dokument ist.

3.5.1 Definition von Namensräumen

Zunächst muss in dem Start-TAG eines Elements der Namensraum definiert werden. Listing 3.14 zeigt hierfür den allgemeinen Aufbau. Die Definition wird als Attribut des Elements `MyElem` angegeben, wobei `xmlns` der reservierter Attributname hierfür ist. Dieser Zeichenkette folgt eine Zuordnung von einem Präfix zu einer URI. Die URI bezeichnet den Namensraum.

Listing 3.14: Definition von Namensräumen

```
1 <MyElem xmlns:prefix=URI>
```

Als Wert einer solchen URI wird gerne der Ort, an dem die verwendete DTD gespeichert ist, angegeben. Es sei jedoch darauf hingewiesen, dass es nicht erforderlich ist, dass unter dieser URI auch tatsächlich die DTD-Datei abgelegt ist. Namensräume dienen lediglich zur symbolischen Unterscheidung der Elemente innerhalb eines XML-Dokuments und haben sonst keine weitere Bedeutung. Das Präfix fungiert als Alias für die URI und kann Element- oder Attributnamen vorangestellt werden, um deren Namensraum anzugeben.

In einem Start-Tag können auch mehrere Namensräume definiert werden (Listing 3.15), wobei zwei Namensräume als gleich angesehen werden, wenn die URIs übereinstimmt, unabhängig von dem jeweiligen Präfix. Das Präfix dient somit nur als Platzhalter oder Kurzschreibweise für die URI.

Listing 3.15: Definition von mehreren Namensräumen

```
<MyElem xmlns:nsp1="http://kwarc.eecs.iu-bremen.de/"
        xmlns:nsp2="http://kwarc.eecs.iu-bremen.de/nmueller/index.html">
```

Abgesehen von folgender Einschränkung kann das Präfix beliebig gewählt werden: `xml` als Präfix ist reserviert und dient dazu, den Inhalt des Elements dem Default-Namensraum zuzuordnen. Ist einem Element nicht explizit ein Namensraum zugeordnet, so unterliegt der Inhalt dem Default-Namensraum, der mit folgender URL assoziiert ist: `http://www.w3.org/XML/1998/namespace`.

3.5.2 Verwendung von Namensräumen

Nachdem ein Namensraum in einem Start-Tag mit einem Präfix definiert wurde, kann er in dem gesamten Inhalt des Elements benutzt werden. Dieser Namensraum gilt auch für den gesamten in das Element eingebetteten Inhalt, sofern die eingebetteten Elemente und Attribute nicht explizit einen anderen Namensraum angeben. Insofern wäre es in dem Beispiel aus Listing 3.16 nicht unbedingt notwendig, bei den Bibliographie-Einträgen jedesmal mit dem vorangestellten Präfix `bibliography` den Namensraum explizit anzugeben.

Listing 3.16: Verwendung von Namensräumen

```
<!-- Start-Tag mit Definition des Namensraums -->
<BIB xmlns:bibliography="http://mydomain.com">
  <!-- Start-Tag mit qualifiziertem Element- und Attributnamen -->
  <bibliography:BOOK bibliography:YEAR="2005" >
    ...
  <!-- End-Tag mit qualifiziertem Elementnamen -->
  </bibliography:BOOK>
</BIB>
```

Diese Darstellungsweise wurde zur Veranschaulichung gewählt, damit ein qualifizierter Name sowohl für Elemente als auch für Attribute angegeben werden kann (Zeile 4). Notwendig jedoch ist, dass das Präfix, wie in der vorletzten Zeile, auch in dem entsprechenden End-Tag wieder angegeben werden muss.

3.6 XMLSchema

In der Beschreibung von DTDs (Abschnitt 3.4) wurde schon darauf hingewiesen, dass diese einige Nachteile aufweisen: Sie haben eine eigene Syntax, und es können keine gleichen Elementnamen für zwei verschiedene Elemente verwendet werden, weil alle Elemente global definiert werden. Desweiteren unterstützen DTDs keine Namensräume (Abschnitt 3.5) und sie erlauben nur sehr rudimentäre Angaben über den Datentyp eines Elements oder Attributs. Aus diesem Grund wurden vom W3C die XMLSchemas entwickelt [XMLb]. Die wichtigsten Vorteile von XMLSchemata sind:

- XMLSchemas sind selbst XML-Dokumente[4].
  Man muss keine grundsätzlich neue Syntax lernen. XML-Werkzeuge können Schemas wie andere XML-Dokumente einlesen und verarbeiten.
- XMLSchemas unterstützen Namensräume.
  Zusätzlich erlauben sie auch die Definition von lokalen Elementen. Solch ein Element kann somit auch denselben Namen haben wie ein anderes lokales Element.
- XMLSchemas unterstützen 44 verschiedene Datentypen.
  Neben den vordefinierten Datentypen für Zahlenformate, `URL`s, Strings, Datum, etc., können zusätzlich eigene Datentypen erstellt werden. Beispielsweise Zahlen von 1 - 31 für Monatstage. Außerdem werden bei der Definition von Datentypen reguläre Ausdrücke unterstützt. Damit kann das Format von Strings oder Zahlen in fast jeder beliebigen Form definiert werden.
- XMLSchemas sind in gewisser Weise objektorientiert.
  Wie bereits in dem vorherigen Punkt erwähnt können eigene Datenypen definiert werden. Von solchen Typen können weitere Typen definieren, die entweder eine Erweiterung oder eine Einschränkung zu den ursprünglichen Typen darstellen.

Durch dieses Ausdrucksstärke von XMLSchemata, kann man somit viel restriktiver und viel detaillierter den Aufbau eines XML-Dokuments bestimmen. Die Folge jedoch ist, dass neben grundsätzlichen Entscheidungen, ob beispielsweise Daten als Kind-Elemente oder als Attribute definiert werden sollen, es noch eine Menge weiterer Entscheidungen zu treffen gibt, besonders auch im Zusammenhang mit Namensräumen. Nicht umsonst ist somit die Liste der Internet-Foren kaum zu überblicken, in denen die wichtigsten Entwurfsentscheidungsmöglichkeiten gegenübergestellt und die Vor- und Nachteile diskutiert werden.

Es bleibt anzumerken: Da XML-Parser zum Zeitpunkt der Entwicklung dieser Arbeit XMLSchemata noch nicht zufriedenstellend unterstützten, wurde zur Überprüfung der

[4] Da das XMLSchema selbst ein XML-Format ist, kann man auch definieren, wie dieses XML-Format aussieht. Das heißt, es gibt ein Schema für Schemas. Anhand dieses Schemas kann man alle Schema-Dokumente auf Gültigkeit überprüfen. Dieses Schema für Schemas ist unter der Namensraum-URI `http://www.w3.org/2001/XMLSchema` definiert. In diesem Fall ist diese URI auch eine gültige URL unter der man einen Link zum Quelldokument des Schemas für Schemas findet.

Gültigkeit der XML-Dokumente DTDs eingesetzt. Es ist aus den in diesem Abschnitt aufgeführten Gründen zu überlegen, in der nächsten Version die DTDs durch entsprechende XMLSchemata abzulösen. Zum einen ist die Entwicklung der XML-Parser enorm vorangeschritten, so dass nun ohne weiteres XMLSchemata verarbeitet werden können und zum anderen wurden die angesprochenen Probleme bezüglich den diversen Entscheidung in OMDOC bereits gelöst.

3.7 APIs für XML-Parser

Wenn eine Applikation XML-Dateien verarbeiten soll, greift man auf XML-Parser zurück. Diese lesen die XML-Dateien ein, überprüfen diese auf Wohlgeformtheit, eventuell auch auf Gültigkeit und geben die eingelesen Daten über eine definierte Schnittstelle an die Applikation weiter. In den folgenden Abschnitten werden verschiedenen Konzepte von XML-Parsers zur Verarbeitung von XML-Dokumenten kurz vorgestellt.

3.7.1 Document Object Model

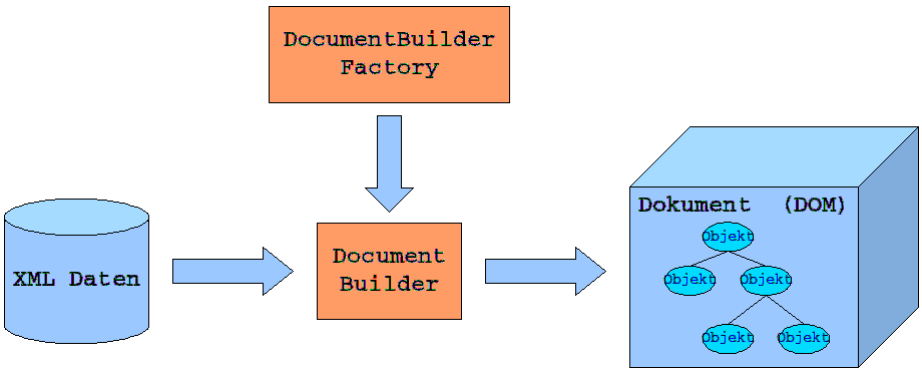


Abbildung 3.17: Das Document Object Model

Die erste Möglichkeit baut auf das DOCUMENT OBJECT MODEL DOM auf (Abbildung 3.17). Hierbei wird eine Baumstruktur aller Elemente, Attribute, Kommentare und der weiteren Inhalte des XML-Dokuments im Speicher erstellt. Die einzelnen Teile des DOM-Baumes werden Knoten genannt. Da der DOM-Baum im Speicher aufgebaut wird, kann auf die einzelnen Knoten wahlfrei und schnell zugegriffen werden. Dies ist der große Vorteil von DOM. Insbesondere, wenn oft auf verschiedene Teile des XML-Dokuments und damit auf verschiedene Knoten des DOM-Baumes zugegriffen werden muss, ist die Verwendung von DOM die erste Wahl. Allerdings gibt es Probleme mit großen Dokumenten. Diese würden sehr viel Speicher verbrauchen, wodurch sich für solche XML-Dokumente die Verwendung von DOM nicht besonders eignet.

3.7.2 Simple API for XML

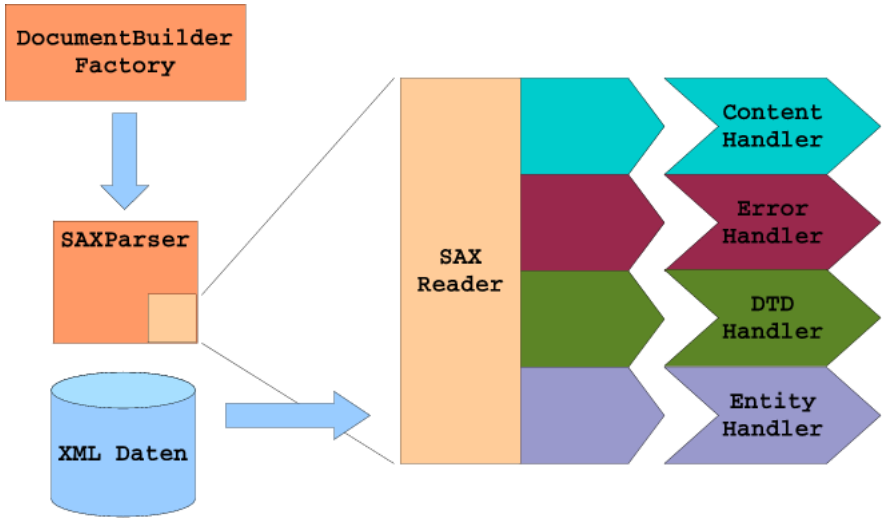


Abbildung 3.18: Die Simple API for XML

Simple API for XML (Sax) ist eine ereignisorientierte Schnittstelle für XML-Parser (Abbildung 3.18). Wenn ein Sax XML-Parser ein XML-Dokument einliest, werden durch die einzelnen Komponenten (`Content Handler`, `DTD Handler` und `Enity Handler`) "Ereignisse" an das Programm gemeldet. So ein Ereignis ist beispielsweise der Dokumentstart, das Auftreten eines bestimmtes Elements oder Attributs, usw. Das Programm, dass das XML-Dokument mittels Sax- basierten XML-Parser einliest, erhält also Ereignisse, die den Aufbau des XML-Dokuments wiedergeben. Das Programm kann die Ereignisse, die es benötigt, weiterverarbeiten, andere kann es ignorieren. Der Vorteil ist klar: Mit Sax kann man beliebig große XML-Dokumente einlesen, da immer nur ein kleiner Teil des Dokuments im Speicher gehalten wird. Allerdings kann mit Sax nicht wahlfrei auf die einzelnen Teile eines XML-Dokuments zugegriffen werden.

Interessant ist, dass Sax nicht von dem W3c entwickelt wurde. Es entstand vielmehr aus der Zusammenarbeit von Entwicklern von XML-Parsern und -Anwendungen, die mit den unterschiedlichen Schnittstellen der XML-Parser zu kämpfen hatten. Die offizielle Webseite von Sax findet man unter `www.saxproject.org`.

3.7.3 Java API for XML Procession

Java API for XML Processing (Jaxp) ist kein eigener Standard wie Sax oder Dom, sondern ist eher eine "Hilfs-API". Sie ermöglicht es, Dom oder Sax mit jedem beliebigen XML-Parsern zu nutzen. Ein weiterer Vorteil ist es, dass man bei einem Wechsel des XML-Parsers den Quellcode nicht ändern muss.

3.7.4 JDOM

Diese API ist die derzeit neueste API für die Bearbeitung von Xml-Dokumenten mit Java. Jdom kann, analog zu Jaxp, mit Dom oder Sax Xml-Parsern verwendet werden. Mit Jdom wird von den Entwicklern das Ziel verfolgt, eine java-spezifische API zum Verarbeiten von Xml bereitzustellen, welche dort hohe Performance-Gewinne gegenüber den vorher vorgestellten APIs Sax und Dom besitzt. In dem Namen Jdom befindet sich zwar der Name Dom, jedoch basiert Jdom weder auf Dom noch auf Sax. Zwar ist Jdom, ebenfalls wie Dom, objektorientiert und es wird gleichsam ein Baum erzeugt, jedoch repräsentiert Jdom ein beliebiges Dokument in Java, welches nicht unbedingt in Xml, sondern lediglich hierarchisch aufgebaut sein muss. Ein weitere großer Vorteil von Jdom ist, dass diese API im Gegensatz zu Dom keine Knotenliste oder Attribute zurückgibt, sondern Java-Klassen, wie `List` oder `Map` benutzt. Desweitern wird nicht der gesamte Baum in dem Speicher gehalten, sondern die Elemente bzw. Attribute werden auf Anfrage der Applikation geladen und nach der Verarbeitung sofort wieder aus dem Speicher freigegeben.

3.8 Css und Xsl

Eines der Hauptanliegen von Xml ist es, die Präsentation von dem Inhalt zu trennen. Das wird dadurch erreicht, dass im Xml-Dokument nur die Daten gespeichert werden. Für die Präsentation werden eigene Dokumente verwendet, die so genannten Stylesheets.

3.8.1 Cascading Style Sheets

Dem Begriff Stylesheet entspricht im Deutschen der Formatvorlage. Dabei handelt es sich um eine Vorlage zur Umwandlung der logischen Auszeichnungen innerhalb eines Markup-Dokuments in physische Auszeichnungen. Cascading Style Sheets (Css) wurde von W3c für die Verwendung in Html-Dokumenten standardisiert (aktuelle Version CSS 2, CSS 3 ist in Vorbereitung). Mit Hilfe von Css wurde die Html-Funktionalität wesentlich erweitert. Formatvorlagen, wie sie im Textverarbeitungsbereich üblich sind, können damit auf ganz ähnliche Weise in Webdokumenten eingesetzt werden. Wo vor Einführung der Css-Formatierungsanweisungen im gesamten Dokument Änderungen vorgenommen werden mussten, genügt nun, aufgrund des Stylesheet-Ansatzes, eine einzige Änderung. Art und Umfang der Implementierung des Css-Konzepts unterscheiden sich jedoch bei den aktuellen Browserversionen teilweise erheblich. So können Festlegungen bezüglich der Formatierung gemäß der Css-Spezifikation können Bestandteil eines Html-Dokuments sein oder in einer externen Datei zusammengefasst werden. In jedem Fall unterstützen sie durch die Verringerung (bzw. im Idealfall sogar Vermeidung) von Formatierungsanweisungen innerhalb von Html-Tags die Trennung von Inhalten und Darstellungs- /Formatierungsvorschriften.

Listing 3.19: "Blaue" CSS-Datei.

```
<STYLE TYPE="text/css">
H1 {
  background-color: #000080;
  font-size: 28pt;
  font-family: Arial;
}
</STYLE>
```

In der CSS-Definition eines HTML-Dokuments werden Element-Eigenschaften mit Gültigkeit für das gesamte Dokument bzw. auch mehrere Dokumente festgelegt. So wird in Listing 3.19 angegeben, dass eine Überschrift erster Ordnung in der Schrift "Arial" mit Punktgröße 28 in Fettschrift vor blauem Hintergrund ausgegeben wird. Sofern der verwendete Browser CSS unterstützt, werden somit, anstelle der voreingestellten Formatierung, als Überschrift erster Ordnung markierte Elemente nun in dieser neu definierten Form angezeigt. Wie schon erwähnt, können solche Cascading Stylesheets Definitionen, statt direkt in den HTML-Sourcecode integriert, in externe Dateien ausgelagert werden. Um ein HTML-Dokument mit einer oder mehreren CSS-Dateien zu assoziieren, können die entsprechende CSS-Definitionen durch sogenannte Links referenziert werden (Listing 3.20). Wenn mit einem `<link...>` eine CSS-Datei referenzieren und

Listing 3.20: Referenz auf eine CSS-Datei.

```
<LINK REL="stylesheet" TYPE="text/css" HREF="blue.css">
```

gleichzeitig ein `<STYLE...>`-Bereich innerhalb des HTML-Dokuments existiert, haben Formate, die direkt innerhalb von `<STYLE...>...</STYLE>` definiert werden, im Konfliktfall Vorrang. Mit dieser Vorschrift ist es somit möglich immer wieder verwendete Formate als Basis-Stylesheet zu importieren und einige davon mit dateispezifischen Formaten zu ergänzten oder zu überschrieben.

Neben der Definition von neuen Formaten für bereits bestehende HTML-TAGs können mit Hilfe der CSS auch sogenannte *Klassen* gebildet werden (Listing 3.21).

Listing 3.21: CSS-Datei mit Klassen.

```
<STYLE TYPE="text/css">
  .wichtig
  {background-color : yellow; color : black }
  .unwichtig
  {background-color : white; color : grey }
</STYLE>
```

Um im HTML-Quellcode diese Formatvorlagen (Klasse "wichtig" und "unwichtig") anwenden zu können, muss dem entsprechenden TAG über das `class` Attribut übergeben werden, welche definierte Klasse verwendet werden soll (Listing 3.22).

Listing 3.22: Html CLASS Attribute.

```
<BODY>
  <H1>Normale Überschrift 1. Ordnung</H1>
  <H1 CLASS="wichtig">Wichtige Überschrift</H1>
  <H1 CLASS="unwichtig">Nicht so wichtige Überschrift</H1>
</BODY>
```

Die Verwendung von Cascading Style Sheets im Zusammenhang mit Xml war seitens des W3c von Beginn der Xml-Spezifikation an vorgesehen, da Xml-Dokumente selbst keine Formatierungsanweisungen enthalten sollten. Die Information über die Art der Darstellung einzelner Elemente sollte über Css mit dem jeweiligen Element verknüpft werden. Über verschiedene Css-Dateien ist es folglich möglich, denselben Inhalt eines Xml-Dokuments unterschiedlich darzustellen. Doch auch hier ergeben sich Defizite von Css im Zusammenhang mit Xml:

- Css ist nicht imstande, eine Einheit (etwa eine Kapitel-Überschrift) zu nehmen und sie an einem anderen Platz in dem Dokument erneut anzuzeigen (etwa in der Kopfzeile).

- Css kann keine Zwillings-Beziehungen schaffen. Zum Beispiel ist es nicht möglich, eine Css-Formatvorlage zu erstellen, die jeden zweiten Absatz im Fettdruck erscheinen läßt.

- Css ist keine Programmiersprache. Es werden keine IF-Abfragen unterstützt, so dass z.B. bedingte Erweiterungen der Formatvorlagen nicht möglich sind.

- Css verfügt nicht über die Fähigkeit, Einheiten durchzuzählen und Werte in Variablen zu speichern. Das bedeutet, dass es nicht einmal möglich ist, gebräuchliche Parameter an einem Platz zu deklarieren, an dem sie leicht verändert werden können.

- Css kann nicht eigenständig Text hervorbringen (etwa Seitenzahlen und dergleichen).

- Css liefert ein simples Kasten-orientiertes Formatierungs-Modell, das für die bisherigen Web-Browser gut geeignet ist, sich aber nicht auf die fortgeschrittenen Auszeichnungsmöglichkeiten ausweiten läßt.

- Css ist auf die Sprachen der westlichen Welt ausgerichtet und setzt voraus, dass die Schriftzeichen in einer horizontalen Abfolge gesetzt werden.

Diese Einschränkungen der Cascading Style Sheets im Gegensatz zu der Offenheit von Xml führten zu der Entwicklung eines erweitertenden Stylesheet-Mechanismus: eXtensible Stylesheet Language (Xsl)

3.8.2 Extensible Stylesheet Language

Die eXTENSIBLE STYLESHEET LANGUAGE ist zwar von der DOCUMENT STYLE SEMANTICS AND SPECIFICATION LANGUAGE (DSSSL), einer Formatvorlagen-Sprache, die ihre Ursprünge in der SGML-Gemeinde hat, abgeleitet, jedoch nach den für XML gültigen Grundsätzen aufgebaut. Die bereits bei XML angeführten Vorteile wie klare Struktur, einfache Interpretierbarkeit gelten daher sinngemäß. XSL bestimmt - gemeinsam mit CSS - das Seitenlayout oder wandelt Dokumente beispielsweise in HTML-fähige Konstrukte um. Die Sprache XSL besteht somit aus zwei wichtigen Komponenten:

1. aus einer Komponente zur Formatierung von XML-Daten (XSL-FO - XSL Formatting Objects), und

2. aus einer Komponente zur Transformation von XML-Daten in andere (XML-) Daten. Für die Transformations-Komponente hat sich die Abkürzung XSLT (XSL Transformation) etabliert.

Wesentliche Grundzüge von CSS bzw. der neueren Version CSS 2 fanden in XSL Eingang, jedoch übernimmt XSL ergänzend die Funktion von "aktiven"[5] XML-spezifischen Formatvorlagen. Abbildung 3.23 zeigt auf welche verschieden Art und Weisen ein XML-Dokument präsentirert werden kann: (1) Wenn das Dokument nicht transformiert werden muss sollte man CSS verwenden. In allen anderen Fällen ist es eine W3C Empfehlung XSLT in einer der folgenden Alternativen zu verwenden: (2) Die Styleeigenschaften zusammen mit dem umstruktuierten Text unter Verwendung von XSL-FO generieren, oder (3) ein neues XML- oder HTML-Dokument erzeugen und eine entsprechende CSS-Datei bereitzustellen [W3C].

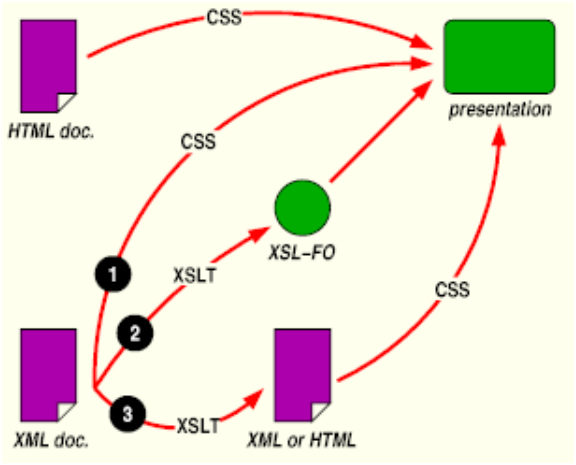


Abbildung 3.23: CSS und XSL [W3C]

[5] Mit der Bezeichnung *aktiv* wird an dieser Stelle auf mit XSLT mögliche Transformationen angespielt.

Im Gegensatz zu CSS verfügt XSL selbst über typische Elemente einer Programmiersprache, so ist es z.B. möglich, bedingte Abfragen zu definieren oder Variablen zu verwenden. XSL gestattet es, die Struktur des XML-Dokuments zur Laufzeit zu modifizieren und Veränderungen auf den XML-Daten durchzuführen. Diese Transformation kann mit Hilfe desselben (!) XSL-Stylesheets sowohl server- als auch clientseitig erfolgen. Die Aufgaben von XSL liegen somit vor allem in zwei Bereichen:

- Die Bildung einer Sprache, mit der sich XML-Dokumente in andere Formate konvertieren lassen (beispielsweise in HTML oder anderes XML, aber eigentlich in beliebige Zielsprachen).
- Die Bereitstellung eines Vokabulars, das den (deskriptiven) TAGs bestimmte Formatierungen zuweist (ähnlich CSS).

Dadurch, dass in XSL auch eine klare Trennung zwischen den Daten und der Präsentation der Daten vorgenommen wird, stellt diese Sprache eine gute Unterstützung in der Entwicklung von XML-Applikationen dar.

3.9 Stärken und Schwächen von XML

In diesem Abschnitt werden, basierend auf den durch diese Arbeit erlangten Erfahrungen des Autors und den vorangegangen Abschnitten, die Stärken (`[+]`) und Schwächen (`[-]`) von XML zusammengefasst.

+ XML ist vom Menschen lesbar und selbstdokumentierend.
  Bei einer aussagekräftigen Wahl der Element- und Attributnamen kann die Bedeutung von Daten innerhalb des XML-Dokuments leicht erkannt werden. Hierbei ist zu empfehlen, englische Bezeichnungen zu verwenden, da XML häufig beim internationalen Austausch von Daten verwendet wird.

+ XML ist für Maschinen (Programme) lesbar.
  Zumindest aus einem bestimmten XML-Dokumenttyp können Informationen, die durch Markup ausgezeichnet sind, extrahiert und weiterverarbeitet werden.

+ XML ist erweiterbar.
  Das heißt, es können selbstdefinierte Element- und Attributnamen verwendet werden. (Im Gegensatz zu HTML.)

+ XML-Dokumente können in Unicode kodiert werden.
  Unicode erlaubt die Verwendung von allen gebräuchlichen Zeichen in fast allen Sprachen und gewährleistet, dass diese auch über Rechnerarchitektur- und Betriebssystemgrenzen hinweg problemlos funktionieren.

+ XML erlaubt es, Typen von XML-Dokumenten zu definieren.
  Eine Familie von XML-Dokumenten gleiches Typs kann gleich behandelt werden.

+ XML-Dokumente können einfach auf Korrektheit überprüft werden.
  Mit XML-Parsern kann bestimmt werden, ob ein Dokument wohlgeformt (Dokument entspricht dem XML-Standard) bzw. gültig (Dokument entspricht einem referenzierten Dokumenttyp) ist.

+ XML erlaubt die Trennung von logischer Struktur, den Daten und deren Präsentation.
  Die logische Struktur ist in einer DTD oder in einem XMLSchema definiert, die Daten befinden sich in einem, dieser logischen Struktur entsprechenden, XML-Dokument und für die Definition der Präsentation dienen Stylesheets (CSS und XSL). Dadurch ist es möglich, dass ein XML-Dokument mit verschiedenen Stylesheets unterschiedlich formatiert bzw. umgekehrt, dass mehrere XML-Dokumente (desselben Typs) mit demselben Stylesheet formatiert werden können und damit gleiches Aussehen erhalten.

+ Für XML gibt es eine Unmenge von Werkzeugen, Anwendungen und Programm-Bibliotheken in fast allen Programmiersprachen.
  Bei eigenen Anwendungen, die XML-Daten verarbeiten sollen, kann man auf diese zurückgreifen und sich dadurch Programmierarbeit ersparen.

+ XML eignet sich hervorragend zum Datenaustausch.
  XML-Dokumente können auf allen Rechnerarchitekturen und Betriebssystemen gelesen werden. XML-Parser können über die in DTDs bzw. XMLSchemata definierte Schnittstelle (Dokumenttyp) die Daten der XML-Dokumente überprüfen und sollten unterschiedliche Dokumenttypen verwendet werden, so besteht zuvor die Möglichkeit, mittels XSLT eine Transformation in den jeweils anderen Typ vorzunehmen.

+ XML ist eine Metasprache.
  Es können weitere Sprachen und Standards definiert werden, die auf XML basieren. Für diese Sprachen bzw. Standards können die vielen bereits vorhandenen XML-Werkzeuge verwendet werden, was die Verarbeitung und Verbreitung erheblich erleichtert.

+ Werden XML-Dokumente beschädigt, so bleiben die nicht beschädigten Teile noch immer lesbar, weil das Dokument aus reinem Text besteht. Beschädigter oder gelöschter Markup kann oft sogar rekonstruiert werden. Bei binären Dokumentformaten ist das meist nicht möglich oder zumindest mit großen Aufwand verbunden.

+ XML ist ein offener Standard und damit nicht an bestimmte Unternehmen oder Software gebunden.

- Obwohl XML grundsätzlich nichts besonders kompliziertes ist, muss man trotzdem mit einem nicht unerheblichen Lernaufwand rechnen. Insbesondere bei der Verwendung von Namensräumen und XMLSchemas ist eine gewisse Erfahrung notwendig, um sie vernünftig einsetzen zu können. Darüber hinaus ist XSLT nicht mit den

üblichen prozeduralen oder objektorientierten Programmiersprachen vergleichbar und deshalb gewöhnungsbedürftig.

- Zur Speicherung von Daten werden seit vielen Jahren relationale Datenbanken eingesetzt. Viele XML-Dokumente lassen sich aber nur sehr schwer und mit hohem Zeitaufwand auf relationale Strukturen abbilden. Allerdings sei erwähnt, dass bereits native XML-Datenbanken zur Verfügung stehen, die direkt mit XML-Daten umgehen können, ohne dass diese zuvor transformiert werden müssen.

- Daten in einer XML-Datei sind zwar durch Markup ausgezeichnet, jedoch ist meist keine, zumindestens intuitive (selbsterklärende) Semantik, gegeben. (Es sei an dieser Stelle an das Beispiel aus Abschnitt 2.5 bezüglich `<i>` *vs.* `<important>` erinnert)

- XML-Dateien sind für den Menschen lesbar und gehen daher nicht sehr sparsam mit Speicherplatz um. Darunter leidet oft auch die Geschwindigkeit bei der Verarbeitung oder der Übertragung. Die Nutzinformationen eines XML-Dokuments könnten leicht in viel kleinere Dateien gespeichert werden. Gerade das Speicherplatzproblem ist nicht zu unterschätzen. Ein Programm könnte die reinen Nutzdaten eines XML-Dokuments auch mit viel weniger Information rekonstruieren: TAGs und Attributnamen könnten durch einfache Trennzeichen ersetzt werden, ähnlich wie das bei der CSV-Darstellung (COMMA SEPARATED VALUES) der Fall ist. Leerzeichen, Tabulatoren und Zeilenvorschübe könnten weggelassen werden. Je nach XML-Dokument ergeben sich hier schon beträchtliche Einsparungspotentiale. Bei den Testdokumenten dieser Arbeit konnte der Autor mit diesen Änderungen die Dateigrösse auf Werte zwischen 27% und 75% der urspünglichen Größe bringen. Es können durchaus Mehrkosten entstehen, wenn ein Programm doppelt soviel Speicherplatz oder doppelt soviel Bandbreite im Netzwerk (internes Netzwerk oder Internet) benötigt. Deshalb wird nach Möglichkeiten gesucht, XML-Dateien zu komprimieren. Beispielsweise werden die XML-Dateien mit den gängigen Komprimierungsverfahren wie GZIP oder ZIP verkleinert. Zu besseren Komprimierungsraten kommt allerdings der Burrows-Wheeler-Algorithmus der z.B. in BZIP2 implementiert ist. Allerdings benötigt der auch mehr Rechenaufwand als GZIP oder ZIP. Eine Komprimierung mit diesen Algorithmen kann eine extreme Verkleinerung bringen, insbesondere gilt: Je mehr (gleiche) XML-Elemente und Attribute im Vergleich zu den Nutzdaten, desto kleiner kann komprimiert werden [Tam00].

3.10 XML Grammatik

```
document          ::= prolog element Misc*
Char              ::= #x9 | #xA | #xD | [#x20−#D7FF] | [#xE000−#xFFFD] | [#x10000−#x10FFFF]
S                 ::= (#x20 | #x9 | #xD | #xA)+
NameChar          ::= Letter | Digit | '.' | '−' | '_' | ':' | CombiningChar | Extender
Name              ::= (Letter | '_' | ':') (NameChar)*
Names             ::= Name (S Name)*
Nmtoken           ::= (NameChar)+
Nmtokens          ::= Nmtoken (S Nmtoken)*
EntityValue       ::= '"' ([^%&"] | PEReference | Reference)* '"' | "'" ([^%&'] | PEReference | Reference)* "'"
AttValue          ::= '"' ([^<&"] | Reference)* '"' | "'" ([^<&'] | Reference)* "'"
SystemLiteral     ::= ('"' [^"]* '"') | ("'" [^']* "'")
PubidLiteral      ::= '"' PubidChar* '"' | "'" (PubidChar − "'")* "'"
PubidChar         ::= #x20 | #xD | #xA | [a−zA−Z0−9] | [−'()+,./:=?;!*#@$_%]
CharData          ::= [^<&]* − ([^<&]* ']]>' [^<&]*)
Comment           ::= <!−−' ((Char − '−') | ('−' (Char − '−')))* '−−>'
PI                ::= '<?' PITarget (S (Char* − (Char* '?>' Char*)))? '?>'
PITarget          ::= Name − (('X' | 'x') ('M' | 'm') ('L' | 'l'))
CDSect            ::= CDStart CData CDEnd
CDStart           ::= '<![CDATA['
CData             ::= (Char* − (Char* ']]>' Char*))
CDEnd             ::= ']]>'
prolog            ::= XMLDecl? Misc* (doctypedecl Misc*)?
XMLDecl           ::= '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
VersionInfo       ::= S 'version' Eq (' VersionNum ' | " VersionNum ")
Eq                ::= S? '=' S?
VersionNum        ::= ([a−zA−Z0−9_.:] | '−')+
Misc              ::= Comment | PI | S
SDDecl            ::= S 'standalone' Eq (("'" ('yes' | 'no') "'") | ('"' ('yes' | 'no') '"'))
element           ::= EmptyElemTag | STag content ETag
STag              ::= '<' Name (S Attribute)* S? '>'
Attribute         ::= Name Eq AttValue
ETag              ::= '</' Name S? '>'
content           ::= (element | CharData | Reference | CDSect | PI | Comment)*
EmptyElemTag      ::= '<' Name (S Attribute)* S? '/>'
doctypedecl       ::= '<!DOCTYPE' S Name (S ExternalID)? S? ('[' (markupdecl | PEReference | S)* ']' S?)? '>'
markupdecl        ::= elementdecl | AttlistDecl | EntityDecl | NotationDecl | PI | Comment
extSubset         ::= TextDecl? extSubsetDecl
extSubsetDecl     ::= ( markupdecl | conditionalSect | PEReference | S )*
elementdecl       ::= '<!ELEMENT' S Name S contentspec S? '>'
contentspec       ::= 'EMPTY' | 'ANY' | Mixed | children
children          ::= (choice | seq) ('?' | '*' | '+')?
cp                ::= (Name | choice | seq) ('?' | '*' | '+')?
choice            ::= '(' S? cp ( S? '|' S? cp )* S? ')'
seq               ::= '(' S? cp ( S? ',' S? cp )* S? ')'
Mixed             ::= '(' S? '#PCDATA' (S? '|' S? Name)* S? ')*' | '(' S? '#PCDATA' S? ')'
AttlistDecl       ::= '<!ATTLIST' S Name AttDef* S? '>'
AttDef            ::= S Name S AttType S DefaultDecl
AttType           ::= StringType | TokenizedType | EnumeratedType
StringType        ::= 'CDATA'
TokenizedType     ::= 'ID' | 'IDREF' | 'IDREFS' | 'ENTITY' | 'ENTITIES' | 'NMTOKEN' | 'NMTOKENS'
EnumeratedType    ::= NotationType | Enumeration
NotationType      ::= 'NOTATION' S '(' S? Name (S? '|' S? Name)* S? ')'
Enumeration       ::= '(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')'
DefaultDecl       ::= '#REQUIRED' | '#IMPLIED' | (('#FIXED' S)? AttValue)
conditionalSect   ::= includeSect | ignoreSect
includeSect       ::= '<![' S? 'INCLUDE' S? '[' extSubsetDecl ']]>'
ignoreSect        ::= '<![' S? 'IGNORE' S? '[' ignoreSectContents* ']]>'
ignoreSectContents ::= Ignore ('<![' ignoreSectContents ']]>' Ignore)*
Ignore            ::= Char* − (Char* ('<![' | ']]>') Char*)
CharRef           ::= '&#' [0−9]+ ';' | '&#x' [0−9a−fA−F]+ ';'
Reference         ::= EntityRef | CharRef
EntityRef         ::= '&' Name ';'
PEReference       ::= '%' Name ';'
```

```
    EntityDecl      ::= GEDecl | PEDecl
65  GEDecl          ::= '<!ENTITY' S Name S EntityDef S? '>'
    PEDecl          ::= '<!ENTITY' S '%' S Name S PEDef S? '>'
    EntityDef       ::= EntityValue | (ExternalID NDataDecl?)
    PEDef           ::= EntityValue | ExternalID
    ExternalID      ::= 'SYSTEM' S SystemLiteral | 'PUBLIC' S PubidLiteral S SystemLiteral
70  NDataDecl       ::= S 'NDATA' S Name
    TextDecl        ::= '<?xml' VersionInfo? EncodingDecl S? '?>'
    extParsedEnt    ::= TextDecl? content
    extPE           ::= TextDecl? extSubsetDecl
    EncodingDecl    ::= S 'encoding' Eq ('"' EncName '"' | "'" EncName "'" )
75  EncName         ::= [A−Za−z] ([A−Za−z0−9._]| '−')∗
    NotationDecl    ::= '<!NOTATION' S Name S (ExternalID | PublicID) S? '>'
    PublicID        ::= 'PUBLIC' S PubidLiteral
```

4 Open Mathematical Documents OMDoc

Wie schon in Kapitel 2 erläutert, wurde als Reaktion auf die Probleme bei grafisch orientiertem Markup (*prozeduralem Markup*) der Ansatz des Semantic Xml entwickelt. Statt der graphischen Erscheinung, beschreibt Semantic Xml die inhaltliche logische Struktur von Dokumenten, basierend auf einer wohldefinierten Menge von Vokabeln mit einer intuitiven (selbsterklärenden) Bedeutung[1]. Auf diese Weise entsteht eine Trennung zwischen Inhalt und Erscheinungsform (Präsentation) des Dokuments. Das Markup im Dokument beschreibt nur noch, welche Rolle bestimmte Teile des Inhalts spielen. Die Erscheinungsform für die Rollen wird dann in separaten Stylesheets festgelegt. Als Modell wird bei Semantic Xml eine Baumstruktur verwendet, in der die inneren Knoten die logischen Elemente und die Blätter nicht weiter untergliederter Texte oder Objekte sind. Mit dieser Separation von Inhalt und Erscheinungsform stellen sich sogar hinsichtlich der Präsentation von Dokumenten weitere Vorteile ein. Bereits auf der Stufe des *deskriptiven Markups* sind folgende Vorzüge gegenüber grafisch orientiertem Markup zu nennen:

- Konsistenz des grafischen Formats.
  Inhalte, die die gleiche Rolle spielen, werden gleich dargestellt.
- Konsistenz bei Präsentation verschiedener Dokumente gleichen Typs.
  Enthalten mehrere Dokumente die gleichen Rollen, so kann ein gemeinsames Stylesheet verwendet werden.
- Präsentation in unterschiedlichen Erscheinungsformen.
  Durch den Austausch des Stylesheets kann die Erscheinungsform konsistent geändert werden, ohne das Dokument zu ändern.

Der Vorteil, der sich dann in Semantic Xml gegenüber *deskriptiven Markup* herausstellt, ist

> die Verwendung einer wohldefinierten Menge an Vokabeln mit einer intuitiven kontextuellen Bedeutung. Die dadurch entstehende Semantik eignet sich insbesondere zur Erstellung von *Ontologien* als Darstellungsweise von Wissensbeständen.

[1] An dieser Stelle sei erneut darauf hingewiesen, dass die intuitive Bedeutung der Vokabeln dann durch *semantisches Markup* klar definiert wird (Abschnitt 2.5).

Der Begriff *Ontologie* hat in der Philosophie seinen Ursprung, stammt aus der griechischen Sprache und bedeutet "Die Lehre vom Seienden". Im Rahmen der Informatik versteht man darunter die Modellierung von Domänen der realen Welt, mit dem Ziel eines strukturierten und fundierten Aufbaus von Wissensbasen, mittels derer rechnergestützt maximales Wissen generiert werden kann. Es handelt sich somit um ein formal definiertes System von Begriffen, Konzepten und Relationen zwischen diesen Begriffen [SS04]. Der Vorteil der Nutzung einer Ontologie bei der Suche nach Informationen wird deutlich, wenn man sich einen einfachen Anwendungsfall ausmalt: Gibt man in einer (semantischen) Suchmaschine "Beweistheorie" ein, so wird man Dokumente nicht finden, welche nur den "Sequenzenkalkül" erwähnen. Die semantische Suche bezieht das in der Ontologie hinterlegte Wissen mit ein und erweitert die Suche zweckmäßig um neue Begriffe. Manchmal ist aus dem Text heraus jedoch nicht ohne weiteres die Zugehörigkeit (Relation) einer Wortfolge zu einem Konzept zu erkennen. Auch das sogenannte Natural Language Processing stößt hier an seine Grenzen. Deshalb müssen die Passagen in Dokumenten mit den nötigen Angaben annotiert werden. Solche Metadaten kennzeichnen das Vorkommen von Ontologie-Entitäten in Dokumenten. Annotationen sind hier als Referenzen auf Einträge in der Ontologie zu verstehen. Dabei lassen sich diese Referenzen auch gebrauchen, um das Dokument als solches zu charakterisieren und zu kategorisieren.

Im Folgenden wird zur mathematischen Wissensrepräsentation die Semantic Xml-Applikation OMDoc [OMD] vorgestellt. Es wird ein Vokabular eingeführt, mit dem es möglich sein wird, mathematische Informationen zu annotieren, so dass das in Kapitel 2 angesprochene Problem der Informationsübermittlung eine Lösung, insbesondere hinsichtlich der Mathematik, erhält. Mit OMDoc wird ein strukturierter und fundierter Aufbau von mathematischen Wissensbasen und darin bestehenden Relationen ermöglicht. Der hier gegebene Überblick wird alle für die Integration in ✓eriFun (Kapitel 5) notwendigen Aspekte von OMDoc behandeln. Um einen tiefergehenden Einblick in diese Sprache zu erlangen, wird der geneigte Leser auf [Koh06] verwiesen.

4.1 Mathematische Wissensrepäsentation mit OMDoc

Mathematik ist eine der ältesten Wissenschaften überhaupt und bildet eine fundierte Basis für eine Vielzahl von Technologien. Da das mathematische Wissen überproportional schnell anwächst, wird immer mehr der Ruf nach einer wohl organisierten Form der Verwaltung dieser Erkenntnisse laut. Die Art und Weise, wie Mathematik auftritt und festgehalten wird, sind mannigfaltig und nur sehr schwer als Ganzes zu fassen. Das schöne jedoch an dieser Wissenschaft ist, dass sich die diversen Erscheinungsformen in einem gleichen: In der formalen Semantik! Semantik als Interpretationsfunktion von einem syntaktischen Bereich in einen semantischen ist hier als eine Funktion von einer Xml Syntax in ein Vokabular[2] zu verstehen. Als digitale Platt-

[2]Eine standardisierte Menge von Begriffen mit Kontextbedingungen (Ontologie).

form der Mathematik gewinnt auch hier wieder das Internet zunehmend an Bedeutung. Zum einen wird es als Präsentationsplattform von e-Learningarchitekturen und zum anderen als Protokoll von diversen mathematischen Softwaresystemen verwendet. Schon jetzt werden hierzu XML-Applikationen verwendet, um den Dokumenten zumindest eine Struktur zu verleihen und diese interdisziplinär verwenden zu können. Der Vorteil einer solchen Plattform ist, dass sich komplexe Systeme nach außen hin in einer sehr einfachen Form präsentieren können und somit eine hohe Benutzerfreundlichkeit erlangen. Besonders dieser Sachverhalt führt zu einer immer größer werden Akzeptanz des Internet als "Netz für mathematisches Wissen". Um diesen mathematischen Informationsaustausch in geregelte Bahnen zu leiten ist eine Markupsprache notwendig, die einen standardisierten Austausch von mathematischen Objekten und Wissen ermöglicht. Die ersten Anfänge lieferten MathML und OpenMath. Beide Sprachen entwickelten eine Repräsentation der Bedeutung von mathematischen Objekten, mit dem Ziel, diese an mathematische Softwaresysteme wie Computeralgebrasysteme (CAS), automatische Beweiser oder Visualisierungssysteme übertragen zu können. MathML und OpenMath ermöglichten, die großen formalen Wissensbestände der Mathematik, die bisher meist in der präsentationsorientierten Sprache LaTeX vorlagen, in eine semantische XML-Repräsentation zu überführen. Der MathML Standard definiert hierfür zwei Teilsprachen: Presentation-MathML und Content-MathML. Presentation-MathML definiert, wie eine feste Menge von mathematischen Symbolen, die in K-12 (Kindergarten bis Klasse 12) Verwendung finden, präsentiert werden. Content-MathML definiert die Semantik für diese Symbole. Für eine Präsentation muss Content-MathML in Presentation-MathML transformiert werden. Der OpenMath-Standard umfasst kein Präsentations-Markup, aber ermöglicht durch den Mechanismus des Content Dictionary (CD) die Bedeutung (neuer) mathematischer Symbole in die OpenMath-Syntax zu spezifizieren. Content Dictionaries sind öffentlich und präsentieren die mathematische Schnittmenge zwischen verschiedenen Applikationen. OpenMath ist eine semantische Markupsprache (hier als Semantic XML-Applikation zu verstehen), die es erlaubt, mathematische Formeln zu strukturieren und als solche zu kennzeichnen. CDs ermöglichen es, wie Bibliotheken der Programmiersprache Java, Spracherweiterungskonzepte auszulagern. Die positive Konsequenz ist, dass OpenMath nicht auf K-12 beschränkt ist (Für einen Überblick der OpenMath-Syntax siehe Abschnitt 4.3). Das Hauptziel von OpenMath war zunächst, einen Standard für Systeme zu schaffen, die mathematische Ausdrücke und Formeln als Eingabe benutzen, so dass diese Standardrepräsentation (mit Hilfe von sogenannten *phrasebooks* (Abbildung 4.2)) in die Eingabesprachen der verschiedenen Systeme übersetzt werden kann und umgekehrt deren Output in OpenMath. Dafür repräsentiert OpenMath ausschließlich mathematische Symbole und Formeln. Content-MathML und OpenMath können teilweise in einander übersetzt werden und sie können wechselseitig in die jeweils andere Sprache eingebunden werden.

OpenMath liefert jedoch lediglich die Grundlage für eine Ontologie mathematischer Dokumente, denn für den Aufbau solcher Ontologien gehören neben den mathematischen Objekten auch Aussagen über die Objekte, wie etwa Theoreme und Beweise, sowie

Eigenschaften und Relationen zwischen den Objekten. Um mathematische Texte, die adaptierbar sein sollen, repräsentieren zu können, bedarf es diverser Erweiterungen.

OMDoc (Open Mathematical Documents) ist eine solche Erweiterung zu OpenMath! Es können nicht nur mathematische Texte als solche gekennzeichnet werden, sondern auch allgemeine Strukturen, Beweise, Beispiele, Erklärungen und Aussagen über Objekte. In dieser Sprache wird eine Infrastruktur zur Kommunikation und zur Speicherung mathematischen Wissens geliefert, die sich im Gegensatz zu Präsentations Markup nicht auf das "Wie", sondern auf das "Was" konzentriert. D.h der Inhalt und die Funktion eines Dokuments stehen im Vordergrund. Um mathematische Dokumente auf diese Weise zu strukturieren und annotieren verwendet OMDoc drei Abstraktionsebenen:

1. **Formeln**
   Zur Darstellung der Atome der Mathematik, die Formeln, verwendet OMDoc den etablierten Standard: OpenMath [Ope] (Abschnitt 4.3). Mit OpenMath ist es möglich, die Struktur von Formeln (*deskriptiver Markup*) sowie deren Kontext (*context Markup*), mit Hilfe von URI Referenzierungen innerhalb der verwendeten Symbole, zu repräsentieren[3].

2. **Aussagen**
   OMDoc ermöglicht auf dieser Abstraktionsebene Aussagen über mathematische Formel und deren Kontext (Theorie) zu formulieren und zu kennzeichnen (Unterabschnitt 4.4.2.1). Auch an dieser Stelle findet sich somit sowohl *deskriptiver* als auch *context* Markup wieder.

3. **Theorien**
   Auf der höchsten Abstraktionsebene können in OMDoc mathematische Theorien (Unterabschnitt 4.4.2) beschrieben und gekennzeichnet werden. Zudem wird die Beschreibung von Theorie-Inklusionen (Unterabschnitt 4.4.2.1), Theorie-Morphismen, und importierten (lokalen/globalen) Theorien zur Verfügung gestellt ([Koh06]). Mit dieser Vorgehensweise wird es ermöglicht das repräsentierte mathematische Wissen in kleine Blöcke zur leichten Wiederverwendung zu kapseln. Theorien stellen in OMDoc den Begriff des Kontextes dar, in dem die zuvor angesprochenen Formeln und Aussagen ihre Gültigkeit finden.

OMDoc ist somit eine Semantic Xml-Applikation, mit deren Grammatik und ausdrucksstarken, wohldefinierten deskriptiven Tags es möglich ist, Formeln spezifische Bedeutungen in verschiedenen Kontexten (Theorien) zuzuweisen. Auf Grund dessen und den oben genannten Abstraktionsebenen liefert OMDoc ein viel breiteres und ausdrucksstärkeres Markupformat, als OpenMath oder MathML.

[3] OMDoc unterstützt zur Repräsention von mathematischen Formeln noch weitere Markupsprachen, wie zum Beispiel MathML. In dieser Arbeit wird der hier erwähnte OpenMath Standard verwendet. Eine vollständige Liste der unterstützten Markupsprachen ist in [Koh06] nachzulesen

4.2 Aufbau eines OMDoc Dokuments

Listing 4.1: Aufbau eines OMDoc Dokuments

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE omdoc SYSTEM "omdoc-vf.dtd">

<omdoc xml:id="VeriFun3.0-2005-08-31"
       version="1.2"
       xmlns="http://www.mathweb.org/omdoc"
       xmlns:dc="http://purl.org/dc/elements/1.1/">

  <metadata>
    <dc:date>2005-09-04T05:18:47</dc:date>
  </metadata>

  <ignore type="todo" comment="Put some life in here!">
    <theory xml:id="groups">
    </theory>
  </ignore>

</omdoc>
```

Das XML-Wurzelelement einer OMDoc Dokument-Instanz ist `omdoc` (Listing 4.1). Nach der XML Spezifikation kann dieses Element alle weiteren in der `omdoc-vf` DTD deklarierten Kind-Elemente enthalten. Die Kind-Elemente, die als direkte Nachkommen von `omdoc` auftreten dürfen, werden in OMDoc als Toplevel-Elemente bezeichnet. Das optionale Attribut `xml:id`, dessen Wert innerhalb des Dokuments eindeutig sein muss, eröffnet die Möglichkeit das gesamte Dokument aus anderen Dokumenten heraus zu referenzieren. Mittels eindeutiger Referenzierung wird in OMDoc ein hohes Maß an Wiederverwendung (Mehrfachverwendung) von (Teil-) Dokumenten ermöglicht (structure sharing). An dieser Stelle sei darauf hingewiesen, dass alle OMDoc Elemente, die das `xml:id` Attribut besitzten, auch mit einem `style` bzw. `class` Attribut versehen werden können. Mit diesem Hack ist es der Semantic XML-Applilation OMDoc möglich, eine Schnittstelle zu Präsentationsinformationen zu liefern. Die in Abschnitt 3.8 kurz vorgestellte Syntax der Cascading Style Sheets kann als Wert dieses Attributes direkt eingesetzt werden. Weitere maßgebliche Attribute des Wurzelelements sind die Deklarationen der verwendeten Namensräume. Sämtliche OMDoc Elemente befinden sich in dem Namensraum mit der URI `http://www.mathweb.org/omdoc`. Mit Hilfe dieser URI wird nicht die aktuelle OMDoc Version denotiert oder eine DTD referenziert, sondern lediglich mögliche Namenskonflikte umgangen (Abschnitt 3.5). Die aktuelle OMDoc Version kann durch ein weiteres optionales `version` Attribut in dem Wurzelelement angegeben werden. Um beschreibende Informationen über das aktuelle OMDoc Dokument enkodieren zu können, die sowohl "Menschen-lesbar" als auch "Maschinen-verstehbar"

sind, erlaubt OMDoc das Einfügen von Metadaten an verschiedenen Stellen. Die Administration einer ganzen Ansammlung von solch beschriebenen Dokumenten, wie zum Beispiel die Kategorisierung nach Autoren, oder eine eingeschränkte Rechtevergabe, wird somit um ein Vielfaches vereinfacht. Als Beschreibungssprache verwendet OMDoc eine der wohl bekanntesten in dem Gebiet der Metasprachen: Doublin Core Meta Data. Alle DC-Elemente könne als Kind-Elemente des OMDoc Elements `metadata` auftreten. Sie befinden sich dem Namesraum `http://purl.org/dc/elements/1.1/`, für welchen OMDoc das Präfix `dc` verwendet. In dem DC Element `date` wird das Erstellungsdatum inklusive des Zeitstempels kodiert. Das Format dieser Zeitangabe muss dem Format des Datentyps `dateTime`, spezifiziert in dem XmlSchema `http://www.w3.org/TR/xmlschema-2/#dateTime` genügen. Weitere Doublin Core Elemente sind unter [Dou] nachzulesen. Neben der expliziten Auszeichnung von (administrativen) Metadaten, erfahren auch Kommentare in OMDoc eine gesonderte Behandlung. Im Gegensatz zu den meisten Xml-Applikationen, in denen Kommentare innerhalb eines Dokuments zwischen `<!--` und `-->` plaziert werden, hat OMDoc hierfür ein spezielles semantisches Markup-Tag bereitgestellt. Alles was normalerweise zwischen den oben genannten Xml-Tags für Kommentare stehen würde, wird in OMDoc mit dem `ignore`-Element ausgezeichnet. Durch die zusätzlichen optionalen Attribute `type` und `comment` kann dem Inhalt dieser Elemente, was jeder wohlgeformter OMDoc Ausdruck sein darf, eine Bedeutung zugewiesen werden. Die das Dokument verarbeitenden Applikationen können auf diese Weise entscheiden, was mit dem Inhalt dieser Elemente während der Dekodierung passieren soll. Kommentare werden folglich, innerhalb der Dekodierung durch einen Xml-Parser (Abschnitt 3.7) nicht ignoriert.

4.3 Repräsentation atomarer mathematischer Objekten

Zur Darstellung von atomaren mathematischen Objekten (Formeln) verwendet OMDoc die etablierte Markupsprache OpenMath. Im folgenden werden die für diese Arbeit relevante OpenMath Elemente beschrieben. Eine detaillierte Beschreibung ist unter [Ope] nachzulesen.

4.3.1 OpenMath-Architektur

In Abbildung 4.2 werden die drei Schichten der OpenMath-Architektur dargestellt. In der privaten Schicht (`private layer`) befinden sich die internen und applikationseigenen Repräsentationen der mathematischen Objekte. Mit Hilfe von den schon erwähnten *phrasebooks* wird die abstrakte Schicht (`abstract layer`) erreicht. Hier befindet sich die Repräsentation der applikationseigenen mathematischen Objekte als OpenMath-Objekte wieder (Unterabschnitt 4.3.2). Da sowohl die applikationseigene Repräsentation als auch die abstrakte OpenMath-Repräsentation intern in dem jeweiligen Programm

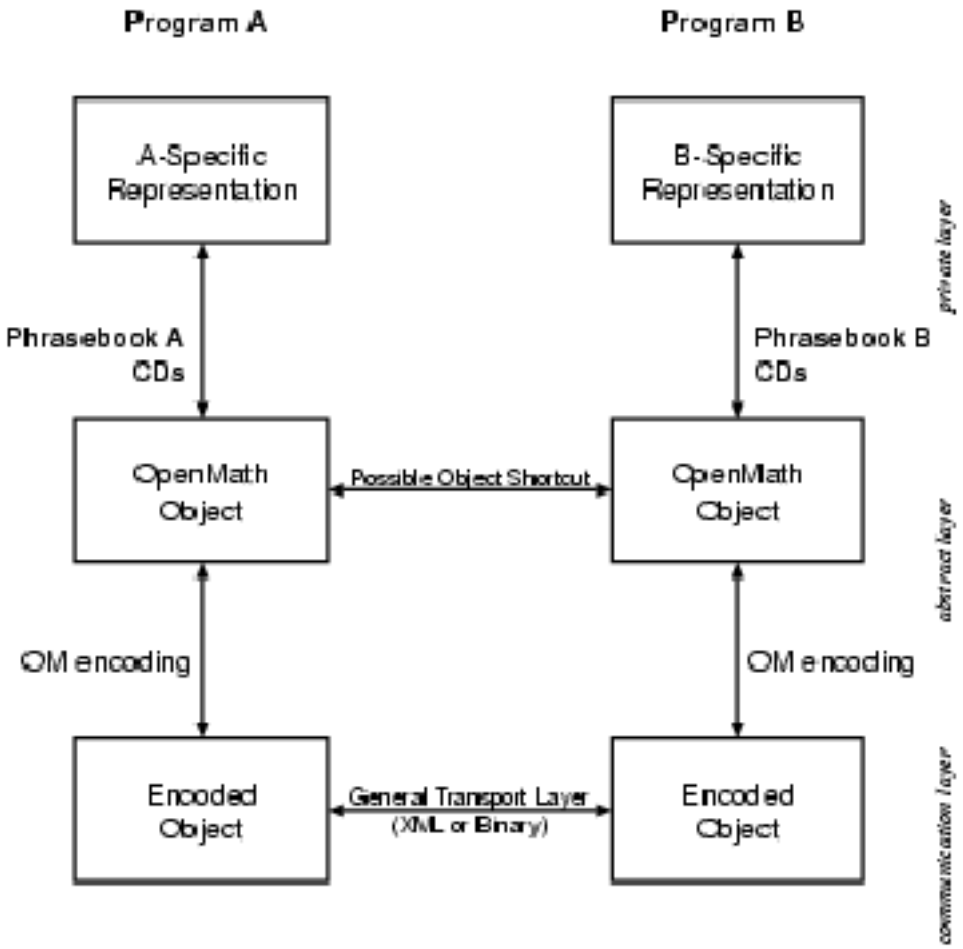


Abbildung 4.2: OpenMath-Architektur [Ope]

gehalten werden, können diese beiden Ebenen unter Umständen gleich sein. Um die repräsentierten Objekte mit anderen Programmen kommunizieren zu können, wird schließlich die dritte Schicht verwendet. In dieser liegen die internen Objekte als Bytestrom vor, so dass sie über diverse Kommunikationsprotokolle interdisziplinär ausgetauscht werden können. Welches Datenformat für die externen Repräsentation verwendet wird, ist jeder Applikation selbst überlassen. Wegen der in den vorangegangen Abschnitten erwähnten Vorteile von Xml wird in dieser Arbeit für die externe Repräsentation, eine Enkodierung in eine Xml-Applikation angegeben (Unterabschnitt 4.3.3).

4.3.2 Formale Definition von OpenMath-Objekten

In OpenMath werden mathematische Objekte als *labelled trees* repräsentiert (Abbildung 4.3). In dem OpenMath-Jargon werden diese Strukturen einfach als OpenMath-

Objekte bezeichnet. In den folgenden Abschnitten wird die abstrakte Definition von OpenMath-Objekten näher erläutert.

4.3.2.1 Atomare OpenMath-Objekte

Die atomaren OpenMath-Objekte bilden die Blätter der *labelled trees*. Ein atomares OpenMath-Objekt besitzt einen der folgenden Typen:

Integer
Hiermit sind die ganzen Zahlen, wie man sie aus der Mathematik kennt, gemeint.

IEEE floating point number
Gleitkommazahlen nach dem IEEE 754-1985 Standard.

Character strings
Eine Unicode Zeichenkette.

Bytearray
Eine Sequenz von Bytes.

Symbol
Eine Symbol stellt in OpenMath ein Einheit, bestehend aus drei Information dar. (1) Der Name des Symbols wird als *Character string* entsprechend der Produktionsregeln aus Unterabschnitt 4.3.2.5 dargestellt (2) Das Content Dictonary gibt den Ort der Definition[4] des Symbols an. Diese Information ist ein Tupel, bestehend aus (2.1) Der Name des Content Dictonary (Produktion entsprechend wie der des Symbolnamens) (2.2) Ein optionales wenn aber angegeben, dann eindeutigem Präfix, genannt `cdbase`, welches zur Disambiguierung verschiedener CDs dient. (3) Optional kann in dem dritten Informationsfeld die *Rolle* des Symbols angegeben werden (Unterabschnitt 4.3.2.4).

Variable
Eine OpenMath-Variable muss einen Namen haben. Dieser Name wird entsprechend den Produktionsregeln aus Unterabschnitt 4.3.2.5 dargestellt.

4.3.2.2 Abgeleitete OpenMath-Objekte

Abgeleitete OpenMath-Objekte werden verwendet, um *fremde* (foreign) Daten, also "Nicht-OpenMath-Objekte", in ein OpenMath-Objekt einzubetten. Ein Beispiel hierfür könnte die Annotation einer Formel mit einer Animation sein. Abgeleitet OpenMath-Objekte werden nach folgender Regel gebildet: Wenn A ein "Nicht-OpenMath-Objekt" ist, dann ist **foreign**(A) ein OpenMath-Objekt. So erstellte Objekte können

[4] Auf die konkrete Syntax einer Symbol-Definition in einem OpenMath Content Dictionary wird in dieser Arbeit nicht weiter eingegangen, da hierfür direkt OMDoc eingesetzt wird.

ein zusätzlichen Enkodierungsfeld besitzen, in dem die Interpretation des Inhaltes des Objektes angegeben werden kann.

4.3.2.3 OpenMath-Objekte

OpenMath-Objekte werden folgendermaßen rekursiv definiert:

1. Atomare OpenMath-Objekte sind OpenMath-Objekte.
   Abgeleitete OpenMath-Objekte sind keine OpenMath-Objekte, sondern können zur Erstellung von OpenMath-Objekten verwendet werden

2. Wenn $A_1, A_2, \ldots, A_n$ mit $n > 0$ OpenMath-Objekte sind, dann ist **application**$(A_1, A_2, \ldots, A_n)$ ein OpenMath-Applikations-Objekt.
   Eine **application** erzeugt ein OpenMath-Objekt basierend auf einer Sequenz von einem oder mehr OpenMath-Objekte. Das erste Kind-Element bezeichnet man mit "head", während die restlichen als Argumente tituliert werden. Mit einem OpenMath-Applikations-Objekt kann man somit u.a. die mathematische Anwendung einer Funktion auf ein Tupel von Argumenten beschreiben. Angenommen das OpenMath-Symbol *sin* sei in einer entsprechenden CD definiert, dann ist **application**(sin, x) ein valides, abstraktes OpenMath-Objekt hinsichtlich des mathematischen Ausdruck $sin(x)$ (Abbildung 4.3).

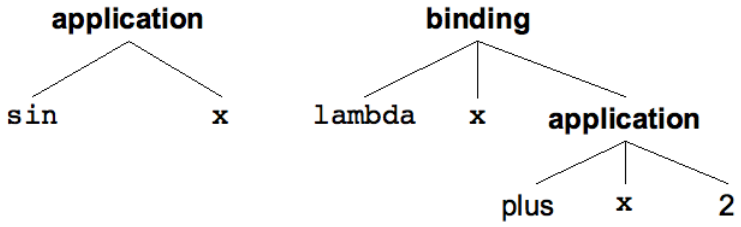


Abbildung 4.3: OpenMath-Applikatons- und -Bindungs-Objekte [Ope]

3. Wenn $S_1, S_2, \ldots, S_n$ OpenMath-Objekte sind, A ein OpenMath-Objekt ist und $A_1, A_2, \ldots, A_n$ mit $n > 0$ OpenMath-Objekte oder abgeleitete OpenMath-Objekte sind, dann ist **attribution**$(\mathbf{A}, \mathbf{S_1 A_1}, \ldots, \mathbf{S_n A_n})$ ein OpenMath-Attribut-Objekt. A ist das attributierte Objekt, $S_1, S_2, \ldots, S_n$ sind die sogenannten Schlüssel und $A_1, A_2, \ldots, A_n$ die dazu assoziierten Werte.

4. Wenn B und C OpenMath-Objekte sind und $v_1, \ldots v_n$ $(n \geq 0)$ OpenMath-Variablen oder OpenMath attributierte Variablen sind, dann ist **binding**$(B, v_1, \ldots v_n, C)$ ein OpenMath-Bindungs-Objekt.
   OpenMath-Bindungs-Objekte werden von einem OpenMath-Objekt und einer Sequenz von Null oder mehreren Variablen gefolgt von einem weiteren OpenMath-Objekt konstruiert. Das erste OpenMath-Objekt ist der Bindungs-Operator, die Argumente 2 bis $n-1$ stellen die zu bindenden Variablen dar und das letzte

Objekt repräsentiert den Rumpf, in dem die Variablen frei vorkommen. In dem Ausdruck $\lambda x.x + 2$ ist die Variable x durch den Bindungs-Operator λ gebunden (Abbildung 4.3). Um Namenskonflikte in den Variablen zu vermeiden sind hier Umbenennungen erlaubt.

5. Wenn S ein OpenMath-Objekt ist und $A_1, \ldots A_n$ $(n \geq 0)$ OpenMath-Objekte oder abgeleitete OpenMath-Objekte sind, dann ist $\mathbf{error}(S, A_1, \ldots A_n)$ ein OpenMath-Fehler-Objekt.

Alle durch die Regeln (2) bis (5) erstellten OpenMath-Objekte nennt man zusammengesetzte OpenMath-Objekte.

4.3.2.4 OpenMath-Symbol-Rollen

Wenn ein OpenMath-Symbol das erste Kind-Element von einer OpenMath-Applikation, -Bindung oder -Fehlers ist, dann bezeichnet man dieses Symbol als konstruierendes Symbol. Mit dem zusätzlichen Attribut `role` kann eine Abgrenzung vorgenommen werden, wie ein zusammengesetztes OpenMath Objekt konstruiert werden sollte[5]. Mögliche Rollen sind:

binder
: Das Symbol darf als erstes Kind-Element eines OpenMath-Bindungs-Objekts verwendet werden.

attribution
: Das Symbol darf als Schlüssel in einem OpenMath-Attribut-Objekt verwendet werden. Diese Art der Attributierung kann durchaus von Applikationen ignoriert werden und sollte somit lediglich informativen Charakter haben.

semantic-attribution
: Analog zu *attribution* mit dem Unterschied, dass diese Attributierung nicht ignoriert werden darf, da hierdurch die Bedeutung des attributierten OpenMath-Objektes verändert wird.

error
: Das Symbol darf als erstes Kind-Element eines OpenMath-Fehler-Objekts verwendet werden.

application
: Das Symbol darf als erstes Kind-Element eines OpenMath Applikation-Objekts verwendet werden.

constant
: Dieses Symbol darf nicht als konstruierendes Symbol verwendet werden.

[5] Wie diese Abgrenzungen durch die jeweilige Applikation interpretiert werden, bleibt einer jeden selbst überlassen. Es handelt sich hierbei um eine Empfehlung als um eine Restriktion.

Ein Symbol kann nicht mehr als eine Rolle besitzen. Ist keine Rolle explizit angegeben, so kann das Symbol an jeder Stelle verwendet werden.

4.3.2.5 OpenMath-Symbol-Namen

Namen von Symbolen, Variablen und Content Dictionaries müssen den Produktionsregeln aus Listing 4.4 genügen.

Listing 4.4: Produktion von OpenMath Namen

```
NameStartChar ::= ":" | [A−Z] | "_" | [a−z] | [#xC0−#xD6] | [#xD8−#xF6] |
                  [#xF8−#x2FF] | [#x370−#x37D] | [#x37F−#x1FFF] |
                  [#x200C−#x200D] | [#x2070−#x218F] | [#x2C00−#x2FEF] |
                  [#x3001−#xD7FF] | [#xF900−#xFDCF] | [#xFDF0−#xFFFD] |
                  [#x10000−#xEFFFF]
Name          ::= NameStartChar (NameChar)*
NameChar      ::= NameStartChar | "−" | "." |
                  [0−9] | #xB7 | [#x0300−#x036F] | [#x203F−#x2040]
```

4.3.3 OpenMath Xml-Enkodierung

Die Motivation, OpenMath-Objekte nach Xml zu enkodieren, wurde maßgeblich zum einen durch das Vorhaben geprägt, ein "Menschen-Lesbares" und "Menschen-Schreibbares" Datenformat zu präsentieren und zum anderen, um eine leichte Integration in Applikationen zu ermöglichen, die bereits Xml als Datenaustauschformat verwenden. Im Folgenden werden die einzelnen Xml-Elemente besprochen die zur Enkodierung von OpenMath-Objekten verwendet werden. Eine kompakte und intuitiv lesbare Übersicht über die gesamte Xml-Enkodierung der OpenMath Grammatik wird in Unterabschnitt 4.3.4 bereitgestellt.

Integers
: werden mit dem `OMI` Element enkodiert. Der Inhalt dieses Tags ist eine Sequenz der einzelnen Ziffern zur Basis 10 ($-?[0-9]+$) oder 16 ($-?x[0-9A-F]+$). Leerzeichen zwischen den einzelnen Ziffern werden ignoriert.

Symbole
: werden mit dem `OMS` Element enkodiert. Dieses Element besitzt die drei Attribute `cd`, `name` und `cdbase`. Der Wert des Attributs `cd` ist das *Content Dictonary*, in welchem das Symbol definiert ist. Der Wert des Attributs `name` ist der Name des Symbols. Das optionale Attribut `cdbase` ist eine URI, die zur Disambiguierung zwischen verschiedenen Content Dictionaries verwendet werden kann. Sollte ein Symbol kein `cdbase` Attribut erhalten, so erbt es dieses von dem ersten Vorgänger

in dem Baum, welcher ein solches Attribut besitzt, sollte ein solches Element existieren. Welche Rolle ein Symbol erhält, wird an dieser Stelle nicht enkodiert, sondern gesondert in dem entsprechenden Content Dictonary gespeichert.

Variablen
werden mit dem `OMV` Element enkodiert. Dieses Element erhält lediglich das Attribut `name`, welches den Namen der Variable als Wert beinhaltet.

IEEE floating point number
werden mit dem `OMF`-Element enkodiert. Dieses Element hat entweder das Attribut `dec` oder `hex`. Der Wert des Attributs `dec` ist die Gleitkommazahl zur Basis 10 $((-?)([0-9]+)?(''.''[0-9]+)?([eE](-?)[0-9]+)?.)$. Der Wert des Attributs `hex` ist die Repräsentation zur Basis 16 von 64bit IEEE Doubles.

Character strings
werden mit dem `OMSTR`-Element enkodiert. Der Inhalt dieses Element ist ein Unicode-Text. Auch hier gelten die Regeln, wie schon in Unterabschnitt 4.3.2.5 erwähnt.

Bytearrays
werden mit dem `OMB` Element enkodiert. Der Inhalt dieses Elements ist ein Base64 enkodierte Zeichenkette. Die Base64-Enkodierung wird in [BF96] definiert.

Applications
werden mit dem `OMA` Element enkodiert. Eine Applikation, deren head das OpenMath-Objekt e_0 und deren Argumente die OpenMath Objekte $e_1 \dots e_n$ sind, wird dann folgendermaßen enkodiert:

$$\texttt{<OMA>}C_0C_1 \dots C_n\texttt{</OMA>}$$

wobei C_i die Enkodierung von e_i darstellt. Beispielsweise wird in Listing 4.5 das Symbol *sin*, welches in dem Content Dictionary *transc*1 definiert ist, mit dem Argument x dargestellt.

Listing 4.5: Xml Enkodierung von **application**(sin, x)

```
<OMA>
  <OMS cd="transc1" name="sin" />
  <OMV name="x" />
</OMA>
```

Errors
werden mit dem `OME`-Element enkodiert. Ein Fehler mit dem Symbol s und dessen Argumente die OpenMath-Objekte $e_1, \dots e_n$ sind, wird dann folgendermaßen enkodiert:

$$\texttt{<OME>}C_sC_1 \dots C_n\texttt{</OME>}$$

wobei C_s die Enkodierung von s und C_i die Enkodierung von e_i darstellt.

Binding
werden mit dem `OMBIND` Element enkodiert. Durch den OpenMath Bindungs-Operator b gebundene OpenMath Variablen $x_1, \ldots, x_n$ in dem Rumpf c werden dann folgendermaßen enkodiert:

$$\texttt{<OMBIND>}B\texttt{<OMBVAR>}X_1 \ldots X_n\ \texttt{</OMBVAR>}C\texttt{</OMBIND>}$$

wobei B, C und X_i die Enkodierungen von b, c und x_i darstellen. Beispielsweise wird in Listing 4.6 der OpenMath Ausdruck
binding$(lambda, x,$ **application**$(sin, x))$ enkodiert.

Listing 4.6: Xml Enkodierung von **binding**$(lambda, x,$ **applicationsin**, **x**$)$

```
<OMBIND>
  <OMS cd="fns1" name="lambda" />
  <OMBVAR>
    <OMV> name ="x" />
  </OMBVAR>
  <OMA>
    <OMS cd="transc1" name="sin" />
    <OMV name="x" />
  </OMA>
</OMBIND>
```

Attributierungen
werden mit dem `OMATTR`-Element enkodiert. Wenn ein OpenMath-Objekt e mit den Paaren $(s_1, e_1), \ldots, (s_n, e_n)$ attributiert ist, dann wird dies folgendermaßen enkodiert:

$$\texttt{<OMATTR><OMATP>}S_1C_1 \ldots S_nC_n\texttt{</OMATP>}E\texttt{</OMATTR>}$$

wobei S_i die Enkodierung von s_i, C_i die Enkodierung von e_i und E die Enkodierung von e darstellt. Beispielsweise wird in Listing 4.7 der Typ einer Variablen enkodiert.

Listing 4.7: Xml Enkodierung von $x : nat$

```
<OMATTR>
  <OMATP>
    <OMS cd="ecc" name="type" />
    <OMS cd="vafp" name="nat" />
  </OMATP>
  <OMV name="x" />
</OMATTR>
```

Referenzen
werden mit dem `OMR`-Element enkodiert. Dieses Element erhält lediglich das Attribut `href`, welches die URI beinhaltet, mit der das entsprechende OpenMath Objekt referenziert wird. OpenMath-Objekte, die durch ein `OMR`-Element dargestellt

werden, sind Kopien des referenzierten Objektes. Die Kopie und das referenzierte Objekt sind strukturell jedoch nicht identisch! Bespielsweise kann der OpenMath Ausdruck

$$\begin{aligned}&\textbf{application}(\quad f,\\ &\qquad\qquad \textbf{application}(f, \textbf{application}(f,a,a), \textbf{application}(f,a,a)),\\ &\qquad\qquad \textbf{application}(f, \textbf{application}(f,a,a), \textbf{application}(f,a,a)))\end{aligned}$$

auf zwei verschiedene Arten enkodiert werden (Listing 4.8).

Listing 4.8: Referenzierung von OpenMath Objekten

```
<OMOBJ version="2.0">                    <OMOBJ version="2.0">
  <OMA>                                    <OMA>
    <OMV name="f" />                         <OMV name="f" />
    <OMA>                                    <OMA id="t1">
      <OMV name="f" />                         <OMV name="f" />
      <OMA>                                    <OMA id="t11">
        <OMV name="f" />                         <OMV name="f" />
        <OMV name="a" />                         <OMV name="a" />
        <OMV name="a" />                         <OMV name="a" />
      </OMA>                                   </OMA>
      <OMA>                                    <OMR href="#t11" />
        <OMV name="f" />
        <OMV name="a" />
        <OMV name="a" />
      </OMA>
    </OMA>                                   </OMA>
    <OMA>                                    <OMR href="#t1" />
      <OMV name="f" />
      <OMA>
        <OMV name="f" />
        <OMV name="a" />
        <OMV name="a" />
      </OMA>
      <OMA>
        <OMV name="f" />
        <OMV name="a" />
        <OMV name="a" />
      </OMA>
    </OMA>
  </OMA>
</OMOBJ>                                 </OMOBJ>
```

In der Verwendung von `OMR`-Elementen muss eine Bedingung berücksichtigt werden: Eine OpenMath-Element darf sich nicht selbst enthalten! Diese Bedingung wird in der OpenMath-Gemeinde der *Acyclicity Constraint* genannt. Neben dem *Acyclicity Constraint* ist noch ein weiterer Punkt bezüglich Referenzierung von gebundenen Variablen zu berücksichten: Namenskonflikte (variable capturing). Da ein `OMR`-Element lediglich ein syntaktischer Referenzierungsmechanismus ist, werden hierbei Namenskonflikte zwischen gebundenen Variablen, oder man könnte auch sagen semantische Referenzierungen, nicht berücksichtigt. Bespielsweise könnte der OpenMath-Ausdruck

$$\begin{aligned}&\textbf{binding}(\ \lambda,\ X,\ \textbf{application}\ f,\\ &\qquad\qquad\qquad\qquad\textbf{binding}(\ \lambda,\ X,\ \textbf{application}(g, X)),\\ &\qquad\qquad\qquad\qquad\textbf{application}(g, X))\end{aligned}$$

wie in Listing 4.9 dargestellt enkodiert werden. Dort kann man erkennen, dass der Term $\textbf{application}(g, X)$ zweifach vorkommt. Zum einen in der expliziten Form, mit der `id` "orig" und zum anderen in der impliziten Form, mit der `id` "copy". In der originalen Version wird die Variable X durch den äußeren Bindungs-Operator mit der `id` "outer" gebunden. Die Kopie dagegen bindet die Variable X durch den inneren Bindungs-Operator mit der `id` "inner". Es kommt somit zu einem Namenskonflikt!

Listing 4.9: Namenskonflikte durch Referenzierung

```
<OMBIND id="outer">
  <OMS cd="fns1" name="lambda" />
  <OMBVAR><OMV name="X" /></OMBVAR>
  <OMA>
    <OMV name="f" />
    <OMBIND id="inner" />
      <OMS cd="fns1" name="lambda" />
      <OMBVAR><OMV name="X" /></OMBVAR>
      <OMR id="copy" href="#orig" />
    </OMBIND>
    <OMA id="orig"><OMV name="g" /><OMV name="X" /></OMA>
  </OMA>
</OMBIND>
```

Wie schon erwähnt kann man dieses Problem sehr leicht mit der α-Umformung umgehen, so dass der Ausdruck dann beispielsweise folgendermaßen dargestellt werden kann

$$\begin{array}{l}\textbf{binding}(\ \lambda,\ X,\ \textbf{application}\ \ f,\\ \qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{binding}(\ \lambda,\ Y,\ \ \textbf{application}(g,Y)),\\ \qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{application}(g,X))\end{array}$$

Mit dem `OMR` Element lassen sich somit, ohne Berücksichtigung dieses Sachverhaltes, schnell semantische Fehler in einen Ausdruck einbauen[6]. Man sollte sich dessen immer bewußt sein, wenn man diese Art des sogenannten "structure sharing" in OpenMath einsetzt.

[6]Semantische Fehler werden nicht von XML-Parsern erkannt! XML-Parser können lediglich Fehler syntaktischer Natur erkennen. Es liegt am Benutzer, solche Namenskonflikte zu vermeiden bzw. aufzulösen

4.3.4 OpenMath Xml-Enkodierung Übersicht

In diesem Abschnitt wird eine Übersicht über die Xml Enkodierung von OpenMath-Objekten dargestellt. Die Bedeutung der einzelnen Tags wurde in Unterabschnitt 4.3.3 erläutert.

```
# OpenMath object constructor
OMOBJ            ::= element OMOBJ { compound.attributes, attribute version { xsd:string }?, omel }

# Elements which can appear inside an OpenMath object
omel             ::= OMS | OMV | OMI | OMB | OMSTR | OMF | OMA | OMBIND | OME | OMATTR | OMR

# things which can be variables
omvar            ::= OMV | attvar
attvar           ::= element OMATTR { common.attributes,(OMATP , (OMV | attvar))}
cdbase           ::= attribute cdbase { xsd:anyURI }?

# attributes common to all elements
common.attributes  ::= (attribute id { xsd:ID })?

# attributes common to all elements that construct compount OM objects.
compound.attributes ::= common.attributes,cdbase

# symbol
OMS              ::= element OMS { common.attributes,
                                   attribute name { xsd:NCName },
                                   attribute cd { xsd:NCName },
                                   cdbase }

# variable
OMV              ::= element OMV { common.attributes, attribute name { xsd:NCName } }

# integer
OMI              ::= element OMI { common.attributes,
                                   xsd:string {pattern = "\s*(-\s?)?[0-9]+(\s[0-9]+)*\s*"}}

# byte array
OMB              ::= element OMB { common.attributes, xsd:base64Binary }

# string
OMSTR            ::= element OMSTR { common.attributes, text }

# IEEE floating point number
OMF              ::= element OMF { common.attributes,
                                   ( attribute dec { xsd:double } |
                                     attribute hex { xsd:string {pattern = "[0-9A-F]+"}}) }

# apply constructor
OMA              ::= element OMA { compound.attributes, omel+ }

# binding constructor
OMBIND           ::= element OMBIND { compound.attributes, omel, OMBVAR, omel }

# variables used in binding constructor
OMBVAR           ::= element OMBVAR { common.attributes, omvar+ }

# error constructor
OME              ::= element OME { common.attributes, OMS, (omel|OMFOREIGN)* }

# attribution constructor and attribute pair constructor
OMATTR           ::= element OMATTR { compound.attributes, OMATP, omel }
OMATP            ::= element OMATP { compound.attributes, (OMS, (omel | OMFOREIGN) )+ }
```

```
# foreign constructor
OMFOREIGN       ::= element OMFOREIGN { compound.attributes,
                                        attribute encoding {xsd:string}?,
                                        (omel|notom)* }

# Any elements not in the om namespace (valid om is allowed as a descendant)
notom           ::= (element * - om:* {attribute * { text }*,(omel|notom)*} | text)

# reference constructor
OMR             ::= element OMR { common.attributes,attribute href { xsd:anyURI }}
```

4.4 Erweiterungen zu OpenMath

Im Folgenden werden die für diese Arbeit relevanten OMDoc-Erweiterungen zu OpenMath näher erläutert. Ein Überblick der Grammatik wird in Abschnitt 4.5 geliefert. Um die Spezifikation der vollständigen OMDoc Grammatik zu erlangen, wird der Leser auf [OMD] verwiesen.

Im Gegensatz zu OpenMath, wo die Konzentration auf den atomaren mathematischen Objekten liegt, geht OMDoc einen Schritt weiter. Hier wird die Möglichkeit eröffnet, nicht nur die Formel selbst zu repräsentieren und zu annotieren, sondern auch Aussagen, Erklärungen, Beschreibungen, Beispiele und Annahmen über solche Formeln zu formulieren.

4.4.1 Mathematische Texte

OMDoc verwendet Commented Mathematical Property (`CMP`) Elemente, um darin einen beliebigen Text zu kapseln. Ein `CMP` Element darf somit auch beliebige andere OMDoc-Elemente enthalten. Die hauptsächliche Anwendung ist, eine informelle Beschreibung für mathematische Aussagen zu geben. Mit dem optionalen `xml:lang` kann die Sprache angegeben werden, in der der Text verfasst wurde. Auf diese Weise ist es möglich, durch mehrere aufeinanderfolgende `CMP`-Elemente, ein Dokument mehrsprachig aufzubereiten. Die Angabe der Sprache wird nach Xml in dem ISO 639 Standard kodiert (`en` := Englisch, `de` := Deutsch, ...). Der Standardwert für dieses optionale Attribut ist `en`. Die Angabe mehrerer aufeinanderfolgender `CMP`s gleicher Sprachen ist nicht erlaubt. Listing 4.10 zeigt ein Beispiel für ein multi-linguale `CMP` Gruppe.

Listing 4.10: Multi-Linguale OMDoc Dokumente

```
<CMP>
  A Function <OMOBJ xml:id="func"><OMV name="f"/></OMOBJ> is monotone if,
  whenever <OMOBJ xml:id="leqvar"><OMA><OMS cd="relation1" name="leq">
  <OMV name="x"/><OMV name="y"/></OMA></OMOBJ>, then
  <OMOBJ xml:id="leqfunc"><OMA><OMS cd="relation1" name="leq"><OMA>
  <OMV name="f"><OMV name="x"/></OMA><OMA><OMV name="f"/>
  <OMV name="y"/></OMA></OMA></OMOBJ>
</CMP>
<CMP xml:lang="de">
  Eine Funktion <OMOBJ><OMR href="#func" /></OMOBJ> ist monoton,
  genau dann wenn <OMOBJ><OMR href="#leqvar" /></OMOBJ> auch <OMOBJ>
  <OMR href="#leqfunc"/></OMOBJ> gilt.
</CMP>
```

An diesem Beispiel kann man auch sehr gut den enormen Vorteil von *structure sharing* ersehen. Lediglich die informellen Teile müssen erneut geschrieben bzw. übersetzt werden. Sämtliche formalen Teile können zwischen den CMPs "geshared" werden.

Der Gegenpart zu einem CMP-Element, oder einer Gruppe von CMP-Elementen, ist ein Formal Mathematical Property (FMP) Element. Mit diesem Element kann rein formal der in einem CMP beschriebene Text ausgedrückt werden. Der Inhalt eines FMP-Elements sind wohlgeformte OpenMath-Ausdrücke. Die Bedeutung des Inhaltes eines CMP-Elements und des direkt darauffolgendem FMP-Elements müssen gleich sein[7]. Auch können hier wieder eine ganze Gruppe von FMP-Elementen angelegt werden, um den Inhalt eines oder mehrerer CMP-Elemente formal zu beschreiben, doch müssen sich die formalen Beschreibungen untereinander entsprechen. So werden in Listing 4.11 zwei FMP-Elemente angegeben, die die Eigenschaft der Monotonie einer Funktion zum Ausdruck bringen. Mit dem zusätzlichen `logic` Attribut kann die zugrunde liegende Logik angegeben werden[8].

Listing 4.11: Multi-Logik OMDoc Dokumente

```
... hier stehen die entsprechenden CMPs ...
<FMP logic="fol">f monoton ⇔ (∀x, y.x ≤ y ⇒ f(x) ≤ f(y))</FMP>
<FMP logic="hol">monton = λf, x, y.impl(x ≤ y, f(x) ≤ f(y))</FMP>
```

FMPs können auch dazu verwendet werden logische Sequenzen zu modellieren. OMDoc stellt hierfür die FMP Kind-Elemente `assumption` und `conclusion` zur Verfügung. Unter dem Begriff Sequenz verstehen wir "Liste von Formeln $\vdash$ Liste von Formeln", wobei eine oder auch beide Listen leer sein können. In der Sequenz $\Gamma \vdash \Delta$ heißt Γ das Antezedens und Δ das Sukzedens. Die inhaltliche Interpretation von $A_1, \ldots A_n \vdash B_1, \ldots B_k$ ist, wenn alle A_i wahr sind, so ist ein B_j wahr. Eine Aussage innerhalb eines `assumption` bzw. `conclusion` Elements muss entweder direkt durch einen gültigen OpenMath-Ausdruck, oder innerhalb eines CMP-Elements formuliert werden. Die Elemente `assumption` bzw. `conclusion` können mit dem optionalen `xml:id` versehen werden, so dass auch hier ein "structure sharing" ermöglicht wird. Insbesondere innerhalb von Beweisen wird sich dies als großer Vorteil erweisen, da solche Sequenzen meist invariant bezüglich eines Beweises sind. Mit dem zusätzlichen `inductive` Attribut kann eine Annahme explizit als Induktionshypothese gekennzeichnet werden. Zur Veranschaulichung wird in Listing 4.12 folgende Sequenz

$$x \neq 0,$$
$$\forall z' : nat, y' : nat.(x + y') + z' = x + (y' + z')$$
$$\vdash \forall x : nat, y : nat, z : nat.(x + y) + z = x + (y + z)$$

[7] Die Überprüfung der Gleichheit der Semantik von einem CMP- und einem darauffolgenden FMP-Element wird nicht überprüft. Analog zu der Bemerkung hinsichtlich *Acyclicity Constraints*, liegt es auch hier am Benutzer, solche Restriktionen einzuhalten.

[8] Zur besseren Lesbarkeit wird in diesem sowie in machen folgenden Beispielen die OpenMath-Syntax durch eine kürzere mathematische Notation ersetzt.

dargestellt. Mit dem OMDoc Element `ref` wird hier die Konklusion referenziert, die bereits an einer anderen Stelle des Dokuments definiert worden ist. In dem notwendigen `xref` Attribut wird die eindeutige `xml:id` des referenzierten Elements angegeben.

Listing 4.12: Sequenzen in OMDoc

```
<FMP>
  <assumption xml:id="vf65b3c481-fdcb-4659-a624-9ff55c202490">
    x ≠ 0
  </assumption>
  <assumption xml:id="vf223f89f0-d751-4723-9025-a89d9df78f24" inductive="yes">
   ∀z' : nat, y' : nat.(x + y') + z' = x + (y' + z')
  </assumption>
  <conclusion xml:id="vf3ac522c9-fba4-4012-9ecb-0b8b4ac56dab">
    <ref xref="#vf6c641dc6-79c3-4baf-92ef-40f23d192175"/>
  </conclusion>
</FMP>
```

4.4.2 Mathematische Theorien

Ein System von bewiesenen Sätzen, die aus einem System von Axiomen folgen, nennt man eine mathematische Theorie. Ein Axiom ist ein Aussage, deren Wahrheitsgehalt nicht bewiesen wird, sondern zugrunde gelegt, angenommen wird. Ein System von Axiomen sollte widerspruchsfrei sein. Die Mathematik bedient sich also vornehmlich Theorien, um Aussagen und Konzepte kontextuell zusammen zufassen. OMDoc bietet mit dem `theory`-Element direkte Unterstützung für dieses Vorgehen. Eine OMDoc Theorie trägt als Attribut eine eindeutige `xml:id`. Mit diesem Zusatz ist es möglich, auf diese Theorie zu verweisen und darin enthaltenen Elemente zu verwenden. Dies spiegelt den Nutzen von CDs wieder, als welche OMDoc Theorien auch verwendet werden können. Zusatzinformationen über eine Theorie können in Form von DC Meta-Daten an das `theory`-Element angefügt werden. Zu dieser Gruppierungsmöglichkeit kommt eine Fülle an Begriffen in OMDoc hinzu, mit denen mathematische Textbausteine explizit mit deren Semantik ausgezeichnet werden können. So wird es mit den in diesem Abschnitt vorgestellten Tags dem Autor eines Dokuments ermöglicht, explizite Angaben über das "Was" einzelner Textbausteine anzugeben. Insbesondere hinsichtlich mathematische Aussagen können konkrete Angaben vorgenommen werden, um was für Aussagen es sich handelt. OMDoc unterscheidet hier im wesentlichen zwischen zwei Gruppen von Aussagen. Die erste Gruppe besteht aus sogenannten *konstitutiven* Aussagen. Solche Aussagen bestimmen den momentanen Kontext oder hinsichtlich der Mathematik vielmehr die momentane Theorie. Mit anderen Worten, Änderungen an diesen Aussagen verändern zugleich die Bedeutung der durch sie bestimmten Theorie. Die zweite Gruppe besteht aus den *nicht-konstitutiven* Aussagen. Solche Aussagen modellieren lediglich explizit mathematische Objekte innerhalb der Theorie, die durch die *konstitutiven* Aussagen implizit gegeben sind. Theorie-konstitutive Elemente müssen somit als

Kind-Elemente in dem `theory`-Element enthalten sein und es dürfen nicht ohne weiteres weitere konstitutive Elemente in eine abgeschlossene Theorie hinzugefügt werden, da dadurch die Bedeutung der Theorie verändert werden könnte. Nicht-konstitutive Elemente können auch ausserhalb einer Theorie definiert werden, mit einer Referenz auf diejenige Theorie, die ihre Semantik bestimmt. Als Beispiel für die *konstitutiven* Elemente der Gruppentheorie stelle man sich die drei wohlbekannten Gruppenaxiome Φ vor. Ein mögliches *nicht-konstitutives* Element, welches explizit dieser Theorie hinzugefügt werden kann, wäre dann ein Lemma φ über die Existenz des Links-Inversen, so dass $\Phi \models \varphi$ gilt.

4.4.2.1 Theorie-konstitutive Aussagen

In OMDoc werden fünf Typen von Theorie-konstitutiven Aussagen unterstützt: Axiome, Symboldeklarationen, Typdeklarationen, Definitionen und ADTs.

`axiom`

In der Theorie der natürlichen Zahlen ist es üblich, sämtliche Eigenschaften der Menge $\mathbb{N}$, über die fünf Peano-Axiome nachzuweisen. Möchte man so zum Beispiel nachweisen, dass es eine eindeutig bestimmte Funktion mit den Eigenschaften der Addition gibt, muss man nur nachweisen, dass das Funktionalgleichungssystem

$$k + 0 = 0 \qquad \text{und} \qquad k + s(l) = s(k + l)$$

genau eine zweistellige Funktion + als Lösung hat. Ohne auf diesen zweigeteilten[9] Beweis weiter einzugehen, betrachten wir hier das nächste theorie-konstitutive OMDoc-Element, mit dem die Beschreibung u.a. der fünf Peano-Axiome in OMDoc ermöglicht wird. Das `axiom`-Element hat ein notwendiges Attribut `name`, mit welchem es eindeutig innerhalb einer Theorie referenziert werden kann. Der Inhalt eines `axiom`-Elements kann aus `CMP`- und/oder `FMP`-Elementen bestehen (Listing 4.13).

Listing 4.13: 1. und 2. Peano Axiom in OMDoc

```
<axiom name="pa1">
  <CMP xml:lang="de">Null ist eine natrürliche Zahl</CMP>
  <FMP>
    <OMA>
      <OMS cd="set1" name="in" />
      <OMV name="0"/>
      <OMS cd="Nat-base" name="Nat"/>
    </OMA>
  </FMP>
</axiom>
```

[9] Zum einen muss man die Eindeutigkeit und zum anderen die Existenz nachweisen.

```
<axiom name ="pa2">
  <CMP xml:lang="de">
    Der Nachfolger jeder natürlichen Zahl ist ein natrürliche Zahl
  </CMP>
  <FMP>
    <OMBIND>
    <OMS cd="pl1" name="forall"/>
      <OMBVAR>
        <OMATTR>
          <OMATP>
            <OMS cd="simpletypes" name="type"/>
            <OMS cd="Nat-base" name="Nat"/>
          </OMATP>
          <OMV name="k"/>
        </OMATTR>
      </OMBVAR>
      <OMA>
        <OMS cd="set1" name="in" />
        <OMA>
          <OMS cd="Nat-base" name="s" />
          <OMV name="k" />
        </OMA>
        <OMS cd="Nat-base" name="Nat"/>
      </OMA>
    </OMBIND>
  </FMP>
</axiom>
```

`symbol`

Das `symbol`-Element deklariert ein neues (mathematisches) Symbol. In dem notwendigen `name`-Attribut wird das Symbol innerhalb einer Theorie eindeutig bestimmt. Da sich der Wert dieses Attributs unter anderem in seiner Anwendung als Wert des Attributs `name` des OpenMath-Elements `OMS` wiederfindet, muss dieser den Produktionsregeln aus Unterabschnitt 4.3.2 für Symbol-Namen genügen. Die Kind-Elemente des `symbol`-Elements bestehen aus einer Menge von `type`-Elementen. Durch diese Menge von `type`-Elementen kann in verschiedenen Typsystemen die Signatur des Symbols angegeben werden. In Listing 4.14 ist beispielsweise die Symboldeklaration der Funktion + angegeben. Auf die genaue Bedeutung des `type`-Elements und dessen Kind-Elemente wird in dem nächsten Punkt weiter eingegangen.

`type`

Wie schon in der Deklaration von Symbolen angesprochen, bietet OMDoc mit dem `type` Element die Unterstützung zur expliziten Typdeklaration. Eine Typde-

klaration denotiert die Semantik eines Symbols oder eines Terms innerhalb einer Domäne. Eine übliche Schreibweise für eine Typdeklaration ist $s : t$. Hiermit wird zum Ausdruck gebracht, dass das Symbol oder der Term s seine denotationale Semantik in der Domäne t findet. In OMDoc wird das `type`-Element verwendet, um eine solche Typdeklaration zu beschreiben. In dem optionalen `system` kann das zugrundeliegende Typsystem angegeben werden. Die Kind-Elemente des `type`-Elements sind gültige OpenMath-Ausdrücke, mit denen der Typ bzw. die Signatur des Symbols angegeben wird. In Listing 4.14 wird die Signatur des Symbols + mit $+ : nat \to nat \to nat$ angegeben.

Listing 4.14: Deklaration von + in OMDoc

```
<symbol name="+">
  <type system="simpletypes">
    <OMOBJ xmlns="http://www.openmath.org/OpenMath">
      <OMA>
        <OMS cd="simpletypes" name="funtype"/>
        <OMS cd="vafp" name="nat"/>
        <OMS cd="vafp" name="nat"/>
        <OMS cd="vafp" name="nat"/>
      </OMA>
    </OMOBJ>
  </type>
</symbol>
```

`definition`

Um das hier als vorletztes aufgeführte theorie-konstitutive Element, die Definitionen, zu beschreiben, verwendet OMDoc das `definition` Element (Listing 4.15). In den beiden notwendigen Attributen `name` und `for` werden zum einen der Name der Definition und das zu definierende Symbol angegeben. Das `definition`-Element unterstützt eine Vielzahl von verschiedenen Mechanismen, um eine konkrete Definition eines Symbols anzugeben. Der hier verwendete Mechanismus und OMDoc Standard-Mechanismus ist der sogenannte `simple`-Mechanismus, in welchem die Definition in Form von gültigen OpenMath-Ausdrücken angeben wird. Mit dem zusätzlichen `existence` Attribut wird die Referenz auf eine Behauptung (assertion) angegeben, die durch einen Beweis sicherstellt, dass dieses Symbol wohldefiniert ist, bzw. im Fall von Funktionssymbolen, dass die Funktion terminiert.

Listing 4.15: Definition von + in OMDoc

```
<definition name="+.def" for="#+"
            type="simple"
            existence="#vf7a44848a-2eb3-4e8f-98e3-a7e4bb5a7664">
  <OMOBJ xmlns="http://www.openmath.org/OpenMath">
    <OMBIND>
      <OMS cd="vafp" name="function"/>
```

```
      <OMBVAR>
        <OMATTR>
          <OMATP>
            <OMS cd="simpletypes" name="type"/>
            <OMS cd="vafp" name="nat"/>
          </OMATP>
          <OMV name="x"/>
        </OMATTR>
        <OMATTR>
          <OMATP>
            <OMS cd="simpletypes" name="type"/>
            <OMS cd="vafp" name="nat"/>
          </OMATP>
          <OMV name="y"/>
        </OMATTR>
      </OMBVAR>
      <OMA>
        <OMS cd="vafp" name="if"/>
        <OMA><OMS cd="vafp" name="is_zero"/><OMV name="x"/></OMA>
        <OMV name="y"/>
        <OMA>
          <OMS cd="vafp" name="succ"/>
          <OMA>
            <OMS cd="VeriFun" name="+"/>
            <OMA>
              <OMS cd="vafp" name="pred"/>
              <OMV name="x"/>
            </OMA>
            <OMV name="y"/>
          </OMA>
        </OMA>
      </OMA>
    </OMBIND>
  </OMOBJ>
</definition>
```

`adt`

Wie die meisten Spezifikationssprachen für mathematische Theorien, ermöglicht auch OMDoc die Definition von Abstract Data Types (ADTs). Eine ADT ist eine Ansammlung von Symbolen und Axiomen, die zur Erzeugung von mathematischen Objekten verwendet werden können. Es wird hierbei zwischen zwei Funktionssymbolen unterschieden. Zum einen existieren Konstruktoren, um den jeweiligen ADT zu erzeugen, zum anderen kann ein Konstruktor Selektoren ent-

halten, mit denen auf die einzelnen Komponenten des jeweiligen Konstruktors zugegriffen werden kann. Mathematische Objekte, die durch Konstruktoren erzeugt werden können, werden als Sorten bezeichnet. Eine solch erzeugte Sorte wird als "frei" angesehen, wenn keine Beziehung zwischen den einzelnen Konstruktoren besteht. Mit anderen Worten, zwei mathematische Objekte, erzeugt durch unterschiedliche Konstruktorterme, können niemals gleich sein. Im anderen Fall nennt man die Sorte "erzeugend".

In OMDoc wird das Element `adt` verwendet, um einen abstrakten Datentyp zu spezifizieren, der wiederum aus mehreren Sorten bestehen kann. Zur Veranschaulichung wird in Listing 4.16 die OMDoc Repräsentation der generischen Sorte

$$list[@ITEM] \equiv \ \emptyset, \quad :: (hd : @ITEM, tl : list[@ITEM])$$

dargestellt und im Folgenden weiter erläutert.

Listing 4.16: Definition der ADT `list[@ITEM]` in OMDoc

```
<adt xml:id="vf7bcddbfb-6860-4f9c-b2b7-c45409ed6b6a.adt"
     parameters="ITEM">
  <sortdef name="list" type="free">
    <constructor name="∅"/>
    <constructor name="::">
      <argument>
        <type system="simpletypes">
          <OMOBJ xmlns="http://www.openmath.org/OpenMath">
            <OMV name="ITEM"/>
          </OMOBJ>
        </type>
        <selector name="hd" total="yes"/>
      </argument>
      <argument>
        <type system="simpletypes">
          <OMOBJ xmlns="http://www.openmath.org/OpenMath">
            <OMA>
              <OMS cd="VeriFun" name="list"/>
              <OMV name="ITEM"/>
            </OMA>
          </OMOBJ>
        </type>
        <selector name="tl" total="yes"/>
      </argument>
    </constructor>
  </sortdef>
</adt>
```

Jede ADT in OMDoc enthält das `xml:id` Attribut, so dass auch hier eine eindeutige Referenzierung erfolgen kann. Mit dem optionalen Attribut `parameters` können der ADT zu verwendende Argumente übergeben werden, wie beispielsweise Typvariablen zur Erstellung von generischen ADTs. Die einzelnen Typvariablen werden in Formen einer durch Leerzeichen getrennten Liste als Wert dieses Attributs eingesetzt. Die in einer ADT enthaltenden Sorten werden in OMDoc durch das `sortdef`-Element gekennzeichnet. Nach obiger Einführung ist offensichtlich, dass OMDoc für dieses Element die Attribute `name` und `type` zur Verfügung stellt. Mit dem Attribut `name` kann der eindeutige Name der Sorte innerhalb der Theorie angeben werden und das Attribut `type` gibt den Typ der Sorte an. Die eine Sorte ausmachenden Konstruktoren werden durch das `sortdef` Kind-Element `constructor` dargestellt. Der Name eines Konstruktors wird in dem entsprechenden Attribut `name` gespeichert. Ein Konstruktor kann in Abhängigkeit seiner Argumente mehrere Selektoren besitzen. Diese werden jeweils durch ein `argument`-Element als Kind-Elemente an den entsprechenden Konstruktor angehängt. Die Kind-Elemente eines `argument` Elements bestehen aus einem Paar. Die erste Komponente stellt den Argumenttyp des Konstruktors an dieser Position dar. Die zweite Komponente repräsentiert den Selektor selbst. Der Typ des Arguments wird in gewohnter Manier mit dem `type`-Element beschrieben. Ein Selektor wird durch das `selector`-Element ausgezeichnet. Ein `selector` enthält das notwendige Attribut `name` für den Namen und ein optionales Attribut `total`. Mit dem letzten Attribut kann die Totalität des Selektors zum Ausdruck gebracht werden.

Ein zentrales Element für die Wiederverwendung ist das `import`-Element; womit die konstitutiven Elemente bereits bestehender Theorien in eine neue Theorie übernommen werden können (Listing 4.17). Das `import`-Element beinhaltet u.a das `from` Attribut. Hier wird in Form einer URI die "Quell-Theorie" angegeben. Die "Ziel-Theorie" erbt auf diese Weise automatisch alle konstitutiven Elemente der Quell-Theorie. Das `import`-Element darf nicht so verwendet werden, dass es zu einem Importzyklus kommt (d.h. dass sich eine Theorie selbst importiert).

Listing 4.17: Theorie Import in OMDoc

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE omdoc SYSTEM "omdoc-vf.dtd">
<omdoc xml:id="VeriFun3.0-2005-11-01"
  xmlns="http://www.mathweb.org/omdoc"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <metadata>
    <dc:date>2005-11-07T11:44:25</dc:date>
  </metadata>
  <theory xml:id="VeriFun">
    <imports from="vafp"/>
  </theory>
</omdoc>
```

4.4.2.2 Nicht theorie-konstitutive Aussagen

In diesem Abschnitt wird die Auszeichnung von mathematischen Aussagen bezüglich einer Theorie behandelt, die implizit durch die darin enthaltenen konstitutiven Aussagen gegeben sind. Durch diese Aussagen werden jedoch keine weiteren Informationen der Theorie hinzugefügt, sondern die impliziten Aussagen werden explizit angegeben. Dass diese expliziten Aussagen auch wirklich den impliziten entsprechen, wird mit mathematischen Beweisen untermauert. Zu den nicht theorie-konstitutiven Aussagen, die in OMDoc eine Auszeichnung erhalten, gehören u.a. die sogenannten Behauptungen (assertion). Mit dem `assertion`-Element werden alle nicht axiomatischen Behauptungen in OMDoc markiert. Mit dem optionalem Attribut `type` kann der Typ der Behauptung näher beschrieben werden. Neben den Typen `theorem` und `lemma` unterstützt auch hier OMDoc eine große Vielzahl von weiteren Typangaben, auf die jedoch hier nicht weiter eingegangen wird. Der Typ `theorem` kennzeichnet einen mathematischen Satz mit einem angehängten Beweis. Der Typ `lemma` stellt einen mathematischen Hilfssatz dar, der gleichwohl durch einen angehängten Beweis nachgewiesen wird. Damit auch diese Elemente referenziert werden können, enthalten auch sie das optionale `xml:id` Attribut. Der Inhalt einer Assertion wird mit `CMP` und/oder `FMP` Elementen beschrieben. In Listing 4.18 wird beispielsweise die Terminierungsaussage der Funktion + dargestellt. Die OMDoc-Repräsentation von Beweisen von mathematischen Aussagen wird in dem nächsten Abschnitt behandelt.

Listing 4.18: Zusicherung der Termination von + in OMDoc

```
<assertion xml:id="vf7a44848a-2eb3-4e8f-98e3-a7e4bb5a7664" type="theorem">
  <CMP>
    <OMOBJ xmlns="http://www.openmath.org/OpenMath">
      <OMS cd="VeriFun" name="+"/>
    </OMOBJ>terminates.
  </CMP>
</assertion>
```

4.4.3 Mathematische Beweise

Der Beweis spielt eine überragende Rolle in der Logik und Mathematik, in der Kriminologie, Rechtswissenschaft und Rechtsprechung und - etwas anders gedeutet - in allen, besonders in den empirischen Wissenschaften, und natürlich in der Erkenntnis- und Wissenschaftstheorie, aber auch natürlich - was oft übersehen wird - im ganz normalen Lebensalltag. In diesem Abschnitt wird die Idee des Beweises im Bereich der Mathematik betrachten und dessen Repräsentation in OMDoc.

Zum Wesen der Mathematik gehört, alles zu beweisen, was behauptet oder benutzt wird. Doch was genau versteht man unter dem Begriff "mathematischer Beweis"? Ein mathematischer Beweis einer Aussage A aus einem Axiomensystem Ax besteht aus einer Folge

von *Schlüssen*, welche von Axiomen aus Ax oder bereits hergeleitetenden Aussagen zu neuen Aussagen führen und schließlich in A enden. Ein Beweis ist somit eine Sequenz von Anwendungen von Schlussregeln, die ihren Ursprung in einem formal definierten Kalkül finden. Die meisten solcher Kalküle haben gemein, dass sie selbst für Beweise von einfachen Sätzen enorm ausarten können. Der Vorteil jedoch darin ist, dass somit formal vollständige Beweise vorliegen, die leicht durch computergestützte Beweisverwahren verifiziert werden können.

In OMDoc wird ein formaler Beweis mit dem Element `proof` eingeleitet. Das `proof` Element enthält optional die Attribute `xml:id` und `for`. Mit dem ersten Attribut wird dem Beweis eine eindeutige `ID` zugewiesen, so dass auch ein Beweis als ganzes referenziert werden kann. In dem `for` wird eine eindeutige URI eingetragen, auf dessen OMDoc-Element sich der Beweis bezieht. Es ist damit auch möglich für eine Behauptung mehrere Beweise anzugeben. Solche OMDoc Elemente dürfen *Assertions* oder einzelne Beweisschritte innerhalb des selben Beweises sein. Durch letztere Funktionalität wird eine hierarchische Anordnung von Beweisen ermöglicht, indem ein Beweisschritt wieder mehrere Teilbeweise beinhalten kann. Einzelne Beweisschritte werden mit dem `derive`-Element als Kind-Element eines Beweises gekennzeichnet. Um nicht nur einen Beweis als ganzes referenzieren zu können, enthält auch dieses Tag das optionale `xml:id` Attribut. Da zu jedem Beweisschritt eine Sequenz in Form von Hypothesen, Induktionshypothesen und der Konklusion assoziiert sein kann, können diese Informationen als `CMP`- und/oder `FMP`-Kind-Elemente an den jeweiligen Beweisschritt angehängt werden. Die Methode, mit der ein Beweisschritt ausgeführt wird, wird als weiteres Kind-Element an das `derive`-Element hinzugefügt. Die verwendete Methode wird durch das `xref`-Attribut referenziert. Die zu einer Methode gehörigen, zusätzlichen Informationen können als Kind-Elemente, gekapselt in `OMOBJ`-Elementen, direkt an die jeweilige Methode angehängt werden.

Um diesen einfachen und doch sehr ausdrucksstarken Mechanismus zu verdeutlichen, wird im folgenden ein Auszug eines Beweises der Kommuntativität der Funktion + angegeben. Dieser Beweis wurde mit dem Theorembeweiser ✓`eriFun` durchgeführt und danach automatisch durch das System in die angegebene OMDoc-Repräsentation überführt. Zur besseren Lesbarkeit wurden zum einen die zusätzlichen Informationen, die an eine Methode angehängt werden können, nachträglich entfernt und zum anderen wurden zusätzliche Kommentare eingefügt und der Inhalt der `FMP`-Elemente erneut in einer leichter zu lesenden mathematischen Notation dargestellt. Bis auf diese nachträglichen Änderungen wurde die dargestellte OMDoc Repräsentation des Beweis vollständig von ✓`eriFun` erzeugt.

```
<assertion xml:id="vf99c19b40-bbbb-4072-8056-e8d51644cc9f" type="lemma">
  <FMP>∀x, y : ℕ.x + y = y + x</FMP>
</assertion>

<proof for="#vf99c19b40-bbbb-4072-8056-e8d51644cc9f" xml:id="vf0bc5bcb5-9bda-4fdd-83a2-7d0e09b99fa8">
 <derive xml:id="vf8e884335-42a3-4cf6-ae47-cf4b21a3bcd2">
   <FMP>
     <conclusion xml:id="vff6f04a36-211f-47a9-9892-e90990f4172d">x + y = y + x</conclusion>
   </FMP>
```

```
<method xref="#Induction">
  Parameter der Induction-Taktik eingeschlossen in OMOBJ Elemente
13
  BASE CASE OF INDUCTION
  <proof xml:id="vffb52dd00-33ce-4753-9829-5cf578aaf71b">
    <derive xml:id="vf72d8f55d-5406-4591-b861-b50559db4e04">
      <FMP>
18      <assumption xml:id="vf08ac1b94-410e-4c2b-b53a-1ebb90c0c3a8">
          x = 0
        </assumption>
        <conclusion xml:id="vf389bf740-e56a-4c24-aa71-d8dc1438d2dd">
          <ref xref="#vff6f04a36-211f-47a9-9892-e90990f4172d"/>
23      </conclusion>
      </FMP>
      <method xref="#Simplification">
        Parameter der Simplification-Taktik eingeschlossen in OMOBJ Elemente
        <proof xml:id="vf0bbdede2-dc98-4f69-977e-cf1b2fbce554">
28        <derive xml:id="vf9fd5c127-eaf5-41fb-867c-fc6ed35380a1">
            <FMP>
              <assumption xml:id="vfbc59add9-5f7f-4635-86eb-9dc5dc8ab367">
                <ref xref="#vf08ac1b94-410e-4c2b-b53a-1ebb90c0c3a8"/>
              </assumption>
33            <conclusion xml:id="vfe86b783e-de78-495a-bcb3-18f8f3bbfbcd">
                y = y + x
              </conclusion>
            </FMP>
            <method xref="#Move Hypotheses">
38            Parameter der Move Hypotheses-Taktik eingeschlossen in OMOBJ Elemente
              <proof xml:id="vfd14e3c85-34c4-432d-9564-abcbadb2b065">
                <derive xml:id="vf12eedce5-fa2e-4541-a337-8f5e5e38c96c">
                  <FMP>
                    <conclusion xml:id="vf14b47463-4897-4c88-9e25-1c95adac6bd9">
43                    if(x = 0, y = y + x, true)
                    </conclusion>
                  </FMP>
                  <method xref="#Induction">
                    Parameter der Induction-Taktik eingeschlossen in OMOBJ Elemente
48
                    BASE CASE OF INNER INDUCTION
                    <proof xml:id="vf29a7a925-65ec-472c-9da7-78fddf08528b">
                      <derive xml:id="vf7588caf0-3ef9-4a72-8083-1ce3afb1a0df">
                        <FMP>
53                        <assumption xml:id="vfea7434c2-2d58-4e04-bf50-99d52f6053e8">
                            y = 0
                          </assumption>
                          <conclusion xml:id="vf80c9498f-f781-4641-a9af-c7d2f8e21ec3">
                            <ref xref="#vf14b47463-4897-4c88-9e25-1c95adac6bd9"/>
58                        </conclusion>
                        </FMP>
                        <method xref="#Simplification">
                          Parameter der Simplification-Taktik eingeschlossen in OMOBJ Elemente
                          <proof xml:id="vf87bd6397-b5d3-4b22-af06-92ed96bad899">
63                          <derive xml:id="vff65ce8d5-a76d-42f3-8402-e3c90d5f74b7">
                              <method>
                                <premise xref="#true_ax"/>
                              </method>
                            </derive>
68                        </proof>
                        </method>
                      </derive>
                    </proof>
73
```

```
                    STEP CASE OF INNER INDUCTION
                    <proof xml:id="vf7d7580e9−f03d−4af6−be35−c9a34186de68">
                      <derive xml:id="vfb37bc7f3−d607−4dda−ae3a−14137aac3fa2">
                        <FMP>
78                        <assumption xml:id="vf1b1e5e0c−008b−474b−9ecd−a506cc159dae">
                            y ≠ 0
                          </assumption>
                          <assumption xml:id="vfed262e70−c03d−4d4a−af9a−d10ab288834d" inductive="yes">
                            ∀x′ : ℕ.if(x′ = 0, pred(y) = pred(y) + x′, true)
83                        </assumption>
                          <conclusion xml:id="vf3b7e64e8−7ce0−4dbf−9ce8−d26c1917bd8e">
                            <ref xref="#vf14b47463−4897−4c88−9e25−1c95adac6bd9"/>
                          </conclusion>
                        </FMP>
88                      <method xref="#Simplification">
                          Parameter der Simplification-Taktik eingeschlossen in OMOBJ Elemente
                          <proof xml:id="vf913f87d8−c9ec−40ed−8ab0−ee33a63e2100">
                            <derive xml:id="vfbfd57a74−c6c7−4d13−9e6e−aaa12a799ffb">
                              <method>
93                              <premise xref="#true_ax"/>
                              </method>
                            </derive>
                          </proof>
                        </method>
98                    </derive>
                    </proof>
                  </method>
                </derive>
              </proof>
103         </method>
          </derive>
        </proof>
      </method>
    </derive>
108 </proof>

    STAPE CASE OF INDUCTION
    <proof xml:id="vf47b18388−17b2−41fe−bb3a−8cad1b26d8ab">...</proof>
</proof>
```

4.4.4 Präsentationen

In den vorangegangen Abschnitten wurden die für diese Arbeit wesentlichen OMDoc-Tags beschrieben, mit denen es ermöglicht wird, den Inhalt und die Struktur von mathematischen Dokumenten zu kennzeichnen. Jedoch war man sich auch in der Entwicklung von OMDoc darüber im Klaren, dass die Notwendigkeit besteht, diverse mathematische Objekte mit weiteren Informationen annotieren zu können. Insbesondere in der Mathematik werden häufig unterschiedliche Darstellungen und Notationen für ein Symbol gewählt. Auch für die Konvertierung eines OMDoc-Dokuments ist es durchaus sinnvoll, die verschiedenen Symbole mit unterschiedlichen Präsentationen versehen zu können. Somit ist es in OMDoc möglich, allgemeine Style- und Notations-Informationen für mathematische Symbole in eigens dafür vorgesehenen `omstyle`- bzw. `presentation`-Elementen zu spezifizieren. Im Weiteren wird nur auf das `presentation`-Element eingegangen, da es für diese Arbeit irrelevant ist, mit welchen zusätzlichen Style-Informationen ein Symbol versehen ist. Das `presentation`-Element, zur Spezifikation der Notation eines Symbols,

enthält ein `for`-Attribut, dessen Wert eine eindeutige URI als Referenz auf das zu spezifizierende Symbol ist. Mit dem Kind-Element `use` und dem erforderlichen Attribut `format` können für unterschiedliche Systeme alternative Notationen angegeben werden. Zur weiteren Spezifikation der Notation eines Symbols kann das `presentation`-Element mit folgenden Attributen versehen werden:

`role`
: Mit diesem Attribut kann spezifiziert werden, unter welcher Anwendung des Symbols diese Notation zum Tragen kommt. Mögliche Werte sind: `applied` (`OMA`), `binding` (`OMBIND`) und `key` (`OMATTR`).

`fixity`
: Mit diesem Attribut kann die *Fixity* eines Symbols spezifiziert werden. Mögliche Werte sind: `prefix`, `infix`, `infixl`, `infixr` und `postfix`.

`bracket-style`
: Mit diesem Attribut kann der zu verwendende Klammerungs-Style spezifiziert werden. Mögliche Werte sind: `lisp` und `math`.

`precedence`
: Mit diesem Attribut kann die Bindungs-Priorität des Symbols spezifiziert werden. OMDoc orientiert sich hier an dem Prolog-Standard: Kleinere Prioritäten bedeuten eine stärkere Bindung.

`lbrack/rbrack`
: Mit diesen beiden Attributen können die zu verwendeten Klammern für das Symbol spezifiziert werden.

In Listing 4.19 ist eine Spezifikation der Notation für das Symbol + innerhalb von ✓`eriFun` angegeben.

Listing 4.19: Notation-Information von + in OMDoc

```
<presentation for="#vf2f90f0c2-02fc-4140-bf5d-1b8e570e50e0"
              role="applied"
              bracket-style="math"
              precedence="20"
              lbrack="("
              rbrack=")"
               fixity ="infixr">
  <use format="VeriFun">+</use>
</presentation>
```

4.4.5 Interpreted Markup

Die Entwickler von OMDoc sind sich darüber bewußt, dass es Sachverhalte gibt, die mit den zur Verfügung stehenden deskriptiven TAGs nicht zum Ausdruck gebracht werden können. Obwohl diese Menge von wohldefinierten Vokabeln sehr wohl für mathematische Text und Konzepte ausreichend ist, ist man der Überzeugung, um die Verbreitung von OMDoc zu fördern, die Möglichkeit des interpreted Markups[10] bereitzustellen. Mit dem `private` Element können system-spezifische Blöcke in einem OMDoc Dokument mit eingefügt werden. In den `data` Kind-Elementen können, durch die zusätzliche Attributierung mittels `pto`, spezifische Programminstruktionen kodiert werden. Dies hat den großen Vorteil, dass nun immer mehr mathematische Computersysteme OMDoc verwenden, da diese ihre system-spezifischen Daten mit enkodieren können. Es wird somit ermöglicht, Daten zu speichern, die diese System unbedingt zu Rekonstruktion des mathematischen Inhaltes benötigen. In Listing 4.20 sind exemplarisch private Informationen für das ✓`eriFun` System dargestellt. Auf die genau Bedeutung dieser privaten Daten wird in dem nächsten Abschnitt weiter eingegangen (Unterabschnitt 5.1.2).

Listing 4.20: ✓`eriFun` private Information in OMDoc

```
<private xml:id="vfd6c5c713-8614-44ad-b36e-803318ae5623">
  <data pto="VeriFun">Arithmetic</data>
  <data pto="VeriFun">&#xd;
   This folder contains the subtheory of arithmetic &#xd;
   required for the subsequent proofs.&#xd;
  </data>
  <data pto="VeriFun"/>
</private>
<private xml:id="vf921e1318-73ca-414b-97ef-c3fb38ef7c0a">
  <data pto="VeriFun">Plus</data>
  <data pto="VeriFun"/>
  <data pto="VeriFun">Arithmetic</data>
</private>
```

[10] An dieser Stelle möchte der Autor darauf hinweisen, dass in verschiedenen anderen Literaturquellen über Markupsprachen *interpreted* Markup auch als *system specific* Markup bezeichnet wird. Man kann die beiden Begriffe in dieser Arbeit synonym verwenden.

4.5 OMDoc Grammatik

Im Folgenden wird ein Überblick der für diese Arbeit relevanten OMDoc-Grammatikelemente in einer kompakten und intuitiven Schreibweise angegeben. Um einen vollständigen Überblick der OMDoc-Grammatik zu erlangen, wird der Leser auf [Koh06] verwiesen.

```
default namespace omdoc      ::= "http://www.mathweb.org/omdoc"

omdocst.scope.attrib         ::= [a:defaultValue = "global"] attribute scope {"global" | "local"}?
idc.attrib                   ::= attribute name {theory-unique}?, id.attrib
gf.attrib                    ::= attribute generated-from {omdocref}
omdocst.constitutive.attribs ::= idc.attrib, gf.attrib?
sym.role.attrib              ::= attribute role {"type" | "sort" | "object" |
                                 "binder" | "attribution" |
                                 "semantic-attribution" | "error"}
omdoc.toplevel.attribs       ::= id.attrib, attribute generated-from {omdocref}?
exists.attrib                ::= attribute existence {omdocref}

omdocmobj.class              ::= legacy | OMOBJ | math
omdocref                     ::= xsd:anyURI

Anything                     ::= (AnyElement|text)*
AnyElement                   ::= element * {AnyAttribute,(text|AnyElement)*}
AnyAttribute                 ::= attribute * { text }*

metadata                     ::= element metadata {id.attrib,
                                   attribute inherits {omdocref}?,
                                   (omdoc.meta.class)*}
omdoc                        ::= element omdoc {group.attribs,
                                 attribute version {xsd:string {pattern = "1.2"}}?,
                                 group.elts}
group.elts                   ::= metadata?,catalogue?,(omdoc.class)*
omdocsth.imports.model       ::= id.attrib,from.attrib,metadata?
imports                      ::= (ss| element imports {omdocsth.imports.model})
theory                       ::= element theory {id.attrib,
                                   attribute cdurl {xsd:anyURI}?,
                                   attribute cdbase {xsd:anyURI}?,
                                   attribute cdreviewdate {xsd:date}?,
                                   attribute cdversion {xsd:nonNegativeInteger}?,
                                   attribute cdrevision {xsd:nonNegativeInteger}?,
                                   attribute cdstatus {" official " | "experimental"
                                                       |"private" | "obsolete"}?,
                                   metadata?,
                                   (omdoc.class | omdocst.constitutive.class)*}
symbol                       ::= element symbol {omdocst.scope.attrib,
                                   omdocst.constitutive.attribs,
                                   [a:defaultValue = "object"] sym.role.attrib?,
                                   metadata?,type*}
type                         ::= element type {omdoc.toplevel.attribs, just-by.attrib?,
                                 attribute for {omdocref}?,
                                 attribute system {omdocref}?,
                                 omdocmtxt.MC.content,
                                 (omdocmobj.class),
                                 (omdocmobj.class)?}
def.simple                   ::= (attribute type {"simple"},exists.attrib?,(omdocmobj.class))
defs.all                     ::= def.informal|def.simple|def.implicit|def.eq
definition                   ::= element definition {omdocst.constitutive.attribs, for.attrib,
                                   omdocmtxt.MC.content,(defs.all)}
assertion                    ::= element assertion {omdoc.toplevel.attribs,
                                   [a:defaultValue = "conjecture"]
                                   attribute type {assertiontype}?,
                                   attribute proofs {omdocrefs}?,
                                   omdocmtxt.MCF.content}
```

```
    assertiontype      ::= "theorem" | "lemma" | "corollary" | "proposition" |
                           "conjecture" | "false-conjecture" | "obligation" |
                           "postulate" | "formula" | "assumption" | "rule"
    CMP                ::= (ss| element CMP {xml.lang.attrib, id.attrib, (omdoc.mtext.class)*})
62  FMP                ::= (ss| element FMP {id.attrib, attribute logic {xsd:NMTOKEN}?,
                                  ((assumption*,conclusion*)|omdocmobj.class)})
    assumption         ::= (ss| element assumption {id.attrib,
                                  attribute inductive {"yes" | "no"}?,
                                  (CMP*,omdocmobj.class?)})
67  conclusion         ::= (ss| element conclusion {id.attrib, (CMP*,omdocmobj.class?)})
    adt                ::= element adt {omdoc.toplevel.attribs,
                           attribute parameters {list {xsd:NCName*}}?, metadata?, sortdef+}
    sortdef            ::= (ss| element sortdef {omdocadt.sym.attrib,
                                  [a:defaultValue = "sort"] attribute role {"sort"}?,
72                                [a:defaultValue = "loose"] attribute type {adttype}?,
                                  metadata?,(constructor | insort)*,recognizer?})
    constructor        ::= (ss| element constructor {omdocadt.sym.attrib,
                                  [a:defaultValue = "object"] sym.role.attrib?,
                                  metadata?,argument*})
77  argument           ::= (ss| element argument {type,selector?})
    selector           ::= (ss| element selector {omdocadt.sym.attrib,
                                  [a:defaultValue = "object"] sym.role.attrib?,
                                  [a:defaultValue = "no"] attribute total {"yes" | "no"}?,
                                  metadata?})
82  proof              ::= element proof {omdoc.toplevel.attribs, fori.attrib,
                                  metadata?,(omtext|symbol|definition|derive|hypothesis)*}
    proofobject        ::= element proofobject {omdoc.toplevel.attribs, fori.attrib,
                                  metadata?,(omdocmobj.class)}
    derive             ::= (ss| element derive {id.attrib,derive.type.attr?,
87                                omdocmtxt.MCF.content,method?})
    hypothesis         ::= (ss| element hypothesis {id.attrib,
                                  attribute inductive {"yes" | "no"}?,
                                  omdocmtxt.MCF.content})
    method             ::= (ss| element method {xref.attrib?, (omdocmobj.class|premise|proof|proofobject)*})
92  premise            ::= (ss| element premise {xref.attrib,
                                  [a:defaultValue = "0"]
                                  attribute rank {xsd:string {pattern = "0|[1-9][0-9*]"}}?})
    private            ::= element private {omdoc.toplevel.attribs,omdocext.private.attrib,
                                  attribute reformulates {omdocref}?,
97                                metadata?,data+}
    data               ::= (ss| element data {id.attrib,
                                  attribute format {xsd:string}?,
                                  attribute href {xsd:anyURI}?,
                                  attribute size {xsd:string}?,
102                               attribute pto {xsd:string}?,
                                  attribute pto-version {xsd:string}?,
                                  attribute original {"external" | "local"}?,
                                  Anything})
    presentation       ::= element presentation {omdoc.toplevel.attribs,
107                               attribute for {omdocref},
                                  role.attrib?,
                                  (xref.attrib?|
                                  ([a:defaultValue = "prefix"] fixity.attrib?,
                                  [a:defaultValue = "("]       lbrack.attrib?,
112                               [a:defaultValue = ")"]       rbrack.attrib?,
                                  [a:defaultValue = ","]       separator.attrib?,
                                  [a:defaultValue = "math"]    bracket-style.attrib?,
                                  [a:defaultValue = "1000"]    precedence.attrib?,
                                  [a:defaultValue = "yes"]     crossref.attrib?,
117                               (use | xslt | style)*))}
    use                ::= (ss| element use {format.attrib, bracket-style.attrib?, fixity.attrib?,
                                  lbrack.attrib?,rbrack.attrib?,separator.attrib?,
                                  attribute element {xsd:string}?,
                                  attribute attributes {xsd:string}?,
122                               crossref.attrib?,
                                  (text | omdocpres.use.mix)*})
```

5 OMDoc Intergration in ✓eriFun

In diesem Abschnitt wird die Integration der Semantic Xml-Applikation OMDoc in den Theorembeweiser ✓eriFun vorgestellt. Der erste Teil beschreibt die Enkodierung von ✓eriFun-Programmen nach OMDoc und im zweiten Teil wird die Rekonstruktion, die Dekodierung von OMDoc nach ✓eriFun erläutert.

5.1 Enkodierung von ✓eriFun nach OMDoc

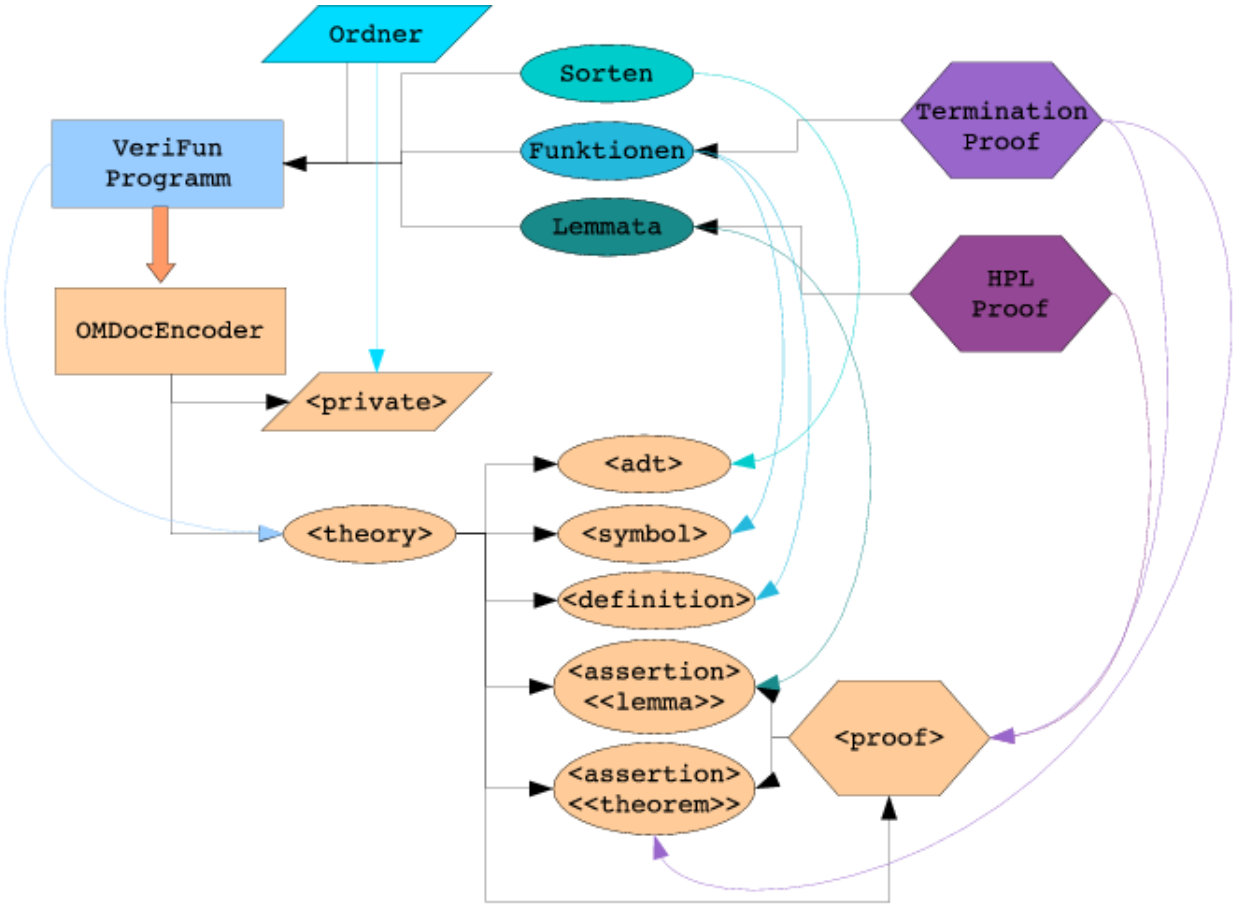


Abbildung 5.1: Der ✓eriFun 2OMDoc Enkoder

In Abbildung 5.1 wird ein grober Überblick der Übersetzung der einzelnen ✓eriFun -Programmelemente nach OMDoc dargestellt. Im oberen Teil dieser Abbildung sind die einzelnen Komponenten dargestellt, aus denen ein ✓eriFun Programm (kurz: Programm) bestehen kann. Die Überführung eines Programms mit all den darin enthaltenen Komponenten in eine gültige OMDoc-Instanz wird durch den in dieser Arbeit erstellten `OMDocEncoder` durchgeführt. In welche OMDoc Markupelemente dieser Enkoder die einzelnen Programmelemente bzw. das gesamte Programm überträgt, wird im unteren Teil der Abbildung dargestellt. Die folgenden Unterabschnitte werden diese Übersetzungen näher erläutern.

5.1.1 Programm

Listing 5.2: OMDoc Repräsentation eines leeren ✓eriFun Programmes

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE omdoc SYSTEM "omdoc-vf.dtd">

<omdoc xml:id="VeriFun3.0-2005-11-18"
       xmlns="http://www.mathweb.org/omdoc"
       xmlns:dc="http://purl.org/dc/elements/1.1/">

  <metadata>
    <dc:date>2005-11-19T01:36:00</dc:date>
  </metadata>

  <private xml:id="vfd4732856-1136-4866-b704-5e2e8e503451">
    <data pto="VeriFun">Program</data>
    <data pto="VeriFun"/>
    <data pto="VeriFun"/>
  </private>

  <theory xml:id="VeriFun">
    <imports from="vafp"/>
  </theory>

</omdoc>
```

In Listing 5.2 ist das Grundgerüst eines ✓eriFun-Programms im OMDoc-Format abgebildet. Sämtliche Programme, die in diesem Format kodiert vorliegen, müssen dem `omdoc-vf`-Dokumententyp genügen. In dieser Dtd sind alle OMDoc-Elemente, Attribute und die zulässige Grammatik spezifiziert, die in der Übersetzung eines Programms nach OMDoc zulässig ist. Es kann somit später in der Dekodierung (Abschnitt 5.2) sichergestellt werden, dass nur gültige `omdoc-vf`-Dokumentinstanzen eingelesen werden.

Die weitere Verarbeitung gültiger Dokumentinstanzen durch den `OMDocDecoder` wird somit enorm vereinfacht, so dass zum Beispiel keine weiteren Überprüfungen hinsichtlich einer korrekten OMDoc-Syntax vorgenommen werden müssen. Sollte die Grammatik und/oder die deskriptiven TAGs nicht korrekt verwendet werden, so wird der eingesetzte validierende XML-Parser eine Exception werfen. Hinzu kommt, dass in dem `xml:id` Attribut des Wurzel Elementes `omdoc` die aktuelle ✓eriFun-Version enkodiert wird. Es läßt sich auf diese Weise zu jeder Zeit feststellen mit welchen Stand des Systems diese Datei erzeugt worden ist. Zusätzlich zu dieser Versionsangabe ist in den dafür vorgesehenen DC-Metadaten (Abschnitt 4.2) der genau Zeitstempel der Erstellung der Datei gespeichert. Mit all diesen Kriterien wird somit eine maximale Robustheit der Daten gewährleistet, so dass die gespeicherten Informationen innerhalb der OMDoc Dateien durch keine Art von Änderungen innerhalb des ✓eriFun Systems oder der Sprache OMDoc selbst gefährdet sind.

In den `private`-Blöcken wird die Ordnerstruktur eines enkodierten Programms gespeichert. Auf die einzelnen Interpretation der `data`-Elemente eines solchen Blocks wird in Unterabschnitt 5.1.2 weiter eingegangen. Die Ordnerstruktur von ✓eriFun-Programmen ist nicht Teil einer OMDoc-Theorie. Die Begründung hierfür liegt zum einen darin, dass es sich hierbei um ✓eriFun-systemspezifische Informationen handelt, die lediglich zur visuellen Gruppierung eines Programms dienen und zum anderen diese Ordner keine weitere, insbesondere den Aufbau einer OMDoc Theorie betreffende Semantik tragen.

Im Endeffekt werden alle Elemente - bis eben auf die zuvor erwähnten Ordner - eines enkodierten Programms innerhalb einer OMDoc-Theorie gekapselt. Da jedoch das Konzept von mathematischen Theorien als solches in ✓eriFun nicht vorhanden ist, erhalten zum einen durch die aktuelle Version des `OMDocEncoder`s generierte OMDoc-Theorien in dem `xml:id`-Attribut immer den festen Wert `VeriFun` und zum anderen wird an dieser Stelle auch keine Unterscheidung zwischen konstitutiven und nicht-konstitutiven Theorieelementen vorgenommen. Auf der Hierarchieebene der direkten `omdoc`-Kind-Elemente, einer durch den `OMDocEncoder` generierten OMDoc-Instanz, werden somit keine weiteren Elemente auftreten (noch würden nachträglich per Hand eingefügte Elemente durch den `OMDocDecoder` eine Berücksichtigung erfahren). Die vordefinierten ✓eriFun-Programmelemente, die Bestandteil eines jeden Programms sind, wurden einmalig[1] in die externe `vafp`-Theorie ausgelagert und werden mittels des `import`-Elements in eine durch den `OMDocEncoder` neu erstellte Theorie importiert. In Folge dessen ist keine durch den `OMDocEncoder` erstellte OMDoc-Theorie leer und jedes nach OMDoc übersetzte Programm besteht aus genau zwei OMDoc Theorien (Dateien): Die (Haupt-)Theorie `VeriFun`, die alle Elemente des enkodierten Programms enhält und die zu importierende Theorie `vafp`, bestehend aus den vordefinierten ✓eriFun-Programmelementen.

[1] Hiermit soll zum Ausdruck gebracht werden, dass die `vafp` Theorie, der vordefinierten ✓eriFun-Programmelemente, nicht immer wieder neu erzeugt wird, sondern während dieser Arbeit durch den Autor per Hand einmal erstellt und den notwendigen Dateien zur Ausführung des ✓eriFun-Systems, hinzugefügt wurde.

5.1.2 Ordner

```
<private xml:id="vf8084160a-eb0f-4fdc-803f-eaca13b594f9">
  <data pto="VeriFun">Program</data>
  <data pto="VeriFun"/>
  <data pto="VeriFun"/>
</private>
<private xml:id="vfa58ed859-71ba-4774-bd76-81872fba7bdf">
  <data pto="VeriFun">Folder A</data>
  <data pto="VeriFun">Comment of Folder A</data>
  <data pto="VeriFun"/>
</private>
<private xml:id="vf9e4bf7c2-4fe8-471b-8c42-ad5f75046f2b">
  <data pto="VeriFun">Folder AA</data>
  <data pto="VeriFun"/>
  <data pto="VeriFun">Folder A</data>
</private>
<private xml:id="vfbd24e25d-13f6-46f5-896c-6464dce7055f">
  <data pto="VeriFun">Folder AB</data>
  <data pto="VeriFun">Comment of Folder AB</data>
  <data pto="VeriFun">Folder A</data>
</private>
<private xml:id="vf4d74846b-7fc1-49ad-996d-525695bb9749">
  <data pto="VeriFun">Folder B</data>
  <data pto="VeriFun"/>
  <data pto="VeriFun"/>
</private>
<private xml:id="vf2fa0e6ff-e5f5-4275-b729-95bc0150a6bb">
  <data pto="VeriFun">Folder C</data>
  <data pto="VeriFun"/>
  <data pto="VeriFun"/>
</private>
```

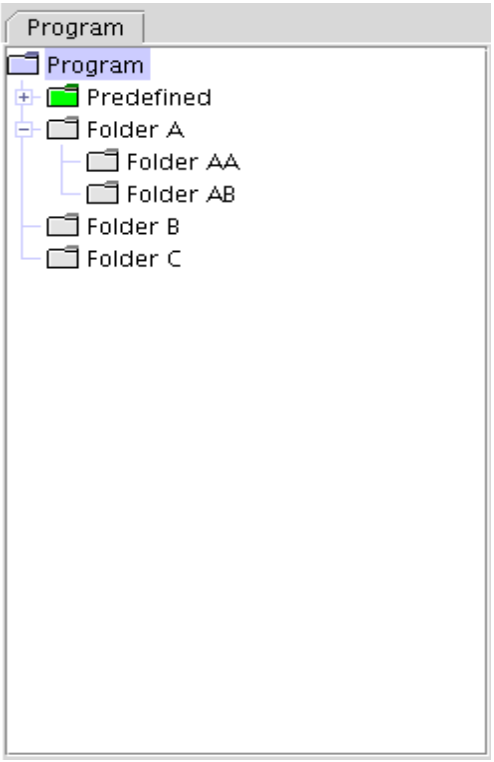


Abbildung 5.3: OMDoc Repräsentation einer ✓eriFun Ordnerstruktur

Die Ordnerstruktur von ✓eriFun wird in OMDoc "interpreted Markup" kodiert. In diesen `private`-Elementen können software-spezifische Anweisungen kodiert werden (Abbildung 5.3). Das `xml:id`-Attribut dient zur eindeutigen Referenzierung der `private`-Daten. Die einzelnen Daten werden mit dem `data`-Element gekennzeichnet. In dem `pto`-Attribut, welches für "private to" steht, kann spezifiziert werden, für welches spezielle Softwareprogramm diese Daten eine Verwendung finden. Der `OMDocEncoder` setzt in diesem Attribut den festen Wert `VeriFun` ein. Die einzelnen `data`-Elemente haben für ✓eriFun die folgende Bedeutung: In dem ersten wird der in dem System vergebene Ordnername kodiert. Der Kommentar zu einem Ordner wird in dem darauf folgenden `data` gespeichert. Um die Ordnerhierarchie abbilden zu können, existiert ein weiteres `data`-Element. In diesem wird der "Vater-Ordner" kodiert. Der aktuelle `OMDocEncoder` verwendet hierzu nicht das `xml:id` Attribut eines `private`-Elements, sondern speichert an dieser Stelle direkt den Namen des "Vater-Ordners" im Klartext. Dies wäre ein erster Ansatzpunkt zur Verbesserung in der nächsten Version. Final sei noch erwähnt, dass die Reihenfolge der einzelnen `data`-Elemente fest durch den `OMDocEncoder` vorgeschrieben ist. Eine Änderung dieser Reihenfolge kann zu Fehlern in der Dekodierung durch den `OMDocDecoder` führen.

5.1.3 Sorten

Listing 5.4: ✓eriFun Repräsentation einer primitiven Sorte

```
structure list <=
  ∅,
  [ infixr ,100] :: (hd : ℕ, tl : list )
```

Die in Listing 5.4 dargestellte ✓eriFun Sorte `list` enthält zwei Konstruktoren ∅ und `::` um diesen Datentyp zu erzeugen. Der Konstruktor ∅ hat die ✓eriFun Standard-Fixity `prefix` und ist von der Stelligkeit 0, enthält somit keine weiteren Selektoren. `::` als weiterer Konstruktor der Sorte `list`, wurde mit der Fixity `infixr` und der Bindungspriorität 100 versehen. Dieser Konstruktor mit der Stelligkeit 2, enthält die beiden Selektoren `hd` und `tl`. Die Typen der Rückgabenwerte eines jeden Selektors werden in ✓eriFun durch einen Doppelpunkt getrennt und direkt hinter den jeweiligen Selektor plaziert. Der Selektor `hd` ist von dem Typ ℕ und `tl` von dem Typ `list`.

Listing 5.5: OMDoc Repräsentation der Sorte aus Listing 5.4

```
<adt xml:id="vf806a4dbf-bf3b-4d0c-976b-d94e009b4d53.adt">
  <sortdef name="vf806a4dbf-bf3b-4d0c-976b-d94e009b4d53" type="free">
    <constructor name="vfcf28e371-2d83-4edb-ae20-8ba831481efa"/>
    <constructor name="vf0dd8da83-01bb-4eb3-9e2c-8971fa95f16b">
      <argument>
        <type system="simpletypes">
          <OMOBJ xmlns="http://www.openmath.org/OpenMath">
            <OMS cd="vafp" name="nat"/>
          </OMOBJ>
        </type>
        <selector name="vf987b8c20-f8a4-46b1-a858-c4f918617c29" total="yes"/>
      </argument>
      <argument>
        <type system="simpletypes">
          <OMOBJ xmlns="http://www.openmath.org/OpenMath">
            <OMS cd="VeriFun" name="vf806a4dbf-bf3b-4d0c-976b-d94e009b4d53"/>
          </OMOBJ>
        </type>
        <selector name="vfcbb7b885-9666-468c-9bbe-9f3837f0f94a" total="yes"/>
      </argument>
    </constructor>
  </sortdef>
</adt>
```

Eine ✓eriFun-Sorte wird in einem OMDoc-`adt`-Element mit korrespondierenden `presentation`-Elementen enkodiert. In Listing 5.5 ist der erste Teil einer solchen Enkodierung nach

OMDoc dargestellt. Die Sorte `list` wird in OMDoc in einem abstrakten Datentyp enkodiert. In dem Element `adt` werden nun alle weiteren die Sorte betreffenden internen Informationen gekapselt. Mit internen Informationen sind an dieser Stelle die abstrakten Informationen zur Erzeugung einer ADT gemeint. Es werden hier keinerlei die Präsentation betreffenden Informationen hinterlegt, noch werden die eigentlichen Namen der Konstruktoren bzw. Selektoren gespeichert. ✓eriFun verwendet hierzu den "externen" OMDoc Präsentations- bzw. Notationsmechanismus (Abbildung 5.6, linker Teil). Neben Sorten wird dieser Mechanismus auch für alle übrigen ✓eriFun -Elemente in der Enkodierung nach OMDoc angewandt. Auf diese Weise ist es ✓eriFun möglich, jegliche Art von Zeichenketten als Namensrepräsentanten zu speichern, die an gewissen Stellen laut der Xml-Spezifikation nicht erlaubt wären. ✓eriFun setzt somit in Attributen wie `xml:id` oder `name` eine eindeutige, systemgenerierte `ID` ein, statt direkt den eigentlichen Symbol-Namen eines Elementes. Das Lesen einer solchen OMDoc-Datei wird zwar auf diese Weise etwas erschwert, man erhält jedoch die Freiheit, sämtliche dem Anwender zur Verfügung stehenden Zeichenketten als Repräsentanten von ✓eriFun -Elementen, enkodieren zu können. Auf `adt`-Element folgt direkt das `sortdef`-Element, mit dem in OMDoc gekennzeichnet wird, dass es sich hierbei um die Definition einer neuen Sorte in der aktuellen ADT handelt. In dieser Sortendefinition werden nun die Konstruktoren mit dem `constructor`-Element und die entsprechenden Selektoren mit dem `selector`-Element aufgelistet. Enthält ein Konstruktor keine Selektoren, so wird dieser in OMDoc als ein leeres Element dargestellt. Der Konstruktor `::` hingegen besitzt die Selektoren `hd` und `tl`. Diese werden dem Konstruktor sowohl in ✓eriFun als auch in OMDoc als Parameter übergeben. Konstruktor-Parameter werden in OMDoc mit dem `argument`-Element gekennzeichnet. Ein Konstruktur-Parameter besteht aus einem Tupel, mit dem Rückgabetyp des Selektors in der ersten Komponente und dem Selektor selbst in der zweiten. Der Typ des Rückgabetyp jedes Selektors wird mit dem `type`-Element gekennzeichnet. Mit dem zusätzlichen Attribut `system` wird zum Ausdruck gebracht, dass es sich hierbei um eine einfache Typdeklaration handelt. In OMDoc existieren mehrere Möglichkeiten, einen neuen Typen einzuführen bzw. einen bestehenden Typ zu verwenden. Da ✓eriFun in der gesamten Enkodierung nur das einfache `simpletypes`-Typsystem verwendet, wird an dieser Stelle nicht weiter auf die anderen Möglichkeiten eingegangen. Der Inhalt von `type`-Elementen im `simpletypes`-Typsystem ist ein `OMOBJ`-Element, in dem das verwendete Sortensymbol als Repräsentant des Rückgabetypes, in dem OpenMath Syntax dargestellt wird. In diesem Beispiel ist der Rückgabetype des ersten Selektors `nat`. Dieses Symbol ist in der Menge der vordefinierten ✓eriFun-Elemente enthalten und denotiert die Menge $\mathbb{N}$ der natürlichen Zahlen. Diese vordefinierten Elemente sind, wie bereits erwähnt, in der Theorie `vafp` enkodiert, so dass das Symbol `OMS` mit dem Namen `nat` auf das CD `vafp` verweisen muss. Mit dieser Referenzierung wird zum einen ersichtlich, an welcher Stelle dieses Symbol definiert wurde und zum anderen können auf diese Weise Namenskollisionen vermieden werden. Es sei darauf hingewiesen, dass die Enkodierung der vordefinierten ✓eriFun -Elemente zum Teil eine Ausnahme gegenüber benutzerdefinierten Sorten darstellt. In Listing 5.5 ist ersichtlich, dass direkt der Name der Sorte `nat` als Wert des Attributs `name` verwendet

wird und nicht wie im Fall neuer benutzerdefinierter Sorten eine eindeutige, systemgenerierte `ID`. In dem Fall der vordefinierten ✓eriFun-Elemente ist dieser Umstand zwar unkritisch, da die entsprechende Übersetzung, zum Beispiel der Sorte `nat` nach $\mathbb{N}$, automatisch durch den Dekoder vorgenommen wird, doch auch dieser Sachverhalt sollte in der nächsten Version zur Verwendung von `ID`s portiert werden. Die zweite Komponente des `argument`-Elements ist die Enkodierung des Selektors selbst. Mit dem leeren `selector`-Element wird der Name und die Totalität des aktuellen Selektors gespeichert. Durch ✓eriFun wird standardmäßig für das Attribut `total` der Wert `yes` eingetragen.

```
<presentation
  for="#vf806a4dbf-bf3b-4d0c-976b-d94e009b4d53"
  role="applied">
  <use format="VeriFun">list</use>
</presentation>
<presentation
  for="#vfcf28e371-2d83-4edb-ae20-8ba831481efa"
  role="applied" bracket-style="math" precedence="1"
  fixity ="prefix" lbrack="(" rbrack=")" >
  <use format="VeriFun">∅</use>
</presentation>
<presentation
  for="#vf0dd8da83-01bb-4eb3-9e2c-8971fa95f16b"
  role="applied" bracket-style="math" precedence="100"
  fixity ="infixr" lbrack="(" rbrack=")">
  <use format="VeriFun">::</use>
</presentation>
<presentation
  for="#vf987b8c20-f8a4-46b1-a858-c4f918617c29"
  role="applied" bracket-style="math" precedence="1"
  fixity ="prefix" lbrack="(" rbrack=")">
  <use format="VeriFun">hd</use>
</presentation>
<presentation
  for="#vfcbb7b885-9666-468c-9bbe-9f3837f0f94a"
  role="applied" bracket-style="math" precedence="1"
  fixity ="prefix" lbrack="(" rbrack=")">
  <use format="VeriFun">tl</use>
</presentation>
```

```
<private for="#vf806a4dbf-bf3b-4d0c-976b-d94e009b4d53">
  <data pto="VeriFun">Comment for list</data>
  <data pto="VeriFun">Program</data>
</private>

<symbol name="vf47e58d4e-1c7f-40a1-938e-2e1b7d7a2791">
  <type system="simpletypes">
    <OMOBJ xmlns="http://www.openmath.org/OpenMath">
      <OMA>
        <OMS cd="simpletypes" name="funtype"/>
        <OMS cd="VeriFun"
             name="vf806a4dbf-bf3b-4d0c-976b-d94e009b4d53"/>
        <OMS cd="vafp" name="bool"/>
      </OMA>
    </OMOBJ>
  </type>
</symbol>
<symbol name="vfcee415d8-9db9-4d08-98c4-5f24668a94be">
  <type system="simpletypes">
    <OMOBJ xmlns="http://www.openmath.org/OpenMath">
      <OMA>
        <OMS cd="simpletypes" name="funtype"/>
        <OMS cd="VeriFun"
             name="vf806a4dbf-bf3b-4d0c-976b-d94e009b4d53"/>
        <OMS cd="vafp" name="bool"/>
      </OMA>
    </OMOBJ>
  </type>
</symbol>
```

Abbildung 5.6: OMDoc Repräsentation zusätzlicher Informationen aus Listing 5.4

Nachdem die abstrakten Komponenten der Sorte `list` enkodiert wurden, werden durch den `OMDocEncoder` sowohl die die Notation betreffenden als auch die systeminternen Informationen bezüglich der aktuellen Sorte gespeichert. Es handelt sich hierbei um die Repräsentation der eigentlichen Elementnamen (Konstruktor, Selektor, Rückgabetype), dem Kommentar, dem zugehörigen ✓eriFun-Ordner und der durch das System erzeugten sogenannten Structure-Tests. In Abbildung 5.6 wird die Enkodierung dieser zusätzlichen Informationen einer Sorte dargestellt. Der linke Teil dieser Abbildung repräsentiert die enkodierten Informationen, die die Notation und Präsentation betreffen (Unterabschnitt 4.4.4). In einem `presentation`-Element werden zusätzliche Informationen des

Symbols durch Attributierung hinzugefügt. Mit dem Attribut `role` wird angeben, unter welchen Umständen diese Präsentation zum tragen kommt. ✓veriFun setzt hierfür als Standardwert `apply` ein, um zu kennzeichnen, dass diese Darstellung des Symbols immer genau dann angewendet werden soll, wenn das Symbol, zum Beispiel durch ein `OMA` OpenMath-Element, appliziert wird. Auch hier ist eine eindeutige Zugehörigkeit durch das Attribut `for` gegeben. In der Dekodierung kann somit das System leicht ermitteln, zu welchem Symbol die aktuelle Präsentation gehört. Der Wert des `precedence` Attributs gibt die Bindungs-Priotität des Symbols an. Hierbei ist zu beachten, dass im Gegensatz zu OMDoc (Unterabschnitt 4.4.4) in ✓veriFun höhere Prioritäten auch stärker binden. Das Attribut `fixity` stellt die *Fixity* des Symbols dar. In der aktuellen ✓veriFun-Version werden die Werte `prefix`, `infix`, `postfix`, `infixl`, `infixr` und `outfix` unterstützt. Bis auf `outfix` werden alle Werte direkt auch von OMDoc zur Verfügung gestellt. Zur Enkodierung von Symbolen mit der Fixity `outfix`, bedient sich ✓veriFun den Attributen `lbrack` und `rbrack`. In den weiteren Attributen wird der Stile (`bracket-style`) der zu verwendeten Klammern und die Klammern selbst (`lbrack` bzw. `rbrack`) angegeben. Für das Attribut `bracket-style` wird der Wert `mathstyle` eingesetzt. Weiter Werte werden durch ✓veriFun zum Zeitpunkt der Arbeit nicht unterstützt. Als Werte von `lbrack`- bzw. `rbrack`-Attributen für die zu verwendenden Klammern setzt ✓veriFun die Werte (und) oder [und] oder den Symbolnamen selbst ein, wenn dieser die *Fixity* `outfix` trägt (Um zum Beispiel die Notation der mathematischen Betragsfunktion $|x|$ zu repräsentieren, wird in die entsprechenden `lbrack`- bzw. `rbrack`-Attribut das Zeichen | eingesetzt). In dem `presentation`-Kind-Element `use` wird dann zu guter letzt der eigentliche Name einer jeden Komponente gespeichert. Als Inhalt eines `use`-Elements ist es erlaubt eine beliebige Zeichenkette zu plazieren, so dass auch eventuelle Sonderzeichen ohne Probleme enkodiert werden können. Mit dem `format`-Attribut können explizit unterschiedliche Repräsentationen angegeben werden. So kann beispielsweise die ✓veriFun Funktion > mit mehreren systemspezifischen Repräsentationen annotiert werden. Man könnte die Repräsentation `gt` für ✓veriFun angeben und zum Beispiel die Repräsentation `>` für eine Darstellung in Html. Auch hier bilden Funktionen mit der *Fixity* `outfix` eine Ausnahme. Damit der `OMDocDecoder` erkennen kann, dass sich in dem aktuell zu dekodierendem Symbol um eine Outfix-Funktion handelt, wird für solche Funktionen das `use` Element als leeres Element dargestellt (Listing 5.7).

Listing 5.7: OMDoc Repräsentation der Outfix-Funktion $|x|$

```
1 <presentation for="#vf3b13f759-e252-4178-b180-d2446ea4d9f9" role="applied"
      bracket-style="math" precedence="1" lbrack="|" rbrack="|" fixity="prefix">
    <use format="VeriFun"/>
</presentation>
```

In dem rechten Teil der Abbildung 5.6 werden die Repräsentationen der übrigen systeminternen ✓veriFun -Informationen dargestellt. In dem oberen rechten Teil ist die Enkodierung des Kommentars und des entsprechenden Ordners zu sehen. Diese Informationen werden analog zu der Enkodierung einer gesamten Ordnerstruktur eines ✓veriFun-Programmes (Unterabschnitt 5.1.2) in einem `private`-Block gekapselt. Durch

das Attribut `for` wird die Referenz zu der zugehörigen Sorte gespeichert, so dass seine eindeutige Zuordnung ermöglicht wird. In den folgenden `data`-Elementen wird zuerst der Kommentar und dann der zugehörige Ordner (auch hier wieder im Klartext) gespeichert. Die Enkodierung der Structure-Tests gestaltet sich entsprechend der Enkodierung einer benutzerdefinierten Funktion (Unterabschnitt 5.1.4) und ist in Abbildung 5.6 im rechten unteren Teil abgebildet. Die einzelnen durch das System erzeugten Structure-Tests werden jeweils durch ein `symbol`-Element dargestellt. Jedoch im Gegensatz zu der Enkodierung von benutzerdefinierten Funktionen werden diese Funktionen lediglich *deklariert* und nicht weiter *definiert*. Denn obwohl diese Structure-Tests während der Dekodierung einer Sorte wieder automatisch durch das ✓eriFun -System erzeugt werden, ist es dennoch unabdingbar, die Deklarationen diese Funktionen zu enkodieren. Der Grund liegt in der Referenzierung dieser Elemente, wenn diese an einer anderen Stelle in dem OMDoc-Dokument bzw. der OMDoc-Theorie verwendet werden. Es würde sich um kein korrektes OMDoc handeln, wenn zwar diese Symbole durch eine anderes OMDoc-Element referenziert werden würden, aber diese an keiner Stelle deklariert worden sind. Auch eine spätere eventuelle Konvertierung in ein anderes Datenformat würde scheitern, da diese Symbole schlichtweg nicht bekannt wären und somit auch nicht repräsentiert werden könnten. Um auch die Repräsentation dieser systemgenerierten Komponenten der aktuellen Sorte zu gewährleisten, werden auch Structure-Tests durch `presentation`-Elemente weiter spezifiziert (Listing 5.8).

Listing 5.8: Structure-Test Notationsinformation aus Listing 5.4

```
<presentation for="#vf47e58d4e-1c7f-40a1-938e-2e1b7d7a2791" role="applied"
    bracket-style="math" precedence="0" fixity="prefix" lbrack="(" rbrack=")">
  <use format="VeriFun">?∅</use>
</presentation>
<presentation for="#vfcee415d8-9db9-4d08-98c4-5f24668a94be" role="applied"
    bracket-style="math" precedence="0" fixity="prefix" lbrack="(" rbrack=")">
  <use format="VeriFun">?::</use>
</presentation>
```

Während dieser Arbeit wurde ✓eriFun unter anderem durch polymorphe Datentypen erweitert. Es ist nun möglich, nicht nur primitive Sorten zu definieren, sondern diese um Typvariablen zu erweitern. Der Vorteil ist offensichtlich: Man muss nun beispielsweise nicht mehr eine `list` für jeden benötigten Datentyp erstellen, sondern definiert vielmehr eine generische `list` und spezifiziert erst in der Applikation dieser `list` um welchen aktuellen Datentyp es sich handelt. Als Beispiel wird in Listing 5.9 die Sorte `list` aus Listing 5.4 als polymorphe Sorte mit der Typvariablen `@value` dargestellt.

Listing 5.9: ✓eriFun Repräsentation einer polymorphen Sorte

```
structure list [@value] <=
  ∅,
  [infixr ,100] :: (hd : @value, tl : list [@value])
```

Diese Erweiterung von ✓eriFun bewirkte auch eine Erweiterung der Sprache OMDoc. Es war bis zu diesem Zeitpunkt nicht direkt möglich, polymorphe Datentypen in OMDoc zu verwenden. Es bestand jedoch die Möglichkeit, in Form von Theorie-Morphismen und parametrisierbaren Theorien den gleichen Effekt zu erzielen. Jedoch zum einen durch den erheblichen Mehraufwand in der Enkodierung und zum anderen durch das Anwachsen der Dateigrößen und insbesondere durch die Offenheit in der Namensgebung der Typvariablen in ✓eriFun, stellt sich schnell heraus, dass dieser Mechanismus nicht zum gewünschten Ziel führen würde. An dieser Stelle wird nicht weiter darauf eingegangen, wie sich diese Alternative dargestellt hätte, sondern es wird direkt die Erweiterung von OMDoc um polymorphe ADTs erläutert.

Listing 5.10: OMDoc Repräsentation aus Listing 5.9

```
<adt xml:id="vf7b9f3e59-e78e-4221-8064-7fa0c5689f5d.adt" parameters="value">
  <sortdef name="vf7b9f3e59-e78e-4221-8064-7fa0c5689f5d" type="free">
    <constructor name="vf8a6673ac-c1d9-4698-b6ee-90213539a984"/>
    <constructor name="vf38164505-4983-417f-8bdc-6a42b046e933">
      <argument>
        <type system="simpletypes">
          <OMOBJ xmlns="http://www.openmath.org/OpenMath">
            <OMV name="value"/>
          </OMOBJ>
        </type>
        <selector name="vf9fc4c672-207f-45c0-ae61-1f675fde7aed" total="yes"/>
      </argument>
      <argument>
        <type system="simpletypes">
          <OMOBJ xmlns="http://www.openmath.org/OpenMath">
            <OMA>
              <OMS cd="VeriFun"
                   name="vf7b9f3e59-e78e-4221-8064-7fa0c5689f5d"/>
              <OMV name="value"/>
            </OMA>
          </OMOBJ>
        </type>
        <selector name="vf55767f3a-b019-4308-88f9-d68ee0db595e" total="yes"/>
      </argument>
    </constructor>
  </sortdef>
</adt>
```

In Listing 5.10 wird die polymorphe Sorte `list` aus Listing 5.9 mit der Typvariablen `@value` im OMDoc-Format vorgestellt. Im Wesentlichen kommt zu der Definition einer ADT in OMDoc durch das `adt` Element ein weiteres Attribut hinzu. Neben der eindeutigen `xml:id` zur Referenzierung erhält dieses Element nun noch das Attribut

parameters. Als Wert ist eine durch Leerzeichen getrennte Liste der Typvariablen erlaubt. Das von ✓eriFun zur Kennzeichnung von Typvariablen vorangestellte @-Zeichen wird nicht mit enkodiert, sondern explizit in der Dekodierung wieder angefügt. Der Name der Typvariablen wird im Klartext gespeichert, was in der nächsten ✓eriFun-Version geändert werden sollte, um auch hier die größtmögliche Offenheit in der Namensgebung zu erzielen. Momentan sind an dieser Stelle keinen Sonderzeichen zugelassen. Der Grund hierfür liegt wiederum in der XML Spezifikation, in der unter anderem manifestiert ist, dass Sonderzeichen in Attributwerten nicht erlaubt sind. Als weitere Modifikation zu der Darstellung einer primitiven Sorte ist die Enkodierung des Rückgabetyps, zum Beispiel der des zweiten Selektors des Konstruktors ::, zu beachten. Es handelt sich hierbei nicht mehr um eine primitive Referenzierung eines OMS-Symbols, sondern vielmehr um die Applikation einer polymorphen Sorte. Das enkodierte Symbol für die Sorte list wird auf die Typvariable @value angewandt. Die restlichen systeminternen und die Notation betreffenden Informationen einer polymorphen Sorte werden analog zu den Daten einer primitiven Sorte gespeichert.

5.1.4 Funktionen

Listing 5.11: ✓eriFun Repräsentation der Funktionen + und ∗

```
function [ infix ] +(x : ℕ, y : ℕ) : ℕ <=
if ?0(x)
  then y
  else +((⁻(x) + y))
end_if

function [ infix ] ∗(x : ℕ, y : ℕ) : ℕ <=
if ?0(x)
  then 0
  else ((⁻(x) ∗ y) + y)
end_if
```

In diesem Abschnitt wird die Enkodierung der von ✓eriFun benutzerdefinierten Funktionen nach OMDoc erläutert. In Listing 5.11 sind zwei Funktionen aufgeführt, die als Beispiel dienen werden. Zum einen die Funktion + als Repräsentant für die natürliche Addition und zum anderen die Funktion ∗, in Abhängigkeit von der Funktion +, für die natürliche Multiplikation. Die Funktion + hat die zwei formale Parameter x und y von dem Typ $\mathbb{N}$. Der Rückgabewert ist ebenfalls von dem Typ $\mathbb{N}$. Das hochgestellte +-Zeichen ($^+$) denotiert die Nachfolger- und das hochgestellte −-Zeichen ($^-$) die Vorgängerfunktion. Für die Funktion ∗ gilt das Analoge.

Der OMDocEncoder berücksichtigt zur korrekten Speicherung von ✓eriFun-Funktionen und sämtlichen anderen Elementen eines ✓eriFun-Programms, die Abhängigkeiten der

Elemente untereinander. In diesem Beispiel wird somit die Funktion + vor der Funktion ∗ enkodiert werden. Der Grund liegt dabei nicht in irgendwelchen OMDoc-Restriktionen, sondern vielmehr ist auf diese Art und Weise eine späterer Dekodierung um einiges einfacher. Der Dekoder kann eine OMDoc Datei bezüglich der Elemente sequentiell einlesen und diese sofort nach der jeweiligen Rekonstruktion in das aktuelle Programm einfügen. Wäre dies nicht der Fall, so müsste der Dekoder die Abhängigkeiten zwischen den Elementen selbst auflösen, was ein erheblicher Mehraufwand wäre und sich in der Performanz negativ niederschlagen würde. Der `OMDocEncoder` enkodiert somit alle Elemente eines aktuellen ✓eriFun -Programms unter einer topologischen Ordnung. Ein Nachteil in dieser Vorgehensweise entsteht jedoch auch: Eine durch den Benutzer vorgenommene Sortierung der Programmelemente geht während der Dekodierung verloren. Auch dieser Nachteil sollte in der nächsten Version des `OMDocEncoders` bzw. `OMDocDecoders` ausgebessert werden.

In Abbildung 5.12 ist die OMDoc-Repräsentation der in Listing 5.11 aufgeführten ✓eriFun Funktion + dargestellt. Die Enkodierung benutzerdefinierter ✓eriFun-Funktionen gliedert sich in zwei wesentliche Teile: Die Funktions*deklaration* und die Funktions*definition*. Mit dem `symbol`-Element wird das neue Funktionssymbol in die aktuelle Theorie eingeführt. Analog zu der Enkodierung von ✓eriFun Sorten (Unterabschnitt 5.1.3) wird auch jedem Funktionssymbol durch den `OMDocEncoder` eine eindeutige `ID` zugewiesen. Zum einen können die einzelnen Funktionssymbole innerhalb von OMDoc direkt referenziert werden und zum anderen wird dadurch weiterhin die Offenheit der Namensgebung für die Elemente in ✓eriFun unterstützt. Da Symbole jedoch in OMDoc kein `xml:id` Attribut besitzen, sondern stattdessen ein eindeutiges `name`-Attribut erhalten, wird diese `ID` als Wert dieses Attributs eingesetzt. Zur Deklaration der Signatur des Funktionssymbols stellt OMDoc verschiedene Vorgehensweisen zur Verfügung. Auch hier verwendet der `OMDocEncoder` wieder analog zu der Enkodierung von ✓eriFun -Sorten (Unterabschnitt 5.1.3), ausschließlich das einfache Typsystem `simpletypes`. Mit dieser Form der Signaturdeklaration werden Funktionssymbole in der "curry"-Schreibweise eingeführt. Die Funktion + trägt somit die Signatur $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$. In OMDoc wird diese Schreibweise innerhalb eines `type` Elements durch das Symbol `funtype` gekennzeichnet. Dieses Symbol erhält als Parameter die einzelnen Typen der Funktionssignatur, so dass in diesem Beispiel die erste beiden Argumente den Typ der formalen Parameter und das letzte Argument den Rückgabetyp der Funktion selbst darstellen. Die Definition der Funktion + wird von dem `OMDocEncoder` direkt im Anschluss an die Funktionsdeklaration gestellt und mit dem OMDoc `definition`-Element eingeleitet. Zur eindeutigen Referenzierung des zugehörigen Symbols bzw. Funktionsdeklaration wird in dem Attribut `for` der Wert des entsprechenden `name`-Attributs gespeichert. Durch die Vergabe der eindeutigen `ID` als OMDoc Name eines jeden Symbols kann es hier zu keinen Verwechslungen kommen. Das Attribut `existence` ist ein Zeiger auf den Existenzbeweis der Funktion. Existenzbeweise sind in ✓eriFun die Terminierungsbeweise (Unterabschnitt 5.1.7) oder Rekursionseliminationsbeweise (Unterabschnitt 5.1.8) von Funktionen. OMDoc verlangt, dass auch jede Funktionsdefinition einen Namen erhält. Der `OMDocEncoder` verwendet hier den gleichen Namen wie der des zugehörigen

```
<symbol
  name="vf7c73dd84-02df-4f34-bedf-38808cea08fd">
  <type system="simpletypes">
    <OMOBJ
      xmlns="http://www.openmath.org/OpenMath">
      <OMA>
        <OMS cd="simpletypes" name="funtype"/>
        <OMS cd="vafp" name="nat"/>
        <OMS cd="vafp" name="nat"/>
        <OMS cd="vafp" name="nat"/>
      </OMA>
    </OMOBJ>
  </type>
</symbol>
```

```
<definition name="vf7c73dd84-02df-4f34-bedf-38808cea08fd.def"
  for="#vf7c73dd84-02df-4f34-bedf-38808cea08fd" type="simple"
  existence="#vf72c05981-ebbd-494f-ba51-62cdcbdfa39e">
  <OMOBJ xmlns="http://www.openmath.org/OpenMath">
    <OMBIND>
      <OMS cd="vafp" name="function"/>
      <OMBVAR>
        <OMATTR>
          <OMATP>
            <OMS cd="simpletypes" name="type"/>
            <OMS cd="vafp" name="nat"/>
          </OMATP>
          <OMV name="x"/>
        </OMATTR>
        <OMATTR>
          <OMATP>
            <OMS cd="simpletypes" name="type"/>
            <OMS cd="vafp" name="nat"/>
          </OMATP>
          <OMV name="y"/>
        </OMATTR>
      </OMBVAR>
      <OMA>
        <OMS cd="vafp" name="if"/>
        <OMA>
          <OMS cd="vafp" name="is_zero"/>
          <OMV name="x"/>
        </OMA>
        <OMV name="y"/>
        <OMA>
          <OMS cd="vafp" name="succ"/>
          <OMA>
            <OMS cd="VeriFun"
                 name="vf7c73dd84-02df-4f34-bedf-38808cea08fd"/>
            <OMA>
              <OMS cd="vafp" name="pred"/>
              <OMV name="x"/>
            </OMA>
            <OMV name="y"/>
          </OMA>
        </OMA>
      </OMA>
    </OMBIND>
  </OMOBJ>
</definition>
```

Abbildung 5.12: OMDoc Repräsentation der Funktionen +

Symbols mit dem zusätzlichem Suffix `.def`. Um den eigentlichen Rumpf einer Funktion zu repräsentieren, werden auch hier wieder von OMDoc unterschiedliche Vorgehensweisen vorgestellt. ✓`eriFun` bedient sich hier der `simple`n Definition. In diesem Typ von OMDoc-Definition wird ein Funktionsrumpf direkt in der OpenMath Syntax gespeichert. Zur Vervollständigung ist in Abbildung 5.13 auch die Funktion $*$ im OMDoc-Format angegeben. An dem Beispiel dieser Funktionsdefinition ist nicht nur die Selbst-Referenzierung bzw. der rekursive Aufruf des zu definierenden Symbols innerhalb des Rumpfs zu erkennen (`vfead69567-0d1c-422d-b6f1-647e2c82e672`), sondern auch die Referenzierung des zuvor neu in diese Theorie eingeführten Symbols +

```
<symbol
  name="vfead69567−0d1c−422d−b6f1−647e2c82e672">
  <type system="simpletypes">
    <OMOBJ
      xmlns="http://www.openmath.org/OpenMath">
      <OMA>
        <OMS cd="simpletypes" name="funtype"/>
        <OMS cd="vafp" name="nat"/>
        <OMS cd="vafp" name="nat"/>
        <OMS cd="vafp" name="nat"/>
      </OMA>
    </OMOBJ>
  </type>
</symbol>
```

```
<definition name="vfead69567−0d1c−422d−b6f1−647e2c82e672.def"
  for="#vfead69567−0d1c−422d−b6f1−647e2c82e672" type="simple"
  existence="#vf29263776−b6cd−48b8−9951−b0b5fb52fdbb">
  <OMOBJ xmlns="http://www.openmath.org/OpenMath">
    <OMBIND>
      <OMS cd="vafp" name="function"/>
      <OMBVAR>
        <OMATTR>
          <OMATP>
            <OMS cd="simpletypes" name="type"/>
            <OMS cd="vafp" name="nat"/>
          </OMATP>
          <OMV name="x"/>
        </OMATTR>
        <OMATTR>
          <OMATP>
            <OMS cd="simpletypes" name="type"/>
            <OMS cd="vafp" name="nat"/>
          </OMATP>
          <OMV name="y"/>
        </OMATTR>
      </OMBVAR>
      <OMA>
        <OMS cd="vafp" name="if"/>
        <OMA>
          <OMS cd="vafp" name="is_zero"/>
          <OMV name="x"/>
        </OMA>
        <OMS cd="vafp" name="zero"/>
        <OMA>
          <OMS
            cd="VeriFun"
            name="vf7c73dd84−02df−4f34−bedf−38808cea08fd"/>
          <OMA>
            <OMS
              cd="VeriFun"
              name="vfead69567−0d1c−422d−b6f1−647e2c82e672"/>
            <OMA>
              <OMS cd="vafp" name="pred"/>
              <OMV name="x"/>
            </OMA>
            <OMV name="y"/>
          </OMA>
          <OMV name="y"/>
        </OMA>
      </OMA>
    </OMBIND>
  </OMOBJ>
</definition>
```

Abbildung 5.13: OMDoc Repräsentation der Funktionen ∗

(`vf7c73dd84-02df-4f34-bedf-38808cea08fd`). Die eigentlichen Symbolnamen, die Notation betreffenden Informationen und sämtliche weitere ✓eriFun-systeminternen Informationen werden, wie für Sorten schon beschrieben, in entsprechenden `presentation`-Elementen enkodiert.

Neben den benutzerdefinierten Funktionen und den aus Sorten generierten Structure-

Tests, existieren in ✓eriFun noch weitere, sogenannte *synthetisierte* Funktionen. Diese synthetisierten Funktionen unterstützen sowohl das System als auch den Benutzer in der Beweisführung von Lemmata bzw. von der Terminierung von Funktionen. Diese Funktionen werden in Abhängigkeit von Eigenschaften benutzerdefinierter Funktionen vollautomatisch durch ✓eriFun erzeugt. In der abstrakten Klasse der synthetisierten Funktionen werden zwei explizite Formen unterschieden: *Domain-* und *Differenz-*Funktionen (Abbildung 5.14). *Domain*-Funktionen werden durch ✓eriFun für partiell definierte Funktionen generiert, so dass wenn ein Aufruf der partiell definierten Funktion $f : \tau_1 \times \cdots \times \tau_n \rightharpoonup \tau$ determiniert ist, der Aufruf der entsprechenden *Domain*-Funktion $\nabla f : \tau_1 \times \cdots \times \tau_n \rightarrow bool$ den Wert *true* liefert. *Differenz*-Funktionen werden durch ✓eriFun für *p-argumentbeschränkte*-Funktionen [WS05] generiert, so dass wenn der Aufruf der *p-argumentbeschränkten*-Funktion $f : \tau_1 \times \cdots \times \tau_n \rightarrow \tau$ einen Wert echt kleiner als der Wert des p-ten Parameters liefert, der Aufruf der entsprechenden *Differenz*-Funktion $\Delta^p f : \tau_1 \times \cdots \times \tau_n \rightarrow bool$ den Wert *true* liefert. Beiden Spezialisierungen

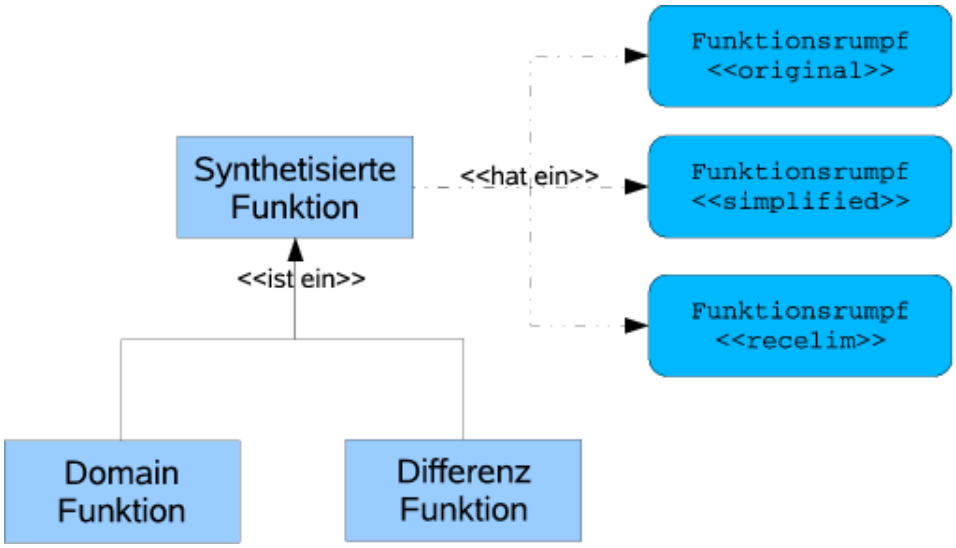


Abbildung 5.14: Synthetisierte ✓eriFun Funktionen

ist gemein, dass für diese Funktionen drei Rümpfe existieren. Der erste Rumpf entsteht direkt aus der Synthese des Systems, der zweite Rumpf stellt als vereinfachter Rumpf die Basis für Rekursionseliminationen dar, woraus der dritte Rumpf entsteht, nachdem die Eliminationen durchgeführt worden sind. Auch diese Funktionen werden nach OMDoc enkdodiert, so dass diese in einer späteren Dekodierung dem ✓eriFun -Dekoder zur Verfügung stehen, um zum einen Vergleiche mit den aus dem System generierten Funktionen vornehmen zu können und zum anderen, wie bereits in der Enkodierung von Structure-Tests angesprochen, eine Transformation in ein anderes Präsentationsformat ermöglichen zu können. Der Unterschied in der Enkodierung solcher Funktionen besteht im Anhängen eines weiteren Suffixes an den Symbol- bzw. Definitionsnamen und der Anzahl der enkodierten Funktionsrümpfe. Um Deklarationen synthetisierter Funktions-

symbole von benutzerdefinierten unterscheiden zu können, erhalten *Domain*-Funktionen das zusätzliche Suffix `dom` und *Differenz*-Funktionen das zusätzliche Suffix `diff`. In den jeweiligen Funktionsdefinitionen werden, entsprechend dem zu enkodierendem Funktionsrumpf, das Suffix `original`, `simplified` oder `recelim` hinzugefügt (Listing 5.15).

Listing 5.15: OMDoc Repräsentation der Funktionen / und Δ^1/

```
========== Enkodierung der Funktion / ==========
<symbol name="vf2c091d8a-a68d-41a6-a62b-da7b556604a6">...</symbol>
<definition name="vf2c091d8a-a68d-41a6-a62b-da7b556604a6.def"
            for="#vf2c091d8a-a68d-41a6-a62b-da7b556604a6"...>...</definition>
<presentation for="#vf2c091d8a-a68d-41a6-a62b-da7b556604a6"...>
  <use format="VeriFun">/</use>
</presentation>
========== Enkodierung der Funktion Δ1/ ==========
<symbol name="vf7eac82e6-ec65-4421-a0d9-7f60478ce29f.dom">...</symbol>
<definition name="vf7eac82e6-ec65-4421-a0d9-7f60478ce29f.dom.original.def"
            for="#vf7eac82e6-ec65-4421-a0d9-7f60478ce29f.dom"...>...</definition>
<definition name="vf7eac82e6-ec65-4421-a0d9-7f60478ce29f.dom.simplified.def"
            for="#vf7eac82e6-ec65-4421-a0d9-7f60478ce29f.dom"...>...</definition>
<definition name="vf7eac82e6-ec65-4421-a0d9-7f60478ce29f.dom.recelim.def"
            for="#vf7eac82e6-ec65-4421-a0d9-7f60478ce29f.dom"...>...</definition>
<presentation for="#vf7eac82e6-ec65-4421-a0d9-7f60478ce29f.dom"...>
  <use format="VeriFun">Δ1/</use>
</presentation>
```

5.1.5 Lemmata

In ✓eriFun werden Lemmata durch das Schlüsselwort `lemma` eingeleitet. Die Definition eines Lemmas besteht aus dessen Namen, den gebundenen Variablen, einschließlich deren Typs und einem booleschen Term als Rumpf. Fast man die in einem Rumpf verwendete, am weitesten außen stehende Relation, in diesem Beispiel wäre das die Gleichheitsrelation "=", als eine Funktion auf, so kann der Rumpf eines Lemmas analog zu dem einer Funktion angesehen werden. Diese Betrachtungsweise des Rumpfes eines Lemmas wird auch in dem `OMDocEncoder` angewandt. In dem Beispiel aus Abbildung 5.16 bedeutet dies, dass in dem Rumpf des Lemmas `+_commutative` die Funktion "=" auf zwei Argumente angewandt wird. Das erste Argument ist $(x + y)$ und das zweite $(y + x)$. Der Vorteil besteht also darin, dass in dem `OMDocEncoder` keine Unterscheidung hinsichtlich der Enkodierung des Rumpfes eines Lemmas und des Rumpfes einer Funktion vorgenommen werden muss. In Abbildung 5.16 ist die OMDoc-Repräsentation des Lemmas `+_commutative` angegeben. Das `assertion` Element dient in OMDoc zur Formulierung jeglicher Art von Behauptungen. Mit dem zusätzlichen Attribut `xml:id` ist auch hier eine eindeutige Referenzierung einer *Assertion* möglich. Da in der Mathematik unterschiedliche Typen von Behauptungen existieren, können diese in OMDoc mit dem Attribut `type` gekennzeichnet werden. ✓eriFun verwendet hier zwei Arten von Klassifikation. Zum einen den Typ `lemma` und zum anderen den Typ `theorem`. Mit dem Typ `lemma` werden die Lemmata aus ✓eriFun gekennzeichnet, der Typ `theorem` existiert zwar in dieser Form nicht direkt in dem System, wird aber durch den `OMDocEncoder` zur Enkodierung von Terminierungsbeweisen (Unterabschnitt 5.1.7) und Beweisen von Rekursionseliminationen (Unterabschnitt 5.1.8) verwendet. Der formale Teil einer *Assertion*,

```
lemma +_commutative <= ∀ x, y : ℕ
  (x + y) = (y + x)
```

```
<assertion xml:id="vf0d942661−9b54−42a3−9ad1−58719257a737" type="lemma">
  <FMP>
    <OMOBJ xmlns="http://www.openmath.org/OpenMath">
      <OMBIND>
        <OMS cd="pl1" name="forall"/>
        <OMBVAR>
          <OMATTR>
            <OMATP>
              <OMS cd="simpletypes" name="type"/>
              <OMS cd="vafp" name="nat"/>
            </OMATP>
            <OMV name="x"/>
          </OMATTR>
          <OMATTR>
            <OMATP>
              <OMS cd="simpletypes" name="type"/>
              <OMS cd="vafp" name="nat"/>
            </OMATP>
            <OMV name="y"/>
          </OMATTR>
        </OMBVAR>
        <OMA>
          <OMS cd="vafp" name="equal"/>
          <OMA>
            <OMS
              cd="VeriFun"
              name="vfd2f0cfc4−aa41−44c7−b628−490646dee781"/>
            <OMV name="x"/>
            <OMV name="y"/>
          </OMA>
          <OMA>
            <OMS
              cd="VeriFun"
              name="vfd2f0cfc4−aa41−44c7−b628−490646dee781"/>
            <OMV name="y"/>
            <OMV name="x"/>
          </OMA>
        </OMA>
      </OMBIND>
    </OMOBJ>
  </FMP>
</assertion>
```

Abbildung 5.16: ✓eriFun und OMDoc Repräsentation des Lemmas +_commutative

was in diesem Fall dem Rumpf des ✓eriFun-Lemmas entspricht, wird aus oben genannten Gründen in gewohnter Weise in OpenMath-Syntax gespeichert. In der Enkodierung des Namens stellen Lemmata jedoch eine Ausnahme gegenüber den bisher erwähnten ✓eriFun-Elementen dar. Es ist in OMDoc nicht zulässig von presentation-Elemente auf ein assertion-Element zu verweisen. In OMDoc tragen *Assertions* keinen Namen, sondern lediglich eine eindeutige ID, so dass keine weiteren die Notation betreffenden Informationen ausgezeichnet werden müssen. Aus diesem Grund unterscheidet sich die Enkodierung der systeminternen Information eines Lemmas zu denen von Funktionen und Sorten. In dem private-Block eines Lemmas wird zusätzlich zu dem Kommentar und dem "Vater-Ordner" auch der Name des Lemmas gespeichert. Dies könnte eine An-

regung zur Erweiterung der Sprache OMDoc sein, da nicht nur in ✓eriFun Lemmata mit einem Namen versehen werden können.

Neben den benutzerdefinierten Lemmata, existieren in ✓eriFun noch zwei weitere: Die *Boundedness*- und die *Projektions*-Lemmata. *Boundedness*-Lemmata werden durch ✓eriFun automatisch in das aktuelle Programm hinzugefügt, wenn durch den Benutzer eine *p-argumentbeschränkte*-Funktion [WS05] $f(x_1 : sort_1, \ldots, x_n : sort_n) : \mathbb{N} <= \ldots$ definiert worden ist. Mit dem entsprechenden durch das System generierten *Boundedness*-Lemma $f\$p\_bounded$ wird folgende Aussage formuliert:

$$\begin{aligned}&\forall q_1, \ldots q_p, \ldots q_n \in \mathbb{N}.\\ &\quad f(q_1, \ldots q_p, \ldots, q_n) \not> q_p \qquad \wedge\\ &\quad \Delta^p f(q_1, \ldots, q_p, \ldots q_n) = true \Rightarrow q_p > f(q_1, \ldots, q_p, \ldots q_n) \qquad \wedge\\ &\quad \Delta^p f(q_1, \ldots, q_p, \ldots q_n) = false \Rightarrow q_p = f(q_1, \ldots, q_p, \ldots q_n)\end{aligned}$$

Projektions-Lemmata werden ebenfalls durch ✓eriFun automatisch in das aktuelle Programm hinzugefügt. In diesem Fall gilt folgende Regel: Wenn durch den Benutzer eine *p-argumentbeschränkte*-Funktion [WS05] $f(x_1 : sort_1, \ldots, x_n : sort_n) : sort <= \ldots$ definiert worden ist ($sort \neq \mathbb{N}$), wird durch ✓eriFun ein *Projections*-Lemma $f\$p\_projection$ dem aktuellen Programm hinzugefügt. Solch ein Lemma beschreibt formal folgende Aussage:

$$\begin{aligned}&\forall q_1 : sort_1, \ldots q_n : sort_n.\\ &\quad \Delta^p f(q_1, \ldots q_n) \Rightarrow f(q_1, \ldots q_n) = q_p\end{aligned}$$

Die auf diese Weise neu in das aktuelle Programm hinzugekommenen Lemmata stellten ein weiteres Hilfsmittel, sowohl dem Benutzer als auch dem System selbst, in der Durchführung von weiteren Beweisen zur Verfügung. Desweiteren können sie zur Formulierung von weiteren Aussagen über die Argumentbeschränktheit der entsprechenden Funktion f eingesetzt werden. Die Enkodierung solcher synthetisierter Lemmata unterscheidet sich von der der benutzerdefinierten lediglich durch das Anhängen eines Suffixes. So wird der `ID` des `xml:id`-Attributs des `assertion`-Elements in beiden Fällen das Suffix `proj` angehängt. In der nächsten Version des `OMDocEncoders` sollte hier in dem Suffix zwischen `proj` und `bounded` unterschieden werden.

5.1.6 HPL Beweise

In ✓eriFun werden zu beweisende Aussagen mit dem sogenannten HPL-Kalkül verifiziert. Dem Anwender stehen hierzu eine Reihe von Taktiken zur Verfügung, die die einzelnen Inferenzregeln des Kalküls zu intelligenten Einheiten zusammenfassen (Abbildung 5.17). Sollte es ✓eriFun nicht gelingen, einen Beweis automatisch verifizieren zu können, so kann der Anwender selbst durch die explizite Anwendung einer Taktik, in den jeweiligen Beweis eingreifen, um das System in der weiteren Durchführung des Beweises zu unterstützen. Als Beispiel ist in Abbildung 5.18 der erste Teil eines Beweises in ✓eriFun für das Lemma aus Abbildung 5.16 dargestellt. Das System wendet die HPL-

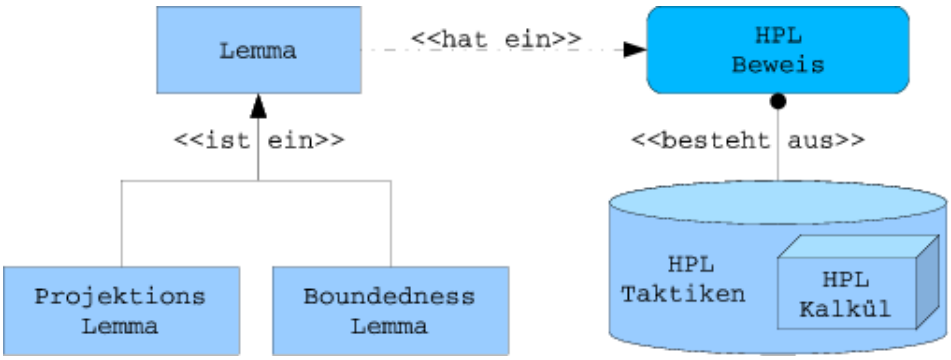


Abbildung 5.17: √eriFun HPL Beweise

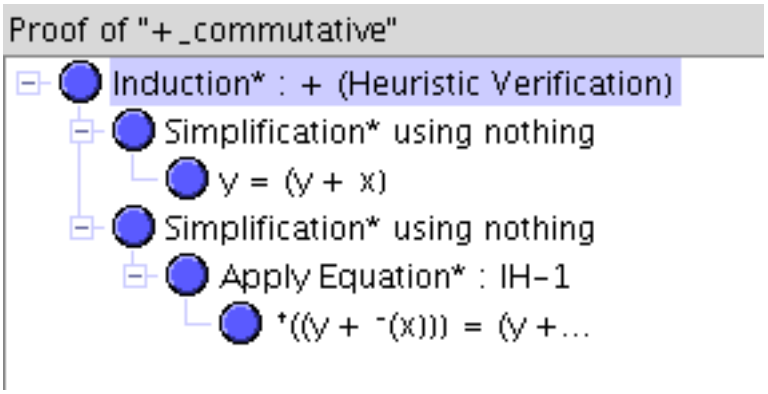


Abbildung 5.18: √eriFun Teilbeweis von Abbildung 5.16

Taktik `Heuristic Verification` an, die u.a die HPL-Inferenzregel `Induction` auf den ersten Beweisknoten anwendet. Die Applikation dieser Inferenzregel hat zwei weitere Beweisknoten zur Folge: Den Induktionsanfang und den Induktionsschritt. In beiden Fällen wird eine sogenannte `Simplification` angewandt, um den zu beweisenden Knoten durch symbolische Auswertungen zu vereinfachen. In dem Induktionschritt versucht √eriFun durch Anwendung der HPL-Taktik `Apply Equation` weitere Vereinfachungen vorzunehmen, um dann, wie auch schon im Basisfall, zu einen Abbruch zu gelangen. An dieser Stelle kann das System keine weiteren heuristisch sinnvollen HPL-Regeln finden, so dass hier ein Eingreifen des Benutzers von Nöten ist. In Abbildung 5.19 ist der vollständige Beweis von Abbildung 5.16, nachdem der Benutzer dem System geholfen hat, dargestellt[2].
An den jeweiligen Stellen, an den √eriFun den Beweis nicht automatisch fortführen konnte, reichte bereits die explizite Anwendung der `Move Hypotheses`-Taktik, um den Beweis vollständig durch das System zu Ende führen zu lassen. Zur Anwendung von den zur Vefügung stehenden HPL-Taktiken bzw. HPL-Inferenzregeln werden von √eriFun Zusatzinformationen benötigt. Diese zusätzlichen Informationen müssen entweder durch

[2]Mit * gekennzeichnete Regelnamen bedeutet, dass diese Regel durch eine Taktik aufgerufen wurde

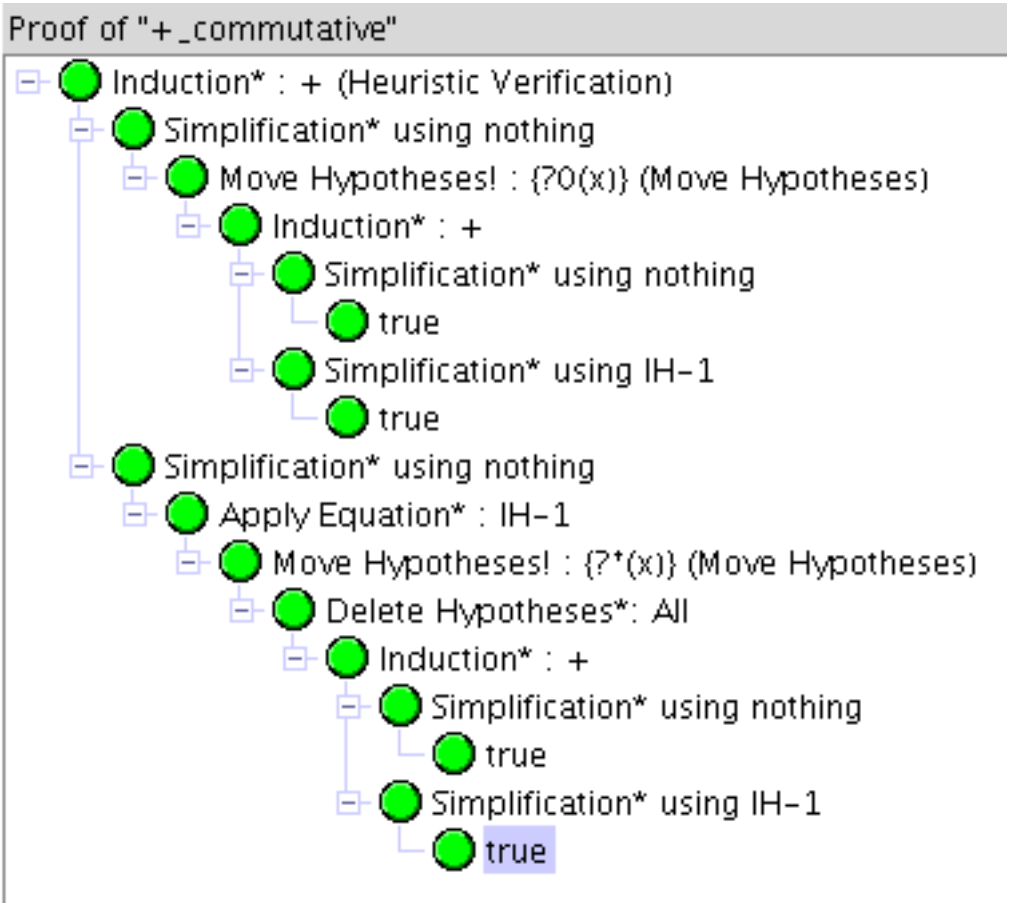


Abbildung 5.19: Vollständiger ✓eriFun Beweis von Abbildung 5.16

den Benutzer explizit angeben werden oder werden durch das System heuristisch berechnet. ✓eriFun speichert zu einem jeden Beweisknoten diese Daten in einem sogenannten `Oracle`, welches der jeweiligen HPL-Taktik bzw. -Inferenzregel durch das System übergeben wird. In Abbildung 5.20 sind die heuristisch durch das System berechneten Daten für die erste Inferenzregel, die Induktion, für den ersten Beweisschritt abgebildet. Das `L` repräsentiert die bei der Berechnung verwendeten Lemmata, `F` die zu verwendende Funktion, `B` das zu verwendende Element, `M` die gewählte Rekursionspositionsmenge, `nu` und `theta` Substitutionen und `R` die Relationsbeschreibung. Neben diesen zusätzlichen Informationen eines Beweisknotens für die Anwendung einer Regel ist jeder Beweisknoten desweiteren mit einer sogenannten Sequenz annotiert. ✓eriFun Sequenzen haben die folgende Form:

$$h_1, \ldots h_n, \forall \ldots ih_1, \ldots, \forall \ldots ih_l \vdash goal$$

Mit $h_1, \ldots h_n$ werden die in diesem Beweisknoten zur Verfügung stehenden *Hypothesen*, mit $\forall \ldots ih_1, \ldots, \forall \ldots ih_l$ die zur Verfügung stehenden *Induktionshypothesen* und mit *goal* die zu zeigende Behauptung denotiert. Für den Induktionsschritt aus Abbil-

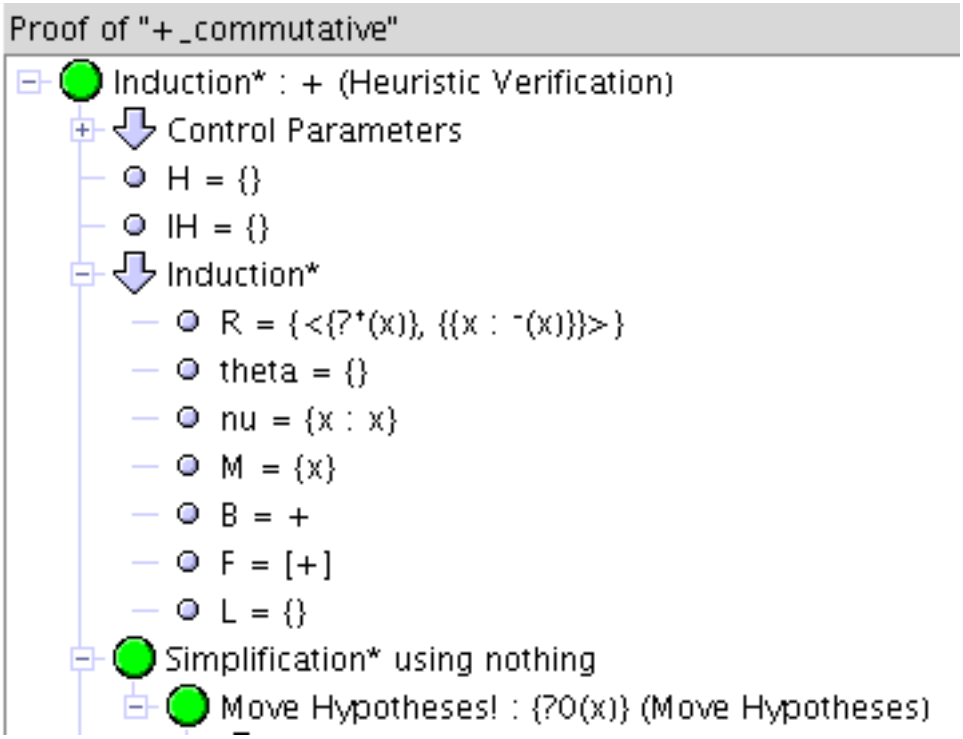


Abbildung 5.20: Ein ✓eriFun Induktions-Orakel

dung 5.19 hat die Sequenz somit folgende explizite Form

$$?^{+}(x),\ \forall y' : \mathbb{N}.(^{-}(x) + y') = (y' +^{-} (x)) \vdash (x + y) = (y + x)$$

In OMDoc wird ein solcher ✓eriFun-Beweisbaum inklusive aller zusätzlichen Informationen (`Oracle`) und Sequenzen innerhalb eines `proof`-Elements repräsentiert. Mit dem Attribut `for` wird angegeben, zu welchem `assertion`-Element dieser Beweis gehört. Die einzelnen Beweisschritte werden mit dem `derive`-Element eingeleitet. Die jeweilige Sequenz eines Beweisknoten wird als `FMP`-Element direkt an das entsprechende `derive`-Element angehängt. Innerhalb eines `FMP` Blockes werden die Hypothesen und Induktionshypothesen mit `assumption` Elementen denotiert. Durch das zusätzliche `inductive` Attribut werden diese untereinander differenziert. Als letztes Element jedes `FMP` Blockes wird das *Goal* in einem `conclusion`-Element dargestellt. Werden Hypothesen, Induktionshypothesen oder das *Goal* in darauffolgenden Beweisschritten unverändert erneut verwendet, so werden diese durch ein `ref`-Element referenziert. Die entsprechende Referenz wird in dem `xref`-Attribut eingetragen. In dem nächsten direkten Kind-Element von `derive` wird enkodiert welche Methode, bzw. hier Regel, auf den aktuellen Goalterm angewandt wurde. OMDoc hat hierfür das `method`-Element vorgesehen. In dem zusätzlichen `xref` Attribut wird die Referenz auf die entsprechende Methode gespeichert. An eine Methode können optional noch Parameter in Form von `OMOBJ`-Elementen hinzugefügt werden. Diese Option wird von dem `OMDocEncoder` verwendet, um die zusätz-

lichen Informationen einer HPL-Regel zu enkodieren. Somit wird jede einzelne Informationen in einem `OMOBJ`-Element gekapselt. Eine solche geordnete Liste von `OMOBJ` Elementen als Ganzes bildet das `Oracle` der jeweiligen Regel. In der momentanen Version des `OMDocEncoder`s bzw. `OMDocDecoder`s muss diese Liste von `OMOBJ` Elementen geordnet sein, da die Informationen durch keinen weiteren semantischen Markup annotiert sind. Die Ordnung wird intern durch den `OMDocEncoder` vorgeschrieben und der `OMDocDecoder` weiß, in welcher Reihenfolge er die Teilinformationen eines `Oracle`s einzulesen hat. Sollte diese Sortierung der Informationen eines `Oracle` beispielsweise per Hand modifiziert werden, so kann eine korrekte Dekodierung nicht mehr gewährleistet werden. Auch das ist ein weiterer Punkt, der in der nächsten Version des `OMDocEncoder`s bzw. `OMDocDecoder`s verbessert werden sollte. Eine Lösung würde sich zum Beispiel in der Erweiterung der Semantic Xml Tags für `Oracle`s wiederfinden. In Listing 5.21 wird nun ein Ausschnitt des Beweises aus Abbildung 5.19 in dem OMDoc-Format dargestellt. Zur besseren Lesbarkeit sind nur die zusätzlichen Informationen für die erste Induktion angeben. Desweiteren wird die Enkodierung des Induktionsschrittes nicht angegeben, da sich dessen Länge alleine über mehrere Seiten erstrecken würde, jedoch keine weiteren Erkenntnisse liefert. Die Enkodierung der jeweiligen Sequenzen, repräsentiert in OpenMath-Syntax, wurde durch eine mathematische Schreibweise abgekürzt.

Listing 5.21: Ausschnitt der OMDoc Repräsentation aus Abbildung 5.19

```
<proof for="#vf4cd5d706-f4fc-4560-8043-8b25cc998e77" xml:id="vff05ea7aa-c073-47d6-8bef-87da9689d9ec">
  <derive xml:id="vf404e14db-6419-4cc1-84ce-141db1e2f537">
    <FMP>
      <conclusion xml:id="vf8da3ef8e-7a3b-4c13-8aff-fe35f193b9bd">
        (x + y) = (y + x)
      </conclusion>
    </FMP>
    <method xref="#Induction">
      <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMSTR>Heuristic Verification</OMSTR>
      </OMOBJ>
      <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMR href="#vf8543e879-bd7b-4bbf-9a48-97b11a4ccc10"/>
      </OMOBJ>
      <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMA>
          <OMS cd="set1" name="set"/>
          <OMATTR>
            <OMATP>
              <OMS cd="simpletypes" name="type"/>
              <OMS cd="vafp" name="nat"/>
            </OMATP>
            <OMV name="x"/>
          </OMATTR>
        </OMA>
      </OMOBJ>
      <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMA>
          <OMS cd="set1" name="set"/>
        </OMA>
      </OMOBJ>
      <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMA>
          <OMS cd="set1" name="set"/>
          <OMA>
            <OMS cd="ecc" name="pair"/>
            <OMATTR>
```

```
38                <OMATP>
                    <OMS cd="simpletypes" name="type"/>
                    <OMS cd="vafp" name="nat"/>
                  </OMATP>
                  <OMV name="x"/>
43              </OMATTR>
                <OMV name="x"/>
              </OMA>
            </OMA>
          </OMOBJ>
48        <OMOBJ xmlns="http://www.openmath.org/OpenMath">
            <OMS cd="vafp" name="true"/>
          </OMOBJ>
          <OMOBJ xmlns="http://www.openmath.org/OpenMath">
            <OMS cd="vafp" name="true"/>
53        </OMOBJ>
          <proof xml:id="vf5512ac9d-a984-4de6-a53d-631389c05e23">
            <derive xml:id="vfd4291d6d-aa7d-4401-b24c-ddd10ac4bd02">
              <FMP>
                <assumption xml:id="vfa7dafa57-acd6-4cae-a440-c3a2d830d623">
58                ?0(x)
                </assumption>
                <conclusion xml:id="vf6769c709-9b82-48fd-ae4c-cbbb170c3576">
                  <ref xref="#vf8da3ef8e-7a3b-4c13-8aff-fe35f193b9bd"/>
                </conclusion>
63            </FMP>
              <method xref="#Simplification">
                Parameter der Simplification-Taktik eingeschlossen in OMOBJ Elemente
                <proof xml:id="vfbf5e7a63-c5c2-4a48-801a-8f7d79962f53">
                  <derive xml:id="vf3ea6ed36-3a5b-459c-80cd-1e6bbafe4c6f">
68                  <FMP>
                      <assumption xml:id="vf44b291a0-cae5-4f90-8945-e402a048191c">
                        <ref xref="#vfa7dafa57-acd6-4cae-a440-c3a2d830d623"/>
                      </assumption>
                      <conclusion xml:id="vf7ae0485b-8790-45e1-a715-52bb2673ab84">
73                      y = (y + x)
                      </conclusion>
                    </FMP>
                    <method xref="#Move Hypotheses">
                      Parameter der Move Hypotheses-Taktik eingeschlossen in OMOBJ Elemente
78                    <proof xml:id="vfae3c6c4a-83ba-4d1d-b113-87190e13ff0d">
                        <derive xml:id="vf9c29ee0e-817d-41de-9092-94a777f24551">
                          <FMP>
                            <conclusion xml:id="vfdf08fceb-06ba-4d20-9ef1-b437905c8c3b">
                              x = 0 ⇒ y = (y + x)
83                          </conclusion>
                          </FMP>
                          <method xref="#Induction">
                            Parameter der Induction-Taktik eingeschlossen in OMOBJ Elemente
                            <proof xml:id="vf99fc64a2-0272-494a-a232-56ca8ab75b5a">
88                            <derive xml:id="vf84c67dd1-4fb0-4a08-b93c-19a00d0ff46b">
                                <FMP>
                                  <assumption xml:id="vfcd870ced-09bb-4385-b486-71d9abb82c85">
                                    ?(y)
                                  </assumption>
93                                <conclusion xml:id="vf341a203f-d4cc-4aa2-9fd6-7c9454c50c4f">
                                    <ref xref="#vfdf08fceb-06ba-4d20-9ef1-b437905c8c3b"/>
                                  </conclusion>
                                </FMP>
                                <method xref="#Simplification">
98                                Parameter der Simplification-Taktik eingeschlossen in OMOBJ Elemente
                                  <proof xml:id="vfa854de66-6058-4261-97d1-771cf887812b">
                                    <derive xml:id="vf1541d18c-eb30-465e-818a-a6afddcc4d66">
                                      <method>
```

```
                                        <premise xref="#true_ax"/>
103                                   </method>
                                    </derive>
                                  </proof>
                                </method>
                              </derive>
108                         </proof>
                            <proof xml:id="vf7c67bb73−50f8−4270−af2b−85d9f5287feb">
                              <derive xml:id="vfc9f1d9d8−8bd2−494f−9c76−624e362295e2">
                                <FMP>
                                  <assumption xml:id="vf7cf934b1−696a−44c2−b7c7−0e73d693cba3">
113                                 $?^{+}(y)$
                                  </assumption>
                                  <assumption xml:id="vf53ca35c4−f5eb−4cf2−8f64−373628fc12f3" inductive="yes">
                                    $\forall x'.x' = 0 \Rightarrow^{-} (y) = ({}^{-}(y) + x')$
                                  </assumption>
118                               <conclusion xml:id="vf6654ebf7−fe32−428e−a50d−edb829aa2f29">
                                    <ref xref="#vfdf08fceb−06ba−4d20−9ef1−b437905c8c3b"/>
                                  </conclusion>
                                </FMP>
                                <method xref="#Simplification">
123                               Parameter der Simplification-Taktik eingeschlossen in OMOBJ Elemente
                                  <proof xml:id="vf8fece78e−f844−43a2−897c−a061913e9ea6">
                                    <derive xml:id="vf730efd8a−f356−4d22−b928−fb66cfce909f">
                                      <method>
                                        <premise xref="#true_ax"/>
128                                   </method>
                                    </derive>
                                  </proof>
                                </method>
                              </derive>
133                         </proof>
                          </method>
                        </derive>
                      </proof>
                    </method>
138                 </derive>
                  </proof>
                </method>
              </derive>
            </proof>
143         <proof xml:id="vf50f77705−6934−4f5c−b04f−c62484c69692">
              ...
            </proof>
          </method>
        </derive>
148   </proof>
```

5.1.7 Terminierungsbeweise

In der Durchführung von Terminierungsbeweisen in ✓eriFun wird der Benutzer durch den sogenannten pE-Kalkül [WS05] unterstützt. Mit Hilfe dieses Kalküls generiert das System auf Grund der formalen Funktionsparameter und dem Funktionsrumpf verschiedene Terminierungsargumente in Form von Maßtermen und dazugehörigen Terminierungshypothesen. Alternativ kann der Benutzer auch interaktiv weitere Maßterme angeben, zu denen ebenfalls die zugehörigen Terminierungshypthesen berechnet werden. Sind für einen Maßterm alle daraus erzeugten Terminierungshypothesen verifiziert, so ist die Terminierung der Funktion bewiesen. Beweise von Terminierungshypothesen werden in ✓eriFun mit dem HPL-Kalkül durchgeführt (Abbildung 5.22). Das ist ein großer Vorteil für den `OMDocEncoder`, da somit keine Unterscheidung zwischen Beweisen für Lemmata und Beweisen für Terminierungshypothesen vorgenommen werden muss. Die

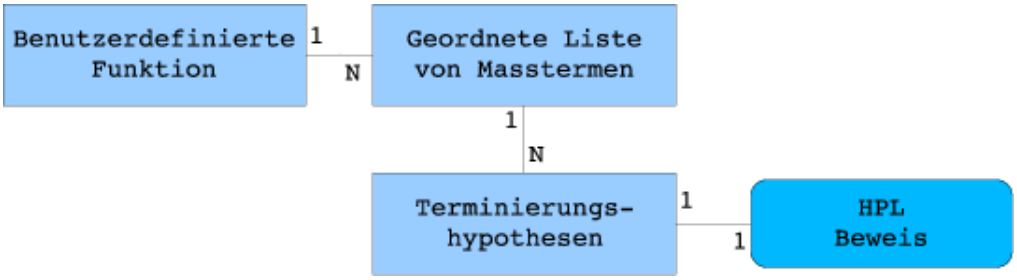


Abbildung 5.22: ✓eriFun Terminierungsbeweise von benutzerdefinierten Funktionen

einzelnen Terminierungshypothesen werden analog zu Lemmata enkodiert. Der einzige Zusatz in der Enkodierung von Terminierungsbeweisen von benutzerdefinierten Funktionen besteht in der Enkodierung der Terminierungsargumente, d.h der Maßterme. Hinzu kommt, dass Beweise in OMDoc immer nur auf ein `assertion`-Element verweisen dürfen und nicht direkt auf ein Symbol oder eine Definition. In Folge dessen wird für jede benutzerdefinierte ✓eriFun-Funktion eine *Assertion* durch den `OMDocEncoder` generiert, in der die informale Behauptung getroffen wird, dass diese Funktion terminiert (Listing 5.23).

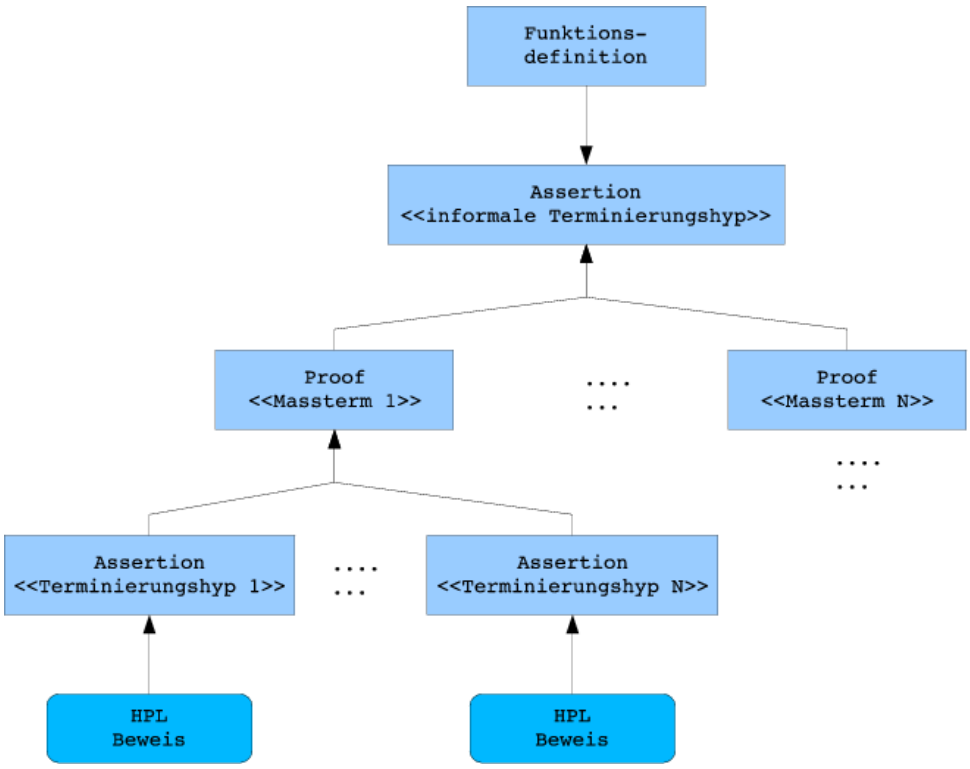
Listing 5.23: Informale Terminierungshypothese in OMDoc

```
<assertion xml:id="vf72c05981-ebbd-494f-ba51-62cdcbdfa39e" type="theorem">
  <CMP>
    <OMOBJ xmlns="http://www.openmath.org/OpenMath">
      <OMS cd="vafp" name="vf7c73dd84-02df-4f34-bedf-38808cea08fd"/>
    </OMOBJ>terminates.</CMP>
</assertion>
```

Im Klartext lautet diese Behauptung: "<*Funktion*> *terminiert*". Diese informale Aussage wird in einem `assertion`-Element beschrieben. Das `assertion`-Element erhält in

dem `type`-Attribut den Wert `theorem`. Auf diese Weise wird es auch dem ✓eriFun-Dekoder ermöglicht, informale Terminierungshypothesen von Lemmata (`type=lemma`) zu unterscheiden. Die informale Aussage, dass eine Funktion terminiert, wird innerhalb des entsprechenden `assertion`-Elements mit einem `CMP`-Element dargestellt. Um in dieser *Commented Mathematical Property* nicht den Namen der betreffenden Funktion angeben zu müssen, wird das Funktionssymbol in OpenMath-Syntax dargestellt. Die Verbindung zwischen solch einer informalen Terminierungshypothese und der betreffenden benutzerdefinierten Funktion, wird in OMDoc mittels des `existence`-Attribut innerhalb des `definition`-Elements hergestellt. Der Wert dieses Attributs ist die Referenz auf die informale Terminierungshypothese (Abbildung 5.12). Auf solch eine generiertes `assertion`-Element kann nun in OMDoc mit `proof`-Elementen verwiesen werden. Der `OMDocEncoder` kapselt jeden Maßterm, bzw. jede Liste von Maßtermen, in einen eigenen Beweis. Dieser Beweis referenziert in dem `for`-Attribut das entsprechende `assertion`-Element. Da Terminierungsargumente in ✓eriFun basierend auf dem pE-Kalkül erzeugt werden und der Benutzer bis auf die Modifikation der Maßterme keine weiteren Eingriffsmöglichkleiten besitzt, wird als verwendete Beweismethode zur Erstellung der Terminierungsargumente der gesamte pE-Kalkül in dem `xref`-Attribut des `method`-Elements angegeben[3]. Analog zu der Parameterübergabe einer HPL Methode, werden die von der Beweismethode zusätzlich benötigten Informationen als eine geordnete Liste von `OMOBJ`-Elementen direkt an das `method`-Element angehängt. Wie auch schon in der Enkodierung von HPL Beweisen, so ist auch hier die Reihenfolge dieser Zusatzinformationen fix durch den `OMDocEncoder` vorgegeben. So wird neben dem eigentlichen Maßterm bzw. der Liste von Maßtermen, der Methode noch zusätzlich eine Liste von Paaren und der Typ des Maßterms selbst übergeben. In den Paaren der Liste wird in der ersten Komponente die Referenz auf das die Terminierungshypothese repräsentierende `assertion`-Element gespeichert. Die zweite Komponente stellt die Position, aus der die Hypothese in dem betreffenden Funktionrumpf gewonnen wurde, dar. Bezüglich des Typs eines Maßterms unterscheidet ✓eriFun zwischen benutzerdefinierten und systemgenerierten Maßtermen. Diese Information wird als zusätzliche Information an die aktuellen Methode angehängt. Der Wert `true` denotiert in diesem Fall, dass es sich um ein systemgeneriertes Terminierungsargument handelt. Zur Veranschaulichung ist im oberen Teil der Abbildung 5.24 ein Überblick der hier beschriebenen OMDoc-Enkodierung von Terminierungsbeweisen benutzerdefinierter Funktionen dargestellt und im unteren Teil ist ein Ausschnitt solch eines konkreten Terminierungsbeweises im OMDoc-Format zu sehen.

[3] Die Unterscheidung zwischen systemgenerierten und benutzerdefinierten Terminierungsargumenten wird direkt an die Maßterme annotiert.




```
<proof xml:id="vf9802b5c8-3350-4a58-a5a8-d22b41319d23" for="#vf72c05981-ebbd-494f-ba51-62cdcbdfa39e">
  <derive xml:id="vfd11265e4-d703-4e2e-8c2a-81deaf45532c">
    <method xref="#PE-Calculus">
      <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMA>
          <OMS cd="vafp" name="measure-term"/>
          <OMATTR>
            <OMATP>
              <OMS cd="simpletypes" name="type"/>
              <OMS cd="vafp" name="nat"/>
            </OMATP>
            <OMV name="x"/>
          </OMATTR>
        </OMA>
      </OMOBJ>
      <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMA>
          <OMS cd="vafp" name="termination-hyps"/>
          <OMA>
            <OMS cd="ecc" name="pair"/>
            <OMR href="#vfd67e73e3-186b-4205-87fe-7381218feca0"/>
            <OMA>
              <OMS cd="vafp" name="position"/>
              <OMI>3</OMI>
              <OMI>1</OMI>
            </OMA>
          </OMA>
        </OMA>
      </OMOBJ>
      <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMS cd="vafp" name="true"/>
      </OMOBJ>
    </method>
  </derive>
</proof>
```

Abbildung 5.24: Enkodierung von Abbildung 5.22 in OMDoc

5.1.8 Rekursionseliminationsbeweise

Für jede benutzerdefinierte Funktion, für die die Terminierung bereits bewiesen wurde, generiert ✓eriFun vollautomatisch die sogenannten synthetisierten Funktionen. Für jede erzeugte synthetisierte Funktion wird desweiteren eine Menge von Rekursionseliminationsformeln erstellt, deren Korrektheit ebenfalls in Form von HPL-Beweisen gezeigt werden kann (Abbildung 5.25). Die Enkodierung von solchen Rekursionseliminationsbeweisen gestaltet sich in ähnlicher Weise wie die der Terminierungsbeweise, nur dass hier die zusätzlichen Informationen nicht aus Maßtermen besteht, sondern aus den Argumenten, die für den Nachweis der Korrektheit der Rekursionseliminationen benötigt werden. In diesen Argumenten, den Rekursionseliminationen, werden für jede Kombination von rekursiven Aufrufen innerhalb des Rumpfes der zugehörigen benutzerdefinierten Funktion zwei Formeln definiert. Eine für den Fall, dass die rekursiven Aufrufe durch `true` und eine für den Fall, dass die rekursiven Aufrufe durch `false` ersetzt werden können. Die einzelnen Formeln werden in gewohnter Weise durch `assertion`-Elemente repräsentiert und deren HPL-Beweis in entsprechenden `proof`-Elementen. Da auch diese Rekursions-

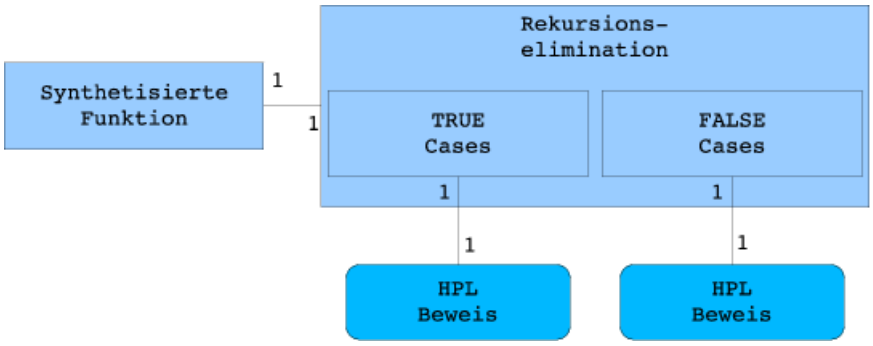
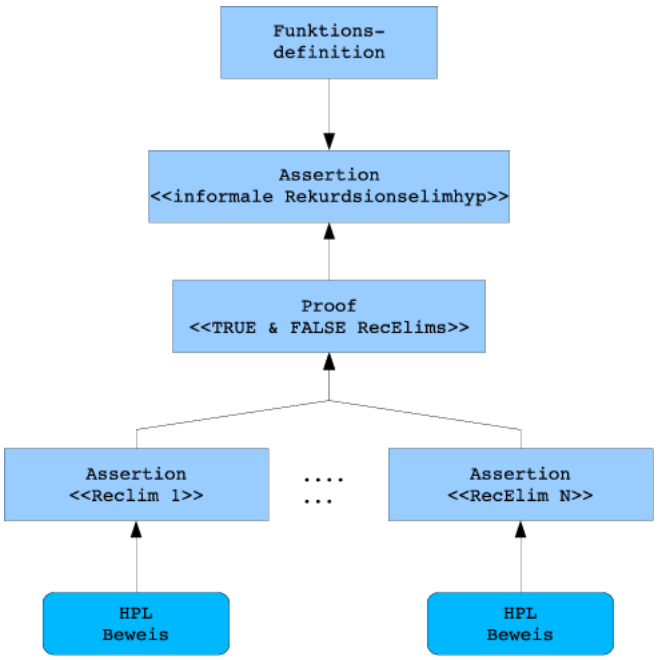


Abbildung 5.25: ✓eriFun -Rekursionseliminationsbeweise von synthetisierten Funktionen

eliminationen in ✓eriFun vollautomatisch erstellt werden, wird als verwendete Beweismethode *System*[4] in dem `xref`-Attribut des `method`-Elements angegeben. Die Parameter für diese Methode setzten sich hier zum einen aus der Information über den Status der synthetisierten Funktion und zum anderen aus zwei Listen von Paaren zusammen. Der Wert für den Status der Funktion wird auf `true` gesetzt, sobald diese in ✓eriFun den *Verified*-Status erhalten hat. Ansonsten ist dieser Wert `false`. In den beiden Listen werden die Rekursionseliminationen für den `true`- und den `false`-Fall unterschieden. In den Paaren der Listen wird in der ersten Komponente die Referenz auf das die Formel repräsentierende `assertion`-Element gespeichert. Die zweite Komponente stellt die Positionenmenge, an der die Formel in dem betreffenden Funktionrumpf gewonnen wurden, dar. Zur Veranschaulichung wird auch hier in dem oberen Teil der Abbildung 5.26

[4]Der Algorithmus für die Rekursionseliminationen trägt keinen speziellen Namen in ✓eriFun.

ein Überblick der in diesem Abschnitt beschriebenen OMDoc-Enkodierung von Beweisen zur Rekursionselimination synthetisierter Funktionen dargestellt und in dem unteren Teil ist ein Ausschnitt solch eines konkreten Beweises im OMDoc-Format repräsentiert.




```
<proof xml:id="vfdab0cfb8-72ed-4130-91d7-56790e033012" for="#vf2909c705-1cc1-4ec7-a00b-7c49fed9dc43">
  <derive xml:id="vf26c74bac-d861-41fc-9c38-7c334fe6e2a5">
    <method xref="#System">
      <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMS cd="vafp" name="true"/>
      </OMOBJ>
      <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMA>
          <OMS cd="vafp" name="recelim-true"/>
          <OMA>
            <OMS cd="ecc" name="pair"/>
            <OMR href="#vf9cd0a44b-b3c8-4c64-8cb3-551d81caba69"/>
            <OMA>
              <OMS cd="vafp" name="position"/>
              <OMA>
                <OMS cd="list1" name="list"/><OMI>3</OMI><OMI>3</OMI>
              </OMA>
            </OMA>
          </OMA>
        </OMA>
      </OMOBJ>
      <OMOBJ xmlns="http://www.openmath.org/OpenMath">
        <OMA>
          <OMS cd="vafp" name="recelim-false"/>
          <OMA>
            <OMS cd="ecc" name="pair"/>
            <OMR href="#vf5896a324-4354-4805-acab-c97c7d5fc03d"/>
            <OMA>
              <OMS cd="vafp" name="position"/>
              <OMA>
                <OMS cd="list1" name="list"/><OMI>3</OMI><OMI>3</OMI>
              </OMA>
            </OMA>
          </OMA>
        </OMA>
      </OMOBJ>
    </method>
  </derive>
</proof>
```

Abbildung 5.26: Enkodierung von Abbildung 5.25 in OMDoc

5.2 Dekodierung von OMDoc nach ✓eriFun

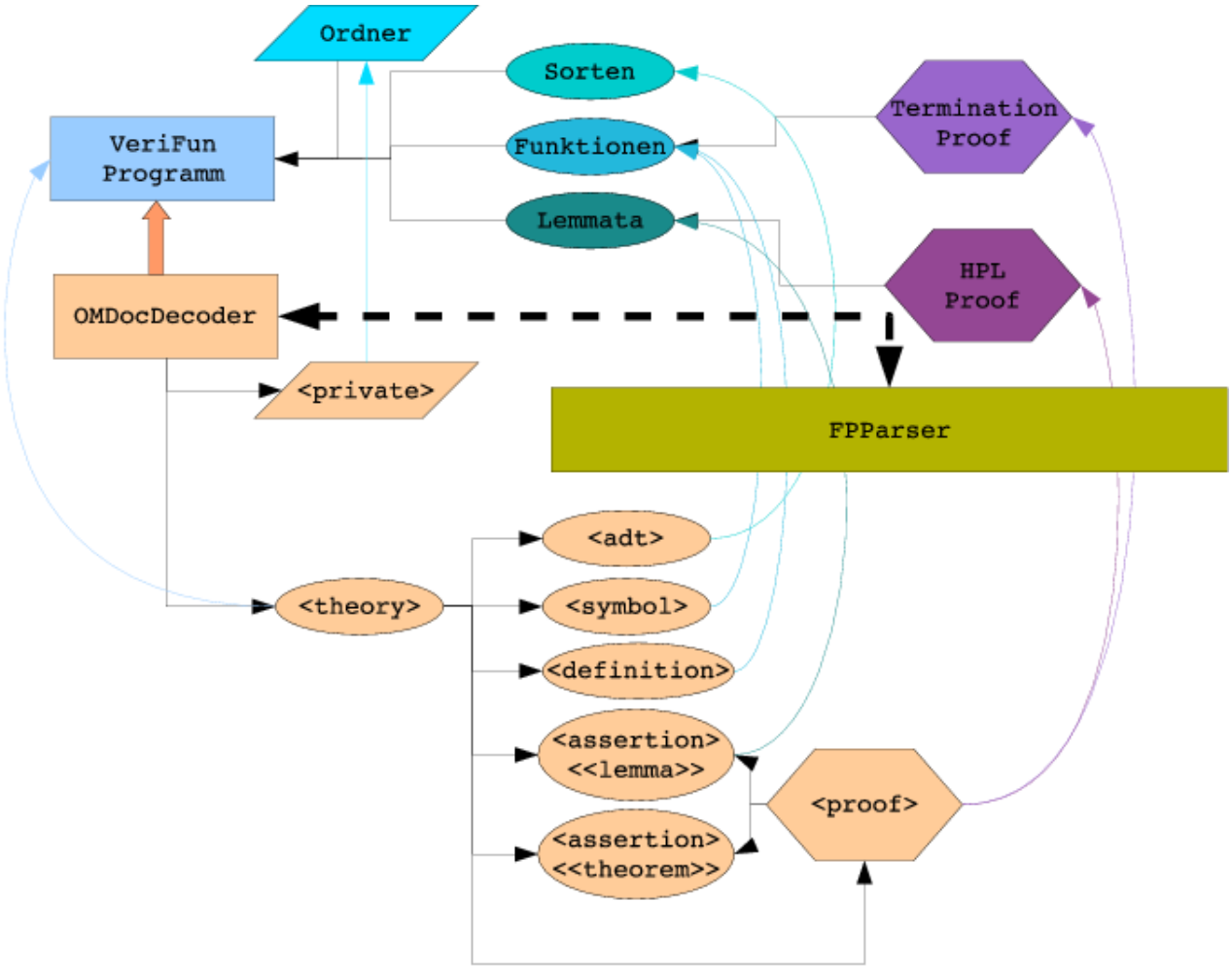


Abbildung 5.27: Der OMDoc2✓eriFun-Dekoder

In Abbildung 5.27 wird ein grober Überblick der Übersetzung der einzelnen OMDoc-Elemente einer gültigen `omdoc-vf`-Dokumentinstanz nach ✓eriFun dargestellt. Im Wesentlichen gleicht diese Darstellung der aus Abbildung 5.1 mit dem Unterschied, dass in diesem Fall offensichtlich die Richtung der einzelnen Pfeile umgekehrt und eine weitere Komponente, der ✓eriFun-`FPParser`, mit aufgenommen wurde. Wie auch schon in Abbildung 5.1 sind auch hier im oberen Teil der Abbildung die einzelnen Komponenten dargestellt, aus denen ein ✓eriFun-Programm bestehen kann. Die Überführung einer `omdoc-vf`-Dokumentinstanz, mit all den darin enthaltenen enkodierten ✓eriFun-Komponenten, in ein ✓eriFun-Programm wird hier durch den in dieser Arbeit erstellten `OMDocDecoder` in Verbindung mit dem in ✓eriFun eingebauten `FPParser` durchgeführt. Welche OMDoc-Markupelemente durch diesen Dekoder in der Überführung von einer

omdoc-vf-Dokumentinstanz in ein ✓eriFun-Programm bzw. in die einzelnen Komponten eines ✓eriFun-Programms berücksichtigt werden, wird in dem unteren Teil der Abbildung dargestellt. Diese OMDoc-Markupelemente werden durch den OMDocDecoder eingelesen, von der OMDoc- bzw. OpenMath-Syntax in die funktionale Sprache $\mathcal{FP}$ übersetzt und zur anschließenden Validierung der Syntax dem eingebauten ✓eriFun -FPParser übergeben. Nach erfolgreicher Dekodierung eines Programmelements wird dieses dem aktuellen Programm hinzugefügt. Direkt im Anschluss einer erfolgreichen Rekonstruktion eines ✓eriFun -Programmelements wird durch den OMDocDecoder im Fall von Lemmata und Funktionen überprüft, ob ein Beweis für dieses Element existiert. Wurde ein solcher Beweis gefunden, so wird dieser unmittelbar nachgespielt. Dies ist auch zwingend erforderlich, da eventuell später einzufügende Programmelemente inklusive deren Beweise, auf diesem Beweis basieren und davon ausgehen, dass dieser bereits zu true evaluierte wurde. Es wurde hier die Formulierung "nach spielen" verwendet, da man sich den OMDocDecoder in der Rekonstruktion sämtlicher Beweise als einen "automatischer Benutzer" vorstellen kann. Man könnte auch sagen, der OMDocDecoder führt jeden durch den Benutzer durchgeführten Mouseklick erneut auf die entsprechenden Beweisknoten aus. In dem gesamten Dekodierungs-Mechanismus ist jedoch zu beachten, dass sich, zum aktuellen Stand die OMDoc-Elemente, um eine erfolgreiche Dekodierung zu erlangen, in topologischer Reihenfolge befinden müssen. Wird zur Erstellung der OMDoc-Dateien der OMDocEncoder verwendet, ist diese Vorbedingung sichergestellt.

Die folgenden Unterabschnitte werden die Übersetzungen von OMDoc nach ✓eriFun näher erläutern und desweiteren auf die Dekodierung von ✓eriFun-Beweisen eingegangen. Es sei an dieser Stelle angemerkt, dass in diesen Erläuterungen an manchen Stellen Wiederholungen des jeweils vorherigen Abschnittes auftauchen werden. Der Autor hat sich für diese Wiederholungen entschieden, um eine angenehmere Lesbarkeit der einzelnen Dekodierungen zu gewährleisten.

5.2.1 Programm

Wie bereits in Unterabschnitt 5.1.1 angesprochen, wird ein ✓eriFun-Programm durch den OMDocEncoder in genau zwei OMDoc-Dateien gespeichert. Die erste Datei, vafp, beinhaltet die in ✓eriFun vordefinierten Programmelemente und in der zweiten Datei, deren Name durch den Benutzer frei gewählt werden kann, ist das gespeicherte ✓eriFun -Programm mit allen darin benutzerdefinierten und durch das System synthetisierten Programmelementen enthalten. In dieser Version des OMDocDecoders wird lediglich die zweite Datei berücksichtigt, da die vordefinierten ✓eriFun-Programmelemente durch das System automatisch einem neuen Programm hinzugefügt werden. Sämtliche Importanweisung der Form <import from="..."> werden somit durch den OMDocDecoder vollkommen ignoriert. Die "Dekodierung" der vordefinierten ✓eriFun-Programmelemente, ist somit fest in dem Dekoder verankert. Dekodierung ist hier in Anführungszeichen gesetzt, da diese vordefinierten Programmelemente im Gegensatz zu allen anderen Programmelementen nicht in dem Sinne dekodiert werden, dass sie aus der vafp Datei

eingelesen und rekonstruiert werden. Der `OMDocDecoder` kennt diese vordefinierten Programmelemente und weiß, dass zum Beispiel das Sortensymbol `nat` in die ✓`eriFun`-Sorte $\mathbb{N}$ übersetzt werden muss. Das hat allerdings zur Folge, dass Änderungen an den vordefinierten Programmelementen, zum Beispiel hinsichtlich der Notation oder Präsentation, auch Änderung in dem `OMDocDecoder`s nach sich ziehen. Dieser Sachverhalt sollte in der nächsten Version des `OMDocDecoder`s verbessert werden, um, unter anderem, auch eine weitere OMDoc Funktionalität, nämlich Theorien zu importieren, gewährleisten zu können. Um ein, in einer `omdoc-vf`-Dokumentinstanz, dekodiertes ✓`eriFun`-Programm zu rekonstruieren, wird diese Datei zuerst von einem validierenden XML-Parser eingelesen und auf dessen Gültigkeit überprüft. Nach einer erfolgreichen Validierung wird durch den `OMDocDecoder` ein leeres ✓`eriFun` Programm $\mathfrak{P}$ erzeugt, in welches zuerst die dekodierten Programmordner (Unterabschnitt 5.2.2) und dann die übrigen Programmelemente - Sorten (Unterabschnitt 5.2.3), Funktionen (Unterabschnitt 5.2.4), Lemmata (Unterabschnitt 5.2.5) und Beweise (Unterabschnitt 5.2.6, Unterabschnitt 5.2.7 und Unterabschnitt 5.2.8) - eingefügt werden. Das durch den `OMDocDecoder` neu erstellte Programm $\mathfrak{P}$ wird der aktuellen ✓`eriFun` Sitzung übergeben und stellt von da an das für diese Sitzung aktuelle Programm dar. Zu beachten ist, dass durch den `OMDocDecoder` in jedem Fall ein neues Programm erstellt wird und der aktuellen ✓`eriFun`-Sitzung übergeben wird. Sollte der XML-Parser während der Validierung einen Fehler melden, so ist $\mathfrak{P}$ leer. Treten in der Traversion der enkodierten Elemente Fehler auf, zum Beispiel durch syntaktische $\mathcal{FP}$ Fehler, so beinhaltet $\mathfrak{P}$ die bis zu diesem Zeitpunkt dekodierten Elemente. In dem Fall einer erfolgreichen und vollständigen Rekonstruktion sind alle ✓`eriFun`-Programmelemente in der durch den `OMDocEncoder` erstellten topologischen Ordnung in $\mathfrak{P}$ enthalten.

5.2.2 Ordner

Die Ordnerstruktur eines ✓`eriFun`-Programms wird in `private`-Elementen enkodiert. Im ersten `data`-Kind-Element wird der Name des Ordners, im zweiten der Kommentar dieses Ordners und im letzten `data`-Kind-Element wird der Name des Vater-Ordners gespeichert. Zur Dekodierung einer ✓`eriFun`-Ordnerstruktur liest der `OMDocDecoder` sequentiell alle direkten `private`-Kind-Elemente des `omdoc`-Elements ein. Die `data`-Elemente eines `private`-Elements werden nur dann eingelesen, wenn das `pto`-Attribut den Wert `VeriFun` enthält, in allen anderen Fällen werden diese nicht berücksichtigt. Die Reihenfolge der `data`-Elemente ist fest durch den `OMDocEncoder` vorgeschrieben und wird strikt von dem `OMDocDecoder` eingehalten. In Folge dessen wird zuerst der Name des Ordners, dann der Kommentar und zuletzt der Name des Vater-Ordners eingelesen. Diese Informationen werden von dem `OMDocDecoder` - ohne Verwendung des `FPParsers` - unmittelbar zur Rekonstruktion des entsprechenden ✓`eriFun`-Ordners verwendet und dem neu erstellten Programm $\mathfrak{P}$ hinzugefügt. Eine Sonderregelung erfährt der Hauptprogrammorder eines ✓`eriFun`-Programms. Dieser Ordner wird durch das System automatisch einem neu erstellten Programm hinzugefügt und darf somit nicht durch den

OMDocDecoder rekonstruiert werden. Für diesen Ordner kann der OMDocDecoder lediglich den Namen und den Kommentar setzen. Identifiziert wird ein Hauptprogrammordner durch einen leeren Vater-Knoten. Sollten in der Dekodierung der Ordnerstruktur Fehler auftreten, so wird das bis zu diesem Zeitpunkt rekonstruierte Programm $\mathfrak{P}$ mit den bereits dekodierten Ordnern an das System übergeben und die Dekodierung wird abgebrochen. Im Fall einer vollständigen und erfolgreichen Dekodierung aller relevanten private-Elemente, ist die komplette ✓eriFun-Ordnerstruktur in $\mathfrak{P}$ enthalten. In dem nächsten Schritt der Dekodierung werden die Programmelemente - Sorten (Unterabschnitt 5.2.3), Funktionen (Unterabschnitt 5.2.4), Lemmata (Unterabschnitt 5.2.5) und Beweise (Unterabschnitt 5.2.6, Unterabschnitt 5.2.7 und Unterabschnitt 5.2.8) - dekodiert und in die Ordnerstruktur von $\mathfrak{P}$ eingefügt. Sind keine weiteren Programmelemente in der einzulesenden OMDoc-Datei enthalten, wird $\mathfrak{P}$ an das System übergeben und die Dekodierung ist abgeschlossen.

5.2.3 Sorten

In der Dekodierung von ✓eriFun-Sorten erzeugt der OMDocDecoder im ersten Schritt aus den OMDoc-Elementen, zur Definition einer ADT, eine Zeichenkette $\mathfrak{Z}$, die die Sorte in der $\mathcal{FP}$-Syntax repräsentiert. Diese Zeichenkette $\mathfrak{Z}$ wird im zweiten Schritt dem FPParser übergeben. Nachdem unter anderem die Korrektheit der $\mathcal{FP}$-Syntax durch den FPParser überprüft worden ist, erzeugt dieser die entsprechende ✓eriFun-Sorte $\mathfrak{S}$ und fügt diese in das neu erstellte Programm $\mathfrak{P}$ ein. In dem dritten und letzten Schritt der Dekodierung einer ✓eriFun-Sorte, wird durch den OMDocDecoder der Kommentar eingelesen - sofern ein solcher für diese Sorte enkodiert wurde - und dem Programmelement $\mathfrak{S}$ zugewiesen. Fehler in einem dieser drei Schritte führen zum sofortigen Abbruch der Dekodierung, das bis zu diesem Zeitpunkt neu erstellte Programm $\mathfrak{P}$ wird an das System übergeben und die Dekodierung ist abgeschlossen. Im Folgenden werden die drei Schritte der Dekodierung einer ✓eriFun-Sorte im Einzelnen erläutert.

Rekonstruktion der $\mathcal{FP}$ Syntax
Eine ✓eriFun-Sorte wird durch den OMDocEncoder innerhalb eines adt-Elements enkodiert (Unterabschnitt 5.1.3). Enthält das aktuell zu dekodierende adt-Element ein parameters-Attribut, wird der Wert dieses Attributs durch den OMDocDecoder eingelesen. Wie bereits in der Enkodierung von ✓eriFun-Sorten erläutert, handelt es sich dabei um eine durch Leerzeichen getrennte Liste der Typvariablen. Als nächster Schritt wird jede einzelne dieser Typvariablen mit dem Präfix '@' versehen, die Leerzeichen zur Trennung der einzelnen Typvariablen werden durch Kommata ersetzt und die modifizierte Liste wird entsprechend der $\mathcal{FP}$-Syntax der Zeichenkette $\mathfrak{Z}$ hinzugefügt. Den Namen einer ✓eriFun-Sorte ermittelt der OMDocDecoder über das xml:id-Attribut des adt-Kind-Elements sortdef. Anhand dieser ID werden die zusätzlichen Informationen wie Präsentation, Ordnerzugehörigkeit und der Kommentar identifiziert. So wird zur Dekodierung des Namens über diese ID das presentation-Element ermittelt, um aus dessen

use-Kind-Element die Zeichenkette auszulesen, die den Namen der ✓eriFun Sorte repräsentiert. Die Konstruktoren mit den entsprechenden Selektoren der ✓eriFun-Sorte werden als Nächstes der Zeichenkette $\mathfrak{Z}$ hinzugefügt. Hierbei dekodiert der OMDocDecoder sequentiell alle constructor-Kind-Elemente des aktuellen sortdef-Elements. Der Wert des name-Attributs des constructor-Elements ist, analog zu dem xml:id-Attribut des adt-Elements, eine ID über welche die zusätzlichen Informationen, wie zum Beispiel Bindungspriorität und *Fixity* des Konstruktors bestimmt werden. Diese Informationen werden aus dem, über die ID gekennzeichnetem, presentation Element ausgelesen. An dieser Stelle sei daran erinnert, das durch OMDoc für das fixity Attribut lediglich die Werte infix, infixl, infixr, prefix und postfix zur Verfügung stehen. Funktionen mit der Fixity outfix werden gesondert behandelt. Für outfix Funktionen wird das linke bzw. rechte Funktionssymbol in den Attributen lbrack bzw. rbrack dekodiert und das entsprechende use Element ist leer. Enthält der aktuell zu dekodierende Konstruktor noch zusätzlich Selektoren, so sind diese als argument Kind-Elemente dem constructor Element angehängt. In dem ersten Kind-Elemente (type) des argument Elements ist der Typ des Selektors enkodiert und in dem zweiten Kind-Element (selector) der Selektor selbst. Hinsichtlich der Dekodierung der zusätzlichen Informationen eines Selektors, werden selector Elemente in gleicher Weise wie constructor Elemente behandelt. Der Typ des Selektors ist innerhalb der type Elements in OpenMath Syntax dargestellt, die durch einen, in den OMDocDecoder, integrierten OpenMathDecoder in die $\mathcal{FP}$ Syntax überführt wird[5]. Nach der Bearbeitung aller Konstruktoren, einschliesslich der zusätzlichen Selektoren, repräsentiert die Zeichenkette $\mathfrak{Z}$ die ✓eriFun Sorte $\mathfrak{S}$ in der $\mathcal{FP}$ Syntax und wird dann, in dem zweiten Schritt der Dekodierung einer ✓eriFun Sorte, dem FPParser übergeben.

Erstellung der ✓eriFun Sorte durch den FPParser
In diesem Schritt verwendet der OMDocDecoder den FPParser als sogenannte "Black-Box". Durch den OMDocDecoder wird die die ✓eriFun Sorte $\mathfrak{S}$ repräsentierende $\mathcal{FP}$ Zeichenkette $\mathfrak{Z}$ und der Name des entsprechenden Programmordners dem FPParser übergeben. Der OMDocDecoder hat an dieser Stelle keine weiteren Eingriffsmöglichkeiten in die Erzeugung von $\mathfrak{S}$. Gelingt die Erstellung, so ist die ✓eriFun Sorte durch den FPParser in den entsprechenden Programmordner des neu erstellten Programms $\mathfrak{P}$ eingefügt worden und der OMDocDecoder fährt mit der Dekodierung des Kommentars fort. Andernfalls wird die Dekodierung abgebrochen und das bis zu diesem Zeitpunkt neu erstellte Programm $\mathfrak{P}$ wird an das System übergeben. Den Name des Programmordners enkodiert der OMDocEncoder über das xml:id Attribut des sortdef Elements. Mit dieser ID wird das zugehörige private Element identifiziert, in dessen zweiten data Element der Name des Programmordners gespeichert ist.

Zuweisung des Kommentars
Der Kommentar einer ✓eriFun-Sorte $\mathfrak{S}$ wird analog zu der Dekodierung des Program-

[5] In der aktuellen Version der OMDocDecoders ist die Funktionalität OpenMath-Syntax zu dekodieren, nicht in einer eigenen Klasse gekapselt. Dem OMDocDecoder wurden lediglich entsprechende Methoden zur Dekodierung von OpenMath Syntax hinzugefügt. Diese Implementierungsentscheidung sollte in der nächsten Version des OMDocDecoders in getrennte Klassen aufgeteilt werden.

mordners dekodiert, mit dem Unterschied, dass dieser in dem ersten `data`-Element des zugehörigen `private`-Elements gespeichert ist. Der eingelesene Kommentar, sofern ein solcher für $\mathfrak{S}$ existiert, wird durch den `OMDocDecoder` direkt der ✓eriFun -Sorte $\mathfrak{S}$ innerhalb des Programms $\mathfrak{P}$ zugewiesen. Nach diesem Schritt ist die Dekodierung einer ✓eriFun -Sorte abgeschlossen.

5.2.4 Funktionen

Die Enkodierung von benutzerdefinierten ✓eriFun-Funktionen ist in zwei Teile untergliedert. In dem ersten Teil wird die Funktions*deklaration* in einem `symbol`-Element und in dem zweiten Teil die Funktions*definition* in einem `definition`-Element enkodiert (Unterabschnitt 5.1.4). In der Dekodierung von benutzerdefinierten ✓eriFun -Funktionen werden durch den `OMDocDecoder` in erster Linie die `definition`-Elemente betrachtet. Durch die eindeutige Referenzierung über das `for`-Attribut eines `definition`-Elements wird das entsprechende `symbol`-Element ermittelt. Ähnlich wie die Dekodierung von ✓eriFun -Sorten, so unterteilt sich auch die Dekodierung von Funktionen in mehrere Schritte. Im ersten Schritt wird für das zu dekodierende `definition`-Element das zugehörige `symbol`-Element ermittelt. Über dieses `symbol`-Element bestimmt der `OMDoc-Decoder` die Signatur - Name und Typ der einzelnen formalen Funktionsparameter und den Funktionstyp - der Funktion $\mathfrak{F}$ und stellt diese in der $\mathcal{FP}$-Syntax dar. In dem nächsten Schritt wird der in dem `definition`-Element in OpenMath-Syntax repräsentierte Funktionsrumpf in die $\mathcal{FP}$-Syntax überführt. Diese Dekodierung der Funktionssignatur und des Funktionsrumpfes in die $\mathcal{FP}$-Syntax, wird als Zeichenkette $\mathfrak{Z}$ dem `FPParser` übergeben. Der `FPParser` überprüft unter anderem die Korrektheit der $\mathcal{FP}$-Syntax, erzeugt nach einer gelungen Überprüfung, die entsprechende ✓eriFun Funktion $\mathfrak{F}$ und fügt diese in das neu erstellte Programm $\mathfrak{P}$ ein. Im Anschluß wird durch den `OMDocDecoder` der Kommentar eingelesen - sofern ein solcher für diese Funktion enkodiert wurde - und dem Programmelement $\mathfrak{F}$ zugewiesen. Fehler in einem dieser Schritte führen auch hier wieder zum sofortigen Abbruch der Dekodierung, das bis zu diesem Zeitpunkt neu erstellte Programm $\mathfrak{P}$ wird an das System übergeben und die Dekodierung ist abgeschlossen. Im Folgenden werden die einzelnen Schritte der Dekodierung einer benutzerdefinierten ✓eriFun -Funktion näher erläutert. In diesem Abschnitt wird jedoch nicht die Dekodierung von Terminierungsbeweisen behandelt, noch wird auf die gesonderte Dekodierung synthetisierter ✓eriFun -Funktionen eingegangen. Die Dekodierung von Terminierungsbeweisen werden in Unterabschnitt 5.2.7 erläutert. Auf synthetisierte Funktionen wird im Rahmen der Rekursionseliminationsbeweise eingegangen (Unterabschnitt 5.2.8).

Rekonstruktion der $\mathcal{FP}$ Syntax
In dem ersten Schritt der Rekonstruktion der $\mathcal{FP}$-Syntax wird über das `for`-Attribut des aktuellen `definition`-Elements das entsprechende `symbol`-Element ermittelt. Die Signatur einer ✓eriFun-Funktion ist innerhalb eines `type`-Kind-Elements dieses `symbol`-Elements in OpenMath-Syntax enkodiert. Der schon angesprochene `OpenMathDecoder`

innerhalb des OMDocDecoders übersetzt diese OpenMath-Syntax in die $\mathcal{FP}$-Syntax. Der Name des Funktionssymbols sowie *Fixity* und Bindungspriorität werden in gewohnter Weise aus dem presentation-Element dekodiert, welches über die in dem name-Attribut des symbol-Elements gespeicherten ID identifiziert wird. Das Resultat dieses Dekodierungsschrittes ist eine Zeichenkette $\mathfrak{Z}$, die die Funktionsdeklaration der ✓eriFun -Funktion $\mathfrak{F}$ in der $\mathcal{FP}$-Syntax repräsentiert. Im zweiten Schritt der Rekonstruktion der $\mathcal{FP}$-Syntax einer ✓eriFun-Funktion wird der Funktionsrumpf eingelesen und übersetzt. Auch diese Übersetzung übernimmt der integrierte OpenMathDecoder, da ein ✓eriFun -Funktionsrumpf gemäß dem OMDoc-simple-Definitionsmechanismus in OpenMath-Syntax repräsentiert wird. Dieser dekodierte Funktionsrumpf wird an die Zeichenkette $\mathfrak{Z}$ angehängt, dem FPParser übergeben und damit ist die Rekonstruktion der $\mathcal{FP}$-Syntax einer ✓eriFun -Funktion abgeschlossen.

Erstellung der ✓eriFun Funktion durch den FPParser
Die Erstellung der ✓eriFun-Funktion $\mathfrak{F}$ verläuft analog zu der Erstellung einer ✓eriFun -Sorte. Auch in diesem Schritt verwendet der OMDocDecoder den FPParser als "Black-Box". Der OMDocDecoder übergibt die Funktion $\mathfrak{F}$ repräsentierende $\mathcal{FP}$ Zeichenkette $\mathfrak{Z}$ und den Namen des entsprechenden Programmordners an den FPParser. Der OMDocDecoder hat an dieser Stelle keine weiteren Eingriffsmöglichkeiten in die Erzeugung von $\mathfrak{F}$. Gelingt die Erstellung, so ist die ✓eriFun-Funktion durch den FPParser in den entsprechenden Programmordner des neu erstellten Programms $\mathfrak{P}$ eingefügt worden und der OMDocDecoder fährt mit der Dekodierung des Kommentars fort. Andernfalls wird die Dekodierung abgebrochen und das bis zu diesem Zeitpunkt neu erstellte Programm $\mathfrak{P}$ wird an das System übergeben. Den Name des Programmordners enkodiert der OMDocEncoder über das name-Attribut des symbol-Elements. Mit dieser ID wird das zugehörige private-Element identifiziert, in dessen zweiten data-Element der Name des Programmordners gespeichert ist.

Zuweisung des Kommentars
Der Kommentar einer ✓eriFun -Funktion $\mathfrak{F}$ wird analog zu der Dekodierung des Programmordners dekodiert, mit dem Unterschied, dass dieser in dem ersten data-Element des zugehörigen private-Elements gespeichert ist. Der eingelesene Kommentar, sofern ein solcher für $\mathfrak{F}$ existiert, wird durch den OMDocDecoder direkt der ✓eriFun -Funktion $\mathfrak{F}$ innhalb des Programms $\mathfrak{P}$ zugewiesen. Nach diesem Schritt ist die Dekodierung einer benutzerdefinierten ✓eriFun -Funktion - bis auf die Dekodierung der Terminierungsbeweise (Unterabschnitt 5.2.7) - abgeschlossen.

5.2.5 Lemmata

Benutzerdefinierte ✓eriFun -Lemmata werden durch den OMDocEncoder in assertion-Elemente enkodiert. OMDoc unterscheidet jedoch zwischen verschiedene Typen von assertion-Elementen. Somit wird zur expliziten Kennzeichnung als Lemma in dem

type-Attribut des zur Enkodierung eines benutzerdefinierten ✓eriFun -Lemmas vorgesehenen assertion-Elements, der Wert lemma gespeichert (Unterabschnitt 5.1.5). In der Dekodierung von ✓eriFun -Lemmata überprüft der OMDocDecoder dieses Attribut, um enkodierte benutzerdefinierte ✓eriFun -Lemmata von enkodierten Theoremen und Terminierungshypothesen unterscheiden zu können. Die Dekodierung von benutzerdefinierten ✓eriFun -Lemmata unterteilt sich, wie auch schon die Dekodierung von ✓eriFun -Sorten, in drei Schritte. Im ersten Schritt wird durch den in dem OMDocDecoder integrierten OpenMathDecoder die $\mathcal{FP}$-Syntax für das benutzerdefinierte ✓eriFun -Lemma $\mathfrak{L}$ aus der OpenMath-Syntax rekonstruiert und im zweiten Schritt als Zeichenkette $\mathfrak{Z}$ an den FPParser übergeben. Der FPParser überprüft unter anderem die Korrektheit der $\mathcal{FP}$-Syntax, erzeugt, nach einer gelungen Überprüfung, das entsprechende ✓eriFun -Lemma $\mathfrak{L}$ und fügt diese in das neu erstellte Programm $\mathfrak{P}$ ein. Im Anschluß wird durch den OMDocDecoder der Kommentar eingelesen - sofern ein solcher für dieses Lemma enkodiert wurde - und dem Programmelement $\mathfrak{L}$ zugewiesen. Fehler in einem dieser drei Schritte führen auch hier wieder zum sofortigen Abbruch der Dekodierung, das bis zu diesem Zeitpunkt neu erstellte Programm $\mathfrak{P}$ wird an das System übergeben und die Dekodierung ist abgeschlossen. Im Folgenden werden die einzelnen Schritte der Dekodierung eines benutzerdefinierten ✓eriFun -Lemmas näher erläutert. In diesem Abschnitt wird jedoch nicht die Dekodierung von HPL-Beweisen benutzerdefinierter ✓eriFun -Lemmata behandelt. Die Dekodierung von HPL-Beweisen wird in Unterabschnitt 5.2.6 erläutert.

Rekonstruktion der $\mathcal{FP}$ Syntax
Im ersten Schritt der Rekonstruktion der $\mathcal{FP}$-Syntax wird über das xml:id-Attribut des aktuellen assertion-Elements das entsprechende private-Element ermittelt. Im Gegensatz zu ADTs und Symbolen sieht OMDoc keine presentation-Elemente für assertion-Elemente vor. Gemäß der OMDoc Spezifikation besitzten Behauptungen keine Präsentationsinformationen, insbesondere keine Namen. Aus diesem Grund werden durch den OMDocEncoder alle, ein benutzerdefiniertes ✓eriFun -Lemma betreffenden, zusätzlichen Informationen wie Name, Ordnerzugehörigkeit und der Kommentar in einem private-Element enkodiert. Die Reihenfolge dieser zusätzlichen Information ist auch hier wieder fest durch OMDocEncoder vorgeschrieben, so dass an dieser Stelle aus dem ersten data-Kind-Element, des über die ID des assertion-Elements identifizierten private-Elements der Name des benutzerdefinierten ✓eriFun-Lemmas ausgelesen wird. In dem nächsten Schritt werden die gebundenen Variablen und der Rumpf des benutzerdefinierten ✓eriFun -Lemmas aus der OpenMath-Syntax in die $\mathcal{FP}$-Syntax übersetzt und als Zeichenkette $\mathfrak{Z}$ an den FPParser übergeben. Die Rekonstruktion der $\mathcal{FP}$-Syntax des benutzerdefinierten ✓eriFun -Lemmas $\mathfrak{L}$ ist nach diesem Schritt abgeschlossen.

Erstellung des ✓eriFun Lemmas durch den FPParser
Die Erstellung des ✓eriFun -Lemmas $\mathfrak{L}$ verläuft analog zu der Erstellung einer ✓eriFun -Sorte oder ✓eriFun -Funktion. Der OMDocDecoder verwendet den FPParser als "Black-Box". Die das ✓eriFun -Lemma $\mathfrak{L}$ repräsentierende $\mathcal{FP}$ Zeichenkette $\mathfrak{Z}$ und der Name des entsprechenden Programmordners wird durch den OMDocDecoder dem FPParser übergeben. Der OMDocDecoder hat an dieser Stelle keine weiteren Eingriffsmöglichkeiten

in die Erzeugung von $\mathfrak{L}$. Gelingt die Erstellung, ist das ✓eriFun -Lemma durch den FPParser in den entsprechenden Programmordner des neu erstellten Programms $\mathfrak{P}$ eingefügt worden und der OMDocDecoder fährt mit der Dekodierung des Kommentars fort. Andernfalls wird die Dekodierung abgebrochen und das bis zu diesem Zeitpunkt neu erstellte Programm $\mathfrak{P}$ wird an das System übergeben. Den Name des Programmordners enkodiert, wie weiter oben bereits erwähnt, der OMDocEncoder über das xml:id-Attribut des assertion-Elements. Mit dieser ID wird das zugehörige private-Element identifiziert, in dessen dritten data-Element der Name des Programmordners gespeichert ist.

Zuweisung des Kommentars
Der Kommentar eines ✓eriFun -Lemmas $\mathfrak{L}$ wird analog zu der Dekodierung des Programmordners dekodiert, mit dem Unterschied, dass dieser in dem zweiten data-Element des zugehörigen private-Elements gespeichert ist. Der eingelesene Kommentar, sofern ein solcher für $\mathfrak{L}$ existiert, wird durch den OMDocDecoder direkt dem ✓eriFun -Lemma $\mathfrak{L}$ innhalb des Programms $\mathfrak{P}$ zugewiesen. Nach diesem Schritt ist die Dekodierung eines benutzerdefinierten ✓eriFun -Lemmas — bis auf die Dekodierung der HPL Beweise (Unterabschnitt 5.2.6) — abgeschlossen.

5.2.6 HPL-Beweise

In ✓eriFun werden sowohl Beweise von benutzerdefinierten und synthetisierten Lemmata als auch Beweise von Terminierungshypothesen und Rekursionseliminationen im HPL-Kalkül durchgeführt. Der OMDocEncoder enkodiert jeden ✓eriFun -Beweis innerhalb eines proof-Elements, welches über die in dem for-Attribut gespeicherte ID mit dem jeweiligen ✓eriFun -Programmelement verknüpft ist (Unterabschnitt 5.1.6). Zur Veranschaulichung der Dekodierung eines HPL-Beweises wird im Folgenden der Fall für ein benutzerdefiniertes ✓eriFun-Lemma $\mathfrak{L}$ betrachtet. Die Dekodierung von HPL Beweisen von synthetisierten Lemmata oder Terminierungshypothesen verlaufen analog. Bevor jedoch der OMDocDecoder mit der Dekodierung eines HPL-Beweises beginnt, werden die ✓eriFun -Systemeinstellungen überprüft, mit denen der Benutzer das Verhalten der Dekodierung von HPL-Beweisen beeinflussen kann. Es stehen hierzu drei Einstellungen zur Verfügung:

always
: Der OMDocDecoder vollzieht genau die in OMDoc gespeicherten Beweisschritte nach - es wird also nicht versucht, zunächst durch das System automatisch neue Beweisschritte berechnen zu lassen, wenn dies in dem in OMDoc gespeicherten Beweis nicht vorgesehen ist.

if needed
: Das System versucht zunächst automatisch einen neuen Beweis zu berechnen. Ist dieser Versuch nicht erfolgreich, so werden die in dem OMDoc-Beweis gespeicherten Beweisschritte und Benutzerinteraktionen verwendet.

on demand

Vor der Dekodierung eines HPL-Beweises wird folgende Abfrage an den Benutzer gerichtet: `Regard user interaction for lemma <Name>`. Diese Abfrage erscheint jedoch nur, wenn Benutzerinteraktionen in dem HPL-Beweis existieren. Beispiel: Nach der Eingabe von dem Lemma +_associative wurde der Beweis automatisch durch das System berechnet - eine Abfrage vor der Dekodierung unterbleibt folglich. Nach der Eingabe des Lemmas +_commutative jedoch musste der Benutzer in den Beweis eingreifen, um das System in der weiteren Durchführung zu unterstützten. In solch einem Fall wird vor der Dekodierung des HPL-Beweises die Abfrage `Regard user interaction for lemma` +_commutative an den Benutzer gerichtet. In dieser Abfrage stehen dem Benutzer die beiden weiteren hier aufgeführeten Alternativen (`always` und `if needed`) zur Auswahl, worauf der `OMDocDecoder` in der Dekodierung des HPL-Beweises entsprechend reagiert.

Da die Systemeinstellung `on demand` gegebenfalls lediglich eine weitere Benutzerabfrage mit den Alternativen `always` und `if needed` nach sich zieht, werden in der weiteren Betrachtung der Dekodierung von HPL-Beweisen nur diese beiden Fälle unterschieden. Gemein ist beiden Alternativen, dass zu Beginn nach einer erfolgreichen Dekodierung des benutzerdefinierten ✓eriFun-Lemmas $\mathfrak{L}$, der `OMDocDecoder` mittels der in dem `xml:id`-Attribut des `assertion`-Elements gespeicherten `ID`, das entsprechende `proof` Element identifiziert. Wurde für das ✓eriFun-Lemma $\mathfrak{L}$ kein Beweis enkodiert, so ist die Dekodierung von $\mathfrak{L}$ an dieser Stelle vollkommen abgeschlossen. Existiert jedoch ein HPL-Beweis, so sind innerhalb des entsprechenden `proof`-Elements, durch den `OMDocEncoder` die einzelnen Beweisschritte (`derive`-Elemente), die Methoden, mit denen der jeweilige Beweisschritt durchgeführt wurde (`method`-Elemente) und die Daten (`Oracle`), die zur Anwendung der jeweiligen Methode benötigt werden (geordnete Liste von `OMOBJ`-Elementen), enkodiert. Wurde durch den Benutzer die Auswahl `always` getroffen, so wird für den initialen Beweisschritt aus dem `xref`-Attribut, des entsprechenden `method`-Elements, die anzuwendende HPL Regel und die Daten für diese HPL-Regel aus den an das `method`-Element angehängten `OMOBJ` Elementen dekodiert. Zur Rekonstruktion des `Oracle`s aus dieses Daten verwendet der `OMDocDecoder` den `FPParser`. Tritt in dieser Rekonstruktion des `Oracle`s eine Fehler auf, so wird der gesamte Dekodierungsvorgang abgebrochen! Hier sei noch mal darauf hingewiesen, dass in dieser Version des `OMDocDecoder`s in der Dekodierung der jeweilige Daten einer HPL-Regel bzw. HPL-Taktik die durch den `OMDocEncoder` strikt vorgeschriebene Ordnung eingehalten werden. Auch hier muss folglich der `OMDocDecoder` diese Ordnung kennen, was in der nächsten Version durch weitere Semantic Xml Tags verbessert werden sollte, um diese engen Koppelung zwischen dem `OMDocEncoder` und dem `OMDocDecoder` aufzulösen. Ist die Dekodierung des `Oracle`s erfolgreich verlaufen, so wird die enkodierte HPL-Regel zusammen mit dem `Oracle` auf den initialen Beweisknoten in dem System angewandt. Lautet die vorgenommene Systemeinstellung `if needed`, so wird analog verfahren bis auf die Tatsache, dass aus dem entsprechenden `method`-Element nicht die nächste HPL-Regel enkodiert wird, sondern die anzuwendende HPL-Taktik inklusive deren Daten. Der Unterschied besteht also darin, dass in dem ersten Fall nur eine HPL-Regel auf den Beweisknoten angewandt

wird, wobei es sich in dem zweiten Fall um die Anwendung einer ganzen HPL-Taktik handelt, die aus mehreren HPL-Regeln bestehen kann. In der weiteren Dekodierung eines HPL-Beweises traversiert der `OMDocDecoder` alle weiteren `derive`- bzw. `method`-Kind-Element des initialen Beweisschrittes und verfährt hinsichtlich der Anwendung einer HPL-Regel bzw. HPL-Taktik analog zu der des initialen Beweisschritts, jedoch mit zwei Erweiterungen: Vor der weiteren Dekodierung weiterer Beweisschritte wird durch den `OMDocDecoder` überprüft, ob die Anwendung der zuletzt angewandten HPL-Regel bzw. HPL-Taktik bereits in dem System zu `true`, also zur Schließung dieses Beweisschrittes, geführt hat. Sollte dies der Fall sein und sich dennoch in dem OMDoc-Beweis weitere Beweisschritte befinden, so bedeutet dies: Der Beweis kann in dem aktuellen, zur Dekodierung verwendeten System kürzer erstellt werden, als mit dem System, mit dem der OMDoc-Beweis enkodiert wurde. Der `OMDocDecoder` wird dann dem Benutzer eine positive Rückmeldung über diese Tatsache liefern und mit dem nächsten Beweisschritt fortfahren. Die zweite Erweiterung — wiederum unabhängig von der gewählten Systemeinstellung — ist, dass vor der Anwendung einer HPL-Regel bzw. HPL-Taktik auf den nächste Beweisknoten, die systemgenerierte Sequenz des jeweiligen Knotens mit der aus dem OMDoc-Beweis verglichen wird. Denn nur wenn diese beiden vollkommen übereinstimmen, d.h. die Hypothesen, die Induktionshypothese und der Goalterm übereinstimmen, kann der `OMDocDecoder` mit der Dekodierung des HPL-Beweises fortfahren. Sollte ein Unterschied bestehen, so wird eine Benutzerinteraktion erzwungen und der `OMDocDecoder` wartet so lange, bis diese abgeschlossen ist, um dann mit der Rekonstruktion des nächsten Beweisschrittes fortzufahren. Sind nach dieser Vorgehensweise alle `derive`- bzw. `method`-Kind-Elemente abgearbeitet, ist die Dekodierung des ✓`eriFun` -Lemmas $\mathfrak{L}$ vollständig abgeschlossen.

5.2.7 Terminierungsbeweise

In der Dekodierung von ✓`eriFun` -Beweisen wird aus der Sicht des `OMDocDecoder`s zwischen zwei Klassen unterschieden: Die Dekodierung von HPL-Beweisen und die Dekodierung von Terminierungs- bzw. Rekursionseliminationsbeweisen (Unterabschnitt 5.2.8). Der Unterschied besteht in den zusätzlichen enkodierten Informationen für die zweite Klasse von ✓`eriFun` -Beweisen. In Fall von Terminierungsbeweisen handelt es sich um Maßterme bzw. eine Liste von Maßtermen, daraus abgeleitete Terminierungshypothesen und die Positionen, aus denen die jeweiligen Hypothese in dem betreffenden Funktionrumpf gewonnen wurden. Da es sich bei diesen Zusätzen um Daten zur Rekonstruktion von Beweisen handelt, werden diese durch den `OMDocEncoder` in separaten `proof`-Elementen enkodiert. Wie bereits in der Enkodierung von ✓`eriFun` -Funktionen (Unterabschnitt 5.1.4) erwähnt, ist es jedoch in der OMDoc-Syntax nicht vorgesehen, dass `proof` Elemente mittels des `for` Attributs direkt auf andere OMDoc Elemente, außer wenn es sich dabei um `assertion`-Elemente handelt, verweisen dürfen. Aus diesem Grund wird durch den `OMDocEncoder` eine informale Terminierungshypothese innerhalb eines `assertion`-Elements enkodiert. Damit der `OMDocDecoder` solche `assertion`-Elemente von denjenigen unterscheiden kann, die eine Enkodierung eines

✓eriFun -Lemmas enthalten, wird in der Enkodierung diesen assertion-Elementen in dem type-Attribut der Wert theorem zugewiesen. Die ID eines solchen Theorems wird in dem existence-Attribut der entsprechenden Funktionsdefinition gespeichert. Auf diese Weise ist es dem OMDocDecoder möglich, einen Terminierungsbeweis für benutzerdefinierte ✓eriFun -Funktion zu identifizieren: Mittels der in dem existence-Attribut gespeicherten ID wird zunächst das assertion-Element mit der informalen Terminierungshypothese ermittelt. Der Inhalt dieses Theorems wird durch den OMDocDecoder nicht weiter betrachtet. Diese informalen Terminierungshypothesen dienen lediglich als Schnittstelle zwischen definition- und proof-Elementen. Somit wird dann für solch ein Theorem, analog zu der Dekodierung von einem HPL Beweis eines ✓eriFun-Lemmas, die proof-Elemente identifiziert, welche in dem for-Attribut die ID dieses Theorems gespeichert haben (Abbildung 5.24). Es ist zu beachten, dass im Gegensatz zur Enkodierung von HPL-Beweisen für ✓eriFun -Lemmata, in welcher einem assertion-Element durch den OMDocEncoder höchstens ein proof-Element zugewiesen wird, in der Enkodierung von Terminierungsbeweisen einem assertion-Element mehrere proof-Elemente zugewiesen werden können. Die Begründung hierfür ist, dass der OMDocEncoder einen jeden Maßterm, bzw. eine jede Liste von Maßtermen, in einen eigenen Beweis kapselt. Der OMDocDecoder muss folglich in der Dekodierung von Terminierungsbeweisen eine Liste $\mathcal{L}$ von proof-Elementen abarbeiten. Für jedes proof-Element aus $\mathcal{L}$ wird in dem ersten Schritt der Maßterme, die aus dem Maßterm entstandenen Terminierungshypothesen und die Position, aus der die jeweilige Hypothese in dem betreffenden Funktionrumpf gewonnen wurden, dekodiert. Zur Rekonstruktion der Maßterme und Terminierungshypothese verwendet auch hier der OMDocDecoder wieder den FPParser. Sollte in einer dieser Rekonstruktionen ein Fehler auftreten so wird der gesamte Dekodierungsvorgang abgebrochen! Sind aus allen proof Elementen der Liste $\mathcal{L}$ erfolgreich alle Maßterme, Terminierungshypothesen und Positionen dekodiert worden, werden im zweiten Schritt diese Informationen mit denen, die nach dem Einfügung einer benutzerdefinierten ✓eriFun -Funktion automatisch durch das System erzeugt werden, verglichen. Zuerst werden die Maßterme betrachtet. Sollten sich die dekodierten Maßteme von denen aus dem System unterscheiden, so werden (1) diejenigen Maßterme aus dem System entfernt, die nicht in der Menge der dekodierten Maßterme vorkommen und (2) diejenigen Maßterme in das System hinzugefügt, die in der Menge der dekodierten Maßterem vorhanden sind, nicht aber in dem System. Als nächstes werden die Terminierungshypothesen einschließlich der entsprechenden Positionen verglichen. Sollte an dieser Stelle ein Unterschied auftreten, so wird eine Warnung an den Benutzer ausgegeben und die Dekodierung dieses Terminierungsbeweises, nicht der gesamte Dekodierungsvorgang, wird abgebrochen. Im Fall der Übereinstimmung wird für jede Terminierungshypothese der referenzierte HPL-Beweis, wie in Unterabschnitt 5.2.6 beschrieben, dekodiert. Ist die Dekodierung der HPL-Beweise aller Terminierungshypothesen erfolgreich, ist die Dekodierung einer benutzerdefinierten ✓eriFun -Funktion vollständig abgeschlossen.

5.2.8 Rekursionseliminationsbeweise

Wie bereits in Unterabschnitt 5.2.7 erwähnt, wird in der Dekodierung von ✓veriFun -Beweisen aus der Sicht des `OMDocDecoder`s zwischen zwei Klassen unterschieden. In diesem Abschnitt wird die Dekodierung der zusätzlichen Informationen von Rekursionseliminationsbeweisen behandelt. Hierbei handelt es sich um jeweils zwei Formeln, die für jede Kombination von rekursiven Aufrufen innerhalb des Rumpfes der zugehörigen benutzerdefinierten ✓veriFun-Funktion durch das System definiert werden. Eine Formel für den Fall, dass die rekursiven Aufrufe durch `true` und eine für den Fall, dass die rekursiven Aufruf durch `false` ersetzen werden können (Unterabschnitt 5.1.8). Auch diese Zusätze werden durch den `OMDocEncoder` in einem separaten `proof`-Element enkodiert. Als Schnittstelle wird erneut eine informale Hypothese in Form eines `assertion`-Elements verwendet. Im Gegenstatz zu der Enkodierung von Terminierungsbeweisen, werden hier jedoch die einzelnen Formeln nicht in einzelne `proof`-Elemente enkodiert, sondern alle Formeln, die entsprechenden Rekursionspositionsmengen und der Status der synthetisierten Funktion werden für einen Rekursionseliminationsbeweis in einem `proof`-Element gespeichert. Die Dekodierung eines Rekursionseliminationsbeweises durch den `OMDocDecoder` gliedert sich dann in drei wesentliche Schritte.

```
1  if (VeriFun.getFunctionStatus() != StatusAP.IGNORED)
   {
     if ((originalFuncBodyVF != originalFuncBodyOMDoc) or (simplifiedFuncBodyVF != simplifiedFuncBodyOMDoc))
       Raise user message!
     else
6    {
       if (VeriFun.getFunctionStatus() == VERIFIED)
         if (recelimFuncBodyVF != recelimFuncBodyOMDoc)
           Raise user message!

11     else if (VeriFun.getFunctionStatus() == StatusAP.READY)
       {
         if ((trueRecElimsOMDoc == trueRecElimsVF) and (falseRecElimsOMDoc == falseRecElimsVF))
         {
           OMdocDecoder.decodeRecursionEliminationProof();
16
           if ((OMDocDecoder.decodeFunctionStatus() == VERIFIED) and (VeriFun.getFunctionStatus() != VERIFIED))
           {
             VeriFun.performRecElim();

21           if (recelimFuncBodyVF != recelimFuncBodyOMDoc)
               Raise user message!
           }
         }
         else Raise user message!
26     }

       else
       {
         if (trueRecElims.isEmpty() && falseRecElims.isEmpty())
31         Raise user message!
       }
     }
   }
```

Im ersten Schritt wird der Status der durch das System automatisch synthetisierten Funktion überprüft. Ist dieser Status *Ignored*, so ist die Dekodierung dieses Rekursionse-

liminationsbeweises abgeschlossen. Ist der Status ungleich *Ignored*, so werden im zweiten Schritt die drei enkodierten Rümpfe (`original`, `simplified` und `recelim`) der synthetisierten Funktion dekodiert. Zur Rekonstruktion der jeweiligen Rümpfe verwendet der `OMDocDecoder` den `FPParser`, so dass auch hier ein Fehler in der Rekonstruktion zum Abbruch des Dekodierungsvorgangs führt. War die Rekonstruktion erfolgreich, werden im dritten Schritt die dekodierten `original`- und `simplified`-Rümpfe mit den automatisch durch das System synthetisierten verglichen. Unterscheiden sich die jeweiligen Paare, so wird dem Benutzer dies mitgeteilt und die Dekodierung dieses Rekursionseliminationsbeweises wird abgebrochen. Stimmen die Paare überein, so werden in dem Fall, dass die automatisch durch das System synthetisierte Funktion den Status *Verified* trägt, die beiden `recelim`-Rümpfe verglichen. Auch wenn an dieser Stelle ein Unterschied auftritt, wird der Benutzer darüber informiert und die Dekodierung dieses Rekursionseliminationsbeweises wird abgebrochen. Stimmen die beiden Rümpfe überein, so ist in diesem Fall für den `OMDocDecoder` diese Dekodierung auch abgeschlossen, da die synthetisierte Funktion bereits den Status *Verified* trägt. In dem Fall, dass die automatisch durch das System synthetisierte Funktion den Status *Ready* trägt, werden die Formeln der Rekursionselimination dekodiert und mit den durch das System automatisch für diese synthetisierte Funktion berechneten verglichen. Stimmen die jeweiligen Formelmengen nicht überein, so wird dies erneut dem Benutzer mitgeteilt und die Dekodierung dieses Rekursionseliminationsbeweises wird abgebrochen. Ist eine Übereinstimmung der beiden Formelmengen vorhanden, so werden die referenzierten HPL-Beweise der Rekursionseliminationsformeln analog zu der in Unterabschnitt 5.2.6 beschriebenen Vorgehensweise, dekodiert. Im Anschluß an die Dekodierung aller HPL Beweise der Rekursionseliminationsformeln wird der Status der Funktion dekodiert. Wenn dieser den Wert *Verified* erhält, jedoch der Status der automatisch durch das System synthetisierten Funktion nicht *Verified* ist, wird zunächst das System veranlasst, eine erneute Rekursionselimination auf der synthetisierten Funktion durchzuführen. Nach dieser Durchführung werden dann die beiden `recelim`-Rümpfe verglichen. In jedem Fall — Übereinstimmung oder Unterscheidung — ist die Dekodierung dieses Rekursionseliminationsbeweises für diesen Fall abgeschlossen. Allerdings wird der Benutzer über eine Unterscheidung zusätzlich informiert. Ist in keinem der hier behandelten Fallunterscheidungen ein Fehler entstanden oder eine Benutzermeldung erzeugt worden, ist der Beweis einer Rekursonselimination einer synthetisierten ✓`eriFun`-Funktion vollständig abgeschlossen.

Zusammenfassung & Ausblick

Mit dieser Arbeit ist die Integration der semantischen Markupsprache OMDoc in den Theorembeweiser ✓eriFun gelungen. In ✓eriFun erstellte Programme und Beweise können nun in einem wiederverwendbaren, "maschinen-verstehbaren", "menschenlesbaren", standardisierten und anwendungsübergreifenden Format repräsentiert werden. Die durch das System erstellen serialisierten und somit versionsabhängigen Dateien werden durch das offene OMDoc-Format abgelöst. Basierend auf dem wohldefiniertem Vokabular von OMDoc steht nun für ✓eriFun die Tür zu dem komplexen Netzwerk der mathematischen Welt offen. Um diese Möglichkeiten jedoch voll ausschöpfen zu können, sind schon während und zum Schluss dieser Arbeit bereits folgende Verbesserungsvorschläge dem Autor selbst und den diese Arbeit betreuenden Personen für die nächste Version des OMDoc Enkoders bzw. Dekoders aufgefallen:

▶ `Enkodierung der ✓eriFun Ordnerstruktur`
Der aktuelle `OMDocEncoder` verwendet zur Referenzierung des "Vater-Ordners" nicht das `xml:id` Attribut des entsprechenden `private`-Elements, sondern speichert als Referenz direkt den Namen des "Vater-Ordners" im Klartext. Dieser Sachverhalt sollte in der nächsten Version auf die Verwendung von `xml:id` Attributen umgestellt werden.

▶ OMDoc-`presentation-Elemente`
Alle in ✓eriFun verwendeten Repäsentanten von Symbolen, Variablen, Typvariablen und Sorten sollten in OMDoc `presentation` Elemente ausgelagert werden und lediglich über das `xml:id`-Attribut referenziert werden. Dies würde die Flexibilität der erlaubten Zeichenketten für Repräsentanten erheblich erweitern.

▶ LaTeX `Konvertierung`
Es sollte die Möglichkeit bestehen, ein in ✓eriFun erstelltes Programm mit all den darin befindlichen Beweisen ausdrucken zu können. OMDoc bietet hierfür eine optimale Ausgangsbasis. Man könnte durch ✓eriFun erstellt OMDoc Dateien via XSLT in LaTeX überführen und diese neue erstellte Datei problemlos in ein Pdf drucken.

▶ `Sortierung`
In ✓eriFun besteht die Möglichkeit, Elemente innerhalb eines Programms per "drag-and-drop" neu zu sortieren. Der `OMDocEncoder` speichert die Elemente jedoch unter einer topologischen Ordnung. Da der `OMDocDecoder` eine OMDoc-Datei sequentiell einliest, kann die durch den Anwender vorgenommene Sortierung nicht rekonstruiert werden.

Würde der `OMDocEncoder` zu jedem Element zusätzlich dessen Index in dem entsprechenden ✓`eriFun` -Programm speichern, wäre es dem `OMDocDecoder` möglich, die ursprünglich Sortierung wieder rekonstruieren zu können.
⋆ *an diese Lösung arbeitet der Autor bereits parallel zu dieser Ausarbeitung*

▶ `Rekursionselimination`
Der `OMDocDecoder` gibt keine Warnung aus, wenn eine rekonstruierte Funktion plötzlich mehr Rekursionseliminationsformeln hat als in der OMDoc-Datei. Solch geartete Warnungen sollten in dem ✓`eriFun` *System Log* angezeigt werden.
⋆ *an diese Lösung arbeitet der Autor bereits parallel zu dieser Ausarbeitung*

▶ `Structure sharing`
Neben dem schon verwendeten *structure sharing* in HPL-Beweisen, sollte dieser Mechaismus auf ein gesamtes OMDoc-Dokument ausgeweitet werden. Die Größe einer solchen Datei würde sich erheblich verkleinern und die Dekodierung beschleunigen.

▶ `Importierte Theorien`
In der aktuellen Version des `OMDocDecoder`s können keine importierten Theorien eingelesen werden. Selbst die `vafp`-Theorie der vordefinierten ✓`eriFun` -Elemente wird nicht eingelesen, sondern ist fest im Dekoder implementiert. Zum einen sollten die vordefinierten Elemente durch den `OMDocEncoder` berücksichtigt werden und zum anderen sollte der `OMDocDecoder` um die Funktionalität erweitert werden, importierte Theorien einlesen zu können. Ein direkter Vorteil dieser Erweiterung wäre, dass nach Modifikationen der vordefinierten Elemente der Dekoder nicht angepasst werden muß und sich die `vafp.omdoc` Datei automatisch anpasst.

▶ `Semantischer Markup von Parametern für Beweismethoden`
Die Parameterübergabe an Beweismethoden erfolgt in Form von geordneten Listen von `OMOBJ` Elementen. Die Ordnung der einzelnen Informationen darf nicht verändert werden, da ansonsten keine korrekte Dekodierung mehr möglich ist. Desweiteren lässt sich nur sehr schwer erkennen, um welche Informationen es sich dabei handelt. Man sollte die Parameter für Beweismethoden mit einem semantischen Markup versehen, um zum einen die strikte Anordnung verwerfen zu können und zum anderen eine bessere "Menschen-Lesbarkeit" auch auf dieser Ebene zu gewährleisten.
⋆ *an diese Lösung arbeitet der Autor bereits parallel zu dieser Ausarbeitung*

▶ `OpenMathDecoder`
Die Funktionalität der OpenMath-Dekodierung sollte aus dem `OMDocDecoder` entkoppelt und so gestaltet werden, dass diese unabhängig von der Übersetzung in die $\mathcal{FP}$-Syntax ist.
⋆ *an diese Lösung arbeitet der Autor bereits parallel zu dieser Ausarbeitung*

▶ `Enkodierung synthetisierter Lemmata`
Die Enkodierung synthetisierter Lemmata unterscheidet sich von der der benutzerdefinierten lediglich durch das Anhängen eines Suffixes. So wird der `ID` des `xml:id`-Attributs des `assertion`-Elements sowohl für *Boundedness*- als auch für *Projektions*-Lemmata das

Suffix `proj` angehängt. In der nächsten Version des `OMDocEncoder`s sollte hier in dem Suffix zwischen `proj` und `bounded` unterschieden werden.

Literaturverzeichnis

[Apa] http://xml.apache.org/

[BF96] Borenstein, N. ; Freed, N.: MIME (Multipurpose Internet Mail Extensions) Part One: Format of Internet Message Bodies. (1996). – http://www.ietf.org/rfc/rfc2045.txt

[BLHL01] Berners-Lee, T. ; Hendler, J. ; Lassila, O.: The Semantic Web: Research and Applications Second European Semantic Web Conference, 2001

[Dou] http://purl.org/dc/elements/1.1/

[GP00] Goldfarb, C. F. ; Prescod, P.: *Das XML-Handbuch. Anwendungen, Produkte, Technologien.* Addison-Wesley, 2000

[HCD04] Hunter, D. ; Cagle, K. ; Dix, C.: *Beginning XML.* Wrox Press, 2004

[HM02] Harold, E. R. ; Means, W. S.: *XML in a Nutshell.* Oreilly, 2002

[Jav] http://java.sun.com

[Koh06] Kohlhase, M.: *OMDoc - An Open Markup Format For Mathematical Documents.* Springer Verlag, 2006

[MBK00] Martin, D. ; Birbeck, M. ; Kay, M.: *XML professionell.* MITP Verlag GmbH, 2000

[Nam] http://www.w3.org/TR/REC-xml-names/

[OMD] http://www.mathweb.org/omdoc/index.html

[Ope] http://www.openmath.org

[Sel] http://www.selfhtml.org

[SS04] Staab, S. ; Studer, R.: *Handbook on Ontologies.* Springer Verlag, 2004

[Sun] http://www.sun.com

[Tam00] Tamm, M.: Packen wie noch nie. Datenkompression mit dem BWT-Algorithmus. In: *c't Magazine für Computer Technik* (2000)

[Ver] http://www.verifun.de

[W3C] http://www.w3.org/Style/CSS-vs-XSL

[Wat02] Watson, D. G.: Brief History of Document Markup / University of Florida. 2002 (Circular 1086). – Forschungsbericht

[WS05] Walther, Christoph ; Schweitzer, Stephan: Automated Termination Analysis for Incompletely Defined Programs. In: Baader, Franz (Hrsg.) ; Voronkov, Andrei (Hrsg.): *Proc. of the 11th Inter. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-11)* Bd. 3452. Montevideo, Uruguay : Springer, 2005, S. 332–346

[XMLa] http://www.w3.org/TR/REC-xml/

[XMLb] http://www.w3.org/XML/Schema

Index

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z