



HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Entwicklung eines browserbasierten Dokumenteditors für (wiss.) Texte mit domänenspezifischen Inhalten

Sven Hodapp

Konstanz, 28. Juli 2014 (ENTWURF)

MASTERARBEIT

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science (M. Sc.)

an der

Hochschule Konstanz

Technik, Wirtschaft und Gestaltung

Fakultät Informatik

Studiengang Master of Science Informatik

Thema: **Entwicklung eines browserbasierten Dokumenteditors
für (wiss.) Texte mit domänenspezifischen Inhalten**

Masterkandidat: Sven Hodapp, Waldenburger Ring 3, 53119 Bonn

Firma: Fraunhofer-Institut für Algorithmen und Wissenschaftliches
Rechnen SCAI

1. Prüfer: Prof. Dr. Marko Boger
2. Prüfer: Dr. Marc Zimmermann

Schlagworte: expose

Ausgabedatum: 1. März 2014
Entwurf-Version: 28. Juli 2014

Kurzreferat (Abstract)

Maximal 1200 Anschläge, alles in einem Abschnitt. (Nach DIN 1422-1).

Ehrenwörtliche Erklärung

Hiermit erkläre ich *Sven Hodapp*, geboren am 16. September 1987 in Singen am Hohentwiel, dass ich

- (1) meine Masterarbeit mit dem Titel

Entwicklung eines browserbasierten Dokumenteditors für (wiss.) Texte mit domänenspezifischen Inhalten

beim Fraunhofer-Institut für Algorithmen und Wissenschaftliches Rechnen SCAI unter Anleitung von Prof. Dr. Marko Boger selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

- (2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 28. Juli 2014

(Unterschrift)

Inhaltsverzeichnis

Kurzreferat (Abstract)	iii
Ehrenwörtliche Erklärung	iv
1 Einleitung	1
1.1 Motivation	1
1.2 Problemstellung	2
1.3 Methode	3
1.4 Anforderungen	3
1.5 Idee	5
1.6 Szenario	6
1.7 Fragen von wissenschaftlichem Interesse	6
2 Theorie	8
2.1 Modellbegriff	8
2.1.1 Modellmerkmale	9
2.1.2 Modell eines Dokuments	10

2.2	Abstrakter Syntaxbaum	11
2.2.1	Allgemeine Implementierung	12
2.2.2	Übertragung auf den Prototypen	12
2.3	Projektionseditoren	13
2.3.1	Übertragung auf den Prototypen	14
2.4	Metamodellbegriff	15
2.4.1	Metamodell des Dokumentmodells	16
2.5	Semiotik	20
2.5.1	Anwendung auf (wiss.) Dokumente	21
2.6	Taxonomie wissenschaftlicher Publikationen	24
2.6.1	Dokumentelemente	25
2.6.2	Taxonomie	27
2.6.3	Dokumentstruktur Matrix	28
2.6.4	Deutung	29
3	Anwendungsfälle	32
3.1	Erstellung wissenschaftlicher Dokumente	32
3.1.1	Szenario: Chemie	33
3.2	UIMA CAS Editor	34
3.2.1	Konzeptuelle Umsetzung	36
3.3	Dokumentation von Spray-Modellen	37
3.3.1	Erhoffter Nutzen	39

3.3.2	Konzeptionelle Umsetzung	39
3.4	Transformation in andere Formate	42
4	Entwicklung des Prototypen	45
4.1	Vorgehen	46
4.1.1	Getestete Spezifikationen	46
4.2	Basiskonzepte	49
4.3	Architektur	50
4.3.1	Wurzel Aktor	51
4.3.2	Basis Aktor	52
4.3.3	Metamodell erweitern	53
4.3.4	Verweisungen auflösen	54
4.4	Eingesetzte Technologien	55
4.4.1	Scala	56
4.4.2	Akka	56
4.4.3	Xitrum	58
4.4.4	Web Standards	58
4.4.5	Dokumentelemente und Domäneneditoren	60
4.4.6	Algorithmen und Datenstrukturen	62
4.5	Erreichter Funktionsumfang	65
4.5.1	Codestatistik	67
	Literaturverzeichnis	68

Kapitel 1

Einleitung

1.1 Motivation

Die Erstellung von qualitativ hochwertigen (wissenschaftlichen) Dokumenten ist keine einfache Sache. Das Computerzeitalter konnte zwar schon viel verbessern, aber es gibt noch immer Situationen, in denen die Dokumentenerstellung sehr mühselig werden kann.

Zum einen kostet es u.U. sehr viel Zeit, Querreferenzierungen im Dokument zu pflegen und konsistent zu halten. Hierfür hat insbesondere LaTeX eine solide Lösung, jedoch stößt man auf der Meta-Ebene (der LaTeX Code selbst) wieder auf das Problem. Beispiel: Man definiert ein Label und referenziert an anderen Stellen darauf. Wird das Label irgendwann umbenannt (weil sich z.B. die Überschrift geändert hat), muss es an allen referenzierten Stellen auch umbenannt werden (klassisches Refactoring) – und das geschieht leider nicht automatisch.

Zum anderen verstehen die Anwendungen oft nur die Domäne „Dokument“, wenn man z.B. chemische Formeln zeichnen will, müssen diese über externe Ressourcen hinzugefügt werden. Dabei gibt es keinen wirklichen Zugriff mehr auf die Meta-Informationen, die eigentlich durch das Domänenmodell – die chemische Formel – mitgeliefert wird. Das kann wiederum zu inkonsistenten

Dokumenten führen. Wenn z.B. nachträglich an der chemischen Formel etwas geändert wird und nur die Abbildung von Benutzer aktualisiert wurde, dann sind u.U. die Verweise (z.B. auf die Masse des chemischen Moleküls) in den beschreibenden Sätzen nicht mehr korrekt. Es fehlt das semantische Verständnis seitens des Dokuments, es weiß also nicht, was die chemische Formel zu bedeuten hat.

Zudem liegt hinter keinem derzeit verfassten Dokument ein echtes/sauberes *Metamodell*. Das heißt ein Modell, welches eine Struktur vorschreibt, wann welche Dokumentbestandteile zulässig sind. Dies kann einen Dokumentenverfasser dabei helfen, ein Dokument (wie z.B. ein Paper der Fraunhofer Gesellschaft) gemäß definierter Vereinbarungen aufzubauen, dabei braucht er kein Wissen über die genaue Vereinbarung und kann dennoch ein formal korrektes Dokument erstellen. Solche Vereinbarungen können z.B. ein Corporate Design oder eine spezielle Reihenfolge von Dokumentelementen sein. Weiterhin kann man ganze Dokumentklassen aus einem solchen Metamodell ableiten; also all jene Dokumente, welchen das gleiche Metamodell übergeordnet ist, gehören somit ganz formal zur gleichen Dokumentklasse. Beispiele für solche Klassen sind: Master-Arbeit, Journal-Paper oder EU-Patent etc.

1.2 Problemstellung

Aus der Motivation erschließen sich folgende Defizite bzw. Problemfelder in bisherigen Systemen:

- (i) Mangel an Konsistenz: In bisherigen Textverarbeitungssystemen ist es sehr leicht, Inkonsistenzen ins Dokument einzubringen.
- (ii) Mangel an Domänenwissen: Dokumente „verstehen“ die Konventionen, Konzepte oder Modelle einer Wissenschaft nicht.
- (iii) Mangel an Semantik: Bedeutungen einzelner Bestandteile eines Dokuments werden oft nicht explizit sichtbar bzw. überhaupt erst verfügbar gemacht.

- (iv) Mangel an Metamodellierung: Es fehlen¹ formale Vereinbarungen, die eine Dokumentenklasse ausmachen und dem Benutzer bei der Strukturierung helfen.

Das wirft die folgende Frage auf: Kann man ein Autorensystem erschaffen, welches die o.g. Defizite aufhebt?

1.3 Methode

In erster Linie soll ein Prototyp entwickelt werden, der einen konkreten Lösungsvorschlag für die Problemstellungen aus Abschnitt 1.2 vorlegt.

Da ein Prototyp wunderbar untersucht werden kann, ist es möglich, theoretische Ideen besser zu veranschaulichen oder überhaupt erst greifbar zu machen. Zudem können unterschiedliche Anwendungsfälle demonstriert werden, um potentiellen Interessenten dieser Technologie Anschauungsmaterial zu liefern. Die Entwicklung eines Protoypen macht bei dieser Masterarbeit Sinn, da am Prototypen:

- theoretische Konzepte aufgezeigt und empirisch validiert² werden können, und
- leicht Anwendungsfälle ausprobiert werden können, um die praktische Tauglichkeit der Softwarearchitektur zu verifizieren³.

1.4 Anforderungen

Bei den Anforderungen an den Prototypen kann man schon konkreter in Richtung Implementierung blicken. Hier die Basisanforderungen, um die Problem-

¹ Oder gehen nicht weit genug, wie z.B. im Falle von LaTeX.

² Funktioniert das Konzept richtig?

³ Verifikation prüft ob ein System richtig gebaut ist. Oder im Falle von Dokumenten: Ist das Dokument zu einer Spezifikation konform?

stellungen zu lösen:

Die kleinste Einheit im System ist das *Dokumentelement*. Beispiele für Dokumentelemente sind: Abschnitte, Absätze, Tabellen, Abbildungen, Fußnoten, etc.

Dokumentelemente sollen in der Lage sein, konkretes „lebendiges“ Domänenwissen zu beinhalten. Das heißt, dass z.B. eine Zeichnung eines chemischen Moleküls im Hintergrund tatsächlich auf einem formalen Molekülmodell basiert.

Reichhaltige Verweisungen unter den Dokumentelementen sollen möglich sein. Das heißt, dass z.B. das formale Modell des Moleküls anderen Dokumentelementen weitere (u.U. berechnete) Informationen bereitstellen kann.

Ein Metamodell definiert die im Dokument vorkommenden Dokumentelemente formal. Das Dokument selbst soll folglich als Modell (Instanz) aufgefasst werden.

Um die Software benutzbar zu machen:

- Es soll ein reaktives System sein, welches sofort auf Eingaben des Benutzers bzw. Veränderungen des Systemzustands reagiert.
- Es soll also ein „lebendiges“ Dokument werden, d.h. es sollen keine (spürbaren) Kompilierzeiten, um das Dokument zu erstellen, entstehen.
- Da es in wissenschaftlichen Dokumenten meist zu komplizierten Verweisungen kommt, müssen diese in jedem Fall konsistent gehalten werden. Es soll daher das Verweisvariablen-Refactoring-Problem abgedeckt werden.
- Es soll ein Einheimischer des WWW werden, d.h. Web Standards sind das Ausgabeformat. Sie übernehmen das Setzen des Dokuments, bieten dem Benutzer eine Editierschnittstelle, ermöglichen Kollaboration und bieten eine ubiquitäre Wissensrepräsentation.

Durch einige dieser Anforderungen gewinnen wir implizit einen Projektionseditor und sogar Potential für ein Dokument „Query Interface“.

1.5 Idee

Im Vordergrund stand die Idee, meine Bachelorarbeit (Hodapp u. a., 2013) so weiter zu entwickeln, dass Dokumente in einem Struktureditor oder Projektionseditor verfasst werden können und mit domänenspezifischen Dokumentelementen interagiert werden kann.

In diesem Kapitel wird also kurz die prinzipielle Idee umrissen, wie man die Problemstellung unter den gegebenen Anforderungen lösen könnte.

Grundsatz der Idee ist, dass jedes Dokumentelement auf ein Aktor abgebildet wird. Ein Aktor kann Nachrichten senden und empfangen, zudem kapselt er einen Zustand und ein Verhalten.

Aktoren können, ebenso wie ein Dokument, hierarchisch angeordnet werden. Das heißt ein Aktor kann Kinder und Geschwister haben. Diese Anordnung ergibt eine baumartige Graphenstruktur, worin ein Aktor einem Knoten entspricht und eine Referenz⁴ auf einen anderen Aktor einer Kante.

Diese Baumstruktur kann als abstrakter Syntaxbaum (AST) der Dokumentenstruktur aufgefasst werden. Der AST kann als ein Modell für ein Dokument aufgefasst werden. Dies wird auf Abbildung 1.1 veranschaulicht. Zudem kann mit Hilfe eines AST ein Struktureditor umgesetzt werden. Der hier mit den Aktoren umgesetzte AST könnte als „reaktiver AST“ bezeichnet werden, da er dank der Aktoren (quasi gratis) besondere Eigenschaften erhält:

- Verteilbarkeit, Fehlertoleranz und massive Parallelität. AST kann z.B. auf ein Cluster gebracht werden, z.B. zur Kompilierung des Dokuments in „der Cloud“; Anwendung von MapReduce auf Dokument oder Dokumentenserie; Jeder Aktor ist autonom, d.h. z.B. asynchrone

⁴ Über eine solche Aktor Referenz können Nachrichten an andere Aktoren übermittelt werden.

Datenbankzugriffe oder schnelle Reaktionszeiten auf Änderungen.

- Möglichkeit zum Query Interface ausgebaut zu werden. Das Dokument bzw. Dokumentelement kann auf Nachrichten reagieren. Beispielsweise zur Aggregation neuer Informationen.
- Möglichkeit mit anderen (Aktoren-) Systemen (z.B. Dokumenten) zu interagieren. Beispielsweise via Akka Remoting Protokoll oder Anbindung an REST-Schnittstellen.

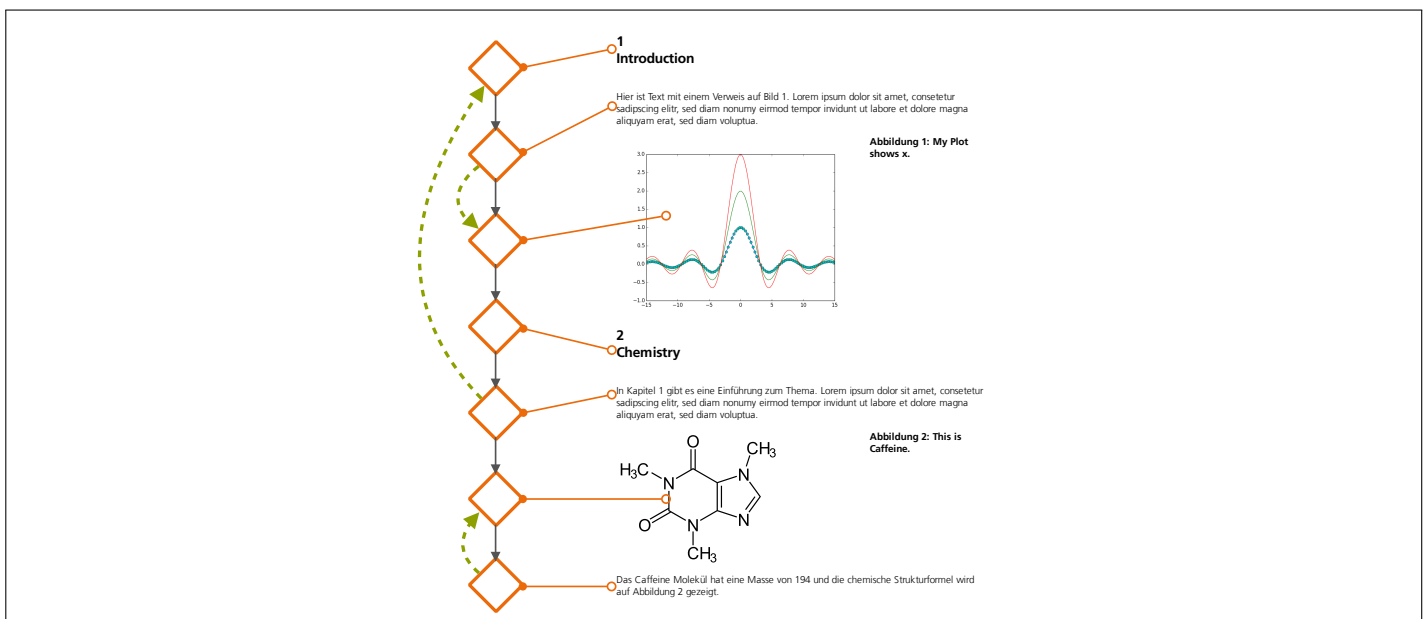


Abbildung 1.1: Modell für ein Dokument. Jeder Aktor (Raute) repräsentiert ein Dokumentelement. Die Aktoren kennen sich (graue Pfeile) und spannen somit einen Graphen auf. Die Aktoren können Nachrichten (grüne Pfeile) austauschen, z.B. um Benummerungen aufzulösen.

1.6 Szenario

1.7 Fragen von wissenschaftlichem Interesse

Bereiche in denen (wiss.) Fragestellungen durch den Prototypen beantwortet werden können:

- Modellgetriebene Software-Entwicklung: Ist es sinnvoll Dokumente als Modell aufzufassen? Wie könnte eine sinnvolle Modellierung von Dokumenten aussehen? Gibt es ein gemeinsames Metamodell?
- Programmiersprachen, Compilerbau: Kann ein Aktorsystem verwendet werden, um einen abstrakten Syntaxbaum zu implementieren? Taugt dieser abstrakte Syntaxbaum als sinnvolle Architekturgrundlage für einen Projektionseditor, und damit auch als Code Generator?
- Knowledge Engineering / Management; Semantik, Ontologie: Brauchen wir mehr explizite Semantik innerhalb von Dokumenten? Ergeben sich Vorteile, wenn diese Semantik direkt vom Autor transportiert wird? Ist es möglich, dass sich Dokumente zu einem gewissen Grad selbst verifizieren⁵ und damit konsistenter machen?
- Bibliotheks- und Informationswissenschaften: Gibt es eine allgemeingültige Taxonomie für wiss. Publikationen?

⁵ Verifikation prüft ob ein System richtig gebaut ist. Oder im Falle von Dokumenten: Ist das Dokument zu einer Spezifikation konform?

Kapitel 2

Theorie

In diesem Kapitel werden theoretische Konzepte erarbeitet.

2.1 Modellbegriff

Modellierung ist das uns angeborene Verfahren, das komplexe Universum auf eine überschaubare Welt zu reduzieren. Indem wir sichtbare und unsichtbare Phänomene auf Begriffe abbilden und nur noch mit diesen umgehen, wird die Gesamtzahl der zu betrachtenden Gegenstände beherrschbar[...] (Ludewig, 2002, S. 7)

Gerade auch in der Wissenschaft spielen Modelle die zentrale Rolle, denn „Das Resultat einer Forschung ist in jedem Falle ein Modell, eine Theorie.“ (Ludewig, 2002, S. 8) Warum sollte man also nicht den Weg ganz konsequent gehen, und ebenfalls die wissenschaftliche Dokumentation selbst als Modellierung betrachten? Warum sollte das Dokument selbst nicht Modelle aus einer Wissenschaft bereitstellen? Das heißt, dass eine Wissenschaftlerin bzw. ein Wissenschaftler seine Modelle direkt im Dokument verwenden kann?

2.1.1 Modellmerkmale

In (Ludewig, 2002, S. 9) ist eine griffige Zusammenfassung der drei zwingenden Merkmale die nach (Stachowiak, 1973) vorliegen müssen, um als Modell zu gelten, aufgeführt. Diese werden hier sinngemäß zusammengefasst und durch Abbildung 2.1 veranschaulicht:

1. **Abbildung:** Ein Modell ist immer ein Abbild eines Originals. Die Abbilder können beliebig geartet sein, d.h. das Modell muss dem Original äußerlich nicht ähneln. Das Original auf welches sich das Modell bezieht, kann neben natürlichen Objekten auch künstlich, geplant oder vermutet sein.
2. **Verkürzung:** Ein Modell erfasst nicht alle Merkmale des Originals. Durch Verkürzung fallen Attribute weg, diese werden übergangen oder präteriert genannt. Das Modell kann jedoch zusätzliche Attribute haben, die so im Original nicht vorkommen, diese werden überflüssig oder abundant genannt.
3. **Pragmatismus:** Ein Modell muss einen Sinn haben, um unter bestimmten Fragestellungen das Original ersetzen können. Das heißt, es wird das Modell statt des Originals untersucht, um daran die Nützlichkeit für eine Zielgruppe (für wen, wann, wozu?) festzustellen.

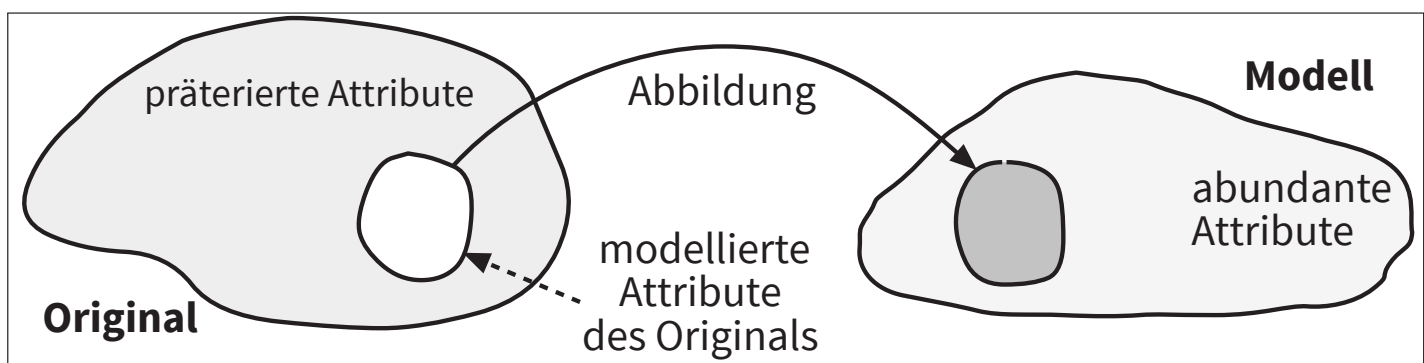


Abbildung 2.1: Schematische Zeichnung der Original-Modell-Beziehung. Entnommen aus (Ludewig, 2002, S. 9).

2.1.2 Modell eines Dokuments

Bevor ein passendes Modell für Dokumente gefunden werden kann, muss man sich bewusst sein dass es dem Abbildungsmerkmal, Verkürzungsmerkmal und dem pragmatischen Merkmal, welche in Kapitel 2.1.1 beschrieben sind, genügen muss.

Das Original ist in jedem Fall ein gesetztes statisches Dokument, z.B. ein auf Papier gedrucktes Buch oder ein (wiss.) Bericht im PDF-Dateiformat.

Dort interessieren uns als Autor in erster Linie aber nur die wirklich inhaltstragenden Attribute des Originals. Auf einer höheren Abstraktionsebene entsprechen diese den Dokumentelementen, wie z.B. Abschnitte, Absätze, Abbildungen etc. Im Falle von z.B. Abschnitten ist nur der Titel maßgeblich, die Benummerung kann auch erst später (durch Berechnung) hinzugefügt werden. Das heißt hier wäre der Titel ein Attribut welches vom Original in das Modell abgebildet wird; die Benummerung wäre jedoch ein abundantes Attribut welches erst innerhalb des Modells erstellt wird. Layoutinformationen, Schriftarten oder Papiersorte spielen dafür keine Rolle und können somit weggelassen werden.

Wenn der Autor an seinem Dokument arbeitet, dann stets über das Modell, wo er sich ausschließlich auf die Dokumentelemente konzentrieren kann. Das Modell kann präskriptiv sein, d.h. aus dem Modell kann wieder ein Original entspringen.

Hier wird der abstrakte Syntaxbaum als Modell für das Dokuments dienen, indem jedes Dokumentelement aus ein Knoten des abstrakten Syntaxbaum aufgefasst wird. Mehr zum abstrakten Syntaxbaum in Abschnitt 2.2.

Beweis

- Abbildungsmerkmal: Der abstrakte Syntaxbaum („das Modell“) ist z.B. ein Abbild eines gedruckten Buches, da Dokumentelemente als Knoten im Syntaxbaum vorkommen.

- Verkürzungsmerkmal: Nur die Attribute aus dem Original werden modelliert, die den maßgeblichen Inhalt des Dokuments transportieren (z.B. Text). Diese Inhalte werden als Dokumentelemente modelliert.
- Pragmatischen Merkmal: Der Autor greift auf das Modell zurück, um daran die herausgelösten und essentiellen Dokumentelemente zu untersuchen bzw. zu manipulieren.

Alle Bedingungen für den Modellbegriff nach (Stachowiak, 1973) sind erfüllt. ▪

2.2 Abstrakter Syntaxbaum

In diesem Abschnitt werden die prinzipiellen Eigenschaften von abstrakten Syntaxbäumen, kurz AST, beschrieben.

Laut (Aho u. a., 2007) ist der AST eine Datenstruktur, die von einem Übersetzer als Zwischenrepräsentation eines Quellcodes generiert wird. Er repräsentiert die hierarchische syntaktische Struktur eines Programms. Aus einer solchen Zwischenrepräsentation wird schlussendlich das Zielfprogramm generiert.

Während der Syntaxanalyse (parsing) werden Syntaxbaum-Knoten erstellt, welche wiederum signifikante Programmkonstrukte repräsentieren. Die Kinder des Knoten sind die bedeutungstragenden Komponenten des Konstrukts. Beispielsweise (s. Abb. 2.2): Gegeben ist ein AST für einen Ausdruck (expression), dann repräsentiert jeder innere Knoten einen Operator und die Kinder des Knoten sind die Operanden. Man beachte jedoch, dass Syntaxbäume für beliebige Konstrukte erstellt werden können und nicht auf Ausdrücke beschränkt sind. Jedes Konstrukt ist durch einen Knoten repräsentiert, dessen Kinder semantisch bedeutungsvolle Komponenten des Konstruktes sind. Durch fortschreitende Analyse können Informationen vom Übersetzer zu den Knoten als Attribute hinzugefügt werden. (ebd.)

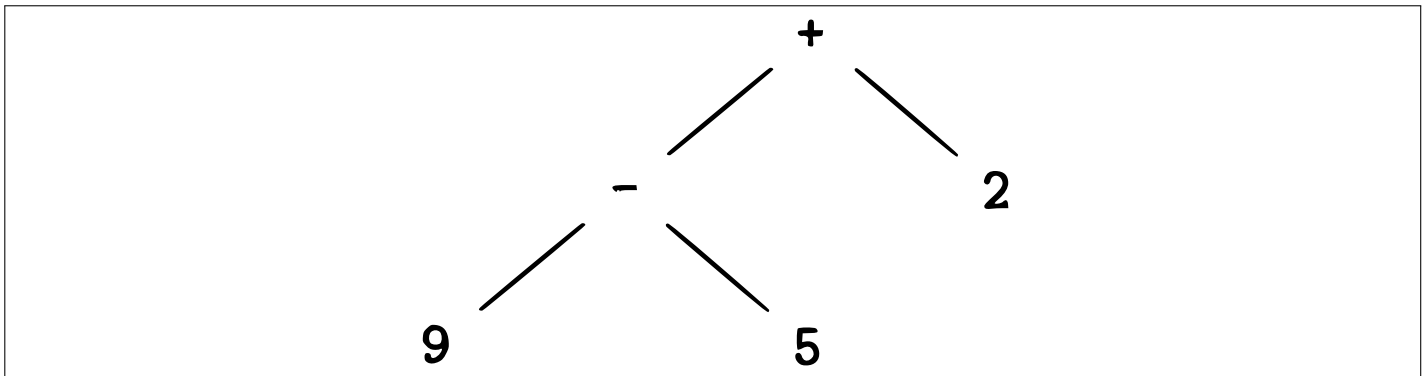


Abbildung 2.2: Abstrakter Syntaxbaum für den Ausdruck 9-5+2. Entnommen aus (Aho, 2007, S. 70).

2.2.1 Allgemeine Implementierung

Wenn ein Operator, also ein innerer Knoten, beliebig viele Operanden, also Kinder des Knoten, haben darf, spricht man von einem n-stelligen bzw. n-ary AST. (Edwards, 2003) Eine solche Struktur ist auf Grafik 2.3 veranschaulicht.

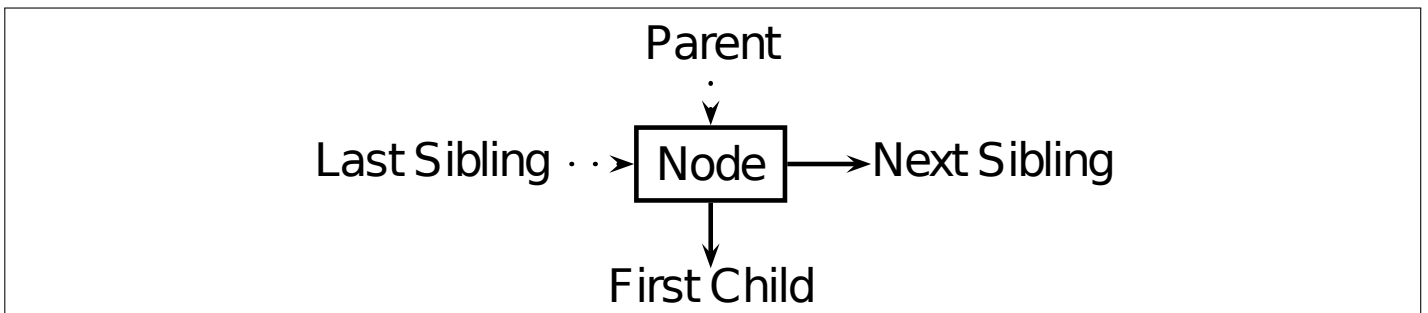


Abbildung 2.3: Generellste Implementierung eines AST. Entnommen aus (Edwards, 2003).

2.2.2 Übertragung auf den Prototypen

Die zentrale Struktur des hier vorgestellten Prototyps ist ebenfalls ein abstrakter Syntaxbaum. Das ist legitim, da Syntaxbäume beliebige Konstrukte repräsentieren können. Jedoch handelt es sich nicht ausschließlich um eine Datenstruktur, da Aktoren als die Baum-Knoten fungieren und Aktoren neben der reinen Datenhaltung auch auf Nachrichten reagieren können. Man könnte quasi davon sprechen, dass der abstrakte Syntaxbaum in dem hier vorgestellten System eine „lebendige Datenstruktur“ ist. Das kann dahingehend von

Vorteil sein, da der AST dadurch befähigt ist, selbstständig eine Analysephase durchzuführen.

In einer solche Analysephase werden u.U. weitere Informationen zu den Knoten hinzugefügt. Beispiel: Eine Hierarchie von Kapiteln kann während der Analyse die jeweils richtige Benummerung ermitteln und jeder Aktor (also Knoten) speichert sich diese Benummerung intern als Attribut ab.

Der Basis Aktor aus Abbildung 2.7 entspricht quasi 1:1 der generellen Implementierung aus Abbildung 2.3. Das hier vorgestellte Metamodell (s. Abschnitt 2.4.1) beschreibt auch eindeutig einen AST. Die Knoten entsprechen den Aktoren, welche wiederum die einzelnen Dokumentelemente des Dokuments repräsentieren. Dieser AST, gesehen als Programmsyntax, beschreibt somit den hierarchischen Aufbau des Dokuments.

Die inneren Knoten, die z.B. Operatoren repräsentieren können, entsprechen hier hierarchiebildenden oder gliedernden Elementen. Diese können (müssen aber nicht) im Dokument sichtbar sein. Beispiele dafür sind z.B. Kapitel, Abschnitte oder die Titelei. Die Kinder dieser Knoten sind bedeutungsvolle Komponenten für den Knoten dahingehend, dass sie entweder weiter die Hierarchie des Dokuments aufspannen und somit das Dokument weiter gliedern oder im Falle von Blättern, die eigentlich inhaltstragenden Elemente sind. Die Blätter müssen auf jeden Fall im Dokument sichtbar sein. Beispiele dafür sind z.B. Absätze oder Abbildungen.

2.3 Projektionseditoren

Für gewöhnlich arbeiten Programmierer mit Quellcode-Editoren, das heißt es werden Schriftzeichen direkt in eine Datei geschrieben. Diese Schriftzeichen bilden die konkrete Syntax einer Programmiersprache. Der Editor kann das arbeiten mit Quellcode erleichtern, indem er z.B. die Schlüsselwörter einer Programmiersprache farbig hervorhebt. Um ein Programm aus der Datei zu generieren, muss ein Übersetzer zunächst eine Syntaxanalyse (parsing) durchführen. Bei dieser Analyse wird die Datei in Wörter, die es in der Programmiersprache gibt, zerhackt. Anhand dieser Wörter kann der Übersetzer

z.B. eine Baumstruktur aufbauen, um die syntaktische Korrektheit des Programms zu prüfen. Das heißt der Übersetzer kann anhand es Baumes feststellen, ob es sich um ein korrektes Programm handelt. Dann kann aus der, als korrekt befundenen, Baumstruktur der eigentliche Maschinencode erzeugt werden.

Projektionseditoren gehen anders vor: Der Editor arbeitet direkt auf dem abstrakten Syntaxbaum (s. Abschnitt 2.2), d.h. der Programmierer editiert nicht mehr eine Datei die durch den Übersetzer in eine Baumstruktur umgewandelt wird, sondern seine editier-Operationen verändern direkt die Baumstruktur des Programms. (Voelter, 2013, S. 68) Nun kann aus der abstrakten Syntax (entspricht der Baumstruktur) eine Variation an konkreten Syntaxen erstellt werden, die sogenannten Projektionen. Diese Projektionen können wieder textuell sein, aber auch grafisch – jedoch sind diese in ihrer Darstellung deutlich flexibler als bei der parserbasierten Editoren. Eine kurze schematische Grafik (Abb. 2.4) verdeutlicht den Unterschied zwischen Quellcode-Editoren und Projektionseditoren.

Im englischen¹ Sprachraum gibt es mehrere Bezeichnungen: structure editor, structured editor oder projectional editor. Man kann diese mit Struktureditor oder Projektionseditor übersetzen.

Die Idee von Projektionseditoren ist nicht neu, bereits 1971 hat Wilfred J. Hansen ein System entwickelt, welches auf solch einem Ansatz beruht. Jedoch haben sich zur damaligen Zeit diese Editor-Formen nicht durchgesetzt, wegen der unzureichenden Benutzerfreundlichkeit. (Gomolka u. Humm, 2013, S. 91) In heutigen Zeiten (insbesondere mit den fortgeschrittenen Web Standards) sollte es aber möglich sein, einen Benutzerfreundlichen Projektionseditor zu bauen, der die Produktivität und den Komfort erhöht.

2.3.1 Übertragung auf den Prototypen

Der hier vorgestellte Prototyp ist augenscheinlich ein Projektionseditor. Sein Basiskonzept ist Dokumente als ein Modell zu sehen. Und das Modell eines

¹ vgl. Wikipedia: http://en.wikipedia.org/wiki/Projectional_editor

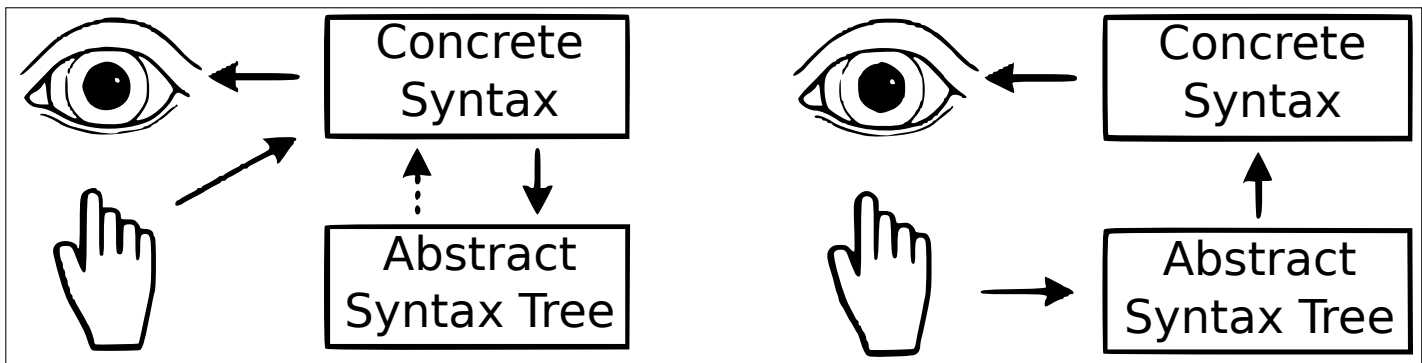


Abbildung 2.4: Parserbasierte Editierung / Quellcode-Editor (links) verglichen mit projektionsbasierter Editierung / Projektionseditor (rechts). Grafiken aus (Voelter, 2013, S. 68) entnommen.

Dokuments erscheint hier in Form eines abstrakten Syntaxbaumes. (s. Kapitel 2.1.2) Der Editor arbeitet also „per Design“ als Projektionseditor. Die Projektionen entsprechen einfachen HTML-Templates, die jedem Dokumentelement sein Aussehen verleihen. Werden die Templates ausgetauscht, können ganz andere Repräsentationen bzw. konkrete Syntaxen des vorliegenden abstrakten Syntaxbaumes möglich sein. Beispielsweise kann der Prototyp aktuell neben der „gesetzten“ Web-Ansicht auch noch LaTeX-Code als Projektion anbieten – das gleiche Dokument in verschiedenen Ansichten oder Projektionen.

2.4 Metamodellbegriff

Werden Modelle und Modellbildung selbst zum Gegenstand der Modellierung, so spricht man von Metamodellen. (Strahinger, 1998, S. 1)

(Strahinger, 1998) hat versucht den Metamodellbegriff anhand der Sprachstufentheorie der Logik, durch Übertragung auf die Modellierungswelt, zu prägen. (Strahinger, 1998, S. 1) erklärt, dass nach (Bühler, 1934) die Sprache drei Funktionen leistet: (1) Darstellung von Sachverhalten, (2) Appell zur Verhaltenssteuerung und (3) Ausdruck von Gefühlen. Für den Metamodellbegriff ist jedoch nur die Darstellungsfunktion interessant. Sprache stellt „ein mögliches Instrument zur Darstellung von Modellen“ (ebd.) dar.

(Strahringer, 1998, S. 1) führt fort, dass in der Logik üblicherweise zwischen Objektsprache und Metasprache unterschieden wird. Die Objektsprache ist Gegenstand der Untersuchung. In der Metasprache erfolgt die Untersuchung. Da die Metasprache selbst auch wieder Gegenstand einer Untersuchung werden kann, ist dieses Prinzip rekursiv anwendbar. Um endlose Rekursion zu vermeiden, sollte als oberstes Glied der Kette ein selbstbeschreibendes Meta-modell stehen. Beispielsweise die Meta Object Facility (MOF) der OMG² geht so vor, um endlose Meta-Rekursionen zu vermeiden.

Wird „die Sprachstufentheorie auf die Modellbildung [...] übertragen, so“ (ebd.) kann im einfachsten Fall ein Metamodell „als ein Modell eines Modells“ (ebd.) beschrieben werden.

Wird die Objektsprache, in der das Modell der untersten Stufe formuliert ist, abgebildet in einem Beschreibungsmodell, so handelt es sich um ein Metamodell. (Strahringer, 1998, S. 3)

„Beschreibungsmodelle dienen der systematischen Beschreibung des betrachteten Gegenstandsbereiches und bestehen aus ausschließlich deskriptiven Satzsystemen.“ Das heißt, ein Beschreibungsmodell ist ein Modell welches so formuliert wird, dass es ein Original anschaulich macht bzw. Eigenschaften des Originals sprachlich spezifiziert. Abbildung 2.5 veranschaulicht den Zusammenhang von Modell und dessen Metamodell.

2.4.1 Metamodell des Dokumentmodells

Damit das Modell formal und damit nicht verschwommen spezifiziert ist, brauchen wir ein Metamodell. Das Metamodell schafft somit auf einer abstrakteren Ebene Klarheit über das Modell.

Auf Abbildung 2.6 ist das eigentliche Modell visualisiert. Dieses soll vom Metamodell beschrieben werden.

² Definiert in ISO/IEC 19508.

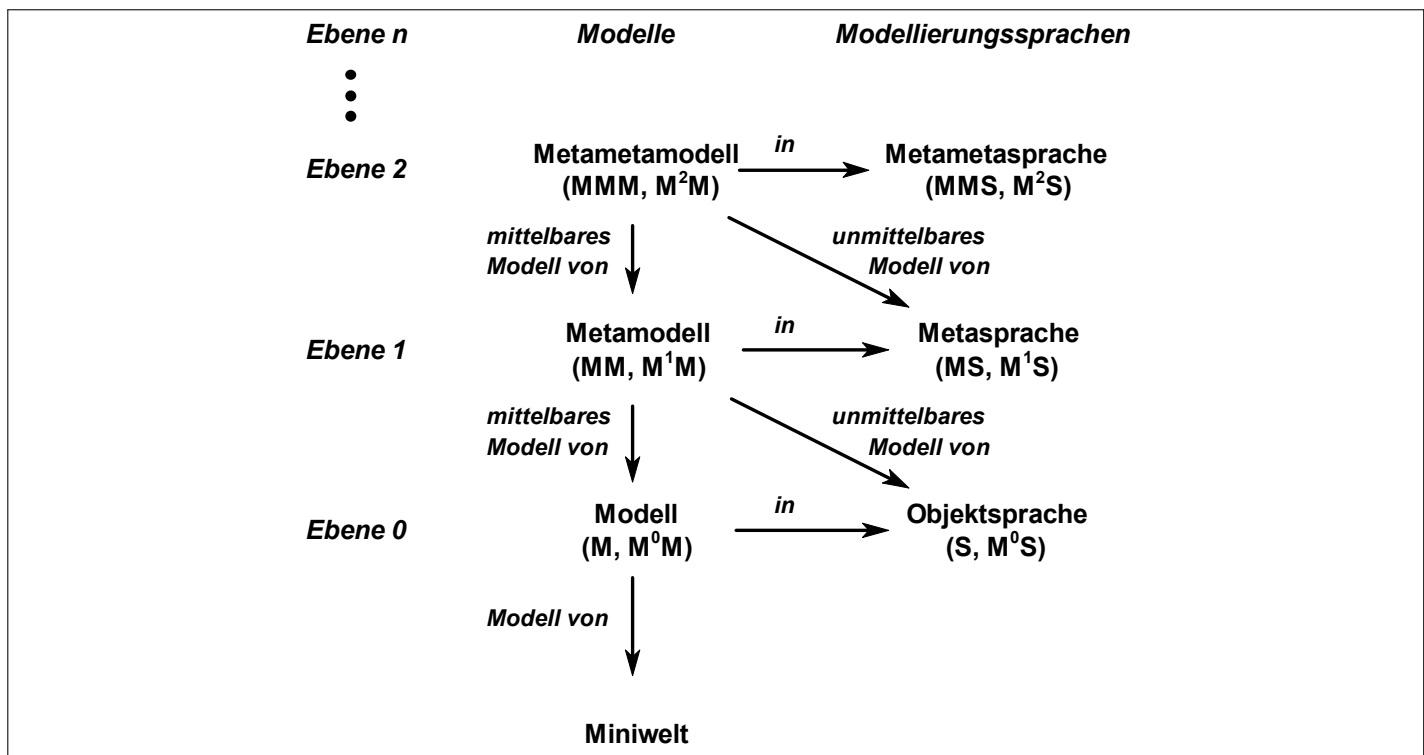


Abbildung 2.5: Der sprachbasierte Metamodellbegriff. Die „Miniwelt“ entspricht dem Original. Grafik entnommen aus (Strahringer, 1998, S. 3).

Abbildung 2.7 zeigt das Modell welches das Dokumentmodell beschreibt, also das „Metamodell“. In der Mitte liegt der Basis Aktor, dieser hält Referenzen zu seinem ersten Kind und zu seinem unmittelbaren Geschwister. Zudem hält der Basis Aktor Instanzen aller verfügbaren Dokumentelemente, wovon jedoch immer nur eine aktiv ist. Welche Dokumentelemente verfügbar sind, wird vom Programmierer spezifiziert – dazu muss er ein Scala Trait (entspricht einem Interface) implementieren und dem Basis Aktor bekannt machen. Man kann sagen, dass er das Metamodell an dieser Stelle (gewollt) verändert. Der Wurzel Aktor hält nochmals alle Topologieinformationen, daher kennt er alle im System vorhandenen Basis Aktoren. Jeder Basis Aktor kennt auch seine zugehörige Wurzel, welche er bei ggf. anstehenden lokalen Topologieveränderungen benachrichtigen muss.

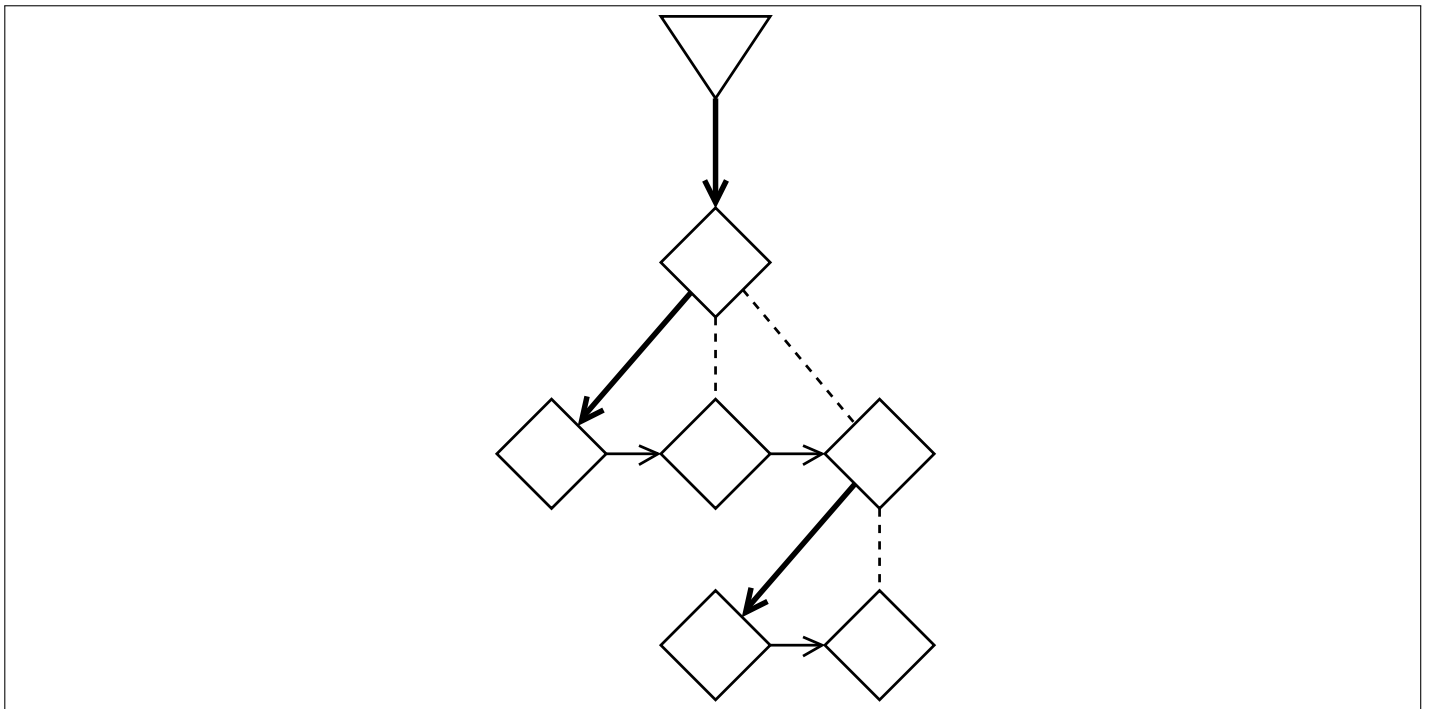


Abbildung 2.6: Das hier verwendete Dokumentmodell. Das Dreieck symbolisiert die Wurzel. Die Raute symbolisieren jeweils einen Actor bzw. ein Dokumentelement. Die stark gezeichneten Pfeile symbolisieren das erste Kind, die gestrichelte Linie symbolisiert die zugehörigen anderen Kinder. Die schwach gezeichneten Pfeile symbolisieren die nächstes Geschwister. Dadurch wird die Dokumenthierarchie aufgespannt.

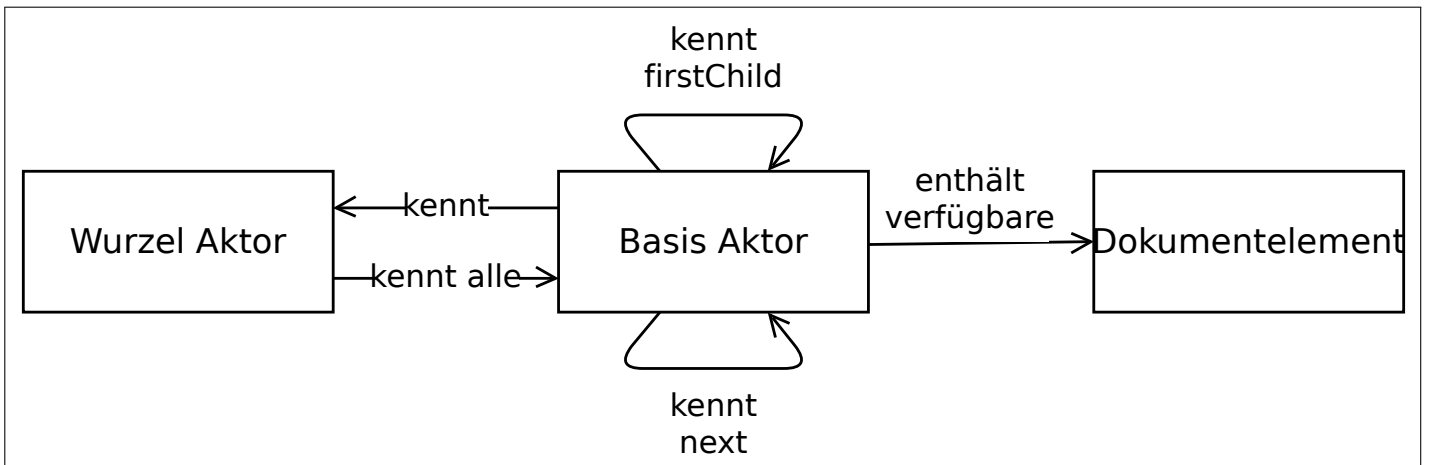


Abbildung 2.7: Das Metamodell des Dokumentmodells.

Beweis

Die Besonderheit bei dem hier entstandenen System ist, dass das Modell gleichzeitig die Sprache ist, in der es modelliert wird. Dies ist möglich, da das

System als Projektionseditor designed ist. Die Objektsprache in der das Modell formuliert ist, entspricht somit quasi der Projektion die direkt aus dem abstrakten Syntaxbaum entspringt. Man könnte von einem „Modell-Objektsprache-Dualismus“ sprechen – es ist gleichzeitig Modell und Objektsprache, je nach Betrachtungspunkt.

Das Modell indem wiederum die Objektsprache modelliert ist, muss ein Beschreibungsmodell sein, damit dies einem Metamodell entspricht. Das hier vorgestellte Metamodell spezifiziert die Eigenschaften des Originals (dies geschieht in den einzelnen Dokumentelementen) und die prinzipielle Struktur (anschaulich gemacht durch Wurzel/Basis Aktor) des abstrakten Syntaxbaumes. Der betrachtete Gegenstandsbereich ist der abstrakte Syntaxbaum, dieser in seiner Gesamtheit ist wiederum das Modell des Dokuments. Auf Abbildung 2.8 ist nochmals eine Übersichtsgrafik der Zusammenhänge von Modell und Metamodell. Somit genügt es dem Metamodellbegriff. ▪

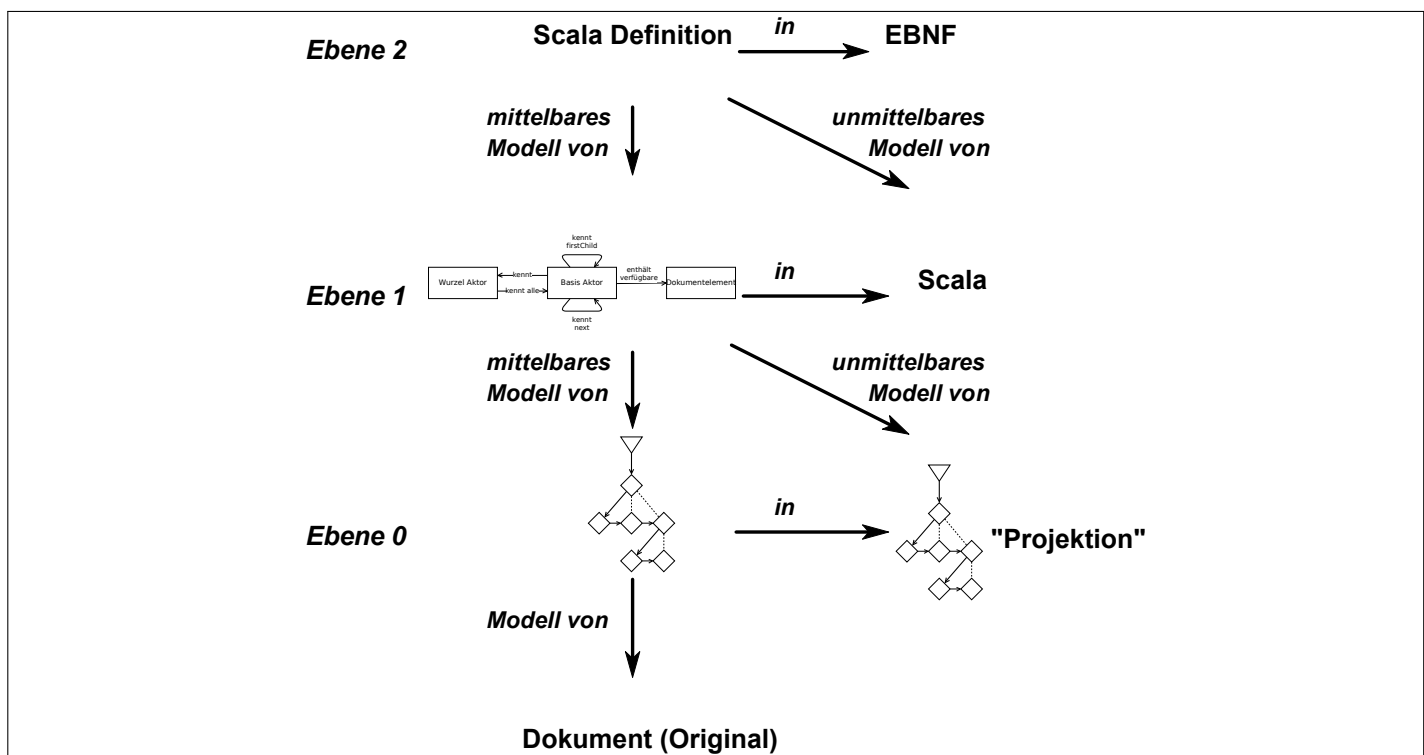


Abbildung 2.8: Übersicht der Zusammenhänge des Dokumentmodells und Dokumentmetamodells. Grafik nach (Strahringer, 1998). Die erweiterte Backus-Naur-Form (EBNF) kann als Metamodell dienen – denn mit EBNF können Programmiersprachen beschrieben werden und zudem ist EBNF in der Lage sich selbst zu beschreiben.

2.5 Semiotik

Semeiotik [sic] ist eigentlich nichts anderes als die allgemeine Lehre von den Sprachen. Ob diese nun künstliche oder natürliche Sprachen sind, spielt keine Rolle. (Malissa, 1971, S. 8)

Die Gesetzmäßigkeiten einer Sprache kann allgemein in vier Hauptteile gegliedert werden, vgl. (Malissa, 1971, S. 8):

- (i) Syntax (Struktur): Beschreibt die Beziehung zwischen Zeichen, Signalen, Symbolen zueinander. Hier kann man sich die Frage stellen: Was ist ein korrekter Aufbau eines Symbols? Zum Beispiel wenn man einen Satz einer natürlichen Sprache als Symbol ansieht, wird ein Satz nur durch eine bestimmten Reihung von Wörtern korrekt.
- (ii) Semantik (Bedeutung): „ist die Lehre von den Zeichen, Signalen, Symbolen und deren Bedeutung.“ (ebd.) Sie stellt also die Beziehung zwischen Symbol und des bezeichneten Objekts her. Hier kann man sich die Frage stellen: Was bedeutet das Symbol? Was ist die Bedeutung?
- (iii) Pragmatik (Funktion): Stellt die Beziehung zwischen Zeichen, Signalen, Symbolen und dem Sender bzw. Empfänger dar. Hier kann man sich die Frage stellen: Was ist der Zweck des Symbols? An wen ist es wozu gerichtet?
- (iv) Sigmatik (Bezeichnung): „stellt die Beziehung zwischen den Zeichen, Symbolen, Signalen usw. und dem, was sie bezeichnen her.“ (ebd.) Hier kann man sich die Frage stellen: Was bezeichnet das Symbol? Was ist das Bezeichnete?

Wenn ein Zeichen, Signal, Symbol, etc. alle der oben genannten vier Aspekte aufzeigt, dann spricht man von einer Information. Im Allgemeinen sind diese vier Aspekte also die Grundlage jeder Information. In Abb. 2.9 sind die Aspekte anhand von Beispielen aus der analytischen Chemie visualisiert.

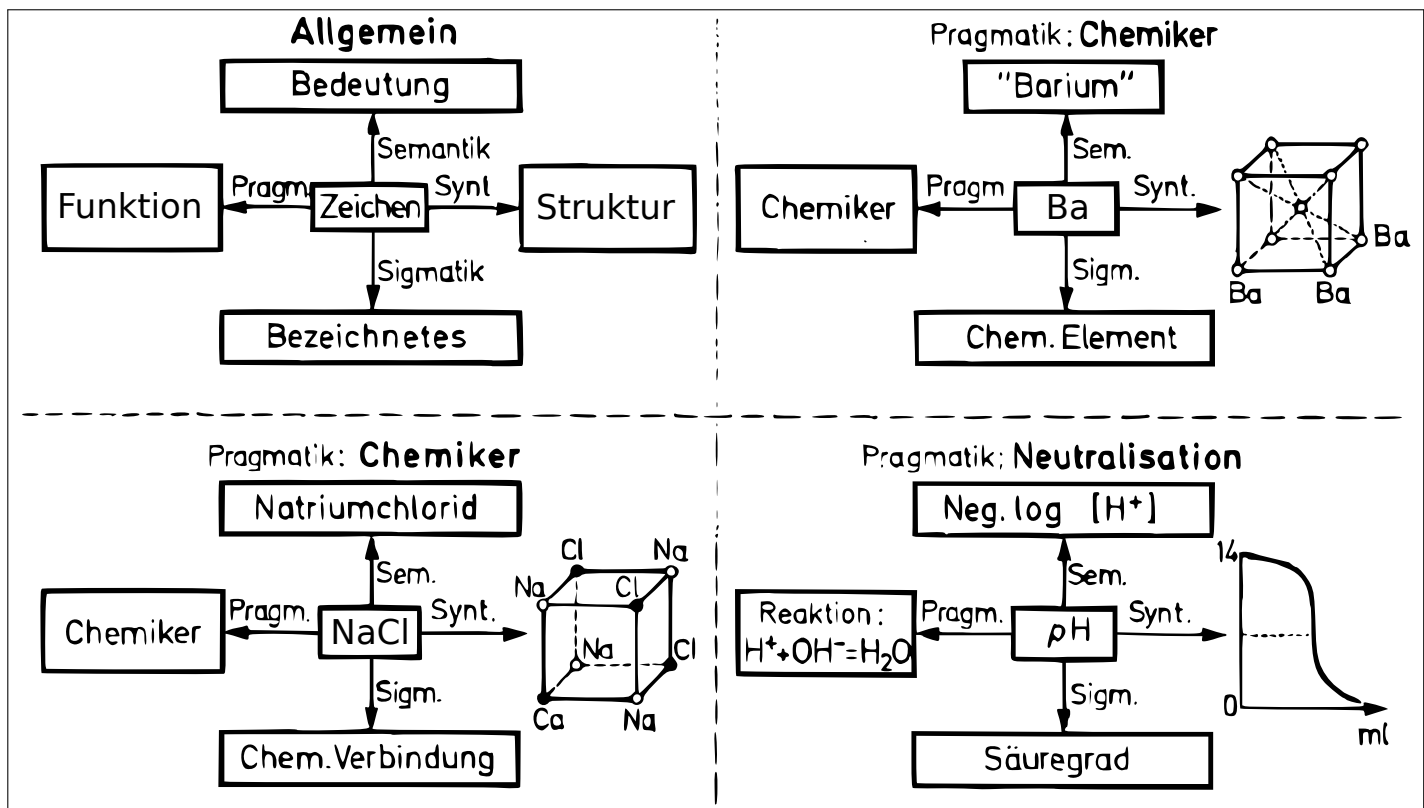


Abbildung 2.9: Die vier Hauptteile der Semiotik grafisch dargestellt mit Beispielen aus der analytischen Chemie. Visualisierung mit einigen Anpassungen aus (Malissa, 1971, S. 9).

2.5.1 Anwendung auf (wiss.) Dokumente

Diese vier Aspekte kommen auch implizit in jedem Text zum tragen, da insb. wiss. Dokumente ein klassischer Informationsträger sind. In der Wissenschaft ist es üblich mit Modellen zu arbeiten. Diese Modelle sollen von anderen Wissenschaftlern möglichst leicht verstanden und benutzt werden. Für das Verstehen eines Modells ist es unerlässlich, die Bedeutungen jedes einzelnen Aspekts des Modells zu erfassen. Damit ein Leser einfach und gezielt Wissen aus dem Dokument ziehen kann, wäre es sicher nützlich, wenn diese Semantik möglichst explizit zur Verfügung stünde. Die Mechanik dazu könnte folgendermaßen aussehen: „Man fährt mit dem Mauszeiger über ein interessantes Objekt und erhält sofort weitere Informationen dazu, um mehr über die Bedeutung des Objektes zu erfahren.“

Wenn man die vier Konzepte allgemein auf den hier vorgestellten Prototypen überträgt, stellt man fest wie sich die Konzepte innerhalb eines Dokuments auswirken. Betrachtet wird ein einzelnes Dokumentelement:

- Zeichen: Entspricht einer Projektion des abstrakten Syntaxbaumes (z.B. ein Abschnitt-Dokumentelement gesetzt als „Kapitel 1. Einleitung“)
- Bezeichnetes: Ist ein Dokumentelement.
- Bedeutung: Erster gliedernder Abschnitt des Dokuments.
- Funktion: Der Autor fungiert während des Schreibens als Sender und Empfänger.
- Struktur: Entspricht dem abstrakten Syntaxbaum selbst bzw. dessen Metamodell.

Man kann argumentieren, dass hinter jedem Dokumentelement (z.B. Abschnitt, Absatz, Tabelle, ...) ein entsprechendes Domänenmodell steckt. Beispielsweise hinter einem Chemie-Dokumentelement stecke ein Moleküleditor, der ein entsprechendes Modell eines Moleküls produziert, sowie eine grafische Repräsentation (Bild eines Moleküls). Idealerweise liegt das Domänenmodell als domänenspezifische Sprache (DSL) vor. Im Bezug auf das Chemie-Dokumentelement könnte das die IUPAC-, Molfile-³ oder SMILES-Molekülnotation sein. Das würde bedeuten, dass der hier vorgestellte Syntaxbegriff auch gleichzeitig das Domänenmodell vertritt.

Das heißt, dass bei der Erstellung eines Dokuments Dokumentelemente vom Autor entworfen werden können, welche Modelle der jeweiligen Wissenschaft enthalten können. Das ermöglicht dem Autor deutlich strukturierter, formaler und konsistenter an die Dokumentation seines Projektes heranzugehen. Zudem werden dem Leser dadurch explizite semantische Informationen bereitgestellt, welche direkt vom dahinterstehenden Modell stammen. Der Leser kann ggf. das Modell seines Kollegen in eigenen Dokumenten wiederverwenden.

³ Das Molfile der Firma MDL repräsentiert chemische Strukturen. Siehe de.wikipedia.org/wiki/Chemoinformatik „Repräsentation chemischer Strukturen“.

Vorteil hierbei wäre, dass weniger Fehler entstehen, da direkt auf die Gedanken (welche in das Modell gegossen wurden) des Original-Autors zurückgegriffen werden kann.

Üblicherweise gibt es unter Dokumentelementen Verweise. Zum Beispiel ein Absatz-Dokumentelement möchte auf die Molekülmasse des Chemie-Dokumentelements verweisen. Dann sieht die Semiotik folgendermaßen aus:

- Zeichen: „Die Molekülmasse ist 144 g/mol.“ (So angezeigt durch den Absatz)
- Bezeichnetes: Chemie(molekül)-Dokumentelement. (Welches das Modell des Moleküls enthält)
- Bedeutung: Masse von Koffein in Gramm pro Mol. (Interessantes Objekt wäre „144 g/mol“)
- Funktion: Verweis
- Struktur: „koffein.masse“ (hieraus folgt „144 g/mol“)

Mechanik: Der Programmierausdruck „koffein.masse“ (koffein wäre der Name des Chemie-Dokumentelements und masse eine bereitgestellte Methode des Chemie-Dokumentelement-Objektes) liefert ein semantisches Objekt, z.B. in Form eines Scala-Objekts des Typs Moleculemass. Moleculemass könnte mit den Properties mass, name, unit ausgestattet sein, um mehr Informationen über seine Bedeutung(en) bereitstellen zu können. Zudem könnte es noch an eine SI-Einheiten-Onologie angekoppelt sein, um die semantischen Informationen mit dem Semantic Web zu teilen. Wenn der Leser mit seinem Mauszeiger über „144 g/mol“ fährt, kann diese Information aus dem Moleculemass-Objekt geholt werden und in Form einer farbigen Annotation angezeigt werden.

2.6 Taxonomie wissenschaftlicher Publikationen

Über die Jahrhunderte in denen wissenschaftliche Texte verfasst wurden, haben sich gewisse Konventionen und Vorgehensmodelle, wie ein Dokument von seinem inneren Aufbau her beschaffen sein soll, herauskristallisiert. Diese Konventionen haben sich sogar über nationale Grenzen hinweg sehr ähnlich entwickelt, was es der Wissenschaftsgemeinde leichter macht, ihr intellektuelles Vermächtnis zu teilen.

In Recherchen ist aufgefallen, dass es schon (wenn auch wenige) Versuche gegeben hat, eine möglichst allgemeingültige Systematik (z.T. formal) des inneren Dokumentenaufbaus herauszudestillieren. Jedoch sind diese Konzepte bisher kaum verbreitet, insbesondere bei der direkten Erstellung von (wiss.) Texten durch den Autor. Heutigen Autorensysteme unterstützen kaum explizite Semantik über das Domänenwissen mit dem sie gerade arbeiten – dies fördert Fehler und Inkonsistenzen im Dokument.

Die DIN-Normen (DIN, 1983a), (DIN, 1983b), (DIN, 1984) und (DIN, 1986) geben Empfehlungen, wie Texte gegliedert und benummert werden sollen bzw. wie im allgemeinen Manuskripte oder Forschungsberichte gestaltet werden sollen. Dies sind keine formalen Vorgaben, sondern vielmehr eine Zusammenfassung der hiesigen guten fachlichen Praxis in Wissenschaft, Technik, Wirtschaft und Verwaltung. Anders hingegen geht (ANSI/NISO, 2012) vor, hier wird formal mittels XML Schemas eine allgemein gültige Menge an XML-Tags für den Aufbau von Fachzeitschriftenartikeln zusammengestellt. Jedoch haben diese auch für anderen Publikationen ihre Gültigkeit, wie z.B. Briefe, Leitartikel oder Buchkritiken. (Shotton u. Peroni, 2014) stellt eine Ontologie (in OWL 2) über einzelnen Komponenten, die in einem Dokument vorkommen können, zur Verfügung. Hier werden die Beziehungen die die Komponenten untereinander pflegen sichtbar gemacht. Es wird dabei zwischen strukturierenden (z.B. Abschnitt, Absatz) und rhetorischen Komponenten (z.B. Einleitung, Abbildung, Anhang) unterschieden.

Hier soll versucht werden die Erkenntnisse der o.g. Quellen zu generalisieren, in der Hoffnung die Basis für ein formaleres bzw. präskriptiveres Autorensystem zu schaffen.

2.6.1 Dokumentelemente

Dokumentelemente bilden das Grundgerüst eines jeden Dokuments. Hier wird versucht den Begriff genauer zu fassen und zu definieren. Es ergeben sich drei abstrakte Ausprägungen der Dokumentelemente:

- **Struktur Elemente:** Repräsentieren Komponenten, die textuellen bzw. grafischen Inhalt des Dokuments beinhalten. Sie tragen maßgeblich zum gedanklichen Gebäude des Dokuments bei.
- **Container Elemente:** Beinhalten andere Dokumentelemente (als Kinder), und haben daher für gewöhnlich keine Repräsentation die aktiv zum eigentlichen intellektuellen Inhalt des Dokuments beiträgt. Sie gruppieren also Elemente zu größeren Dokument-Bestandteilen; dies gibt Vorteile bei der Verarbeitung und Semantik des Dokuments. Beispielsweise werden Dokumente oft in Titelei und Hauptteil unterteilt, wobei die Titelei gerne römisch nummeriert wird und der Hauptteil arabisch. Hat man hier eine explizite Hierarchievorgabe zur Hand, erleichtert das die Verarbeitung; wie im Beispiel die gewollten Nummerierungskonventionen durchzusetzen.
- **Metadaten Elemente:** Sind Elemente die andere Elemente beschreiben bzw. allgemein gesprochen enthalten sie Daten über Daten. Beispielsweise eine Literatureintrag beschreibt lediglich einen Original-Artikel; oder eine Fußnote enthält eine Anmerkung über ein anderes Dokumentelement (die Fußnote verändert den Inhalt den das Dokument transportieren will an sich nicht, aber sie macht ihn verständlicher).

Die Dokumentelemente können zudem auch noch Attribute oder Properties enthalten. Diese enthalten Fakten über das betreffende Element, z.B. ob es sich bei einer Liste um eine geordnete Liste oder ungeordnete Liste handelt. Im Falle des hier vorgestellten Prototypen, kann man argumentieren, dass in den Attributen auch das jeweils vorliegende Domänenmodell des Dokumentelements abgelegt wird.

In DoCO wird ein Unterschied zwischen strukturierenden (z.B. Abschnitt, Absatz) und rhetorischen Komponenten (z.B. Einleitung, Diskussion, Abbildung) unterschieden. Hier wird jedoch nur das Struktur Element benötigt, da die rhetorische Bedeutung über die Attribute des Elements hinzugefügt werden kann. Allgemein gesprochen gehört die rhetorische Bedeutung zur Semantik des Elements. Die Semantik des Elements wird wiederum maßgeblich über das Domänenmodell gewonnen und dieses ist wiederum in den Attributen anässig. Daher muss kein Unterschied zwischen strukturierenden und rhetorischen Komponenten gemacht werden.

Bedeutung für das Dokumentmodell

Im Dokumentmodell, also dem abstrakten Syntaxbaum entsprechen die Struktur Elemente den Blättern und die Container Elemente entsprechen den inneren Knoten. Metadaten Elemente kommen entweder wie Struktur Elemente vor oder werden in ein extra „Metadokument“ ausgelagert, falls die Metadaten Elemente nur einzelne Attribute bereitstellen sollen und nicht in Gänze angezeigt werden.

Verweisungen

(Shotton u. Peroni, 2014) und (ANSI/NISO, 2012) modellieren mehr Dokumentelemente als in der vorgestellten Taxonomie (s. Abschnitt 2.6.2). Das liegt zum einen daran, dass hier viele dieser „Elemente“ in dem hier vorgestellten Modell als Attribute abgelegt werden. Hier treten nur die Komponenten explizit als Dokumentelement auf, bei denen es Sinn macht auch als ein Knoten des abstrakten Syntaxbaumes abgebildet zu werden. Zum anderen werden in den Modellen von (ebd.) Verweisungen, die sich innerhalb des Dokuments abspielen (z.B. auf Kapitel oder Abbildungen), auch explizit als Dokumentelement modelliert. Dies macht aber in dem hier vorgestellten Dokumentmodell wenig Sinn.

Denn hier treten Verweisungen nicht in Form von Knoten auf, sondern sie entsprechen den Kanten des Dokumentengraphs. Wenn ein Knoten (also Do-

kumentelement) eine Information von einem anderen Knoten erhalten will, werden Nachrichten mit den Informationen ausgetauscht. Durch diesen Mechanismus werden die Verweisungen im Dokument schlussendlich aufgelöst. Wir haben also ein ganz klares Modell: Inhalte in die Blätter, Hierarchie in die inneren Knoten, Semantik und Modelle in die Attribute und Verweisungen fließen über die Kanten, also in der Implementierung des Prototypen als Nachrichten-Kommunikation.

2.6.2 Taxonomie

In der Tabelle auf Abb. 2.10 sind nochmals alle drei Ausprägungen der Dokumentelemente (s. Abschnitt 2.6.1) übersichtlich dargestellt. Sie sind von abstrakt zu konkreter werdend geordnet. In der vorletzten Spalte sind die eigentlichen Kernelemente aufgezeigt, welche auch so im Dokument vorkommen können. In der letzten Spalte sind mögliche Attribute, Properties bzw. Methoden aufgelistet, die dem Element zusätzliche (semantische) Informationen hinzufügen.

Mit den in der Tabelle (Abb. 2.10) aufgelisteten Dokumentelementen kann man ein typisches, aber sehr allgemein gehaltenes, wissenschaftliches Dokument aufbauen. Die Gesamtheit der hier vorliegenden Dokumentelemente beschreibt somit eine ganz bestimmte Dokumentklasse, welche als allgemeiner „wissenschaftlicher Bericht“ oder „Report“ bezeichnet werden kann. Aber es gibt auch noch beliebige andere Klassen (z.B. Patente, Normen, Software-dokumentation, etc.), die möglicherweise einen (leicht) anderen Aufbau haben und somit ggf. andere Dokumentelemente benötigen. Neu definierte Dokumentelemente (z.B. für eine andere Dokumentklasse) sollten sich leicht in diese Taxonomie eingliedern lassen. Damit liegen sie nicht mehr als implizite Aufbauvorschrift (gemeint ist so etwas wie ein Corporate Design) vor, sondern sind explizit, für die Nutzung in einem Autorensystem welches die Taxonomie umsetzt, formalisiert.

abstract to concrete elements		attributes	
Container	Start Point	Root	
	Component	Front Matter	
		Body Matter	
		Back Matter	
	Outline	Part	title, numbering
		Chapter	title, numbering
		Section	title, numbering
		Subsection	title, numbering
		Subsubsection	title, numbering
Structure	Text	Paragraph	text
		List	texts
		Quotation	text, referenceitem
	Captioned Box	Figure	title, numbering, caption
		Table	title, numbering, caption
		Formula	title, numbering, caption
		Code	title, numbering, caption
Meta	Bibliographic (Structure)	Document Title	title, subtitle
		Autor	name
		Abstract	text
	Indexing (Container)	Table of Contents	outlines
		Bibliography	referenceitems
		List of Figures	figures
		List of Tables	tables
	Annotation (Structure)	Emphasize	text
		Footnote	text
		Marginal Note	text
	Other	Reference Item	author, year, title, ...

Abbildung 2.10: Taxonomie des inneren Aufbaus von wissenschaftlichen Dokumenten. Sortiert von abstrakt zu konkret.

2.6.3 Dokumentstruktur Matrix

Damit das Autorensystem beim Erstellen eines formal korrekten Dokuments helfen kann, wird ein Regelwerk benötigt welches vorschreibt „welches Element nach wem erlaubt ist“ (erlaube Geschwister-Beziehungen) und „wo darf welches Element vorkommen“ (erlaube Eltern-Kind-Beziehungen).

Durch diese Formalisierung kann der Autor wie auf Schienen durch den Aufbau geführt werden. Er muss sich also nicht mehr selbst darum kümmern die impliziten Vorgaben der Dokumentklasse durchzusetzen. Das heißt er macht weniger Fehler beim Aufbau des Dokuments, da das System assistierend zur Seite steht, um die gewählte Dokumentklasse ihrem Regelwerk konform zu

halten und er sich stärker auf die Erstellung des Inhalt konzentrieren kann. Zudem liefert jedes Dokumentelement ein Set an Attributen mit, was es ermöglicht mehr explizite Semantik in das Dokument zu bringen und das Dokument insgesamt konsistenter zu halten.

Erlaube Verschachtelungen

Auf Abbildung 2.11 sind die erlaubten Eltern-Kind-Beziehungen der vorliegenden Dokumentklasse definiert. Die hier dargestellte Matrix ist eine Vereinfachung, also nur eine Auswahl an Dokumentelementen, der Taxonomie aus Tabelle auf Abb. 2.10. Diese Matrix definiert die erlaubte Schachtelung der vorliegenden Dokumentklasse.

Erlaube Abfolgen

Auf Abbildung 2.12 sind die erlaubten Nachfolger-Vorgänger-Beziehungen der hier beispielhaft vorliegenden Dokumentklasse als Matrix dargestellt. Hier wird also die Frage geklärt, welches Dokumentelemente aneinander gereiht werden dürfen, um ein wohlgeformtes Dokument der vorliegenden Dokumentklasse zu bilden.

2.6.4 Deutung

Wenn man also ein Set an Dokumentelementen definiert und in die Taxonomie eingegliedert hat, zudem noch die erlaubten Beziehungen zwischen den Dokumentelementen in Matrixform definiert hat, dann erhält man eine präzise definierte Dokumentklasse. Nützlich ist das nicht nur um dem Autor „Schienen“ für die Dokumenterstellung bereitzustellen, sondern auch für Klassifikationsaufgaben. Beispiel: Im Zuge von Information Retrieval hat man verschiedene mit OCR aufbereitete PDFs, welche man gerne einer Dokumentklasse zuordnen möchte, um sie wieder in ein „sauberes“ oder aktuelles Template zu überführen. Wenn nun ein Klassifikator die aufbereiteten PDFs in

darf (als direkte Eltern haben) Kind haben	Root	Front Matter	Body Matter	Back Matter	Section	Subsection	Paragraph	List	Figure	Table	Abstract	Table of Contents	Bibliography	Footnote
Root		1	1	1										
Front Matter											1	1		
Body Matter					1									
Back Matter													1	
Section						1	1	1	1	1				
Subsection							1	1	1	1				
Paragraph														
List														
Figure														
Table														
Abstract														
Table of Contents														
Bibliography														
Footnote														

Abbildung 2.11: Stellt die erlaubten Eltern-Kind-Beziehungen zwischen den Dokumentelementen dar. Hier wird also definiert mit welchen Elementen die Hierarchie des Dokuments aufgespannt werden darf.

ihre Dokumentelemente zerlegt, kann über die Gesamtheit der im PDF gefundenen Dokumentelemente und deren Anordnung bzw. Verschachtelung, dank der aufgestellten Taxonomien bzw. Matrizen, auf die Dokumentklasse (z.B. EU-Patent, US-Patent, DIN-Norm, etc.) zurückgeschlossen werden.

	das Vorgänger haben		das Nachfolger haben													
	Root	Front Matter	Body Matter	Back Matter	Section	Subsection	Paragraph	List	Figure	Table	Abstract	Table of Contents	Bibliography	Footnote		
Root																
Front Matter			1													
Body Matter				1												
Back Matter																
Section					1											
Subsection						1	1	1	1	1						
Paragraph						1	1	1	1	1						
List						1	1	1	1	1						
Figure						1	1	1	1	1						
Table						1	1	1	1	1						
Abstract												1				
Table of Contents											1					
Bibliography																
Footnote																

Abbildung 2.12: Stellt die erlaubten Geschwister-Beziehungen (Nachfolger/Vorgänger) zwischen den Dokumentelementen dar. Hier wird also definiert welche Elementen aneinandergereiht werden dürfen.

Kapitel 3

Anwendungsfälle

3.1 Erstellung wissenschaftlicher Dokumente

In erster Linie soll das System Autoren bei der Erstellung hochwertiger wissenschaftlicher Berichte helfen.

Diese Masterarbeit selbst, wurde mit dem mit dem in ihr beschriebenen Prototypen erstellt. Dies veranschaulicht die Tragfähigkeit der Konzepte, bzw. die Stabilität der momentanen Prototypen-Implementierung.

Damit das Dokument später auch der Hochschule bzw. Fakultät vorgelegt werden kann, muss es auch noch in einem printfähigen Format (insb. mit Paginierung) vorliegen. Da zur Zeit noch keine Paginierung der Web-Ansicht erfolgt, muss auch noch auf ein anderes Werkzeug zurückgegriffen werden, um ein printfähiges Format zu erhalten. Da das System, dank des abstrakten Syntaxbaumes, fähig ist das Dokument in verschiedene Ansichten zu präsentieren, kann auch LaTeX-Code generiert werden. Nähere Erklärungen zu solchen Transformationen in Abschnitt 3.4. Das heißt die Print-Version der Masterarbeit kann zudem noch via LaTeX erzeugt werden. In Zukunft sollte es jedoch kein Problem darstellen eine Print-Paginierung in die Web-Ansicht zu integrieren, dazu kann der Layouter „scaltex.js“, welcher in der Bachelorarbeit (Hodapp u. a., 2013) entstanden ist, eingesetzt werden.

Um einen wissenschaftlichen Text, wie eine Masterarbeit, verfassen zu können, müssen einige (Basis) Dokumentelemente vom Prototyp bereitgestellt werden, zudem müssen Verweise unter den Dokumentelementen möglich sein. Beispielsweise (mit den Implementierungsnamen): Chapter, Section, SubSection, SubSubSection und Figure mit automatischer Benummerung; Paragraph, List, ArabicList, RomanList, Quotation, Footnote; FrontMatter, BodyMatter, BackMatter.

3.1.1 Szenario: Chemie

Für Demonstrationszwecke wurde auch ein Chemie-Dokumentelement erstellt, welches einen Moleküleditor anbietet. Damit wäre der Autor komfortabel in der Lage in seinem Dokument direkt chemische Moleküle zu zeichnen und darauf zu verweisen. Der Moleküleditor produziert Molfiles¹ als Ausgabe- bzw. Austauschformat, dieses Molfile ist also ein (persistentes) Modell für ein Molekül. Das Chemie-Dokumentelement könnte noch an eine Chemie-Bibliothek, die Molfiles verarbeiten kann, angebunden werden. Eine solche Programmbibliothek kann weitere Informationen zu den verwiesenen Molekülen bereitstellen, und damit einen erheblichen Beitrag zur expliziten Semantik und Konsistenz des Dokument beitragen.

Auf Abbildung 3.1 ist ein durch das Chemie-Dokumentelement generiertes Molekül abgebildet. Wenn man darauf klickt, wird der Domäneneditor Ketcher geöffnet (s. Abb. 3.2). Ketcher² ist ein in Javascript geschriebener Moleküleditor für den Webbrowser. Das Chemie-Dokumentelement umhüllt diesen Moleküleditor lediglich und verwendet die Ketcher-API, um ein svg-Bild aus dem Molekül für die reine Ansicht zu generieren.

¹ Das Molfile der Firma MDL repräsentiert chemische Strukturen. Siehe de.wikipedia.org/wiki/Chemoinformatik „Repräsentation chemischer Strukturen“.

² Webseite des Ketcher Projekts: <http://ggasoftware.com/opensource/ketcher>

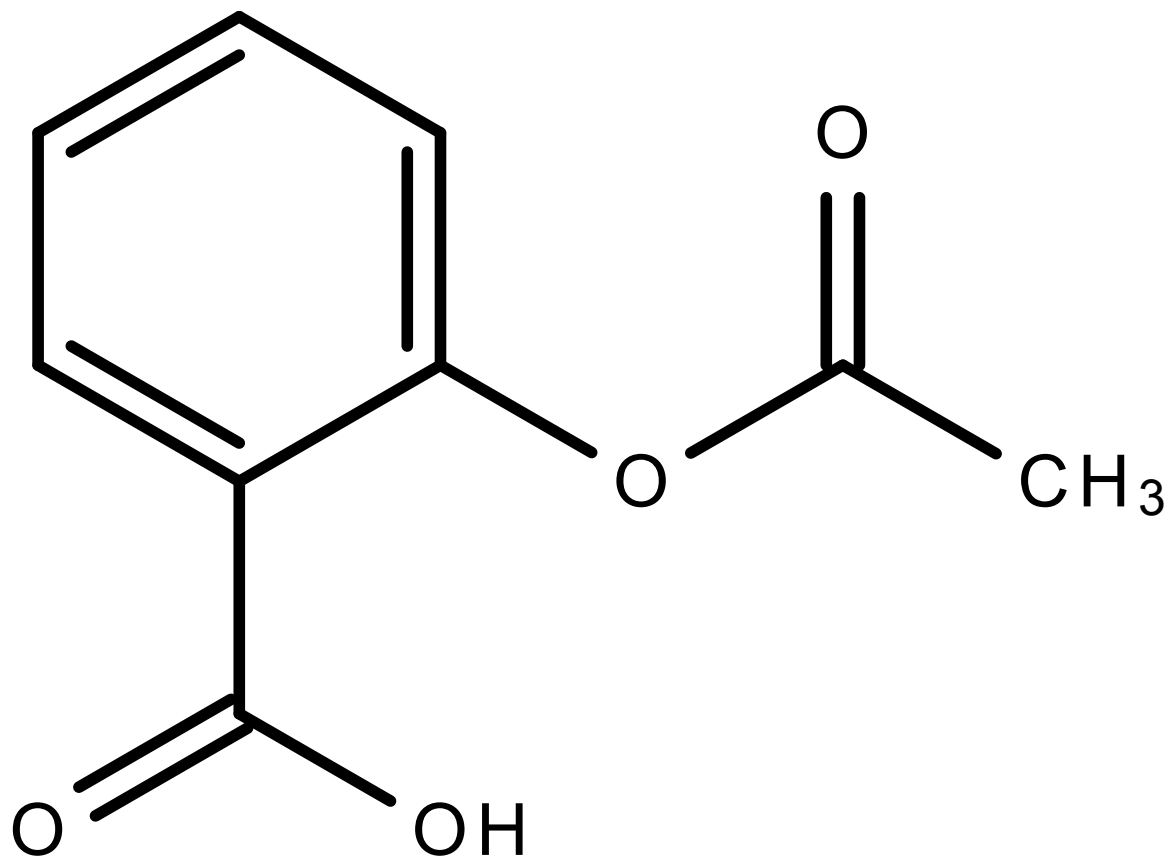


Abbildung 3.1: Generiertes Molekül Aspirin.

3.2 UIMA CAS Editor

Apache UIMA³ ist ein Softwaresystem mit dessen Hilfe neues, für den Benutzer relevantes, Wissen aus großen unstrukturierten Datenmengen gewonnen werden kann.

Ein einfacher Text kann um Anmerkungen, den Annotationen, angereichert werden. Diese Annotationen sorgen dafür einzelne Entitäten des Texts zu identifizieren, wie z.B. Personen, Orte, Organisationen etc. oder Relationen wie

³ Webseite des Apache UIMA (Unstructured Information Management Architecture) Projekts: ui-ma.apache.org

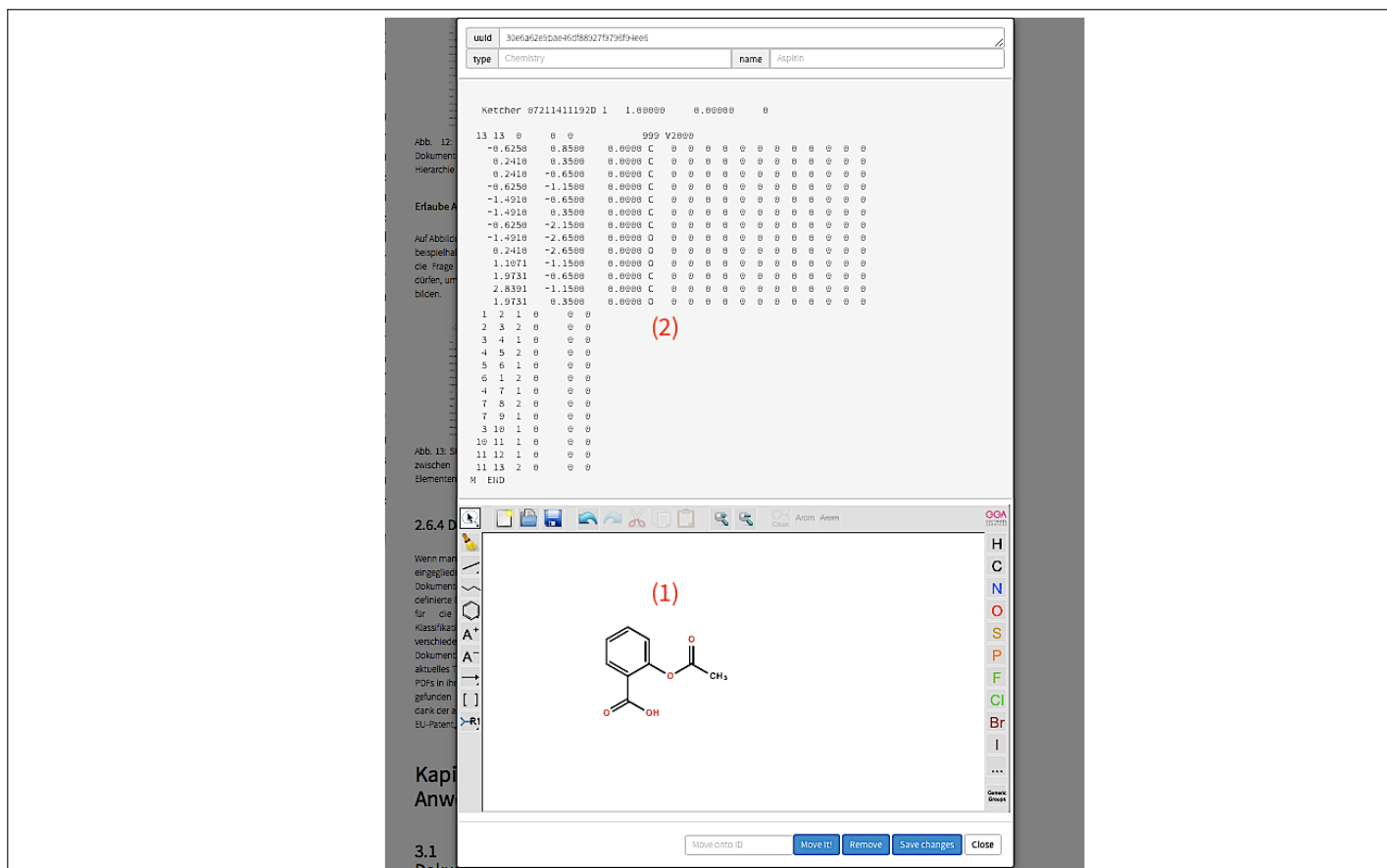


Abbildung 3.2: Bildschirmaufnahme der Editier-Ansicht des Chemie-Dokumentelements. Es ist der Domäneneditor „Ketcher“ (1) zu sehen, der gerade das Aspirin Molekül anzeigt und zum editieren einlädt. Zudem wird das Modell (2) des Aspirin Moleküls im Molfile-Format angezeigt.

„arbeitet für“.

CAS (Common Analysis System) Objekte bieten Zugriff auf das Typsystem, Indexe, Iteratoren und Filter von UIMA. Insbesondere ist es möglich Annotationen mit dem CAS Objekt zu erstellen. Das heißt in einem CAS Objekt kann ein Dokument mit Annotationen gespeichert werden. Ein solches CAS Objekt kann in das XML Metadata Interchange (XMI) Format serialisiert werden, wodurch es möglich ist annotierte Dokumente zu persistieren und auszutauschen.

Mit einem CAS⁴ Editor würde ein Autor also gerne im betrachteten Dokument

⁴ UIMA CAS Objekt Dokumentation: uima.apache.org/downloads/releaseDocs/2.3.0-

mit Annotationen arbeiten:

1. Annotationen anzeigen,
2. Annotation bearbeiten und
3. Annotationen hinzufügen.

3.2.1 Konzeptuelle Umsetzung

Für jeden Annotations-Typ (z.B. Person, Ort, etc.) müsste ein entsprechendes Dokumentelement angelegt werden, d.h. das Metamodell des Dokuments muss entsprechend erweitert werden. Wenn das Metamodell um die gewünschte Menge an Annotations-Typen erweitert ist, kann der Prototyp automatisch mit den neuen Dokumentelementen bzw. Annotationen umgehen. Da jedes Dokumentelement das Modell seiner Domäne vertritt, können hier auch zusätzliche (semantische) Informationen, die für die jeweilige Annotation wichtig sind, abgelegt werden.

Annotation bedeutet „Anmerkung“, „Beifügung“, „Hinzufügung“. In diesem Sinn haben Annotationen bei Stichworten, Begriffsklärungen oder ausführlichen Texten den Charakter der Erklärung beziehungsweise Ergänzung. Annotationen halten Dinge fest, die zwar nicht als wesentlich für das Hauptstichwort oder den Haupttext erachtet werden, aber wichtige Zusatzinformationen darstellen. Sie sind es immerhin wert, ausdrücklich festgehalten zu werden, und auf diese Weise erhalten die bezeichneten Inhalte einen Platz in der Ordnung des Ganzen, ohne die Struktur zu stören oder die Sinnlinie der Aussage zu unterbrechen. (Wikipedia, 2013)

Das heißt, Annotationen sind nicht zwingender Bestandteil des eigentlichen Dokuments, welches den intellektuellen Inhalt liefert. Sie sind eher Stützen

incubating/docs/api/org/apache/uima/cas/CAS.html

welche das Dokument leichter verständlich machen. Daher können sie den Metadaten Dokumentelementen zugeordnet werden.

Für diesen Zweck wurde im Prototyp ein Metadokument implementiert. Dieses enthält Zusätze oder Aussagen, die vom eigentlichen Dokument separiert sein sollen. Das Metadokument erbt die gesamte Infrastruktur des Hauptdokuments. Sie haben einen gemeinsamen Dokumentelement-Pool und können daher gegenseitig auf die vorkommenden Dokumentelemente zugreifen (z.B. via Verweis). Schöner Nebeneffekt: Die Annotationen können im Metadokument übersichtlich angeordnet und dort ggf. dokumentiert werden.

Annotationen werden durch ein „mouse over“ im Browser farbig hervorgehoben (s. Abb. 3.3). Im Metadokument können neue Annotationen hinzugefügt werden und via Verweis in ein beliebiges Dokumentelement gebracht werden. Im Metadokument können sie auch bearbeitet werden. Für Demonstrationszwecke wurde das Dokumentelement „Annotation“ implementiert. Als Beispieldokumente dienen Patentschriften die durch das vom Fraunhofer SCAI geleitete Projekt *UIMA-HPC*⁵ digitalisiert, durchsuchbar und annotiert wurden. Das SCAI.BIO abteilungsinterne Werkzeug „The Interfacer“ kann CAS Objekte (in denen die Patentschriften u.a. vorliegen) in Datenstrukturen des Prototypen übersetzen und somit für den Prototypen mühelos nutzbar machen.

3.3 Dokumentation von Spray-Modellen

Spray⁶ ist ein Forschungsprojekt im Bereich der modellgetriebenen Softwareentwicklung, an der HTWG Konstanz. Spray erstellt Editoren für visuelle domänenspezifische Sprachen (DSL). Das heißt mit Spray können z.B. Diagramme wie Petrinetze, UML, etc. formal definiert werden und aus diesen Definitionen kann ein Editor generiert werden. Mit einem solchen Editor kann dann das jeweilige Diagramm gezeichnet bzw. modelliert werden kann. Vorteil ist, dass durch die Grammatik der visuellen DSL genau spezifiziert ist, welche Verbindungen zwischen den einzelnen Knoten erlaubt sind und welche nicht,

⁵ Webseite des UIMA-HPC Projekts: uima-hpc.org

⁶ Webseite des Spray Projekts: eclipselabs.org/p/spray/

Meta Dokument Ansicht

1.0 Annotationen

Annotation: DE 3144021 A1

Annotation: DE 3144021 A1

Annotation: DE-OS 30 08 588

Annotation: DE—OS 30 14 669

Annotation: EP-A 0004 399

Annotation: GB-A 80 12 242

Annotation: Mol HTP in 1000

Annotation: DE-OS 30 06 268

Annotation: US 3 271 147

Annotation: US 3 271 148

Annotation: DE-OS 2 315 304

Annotation: DE-OS 2 941 818

Haupt Dokument Ansicht

Farbstoffe dienen vorzugsweise langkettige quaternäre Ammonium- oder Phosphoniumverbindungen, z. B. solche, wie sie beschrieben sind in [US 3 271 147](#) und [US 3 271 148](#). Ferner können auch bestimmte Metallsalze und deren Hydroxide, die mit den sauren Farbstoffen schwerlösliche Verbindungen bilden, verwendet werden. Weiterhin sind hier auch polymere Beizmittel zu erwähnen, wie etwa solche, die in [DE-OS 2 315 304](#), [DE= 05.2 631 521](#) oder [DE-OS 2 941 818](#) beschrieben sind. Die Farbstoffbeizmittel sind in der Beizmittelschicht in einem der üblichen hydrophilen Bindemittel dispergiert, z.B. in Gelatine, P_{an}notation, ganz oder partiell hydrolysierten Celluloseestern. Selbstverständlich können auch manche Bindemittel als Beizmittel fungieren, z. B. Mischpolymerisate oder Polymerisatgemische von Vinylalkohol und N-Vinylpyrrolidon, wie beispielsweise beschrieben in der [DE-AS 1 130 284](#), ferner solche, die Polymerisate von stickstoffhaltigen quaternären Basen darstellen, z. B. Polymerisate von N-Methyl-vainylpyridin, wie beispielsweise beschrieben in [US 2 484 430](#). Weitere brauchbare beizende Bindemittel sind beispielsweise Guanylhydrazonderivate von Alkylvinylketonpolymerisaten, wie beispielsweise AG 1813 10 15 20 25 äqu-c 1 1 4 n-o „a_- m94“ 3144021 beschrieben in der [US 2 882 156](#) oder Guanylhydrazonderivate von Acylstyrolpolymerisation, wie beispielsweise beschreiben in [DE-OS 2 009 498](#). Im allgemeinen wird man jedoch den zuletzt genannten beizenden Bindemitteln andere Bindemittel, z. B. Gelatine, zusetzen. Darüber hinaus kann die Bildempfangsschicht oder eine hierzu benachbarte Schicht Schwermetallionen, insbesondere Kupfer- oder Nickelionen enthalten, falls bei der Entwicklung durch Schwermetallionenkomplexierbare diffusionsfähige Farbstoffe oder Farbstoffvorläufer freigesetzt werden. Die Metallionen können in der Bildempfangsschicht in komplex gebundener Form vorliegen, z. B. gebunden an bestimmte Polymerisate wie etwa beschrieben in Research Disclosure 18 534 ([Sept. 1979](#)) oder in DE-os 30 02 287.7. Sofern die Bildempfangsschicht auch nach vollendeter Entwicklung in Schichtkontakt mit dem lichtempfindlichen Element verbleibt, befindet sich zwischen ihnen in der Regel eine alkalidurchlässige pigmenthaltige lichtreflektierende Bindemittelschicht, die der optischen Trennung zwischen lichtempfindlichem Element und Bildempfangsschicht und als Bildhintergrund für das übertragene farbige Äquidensitenbild dient. Eine solche lichtreflektierende Schicht kann in bekannter Weise bereits in dem farbfotografischen Aufzeichnungsmaterial vorgebildet sein oder aber

Abbildung 3.3: Entnommen aus Screenshots des Prototypen. Es wird dargestellt, wie Annotationen vom Meta Dokument in das Haupt Dokument (über Verweisung) fließen können. Im Haupt Dokument werden bei „mouse over“ alle Annotationen vom gleichen Annotations-Typ „Annotation“ farbig hervorgehoben. Es ist möglich beliebige Annotations-Typen (als Dokumentelement) zu kreieren.

d.h. der Diagramm Editor kann zu einem gewissen Grad die Korrektheit des Modells überprüfen.

Das heißt ein mit Spray generierter Editor arbeitet auf einer abstrakteren Ebene – der Modellebene. Das Modell selbst sollte jedoch auf einer noch abstrakteren Ebene beschrieben werden: in einer natürlichen Sprache. Andernfalls ist es einem Menschen kaum möglich ein Modell und dessen Zweckbezug, ohne große Hürden, zu verstehen. Einfacher ausgedrückt: das Modell muss dokumentiert werden, damit andere Projektbeteiligte auch etwas damit anfangen können. Man kann also argumentieren, dass Dokumentation in der Abstraktionsebene nochmals über dem eigentlichen Modell liegt, und

das Modell um benötigte sprachliche Erklärungen anreichert. Die Dokumentation fügt also explizit den Pragmatismus zum Modell hinzu: für wen, wann und wozu gibt es das Modell?

3.3.1 Erhoffter Nutzen

Die hier vorgestellten Konzepte bzw. Prototyp ermöglichen quasi das gleichzeitige Modellieren und Dokumentieren. Die Dokumentationssoftware ist in der Lage direkt mit dem Modell zu interagieren und daher ist die Dokumentation in einem sehr hohen Maß konsistent. Zudem ist das Dokumentieren dann kein isolierter Prozess mehr „den man machen muss“, sondern gehört zum Arbeitsablauf der Modellierung dazu. Das heißt von der Idee bis zum fertigen Modell kann die Dokumentation das Gedankengebäude des Entwicklers sinnvoll unterstützen.

Und ich persönlich denke, dass sich durch ein solches Dokumentationswerkzeug ein System wie Spray von anderen Lösungen absetzt und daher ein großes Erfolgspotential entwickeln kann. Gibt es bereits ein solches Werkzeug im Bereich der modellgetriebenen Softwareentwicklung, wo Dokumentation und Modellierung derart integriert werden können und zudem in einem veröffentlichungsbereiten Format vorliegen, welches wissenschaftlichen Standards entspricht?

3.3.2 Konzeptionelle Umsetzung

Der vorgestellte Anwendungsfall wurde beispielhaft im hier entwickelten Prototypen umgesetzt. Das Forschungsteam um Spray hat die Software von einer reinen Java/Eclipse Implementierung ins Web gebracht, das heißt die generierten Diagramm-Editoren liegen auch als Webanwendung⁷ mit REST-Schnittstelle vor. Der Prototyp selbst ist auch mit Web Standards gebaut und kann daher recht mühelos mit „Spray Online“ interagieren.

⁷ Momentan liegt die Spray Online Webanwendung nur intern im VPN-Netz der HTWG Konstanz vor.

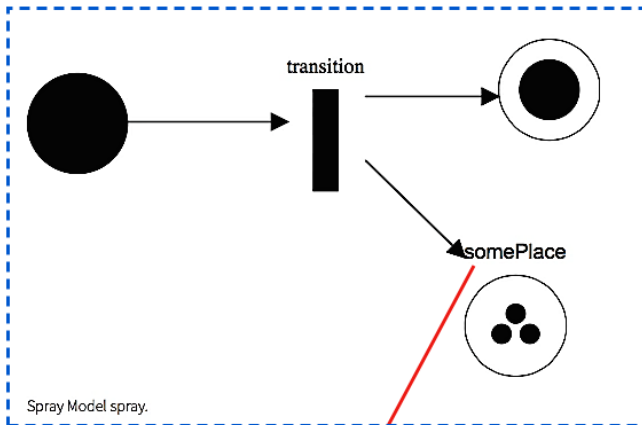
Es wurde in Wrapper entwickelt, der es ermöglicht den Spray Online Editor in Form eines Dokuments zu anbieten. Jedes Dokumentelement kann beliebige Ansichten oder Projektionen haben. Hier gibt es zum einen eine Editierungsansicht (dort können DSLs oder Domäneneditoren platziert werden) und zum anderen eine reine Anzeigeansicht (z.B. als Webseite). Das Spray-Dokumentelement bettet den originalen Spray-Editor in Form eines iframes in die Editierungsansicht ein. Der Benutzer kann dann direkt Änderungen an seinem Spray-Modell vornehmen. Wenn der Benutzer die Änderungen speichert, holt sich das Spray-Dokumentelement den aktuellen Zustand des Modells aus der Spray-Persistenz via REST-Schnittstelle. Im Zustand des Modells sind alle Informationen über das Modell gespeichert, wie z.B. die Namen der einzelnen Diagramm-Knoten. Dadurch dass das Spray-Dokumentelement Zugriff auf die Attribute des Spray-Modells hat, kann im Dokument auf diese Attribute durch Verweisungen zugegriffen werden. Mit diesem Zugriff kann das Dokument mit dem Modell konsistent gehalten werden und es besteht die Möglichkeit Semantik explizit zu machen, indem zusätzliche Informationen über die Bedeutung des referenzierten Attributes angezeigt werden. Für die Anzeigeansicht holt sich das Spray-Dokumentelement ein svg-Bild des Diagramms – dieses kann über einen HTTP-Request vom Spray Online Editor abgefragt werden.

Auf Abbildung 3.4 ist das Szenario aufgezeigt. Ziel ist es, Attribute aus dem Spray-Modell als Verweis innerhalb des Textes anzuzeigen. Hier im Beispiel soll der Name einer Stelle aus einem Petrinetz im Text referenziert werden.

Wenn der Benutzer das Spray-Dokumentelement anklickt, öffnet sich eine Editieransicht in der das Modell manipuliert werden kann. (Dargestellt auf Abbildung 3.5) Dort wird der (1) Domäneneditor „Spray Online“ dargestellt. Der Editor hat ein Petrinetz-Modell geladen, welches vom Benutzer manipuliert werden kann. Das Spray-Dokumentelement kann (2) einen Namen erhalten, um es später beim Verweisen auf das Modell (komfortabel) wieder zu finden. Wenn der Benutzer (3) die Änderungen speichert, holt sie das Spray-Dokumentelement die aktuellen Informationen über den momentanen Zustand des Modells (wie z.B. die Namen der einzelnen Diagramm-Komponenten).

Kapitel 1 Using Spray

Via default fa5a62f0-1541-4551-9203-843e61ac7253 on 141.37.31.44:9000 is opened. But it is possible to change the accessed uuid.



1.1 Reference The Model

Now access the model. We have a place "somePlace".

Abbildung 3.4: Im blau gestrichelten Kasten ist das Spray-Dokumentelement in seiner Anzeigeansicht zu sehen. Szenario: Der Name einer Stelle des Petrinetz-Diagramms soll im Text erwähnt werden (roter Pfeil).

In Abbildung 3.6 wird die Editieransicht eines Abschnitts gezeigt. Dieser Abschnitt soll auf das Modell verweisen, und sich bei Änderungen des Modells selbst aktualisieren zu können. Der Programmausdruck holt sich die entsprechenden Informationen über das Modell direkt beim Spray-Dokumentelement ab. Der rot unterstrichene Teil-Programmausdruck zeigt alle verfügbaren Modell-Informationen (s. Pfeil), die es gerade über alle Stellen im Petrinetz gibt, an. So kann sich der Autor auch eine einzelne Stelle des Petrinetzes herauspicken und ihren (aktuellen) Namen abfragen. Abbildung 3.4 zeigt das Resultat.

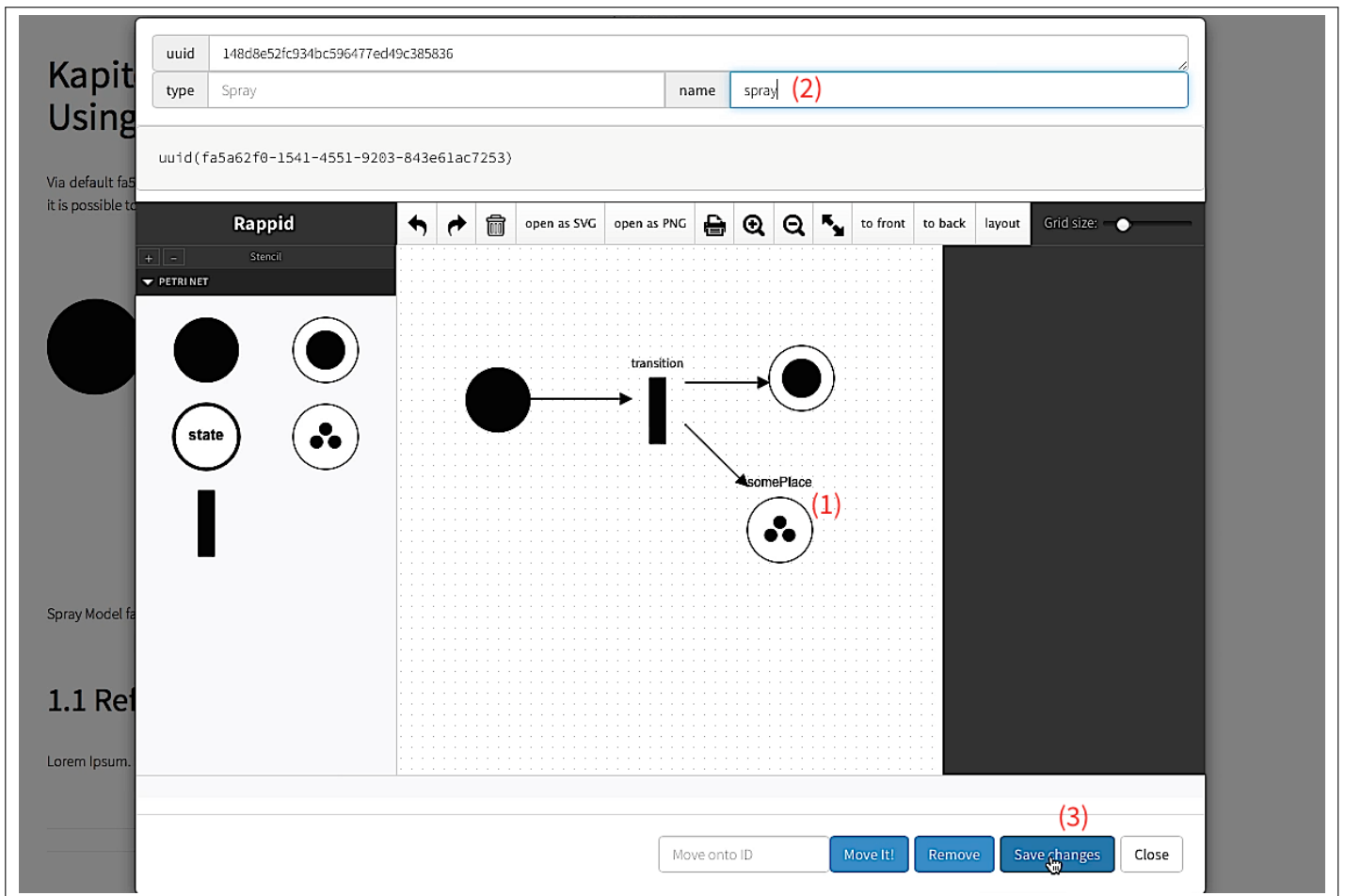


Abbildung 3.5: Ansicht zum Bearbeiten des Spray-Dokumentelement. In (1) ist der Domäneneditor, welcher das Modell in Diagrammform anzeigt und durch den Benutzer manipulierbar macht. Das Modell kann benannt werden für die spätere Verweisung im Text (2). Speichern der Veränderungen (3) veranlassen das Spray-Dokumentelement die aktuellen Modell-Informationen zu laden.

3.4 Transformation in andere Formate

Dokumente in andere Formate zu überführen war schon immer notwendig, und in modernen Zeiten immer wichtiger. Beispielsweise werden von wissenschaftlichen Berichten (wie Diplomarbeiten) Manuskripte angefertigt. Diese werden später vom Verleger überarbeitet und in ein Buch (oder eBook) gesetzt, vgl. (DIN, 1983b). Daran hat sich nichts geändert, nur dass die Schreibmaschine durch ein Textverarbeitungsprogramm ersetzt wurde und der Vorgang dadurch etwas komfortabler geworden ist.

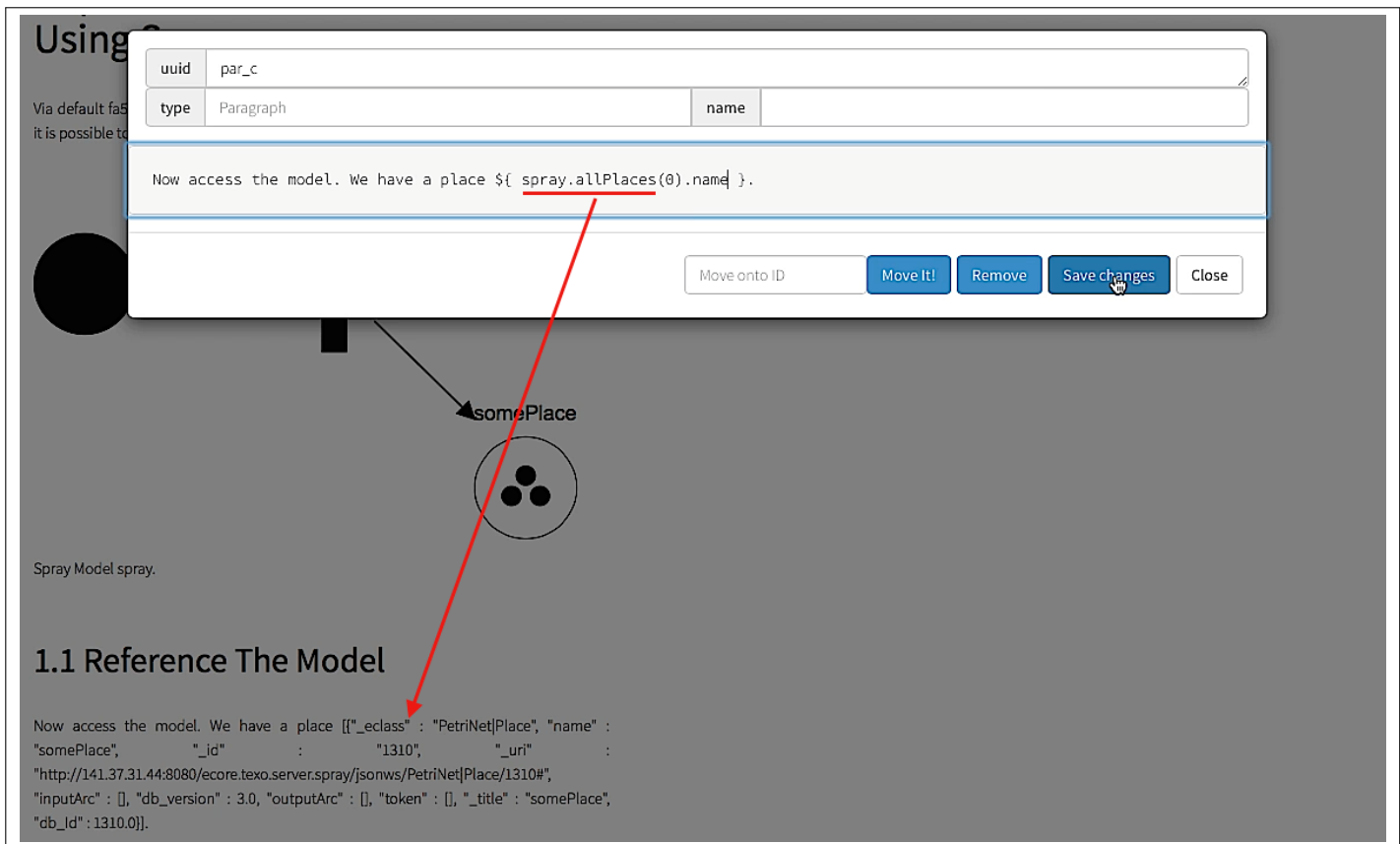


Abbildung 3.6: Hier wird auf das Spray-Dokumentelement verwiesen. Mit einem Programmausdruck kann der aktuelle Zustand des Modells abgefragt werden und bei Bedarf auf einzelne Attribute verwiesen werden. Bei Modelländerungen (z.B. Umbenennungen eines Diagrammknotens) bleibt der Text konsistent.

Für Format Transformationen gibt es in jüngster Zeit auch noch andere Anwendungsgebiete. Beispielsweise möchte man historische Dokumente digitalisieren und mit möglichst vielen Zusatzinformationen anreichern (vgl. Abschnitt 3.2). Oder möchte Dokumente in digitalen Langzeitarchiven speichern. Oder möchte Dokumente von alten Layout-Konventionen in die Neuen überführen. Oder möchte gleichzeitig Dokumente sowohl für legacy Software als auch für aktuelle Software zugänglich halten. Oder möchte das Dokument für andere Arbeitsgebiete aufbrechen, wie z.B. Knowledge Discovery, Semantic Web. Oder möchte ein Dokument auf ein bestimmtes Publikum spezialisieren oder in eine andere Dokumentklasse überführen (Journal Artikel in eine Präsentation). All das erfordert u.U. vollkommen unterschiedliche Formate!

Hier kommt der abstrakte Syntaxbaum (AST) ins Spiel: Ein AST ist innerhalb eines Übersetzers eine Zwischenrepräsentation für ein Programm, woraus schlussendlich Maschinencode produziert wird – Details in Abschnitt 2.2. Hier wird direkt auf dem AST gearbeitet (Projektionseditor). Der AST repräsentiert zudem auch direkt das Modell des Dokuments. Und jedes Dokumentelement hat wiederum Wissen über seine Domäne. Es sind also genügend Informationen vorhanden, um quasi beliebige Formate zu produzieren. Es muss also nur eine weitere Projektion zum AST hinzugefügt werden (als Template), damit ein entsprechendes Format erzeugt werden kann. Das heißt statt klassisch Maschinencode zu produzieren, kann hier auch ein anderes Dokumentformat produziert werden, wie z.B. (ANSI/NISO, 2012) konformes XML, XMI für CAS Objekte, ePub, etc.

Hier wurde beispielhaft, neben der HTML-Projektion mit Editier-Funktion, LaTeX-Code als Projektion umgesetzt. Aus diesem LaTeX-Code kann wiederum ein (paginiertes) PDF generiert werden. Dies wird auch so genutzt, um diese Masterarbeit zu schreiben – siehe Abschnitt 3.1.

Kapitel 4

Entwicklung des Prototypen

Um die wissenschaftlichen Fragen zu **beantwortet wurde** als Methode die Entwicklung eines Prototypen gewählt. Anhand des Prototypen können Aussagen getroffen werden, ob die vorgestellten Ideen und Konzepte tragfähig sind. Zudem können die Konzepte verfeinert und **neue** Impulse für neue Ideen bzw. Theorien gefunden werden.



Zunächst wird das Vorgehen mit den Test-Spezifikationen erläutert. **Gefolgt von** **den** Basiskonzepten, die dem Prototypen inne wohnen. Daraus ergibt sich eine Architektur, die von Grob zu Detailreich beleuchtet wird. Am Ende des Kapitels wird der erreichte Funktionsumfang diskutiert und mit einer kleinen Statistik über den Codeumfang des Prototypen abgeschlossen.



Der Quellcode ist auf GitHub öffentlich zugänglich und wird via Zendodo.org zitierbar gemacht, d.h. Version 0.6.0 des Prototypen erhält eine DOI (digital object identifier) und liegt im Langzeitarchiv von Zenodo. Im Quellcode ist eine README **Datei welche** als Anhang für die Masterarbeit dient. Dort ist u.a. beschrieben, wie die Software installiert werden kann, welche Releases (Meist Zwischenstände für Vorträge) es gegeben hat.

4.1 Vorgehen

Als Entwicklungsmethode wird das Behavior-Driven development (BDD) eingesetzt, welches von (North, 2006) geprägt wurde. (Unit) Tests werden dort in Form von Spezifikationen verfasst. Jede Spezifikation soll ein bestimmtes Verhalten der Software dokumentieren und verifizieren. Das Framework ScalaTest¹ bietet die Möglichkeit nach BDD Softwaresysteme zu entwickeln. Die Aktoren-Implementierung des Prototypen wird mit ScalaTest getestet.

Bei jedem git push auf GitHub wird automatisch Travis CI aktiv. Hierbei handelt es sich um eine Continuous Integration Platform, welche die richtige Umgebung zum Ausführen von ScalaTest² einrichtet. Es werden automatisch die benötigten Scala Abhängigkeiten heruntergeladen und kompiliert, sowie eine CouchDB für die Tests zur Persistenz bereitgestellt. Ob die Tests fehlschlagen oder glücken, wird in einem Protokoll festgehalten.



4.1.1 Getestete Spezifikationen



Hier sind die 41 Spezifikationen die an den Prototypen gestellt sind aufgelistet. Die Spezifikationen sind in englischer Sprache gehalten, da dies die übliche Weise ist, Software zu dokumentieren. Die Spezifikationen sind so formuliert, dass sie möglichst selbsterklärend sind. Nachfolgend die Ausgabe von ScalaTest:

RootSpec:

The Root Actor

- should hold topological informations about the document
- should digg for the firstChilds
- should digg for the next references
- should be able to calculate the correct order of the document elements
- should be able to insert (new) elements and remove elements from topology

¹ Tests als Spezifikationen aus der Dokumentation von ScalaTest:
http://www.scalatest.org/user_guide/tests_as_specifications

² Travis CI für das Git Repositorium des Prototypen: <https://travis-ci.org/themeriusscaltex-2>

- should setup the document actors with a given topology
- should be able to initiate the Update through the entire document
- should pass messages to the selected id

RemoveElementSpec:

Remove of a non-leaf

- should repair the topology and put the removed subtree into the graveyard

Remove of a leaf

- should repair the topology and put the removed element into the graveyard

MoveElementSpec:

Move of a non-leaf

- should change the topology (hang a entire subtree to a destination)
- should change the topology (hang a entire subtree to a destination II)
- should move the entire subtree into another hierarchy level

Move of a leaf

- should change the topology and recreate the actor at the desired position
- should also work if moved to a position of a first child

ReportSpec:

A REPORT document meta model element

- should save it's unique id into it's state
- should be able to change it's current assigned document element
- should be able to obtain a reference to the next actor
- should send update to the next actors
- should have a content (source, representation and result from evaluation)
- should send deltas when the topology changes
 - + When a new element is inserted after a leaf
 - + Then updater should receive a delta (topology change set)
 - + And the new-elem should register itself to root
 - + When a new element is inserted after a non-leaf
 - + Then updater should receive a delta to the lastChild
 - + When a new element is inserted as first child (extend hierarchy)
 - + Then updater should receive a delta

A element with a variable (short) name

- should be part of the base actor state

- should be changeable

OutlineSpec:

The SECTION document elements

- should have 'title' and 'numbering' properties
- should change the 'title' attribute when the content changes
- should be able to discover it's (primary) section number
- should be able to discover it's (secondary) section number
- should be able to discover it's (tertiary) section number

TableOfContentsSpec:

The TableOfContents element

- should ask every Chapter and *Section within BodyMatter

InterpreterSpec:

A INTERPRETER

- should evaluate a string which contains scala code
- should return None if evaluation fails

DiscoverReferencesSpec:

The References Discovery

- should find all ActorRefs within a Sting
- should generate code of the classes the user may use
- should collect the complete code and put the evaluated repr into contentRepr

Unified content

- should be an array with the expression details

CouchDbSpec:

Root

- should be able to load a document from CouchDB persistance
- should NOT integrate _id and _rev from CouchDB into the topology
- should persist the topology on changes

Each such created element

- should aquire it's state from persistance 'to life'
- should save state, if changed, back to persistance
- should save state if it's not persisted yet

4.2 Basiskonzepte

Der Prototyp versucht die Anforderungen aus Abschnitt 1.4 umzusetzen. Hier nochmals kurz in Stichpunkten zusammengefasst:

1. Abstraktion für Dokumentelemente
2. Domänenspezifische Inhalte in Dokumentelementen
3. Reichhaltige Verweisungen für explizite Semantik und mehr Konsistenz
4. Metamodell definiert erlaube Dokumentelemente
5. Projektionseditor für Dokumente, mit Web Standards umgesetzt

(1) Es ist möglich ein Dokument als abstrakten Syntaxbaum darzustellen, wobei jeder Knoten einem Dokumentelement entspricht. (2) Jedes Dokumentelement kann Teile seiner Domäne in den Attributen abspeichern und einen Domäneneditor, der im Browser ausgeführt wird, bereitstellen. (3) Dokumentelemente können durch Nachrichtenaustausch Attribute eines anderen Dokumentelements abfragen und in den eigenen Kontext einbauen. Diese Abfragen werden in der Programmiersprache Scala notiert und von einem Scala-Interpreter ausgewertet. (4) Jedes Dokumentelement implementiert eine Scala **Klasse wodurch** es repräsentiert wird. Der Scala-Interpreter kann diese Klassen instanziiieren und ausführen. Das Resultat kann z.B. an ein anderes Dokumentelement geschickt werden. (5) Nachrichten aus einem Websocket können zum abstrakten Syntaxbaum geleitet werden, und umgekehrt. Dieser kann darauf reagieren bzw. entsprechende Maßnahmen ausführen. Beispielsweise kann er sich selbst modifizieren (Knoten hinzufügen/entfernen) oder analysieren (Neue Attribute der Knoten entdecken, wie z.B. Benummerungen zu Abschnitten hinzufügen).

Bedient wird das Autorensystem über den Browser. **Es spiegelt direkt eine Repräsentation** des abstrakten Syntaxbaumes des Dokuments wieder. Der Benutzer arbeitet also direkt mit abstrakten Syntaxbaum, bekommt jedoch nur ausgewählte bzw. relevante Attribute als „konkrete Syntax“ angezeigt. Der Autor bekommt also eine möglichst komfortable Ansicht des Dokuments. Wenn



er auf ein Dokumentelement klickt, wird ein Fenster geöffnet. Dort werden nähere Informationen zum Dokumentelement eingeblendet und das Dokumentelement kann dort bearbeitet werden. In dieser Ansicht können neben textuellen Eingaben (z.B. in Form von DSLs) auch domänenspezifische Editoren (wie z.B. ein chemischer Moleküleditor) angezeigt werden.

4.3 Architektur

Auf Abbildung 4.1 ist eine Übersicht über die Architektur des Prototypen. Der Dreh- und Angelpunkt ist das Dokumentmodell (vgl. Abschnitt 2.1.2), mit dem der Autor quasi direkt kommunizieren kann. Das Dokumentmodell wird von seinem Metamodell (vgl. Abbildung 2.7) abgeleitet, dort sind u.a. die einzelnen Dokumentelemente definiert.

Jedes einzelne Dokumentelement, **also Aktor**, persistiert bei Änderungen des Zustandes diesen in CouchDB. CouchDB ist eine dokumentenorientierte NoSQL-Datenbank, welche JSON-Objekte speichern kann. Eine Besonderheit der Datenbank ist, dass jede Änderung am JSON-Objekt mit einer neuen Version versehen wird, d.h. es ist möglich die Entstehungsgeschichte eines jeden Dokumentelements nachzuvollziehen. Es ist also **möglich ein** Dokument zerstörungsfrei zu erstellen. Neben der Versionskontrolle ist die REST-Schnittstelle zur Datenbank ein Basiskonzept von CouchDB, s. (Anderson u. a., 2010).



Xitrum ist ein controller-first Web **Framework geschrieben** in Scala. Die Aufgabe von Xitrum **ist Websockets** für die Kommunikation mit dem Browser bereitzustellen und **die** statische Dateien, wie HTML-Templates, CSS, JavaScript oder Bilder auszuliefern. Das Dokumentmodell hat einen Update Aktor, welcher die Änderungen vom Browser bzw. vom Modell handhabt. Der Update Aktor kann mit mehreren Websockets gleichzeitig sprechen. Dies ermöglicht es, dass mehrere Browser das gleiche Dokument betrachten können und live die Änderungen der anderen Parteien mitbekommen. Jeder Browser ist mit einem Websocket verbunden.



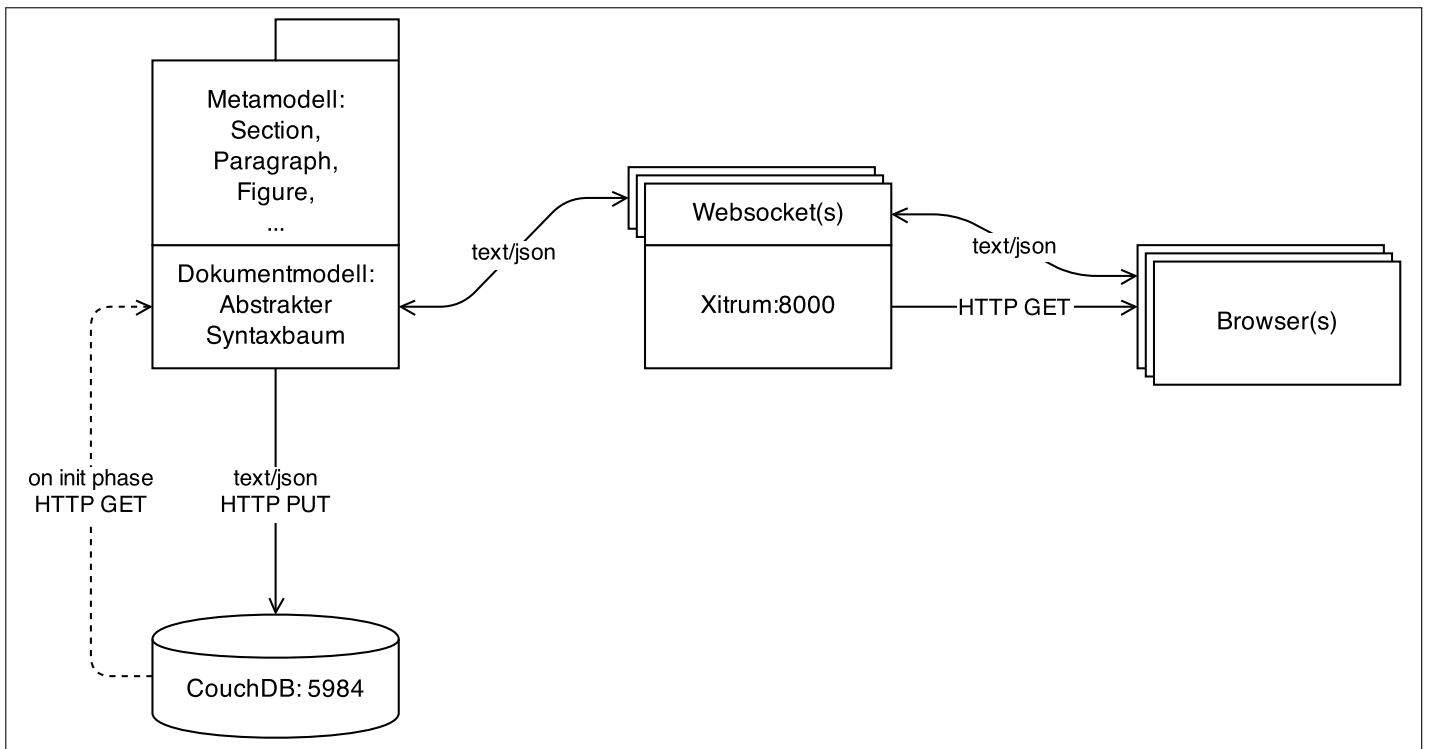


Abbildung 4.1: Die Architektur des Prototypen. Die Pfeile stellen die Kommunikation zwischen den einzelnen Komponenten **dar. Die** Kommunikation läuft zum einen über das Websocket-Protokoll und zum anderen über HTTP-Calls. Die Daten werden stets als JSON-String übertragen, außer bei der Auslieferung der statischen Daten durch Xitrum. Der gestrichelte Pfeil ist eine einmalige Kommunikation, die nur zum Initialisieren des Actor Systems (bzw. abstrakten Syntaxbaumes) verwendet wird. Bei der Initialisierung holt sich jedes Dokumentelement den gespeicherten Zustand aus der Datenbank. Wenn sich der Zustand eines Dokumentelements ändert, wird dies in der Datenbank als neue Version gespeichert.

4.3.1 Wurzel Aktor

Der Wurzel oder Root Aktor enthält die Topologie des Dokuments. Zudem werden hier alle **Aufgaben die** mit der Topologie **zusammenhängen verarbeitet**, da der Root Aktor als einziger Aktor die Gesamtübersicht über das Dokument hat. Zwar speichert jeder Dokumentelement-Aktor seine lokalen Beziehungen (Kinder, Geschwister), aber diese werden nicht in die Datenbank geschrieben. Somit wird verhindert, dass durch eine Wiederherstellung einer Version eines Dokumentelements irgendwelche Topologie-Lücken aufgerissen werden. Nur der Root Aktor speichert topologische Informationen in der Datenbank ab. Wenn Versionen des Root Aktors wiederhergestellt werden, wird

somit ausschließlich die Topologie wiederhergestellt. Das heißt die Topologie kann unabhängig von den aktuellen Inhalten der einzelnen Dokumentelemente restauriert werden. Stichwortartig zusammengefasst:

- Enthält die (gesamte) Topologie des Dokuments
- Persistiert die Topologie in die Datenbank
- Kann Topologie-Änderungen initiieren (wie z.B. hinzufügen, entfernen und verschieben von Knoten)
- Kann Topologie-Informationen bereitstellen (wie z.B. Position der einzelnen Dokumentelemente)
- Ist der Einstiegspunkt für das Dokument

4.3.2 Basis Aktor

Der Basis Aktor repräsentiert einen Knoten im abstrakten Syntaxbaum. Zudem enthält der Basis Aktor eine Instanz einer Scala Klasse, welche das Dokumentelement mit all seinen Attributen und Methoden beschreibt. Der Basis Aktor trägt also das Dokumentelement quasi huckepack. Der Basis Aktor verwaltet die Nachrichtenkommunikation mit den anderen Aktoren und hält Beziehungen zu seinen Kindern (insbesondere dem ersten Kind) und seinen Geschwistern (insbesondere der Nachfolger). Der Basis Aktor kann bei Änderungen kann eine Analysephase des gesamten abstrakten Syntaxbaumes auslösen. Dort werden die Dokumentelemente aufgerufen zu überprüfen, ob noch alle Attribute aktuell sind. Beispielsweise kann ein Abschnitt-Dokumentelement seine Benummerung nochmals überdenken. Nochmals in Stichworten:

- Ist ein Knoten des abstrakten Syntaxbaumes
- Trägt eigentliche Dokumentelemente (mit ihrem Domänenwissen) huckepack
- Verwaltet die Nachrichtenkommunikation

- Kennt die lokale Topologie des Dokuments (sein erstes Kind, sein nächstes Geschwister)
- Bei Änderungen wird eine Analysephase ausgeführt, die auch an das Dokumentelement getragen wird

4.3.3 Metamodell erweitern

Ein Teil des Metamodells sind die Dokumentelement Definitionen. Hier wird spezifiziert, welche Dokumentelemente im eigentlichen Dokumentmodell vorkommen dürfen. Weiter wird dort spezifiziert, welche Eigenschaften das Dokumentelement besitzt. Jedes Dokumentelement implementiert das gleiche Interface:

```
class Figure extends DocumentElement {

  // 1. Attribute
  this.state.title = ""
  this.state.numbering = ""

  // 2. Analysephase
  def _gotUpdate(actorState: Json[_], refs: Refs) = {
    // ...
    super._gotUpdate(actorState, refs)
  }

  // 3. Analysephase
  def _processMsg(m: M, refs: Refs) = {
    // ...
  }
```

}

Im o.g. Quelltext wird ein Dokumentelement „Figure“ (Abbildung) spezifiziert. Es hat (1) ein Menge an Attributen festgelegt, die während der z.B. der Analysephase ändern können. Wenn der Basis Aktor über Änderungen **benachrichtigt** wird, teilt er das über Methode (2) auch dem Dokumentelement mit. Es erhält Zugriff auf den Zustand des Basis Aktors (hier ist z.B. contentSrc gespeichert, also die rohen Eingaben des Autors) und auf die Referenzen die der Aktor zu anderen Aktoren pflegt. Auf diese Weise kann das Dokumentelement Anfragen an andere Aktoren bzw. Dokumentelemente, im Form von Nachrichten, stellen. Zudem kann es vorkommen, dass es Nachrichten gibt, die an das betreffende Dokumentelement gerichtet sind. Diese werden in (3) verarbeitet. Beispielsweise kann Methode (2) einer Abbildung eine Nachricht an alle anderen Abbildungen schicken, um die korrekte Benummerung herauszufinden. Diese Nachricht würde bei jeder anderen Abbildung in Methode (3) zur weiteren Verarbeitung landen.


Beliebige weitere Methoden, die für das Dokumentelement bzw. dessen Domäne nützlich sein könnten, können an diese Scala Klasse hinzugefügt werden. Das ermöglicht den Aktoren reichhaltige Verweisungen untereinander durchzuführen. Es wird also ermöglicht, dass ein Dokumentelement aus den Attributen eine Information berechnen kann und diese an den anfragenden Aktor weitergibt.

4.3.4 Verweisungen auflösen

Reichhaltige Verweisungen sind ein grundlegendes Konzept des Prototypen. Hieraus entstehen neben den üblichen Verweisungen, wie man sie aus wiss. Dokumenten kennt (Verw. auf Abbildungen, Abschnitte oder Literatur), neue Möglichkeiten für Semantik und Konsistenz.

Dokumentelemente liegen als Scala Klasse vor und eine Verweisung greift darauf direkt zu. Das heißt innerhalb eines Textes wird mit einem Scala Programmausdruck notiert, auf welches Attribut oder Methode zugegriffen werden soll und welches Dokumentelement adressiert werden soll.

Prinzipieller Ablauf:

1. Ein Basis Aktor findet einen Programmausdruck in seinem, durch den Autor eingegeben, Text (contentSrc).
2. Der Basis Aktor findet heraus, an welche anderen Basis Aktoren der Programmausdruck gerichtet ist.
3. Die anderen Basis Aktor schicken ihren jeweils aktuellen Zustand, in Form von generiertem Scala Code, an den anfragenden Aktor.
4. Der Basis Aktor sammelt den Code der anderen Aktoren ein. Das heißt er hat nun den Code, um alle Verweise die er gefunden hat aufzulösen. Der aggregierte Code wird an einen Scala Interpreter (auch ein Aktor) weitergeleitet.
5. Der Scala Interpreter instanziiert die Scala Klassen mit dem Zustand der anderen Aktoren.
6. Das Resultat des Interpreters wird, bei gelingen, an die Stelle des  Programmausdrucks kopiert.
7. Wenn alle Programmausdrücke aufgelöst sind wird der neue String als contentRepr bereitgestellt.
8. Neben contentRepr gibt es contentUnified. Dies ist eine JSON-Struktur, welche den Text, die Programmausdrücke sowie die jeweiligen Resultate enthält. (An dieser Stelle könnten, für mehr Semantik, Scala Typesystem Informationen bereitgestellt werden.)
9. Die Anzeige im Browser kann contentSrc, contentRepr, contentUnified und alle anderen Attribute des Dokuments elements nach belieben verwenden. (In einem Template.)

4.4 Eingesetzte Technologien

Hier soll geklärt werden, welche Technologien eingesetzt werden, um den Prototypen zu bauen. Zudem wird begründet, warum genau diese

Technologien ausgewählt wurden.

4.4.1 Scala

Scala ist eine statisch typisierte Sprache, welche auf der Java Virtual Machine ausgeführt wird. Scala verheiratet auf sehr geschickte **weise** das funktionale mit dem objektorientierten Programmierparadigma. Es ergibt sich eine an Java angelehnte, aber sehr aufgeräumte Syntax. Das ermöglicht sehr mächtige und abstrakte Programmierkonstrukte. Die Besonderheiten dabei sind, dass Methoden wie Operatoren auftreten können, Klammern, Semikolons oder Punkte können oftmals weggelassen werden. Dies ermöglicht ein sehr klares und sauberes Sprachbild und eignet sich wunderbar für interne DSLs, s. (Hodapp u. a., 2013).

Zudem gibt es für Scala noch das Akka Toolkit für Aktoren. Diese Implementierung ist neben der bekannten Aktoren-Implementierung aus Erlang die Erfolgreichste auf dem Markt. Scala hat ein ausgesprochen mächtiges Typsystem, welches für zusätzliche Informationen (u.a. Semantik) angezapft werden kann. Diese Eigenschaft könnte für die Zukunft des Projektes interessant sein.

4.4.2 Akka

Akka ist ein in Scala geschriebenes Toolkit. Es bietet Lösungen zur nebenläufigen und verteilten Programmierung an. Akka ermöglicht den Entwicklern, diese schwierige Aufgabe, effizient zu meistern. Insbesondere Skalierungsprobleme sollen vereinfacht werden. Die Basiseinheit mit der vom Programmierer gearbeitet wird, sind die Aktoren. Die ursprüngliche Idee des Aktor Modells stammt von (Hewitt u. a., 1973). Ein Aktor kann nach Empfangen einer Nachricht folgendes tun, vgl. (Hewitt, 2010):

- Senden von Nachrichten an Adressen von Aktoren
- Erstellen von neuen Aktoren

- **Entscheiden wie** mit den nächsten Nachrichten die empfangen werden umgegangen wird

Daraus ergeben sich die Kernoperationen der Akka Aktoren (Roestenburg u. a., 2014, S. 23):

- CREATE: Ein Aktor kann einen anderen Aktor erstellen.
- SEND: Ein Aktor kann Nachrichten zu einem anderen Aktor senden.
- BECOME: Das Verhalten (wie auf Nachrichten reagiert wird) von einem Aktor kann dynamisch verändert werden.
- SUPERVISE: Ein Aktor kann seine Kinder überwachen und bei deren Fehlfunktion (z.B. Neustart des Kindes) eingreifen.

Ein Aktor entspricht also quasi einem leichtgewichtigen Thread. Ein Betriebssystem (Linux x64) kann ca. 4096 normale Threads pro ein Gigabyte Arbeitsspeicher halten, aber jedoch ca. 2.7 Millionen Akka Aktoren (Roestenburg u. a., 2014, S. 20). Das Aktoren Modell wurde erstmals sehr erfolgreich von Ericsons Programmiersprache Erlang eingesetzt, im ADX301 switch mit einer Verfügbarkeit von 99.9999999 Prozent (Roestenburg u. a., 2014, S. 12). Daher ähnelt die API der Akka Aktoren auch der API der erfolgreichen Erlang Aktoren (Typesave Inc., 2014, S. 69). Laut Akka-Projektwebseite <http://akka.io/> setzen viele Firmen das Toolkit produktiv ein, was die Reife des Projekts verdeutlicht.

(Typesave Inc., 2014, S. 7) nennt einige Anwendungsszenarien von Akka, aber der Einsatz als Fundament für einen abstrakten Syntaxbaum ist dort nicht aufgeführt. Das Aktoren-Modell sollte sich jedoch für den Bau eines abstrakten Syntaxbaumes eignen, da die Aktoren selbst ein hierarchisches System organisieren können bzw. im Fall von Akka auch explizit eines sind (Roestenburg u. a., 2014, S. 22). Der hier entwickelte Prototyp wird Akka verwenden, um einen abstrakten Syntaxbaum zu bauen und prüfen, ob das eine gute Idee ist.

4.4.3 Xitrum

Xitrum ist ein controller-first Web Framework geschrieben in Scala. Es zeichnet sich durch eine sehr klare und verständliche API aus. Akka Aktoren werden von Xitrum eingesetzt um Requests zu handhaben, insbesondere die WebSocket-API von Xitrum setzt auf Aktoren. Dies ermöglicht eine sehr einfache Anbindung an das Dokumentmodell aka. abstrakter Syntaxbaum aka. Aktorensystem.

Es wurden auch noch andere Scala Web Frameworks untersucht und auf Tauglichkeit geprüft:

- Spray.io v1.2.0: Setzt komplett auf Akka, auch als Webserver. Unterstützt keine Websockets; ist jedoch für die Zukunft geplant. Wird seit 2011 entwickelt. (Entnommen aus Changelog)
- Play Framework v.2.2.2: Sehr vielversprechend und stabil mit großer Entwicklergemeinde. Jedoch läuft der Scala Interpreter hier nicht korrekt, da eine extra VM gestartet wird, was durch das sbt plugin von Play geschuldet ist. Wird seit 2007 entwickelt. (Entnommen aus Wikipedia) Als Webserver wird Netty eingesetzt.
- Socko Web Server v0.4.1: Interessante Funktionen, aber (für mich) eine schwierige und umständliche API. Wird seit ca. 2012 entwickelt. (Entnommen aus GitHub Commit Logs) Als Webserver wird Netty eingesetzt.
- Xitrum v.3.13: Mit Abstand die beste und schönste API und wird schon seit 2010 aktiv entwickelt. Wird laut Author in echten Projekten schon erfolgreich eingesetzt. Die Dokumentation ist auch recht umfangreich und gut verständlich. Als Webserver wird Netty eingesetzt.

4.4.4 Web Standards

Mit Web Standards ist die Benutzeroberfläche, welche sich im Browser abspielt, realisiert. Das heißt die Anzeige des Dokuments, sowie die

Editor-Schnittstelle für den Autor sitzt im Browser. Das „Web“ besteht aus vielerlei Standards verschiedener Standardisierungsgremien wie, z.B. W3C, IANA, ECMA, Unicode oder IETF, siehe (Sikos, 2011, S. 6). Im Prototyp wurden vor allen Dingen HTTP, Websockets, HTML5, CSS3 und ECMAScript Edition 5 (JavaScript) verwendet.

Websockets

Websockets³ sind noch ein relativ junger Standard, die technischen Anforderungen stehen schon fest, jedoch fehlt noch die endgültige Ratifizierung (Candidate Recommendation zu Recommendation). Viele Browser haben daher schon die WebSocket API implementiert.

Über einen WebSocket werden Daten als JSON-String von und an den abstrakten Syntaxtree übermittelt. Der Vorteil der Websockets ist, dass die Verbindung zum Browser, und damit zum Autor, immer offen bleibt, also nicht bei jeder Aktion des Autors neu aufgebaut werden muss. Des weiteren ist die Verbindung bidirektional, das heißt der Server kann dem Client bei Änderungen benachrichtigen – der heißt der Client muss kein Polling betreiben um über serverseitige Änderungen benachrichtigt zu werden. Wenn der abstrakte Syntaxbaum (Teil des Servers) seine Analysephase abschließt, kann er sofort den Browser über die Änderungen benachrichtigen.

Javascript Module

JavaScript steht in der Kritik ein schlechtes Abhängigkeits- und Modulmanagement zu haben. Das heißt der Programmierer von client side JavaScript gerät oft in Konflikt mit Namespace Pollution⁴. CommonJS versucht Empfehlungen für JavaScript-APIs zu geben. Dazu gehört auch Asynchronous Module

³ Technische Spezifikation der WebSocket API: <http://www.w3.org/TR/2012/CR-websockets-20120920/>

⁴ Von Global Namespace Pollution spricht man dann, wenn Variablen im ersten bzw. globalen Namensraum angelegt werden. Auf diesen Namensraum können alle Funktionen eines Programms zugreifen. Wenn eine fremde JavaScript-Bibliothek, die man im eigenen Programm verwenden will, viele Variablen global hält, gibt es die Möglichkeit unwissend diese zu überschreiben – also eine potentielle Fehlerquelle.

Definition (AMD)⁵, um Module in JavaScript zu definieren und deren Abhängigkeiten zu managen, um die Namespace Pollution zu vermeiden. Require.js⁶ ist ein Projekt, welches die AMD Spezifikationen für client side JavaScript, also für den Browser, umsetzt. Der Prototyp setzt Require.js ein, um strukturiert mit dem client side JavaScript Programmcode umgehen zu können.

Bower ist ein Kommandozeilen Programm, welches client side JavaScript Bibliotheken automatisch herunterladen und verwalten kann. Es ist also ein Werkzeug um Abhängigkeiten aufzulösen. In der bower.json Datei wird spezifiziert, welche Bibliotheken in welcher Version von Bower heruntergeladen werden sollen. Bower wird auch vom Prototypen verwendet, um die eingesetzten JavaScript Bibliotheken zu verwalten:

- requirejs: JavaScript Module Definitionen
- ketcher: Chemischer Moleküleditor (Ein Domäneneditor)
- bootstrap: Layout Framework
- at.js: Autovervollständigung
- jquery: Komfortable API zur XML-Baummodifikation (DOM)
- mustache.js: Template Engine

4.4.5 Dokummentelemente und Domäneneditoren

Um dem Autor eine übersichtliche Repräsentation bzw. die Möglichkeit zur Bearbeitung eines Modells einer Domäne zu geben, sind die hier sogenannten Domäneneditoren nützlich. Jedes Dokumentelement kapselt eine spezifische Domäne, sei es z.B. „Chemiemolekül“, „Petrinetz“, oder „Abschnitt“, „Nummerierte Liste“ oder eine „Quellenangabe“. Für jede dieser Domänen gibt es ein Modell auf das sie sich beziehen. Dieses Modell kann textuell (als DSL) oder grafisch beschrieben werden. Ein Domäneneditor vereinfacht die Interaktion

⁵ CommonJS: Asynchronous Module Definition API-Spezifikation:
<http://wiki.commonjs.org/wiki/Modules/AsynchronousDefinition>

⁶ Require.js Projektwebseite: <http://requirejs.org/>

zwischen Autor und Modell. Im einfachsten Fall bietet der Domäneneditor eine DSL an, für komplexere Modelle, wie die eines Chemiemoleküls, bieten sich grafische Editoren (umgesetzt in Web Standards) an.

Dokumentelemente bestehen aus einer konkreten Scala Klasse (siehe 4.3.3) auf Serverseite. Der Client (also Browser) erhält lediglich den aktuellen Zustand als JSON-String des Dokumentelements. Für jedes Dokumentelement gibt es auf Clientseite mindestens ein Mustache HTML-Template. Das Template rendert dies direkt im Browser zu einer konkreten Repräsentation. Das heißt eine beliebige andere Repräsentation wird durch anpassen des Templates erreicht.

Ketcher

Ketcher⁷ ist ein für sich allein stehender, in JavaScript geschriebener, Editor für chemische Moleküle. Damit lassen sich Moleküle darstellen und zeichnen. Ein chemisches Molekül kann als Molfile⁸ abgespeichert werden. Das Molfile kann als Modell des Moleküls betrachtet werden und wird hier als Attribut im Dokumentelement abgespeichert.

Ein Wrapper macht Ketcher als Domäneneditor nutzbar. Neben dem der Scala Klasse und dem HTML-Template existiert noch ein JavaScript-Modul welches die Logik enthält, um Ketcher kompatibel zur Editor-Schnittstelle zu machen.

Spray

Spray ist eine Web Applikation, welches es ermöglicht formale Diagramme zu zeichnen. Ausführliche Erklärungen in Kapitel 3.3.

Anderes als bei Ketcher handelt es sich hier um eine Web Applikation, die auf einem anderen Server läuft. Das heißt hier gibt es keinen direkten Zugriff auf

⁷ Webseite des Ketcher Projekts: <http://ggasoftware.com/opensource/ketcher>

⁸ Das Molfile der Firma MDL repräsentiert chemische Strukturen. Siehe de.wikipedia.org/wiki/Chemoinformatik „Repräsentation chemischer Strukturen“.

den JavaScript Quellcode, aber APIs um die Web Applikation anzusprechen. Auch hier wurde ein Wrapper gebaut, der sogar sehr ähnlich zu dem von Ketcher aufgebaut ist.

Markdown und BibTeX

Die Listen Dokumentelemente nehmen Markdown als DSL an. Eine Liste konvertiert mit Hilfe einer Scala Bibliothek aus der Markdown-Notation eine XML-Notation. Die XML-Notation wird in Attribute des Dokumentelements umgewandelt, so dass ein Template diese nach belieben benutzen kann.

Die Buchreferenzen Dokumentelemente nehmen BibTeX als DSL an. Eine Scala BibTeX Bibliothek verwandelt die BibTeX-Notation in Attribute des Dokumentelements, so dass ein Template diese nach belieben darstellen kann. Explizit werden authors, title und year als Attribute extrahiert.

4.4.6 Algorithmen und Datenstrukturen

Hier werden einige wichtige Algorithmen kurz erklärt, die während der Entwicklung entstanden sind. Dazu gehört ein Algorithmus zur effizienten Benummerung von Kapiteln bzw. Abschnitten mittels Nachrichtenaustausch. Eine Datenstruktur zur Speicherung der Dokumenten-Topologie mit einem Algorithmus zur Auswertung der Reihenfolge der einzelnen Elemente der Topologie. Und die Funktionsweise der Code Generierung zur Auflösung der „reichhaltigen“ Verweisungen.

Benummern im gerichteten Graph

Abbildungen und Kapitel bzw. Abschnitte werden in wissenschaftlichen Dokumenten für gewöhnlich benummert. Der abstrakte Syntaxbaum muss also während der Analysephase die genauen Benummerungen einzelner Dokumentelemente (je nach Typ) bestimmen können. Hier wird ein Algorithmus

vorgestellt der durch Nachrichtenaustausch innerhalb eines gerichteten Graphen Benummerungen zuweisen kann.

Gegeben ist ein gerichteter Graph $G = (V, E)$. Wobei $V = a, b, x, c$ die Menge der Knoten (Dokumentelemente) darstellt und $E = (a, b), (b, x), (x, c)$ die Kanten (Nachrichtenkanal). a, b, c sind die Abschnitte die benummert werden sollen.

Einfache Visualisierung von G : $a \rightarrow b \rightarrow x \rightarrow c$.

a schickt die Nachricht "Durchzählen(1)" durch G .

b erhält eine Nachricht von a .

b erhöht Durchzählen um 1 auf Durchzählen(2).

b setzt seinen eigenen Nummerzähler auf Durchzählen(2).

b schickt "Durchzählen(2)" weiter durch den Graph.

x reicht die "Durchzählen" Nachricht weiter, ohne Aktion.

c erhält eine Nachricht die von a und b (etc.) bearbeitet wurden.

c erhöht Durchzählen um 1 auf Durchzählen(3)

c setzt seinen eigenen Nummerzähler auf Durchzählen(3).

Da c keinen Nachfolger mehr hat, kann nichts weitergeleitet werden.

Damit werden maximal $|E|$ Nachrichten benötigt, um für jeden Abschnitts-Knoten die korrekte Benummerung zu bestimmen. Die Komplexität bezüglich der Laufzeit beträgt also $O(|E|)$.

Analog kann für Unterabschnitte und Unterunterabschnitte mit der Nachricht Durchzählen(1, 1, 1) begonnen werden. Ein Unterabschnitt würde also nur die mittlere 1 inkrementieren und die anderen unangetastet lassen.

Das ist ein recht natürlicher Ansatz, um die Benummerungen zu bestimmen. Auf diese Art können Abschnitte sowie Unterabschnitte gleichzeitig mit einer einzigen Nachrichtenkette aktualisiert werden. Der Aufwand bleibt also bei $|E|$ Nachrichten.

Topologie speichern und verarbeiten

In einer JSON Datenstruktur wird die Topologie des Dokuments festgehalten. Als Schlüssel dient die ID des Dokuments und als Wert ein Verweis auf sein nächstes Geschwister bzw. sein erstes Kind (vgl. Abbildung 2.3):

```
{
  "root": {
    "next": "",
    "firstChild": "front matter"
  },

  "front matter": {
    "next": "body matter",
    "firstChild": "sec a"
  },

  "sec a": {
    "next": "par a",
    "firstChild": ""
  },
  ...
}
```

Die (flache) Reihenfolge der Dokumentelemente kann mit dieser Datenstruktur, mit Tiefensuche in die Kinder und anschließende Breitensuche in die Geschwister, bestimmt werden. Diese Information ist wichtig für die Benutzeroberfläche, um das Dokument in der richtigen Reihenfolge anzuzeigen. Weiß die Benutzeroberfläche einmal die Dokumentelement-Reihung, reichen einfache Deltas über die Hierarchieänderungen aus. Der Algorithmus ist exemplarisch als JavaScript Code im REPO/TODO verfügbar. Der Root Akteur enthält eine entsprechende Scala Implementierung dieses Algorithmus.

Code Generierung für Verweise

In Abschnitt 4.3.4 wurde bereits angesprochen wie das Vorgehen ist, um Verweise, die zwischen Dokumentelementen bestehen, aufzulösen.

4.5 Erreichter Funktionsumfang

Mit dem Prototyp ist es möglich eine Masterarbeit zu schreiben. Das heißt dass von der prinzipiellen Funktionalität alles da ist, um wissenschaftliche Texte zu verfassen. Durch die Möglichkeit Projektionen in andere Formate zu erhalten, kann auch LaTeX Code generiert werden, um (noch) fehlende Funktionalität (insb. Paginierung) auszugleichen. Jedoch hat LaTeX nur eine beschränkte Menge an „Dokumentelementen“ im Vergleich zum Prototyp – es fehlen nunmal die domänenspezifischen Inhalte. Wenn diese Benutzt werden sollen, müssen diese mehr oder weniger von Hand an LaTeX nachgereicht werden, z.B. in Form von statischen Bildern. Beispiel: Das Chemiemolekül wird auch als SVG-Bild bereitgestellt, damit LaTeX dieses verarbeiten kann, muss es aus der Datenbank geholt und in ein pdf konvertiert werden. Rein vom Konzept her ist es aber leicht möglich solche Aufgaben besser zu automatisieren. Beispielsweise holt ein Shell-Skript via curl das SVG-Bild aus der Datenbank und legt es in einen Ordner ab, damit der LaTeX-Compiler es verwenden kann. Es macht Spaß mit den Prototyp Dokumente zu erstellen und zu sehen, wie von Zauberhand gleichzeitig auch der passende LaTeX-Code generiert wird. Er bietet eine übersichtliche Darstellung des Dokuments und ist bereits einigermaßen bequem zu bedienen, insbesondere dank der Autovervollständigung. Die Basiskonzepte aus Abschnitt 4.2 konnten somit umgesetzt werden.

Am Anfang der Masterarbeit war noch nicht klar, ob das Vorhaben überhaupt gelingt. In dieser Phase ist ein erster Prototyp entstanden. Dieser hat gezeigt, dass das Konzept Potential hat, tragfähig zu sein. Ein zweiter Prototyp ist entstanden, mit einer deutlich klareren und verbesserten Architektur. In die zwei Prototypen ist insgesamt ca. 4 Monate Entwicklungsarbeit und Konzeption geflossen, es wurde agil gearbeitet, um verschiedene Konzepte „unbürokratisch“ ausprobieren zu können. Der zweite Prototyp hat die Konzepte immer klarer

werden lassen, so dass daraus eine gut strukturierte Theorie gefolgert und nachvollzogen werden kann. Aber die neuen Erkenntnisse zeigen auch, dass der Prototyp nicht an allen Stellen die Theorien konsequent umsetzt.

- Jedes Dokumentelemente repräsentiert ein Modell.
- Jedes Modell kann als DSL ausgedrückt werden. Textuell oder grafisch.
- Ein Modell kann in Form von Attributen (des Dokumentelements) gespeichert werden. Die Attribute entsprechen quasi der ausgewerteten DSL. Die DSL selbst wird auch als Attribut gespeichert.

Jedoch stößt der Prototyp an einigen Stellen auch auf seine Grenzen, gerade hinsichtlich der Performanz bei großen Dokumenten. Eine Vermutung ist, dass die eingesetzte JSON-Bibliothek, die intern sehr oft aufgerufen wird, nicht sonderlich schnell ist – im Fokus stand zunächst eine Bibliothek mit einer übersichtlichen und flexiblen API zu verwenden. Die Architektur selbst ist auch noch nicht modular genug, d.h. der Basis Aktor sowie Wurzel Aktor sind zu monolithisch. Wenn das Metamodell verändert wird, muss das System neu kompiliert werden. Und z.Z. kann nur ein Dokument geladen werden; eine Unterscheidung zwischen verschiedenen Benutzern gibt es auch noch nicht. Wenn die hier vorgestellten Konzepte in einer Produktiven Umgebung eingesetzt werden sollen, ist eine Neuauflage der Software empfehlenswert.

Jedoch hat der Prototyp das gezeigt, was er zeigen soll: Es ist möglich mit Akka Aktoren einen abstrakten Syntaxbaum zu bauen, welcher als Modellgrundlage für Dokumente dient. Zudem ist es sehr nützlich auch domänenspezifische Inhalte als spezialisierte Dokumentelemente zu modellieren. Dadurch erhält der Autor mehr Konsistenz und Semantik im Dokument – dadurch wird einem Dokument ermöglicht sich selbst zu einem gewissen Grad zu verifizieren und damit weniger fehleranfällig zu machen. Außerdem ist es sehr für einen Autor sehr bequem, wenn ein von ihm häufig verwendetes Modell als fertiges und sorgenfreies Dokumentelement vorliegt; welches gleichzeitig Visualisierungen in Veröffentlichungsqualität liefert.

4.5.1 Codestatistik

Zirka 4 Monate intensive Entwicklung am Prototypen, danach nur noch kleine Bugfixes oder Einführung neuer Dokumentelemente. Insgesamt wurden über 150 git commits gemacht. 41 Testspezifikationen bestehend aus rund 1600 Code Zeilen werden dem Aktorsystem abverlangt. Die resultierende Software besteht aus: Zirka 500 Zeilen Code gehen an die in JavaScript geschriebene Benutzeroberfläche (ohne Template- oder Style-Code). Aus zirka 1000 Zeilen Code besteht das Aktorsystem, ohne Dokumentelement-Definitionen und ohne Server Code. Es wurden insgesamt sechs Releases erstellt, welche in diversen (internen) Vorträgen vorgeführt wurden.

Literaturverzeichnis

- [Aho u. a. 2007] AHO, Alfred V. ; LAM, Monica S. ; SETHI, Ravi ; ULLMAN, Jeffrey D.: *Compilers. Principles, Techniques, and Tools*. 2. Aufl. Boston : Pearson/Addison Wesley, 2007. – ISBN 978-0-321-48681-3
- [Anderson u. a. 2010] ANDERSON, J. C. ; LEHNARDT, Jan ; SLATER, Noah: *CouchDB: The Definitive Guide*. Sebastopol : O'Reilly Media, Inc., 2010 <http://guide.couchdb.org/index.html>. – ISBN 978-1-449-38293-3
- [ANSI/NISO 2012] ANSI/NISO: Z39.96-2012: *JATS. Journal Article Tag Suite*. National Information Standards Organization, 2012. – ISBN 978-1-937522-10-0
- [Bühler 1934] BÜHLER, Karl: *Sprachtheorie. Die Darstellungsfunktion der Sprache*. Jena : Fischer, 1934
- [DIN 1983a] DIN: 1421: Abschnitte, Absätze, Aufzählungen. In: *Gliederung und Benummerung in Texten* (1983), Januar
- [DIN 1983b] DIN: 1422-1: Gestaltung von Manuskripten und Typoskripten. In: *Veröffentlichungen aus Wissenschaft, Technik, Wirtschaft und Verwaltung* (1983), Februar
- [DIN 1984] DIN: 1422-3: Typographische Gestaltung. In: *Veröffentlichungen aus Wissenschaft, Technik, Wirtschaft und Verwaltung* (1984), April
- [DIN 1986] DIN: 1422-4: Gestaltung von Forschungsberichten. In: *Veröffentlichungen aus Wissenschaft, Technik, Wirtschaft und Verwaltung* (1986), August
- [Edwards 2003] EDWARDS, Stephen A.: *Abstract Syntax Trees*. Version: 2003. <http://www1.cs.columbia.edu/~sedwards/classes/2003/w4115f/ast.9up.pdf>. Vorlesungsskript der Columbia University

- [Gomolka u. Humm 2013] GOMOLKA, Andreas ; HUMM, Bernhard: Structure Editors: Old Hat or Future Vision? Version: 2013. http://dx.doi.org/10.1007/978-3-642-32341-6_6. In: MACIASZEK, LeszekA. (Hrsg.) ; ZHANG, Kang (Hrsg.): *Evaluation of Novel Approaches to Software Engineering* Bd. 275. Springer Berlin Heidelberg, 2013. – ISBN 978-3-642-32340-9, 82-97
- [Hewitt 2010] HEWITT, Carl: Actor Model for Discretionary, Adaptive Concurrency. In: *CoRR* abs/1008.1459 (2010)
- [Hewitt u. a. 1973] HEWITT, Carl ; BISHOP, Peter ; STEIGER, Richard: A Universal Modular ACTOR Formalism for Artificial Intelligence. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1973 (IJCAI'73), 235-245
- [Hodapp u. a. 2013] HODAPP, Sven ; BOGER, Marko ; ZIMMERMANN, Marc: Vergleich von interner und externer DSL-Technologie zur Entwicklung eines Textsatzsystems zur automatischen Dokumentengenerierung. (2013), Jan. <http://dx.doi.org/10.5281/zenodo.10940>. – DOI 10.5281/zenodo.10940
- [Ludewig 2002] LUDEWIG, Jochen: Modelle im Software Engineering - eine Einführung und Kritik. In: GLINZ, Martin (Hrsg.) ; MÜLLER-LUSCHNAT, Günther (Hrsg.): *Modellierung* Bd. 12, GI, 2002 (LNI). – ISBN 3-88579-342-3, S. 7-22
- [Malissa 1971] MALISSA, Hanns: Automation in und mit der Analytischen Chemie IV. In: *Fresenius' Zeitschrift für analytische Chemie* 256 (1971), Nr. 1, 7-14. <http://dx.doi.org/10.1007/BF00537872>. – DOI 10.1007/BF00537872. – ISSN 0016-1152
- [North 2006] NORTH, Dan: Behavior Modification. In: *BETTER SOFTWARE MAGAZINE* (2006), März. <http://dannorth.net/introducing-bdd/>
- [Roestenburg u. a. 2014] ROESTENBURG, Raymond ; BAKKER, Rob ; WILLIAMS, Rob: *Akka in Action*. Early Access Edition (Version 14). Birmingham : Manning Publications Company, 2014. – 475 S. <http://www.manning.com/roestenburg/>. – ISBN 9781617291012
- [Shotton u. Peroni 2014] SHOTTON, David ; PERONI, Silvio: *Doco, the Document Components Onology*. <http://purl.org/spar/doco>. Version: 2014
- [Sikos 2011] SIKOS, Leslie F.: *Web Standards. Mastering HTML5, CSS3, and XML*. 1. Aufl. Berkeley, CA : Apress, 2011. <http://dx.doi.org/10.1007/978-1->

4302-4042-6. <http://dx.doi.org/10.1007/978-1-4302-4042-6>. – ISBN 978-1-4302-4041-9

[Stachowiak 1973] STACHOWIAK, Herbert: *Allgemeine Modelltheorie*. Springer Vienna, 1973 <https://archive.org/details/Stachowiak1973AllgemeineModelltheorie>. – ISBN 3-211-81106-0

[Strahinger 1998] STRAHINGER, Susanne: Ein sprachbasierter Metamodellbegriff und seine Verallgemeinerung durch das Konzept des Metaisierungsprinzips, 1998 (Modellierung 98: Proceedings des GI-Workshops in Münster, März 1998. Hrsg.: K. Pohl (u.a.))

[Typesave Inc. 2014] TYPESAVE INC.: Akka Scala Documentation. Release 2.3.3. (2014), Mai

[Voelter 2013] VOELTER, Markus: *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. CreateSpace Independent Publishing Platform, 2013 <http://dslbook.squarespace.com>. – ISBN 9781481218580

[Wikipedia 2013] WIKIPEDIA: *Annotation* — *Wikipedia, Die freie Enzyklopädie*. <http://de.wikipedia.org/w/index.php?title=Annotation&oldid=124988519>. Version: 2013. – [Online; Stand 22. Juli 2014]