HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Wissenschaftliche Dokumente mit domänenspezifischen Inhalten: Entwicklung eines allgemeinen Modells für wissenschaftliche Dokumente umgesetzt als browserbasierter Editor

Sven Hodapp

Konstanz, 22. August 2014

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:	**Wissenschaftliche Dokumente mit domänenspezifischen Inhalten: Entwicklung eines allgemeinen Modells für wissenschaftliche Dokumente umgesetzt als browserbasierter Editor**
Masterkandidat:	Sven Hodapp, Hohentwielstraße 2, 78247 Hilzingen
Firma:	Fraunhofer-Institut für Algorithmen und Wissenschaftliches Rechnen SCAI
1. Prüfer:	Prof. Dr. Marko Boger
2. Prüfer:	Dr. Marc Zimmermann
Schlagworte:	Wissenschaftliches Manuskript, Textverarbeitung, Modelltheorie, Taxnomomie, Software Engineering, Projectional Editing, Domain-Specific Languages, Domain Modelling
Ausgabedatum:	1. März 2014
Abgabedatum:	22. August 2014

Kurzreferat

Aktuelle Textverarbeitungsprogramme haben gerade bei der Erstellung von wissenschaftlichen Dokumenten große Defizite, da dort oft formale Modelle, wie z.B. Chemie-Moleküle, aus dem jeweiligen Fachgebiet (Domäne) benutzt werden müssen. Heutige Systeme können mit diesen Modellen nicht direkt interagieren, was oft zu Inkonsistenzen im Dokument führt. In dieser Arbeit wird der Prototyp eines Dokumenteditors entwickelt, welcher domänenspezifische Dokumentelemente einführt, und so formale Modelle direkt bei der Bearbeitung des Dokuments verwendbar macht. Die Dokumentelemente (z.B. Absätze, Abbildungen etc.) können auch domänenspezifische Inhalte repräsentieren, z.B. ein Molekül. Wenn ein Dokumentelement auf ein anderes verweist, kann direkt auf Attribute des darunterliegenden Modells zugegriffen werden; bspw. auf den Namen eines Moleküls. Der Prototyp beweist seine Praxistauglichkeit anhand von vier konkreten Anwendungsfällen. Aus dem Prototyp ist eine allgemeine Theorie über den inneren Aufbau von Dokumenten entwickelt worden und tangiert die Bereiche (Meta-)Modellierung, Projektionseditoren, Semiotik und Taxonomie. Es gibt vielfältige Weiterentwicklungsmöglichkeiten, wie Ausbau zur Open Access Platform mit erweiterten Autorenfunktionen, Zentrale für Format-Transformationen oder Literate Programming IDE.

Abstract

Current word processors have deficits for creating academic writings, because they use often formal models like chemical molecules of a specific domain. But current systems don't interact with these models, which often leads to inconsistent documents. In this work a document editor prototype was developed, which introduces domain specific document elements to enable direct usage of such models within the document. Document elements (e.g. paragraphs, figures, etc.) can represent domain specific content, such as a molecule. If one document element refers to another one, it can directly access model attributes of the other one, like the name of the molecule. The prototype proofes it's practial benefit by four different use cases. With the help of the prototype it was possible to retrieve general theories about the inner construction of documents, it covers the areas: (meta) modelling, projectional editing, semiotics and taxonomy. It is possible to envolve this thoughts to build a open access platfom with extended authoring abilities, central for format transformations or literate programming IDE.

Ehrenwörtliche Erklärung

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

(1) meine Masterarbeit mit dem Titel

Wissenschaftliche Dokumente mit domänenspezifischen Inhalten: Entwicklung eines allgemeinen Modells für wissenschaftliche Dokumente umgesetzt als browserbasierter Editor

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, 22. August 2014

(Unterschrift)

Inhaltsverzeichnis

Abbildungsverzeichnis

Glossar

Abstrakter Syntaxbaum Datenstruktur die Programmcode auf abstrakte Weise repräsentiert.

Aktor Entitäten mit nebenläufiger Rechenfähigkeit, die über Nachrichtentausch kommunizieren.

AST Abstract Syntaxtree bzw. abstrakter Syntaxbaum.

Axiom „Axiome sind Aussagen, die in der Domäne immer wahr sind“ (Drachenfels, 2013, S. 3-4).

Baum Ein zusammenhängender kreisfreier Graph.

Dokumentelement Grundlegende Bestandteile des inneren Aufbaus von Dokumenten. Beispiele hierfür sind Kapitel, Absätze oder Abbildungen etc.

Domäne Bezeichnung eines Fachgebietes bzw. Wissensgebietes.

Domäneneditor Ein grafischer Editor der es ermöglicht ein Modell eines spezifischen Fachgebiets zu manipulieren.

DSL Domain Specific Language. Eine Programmiersprache die auf eine spezifische Anwednungsdomäne zugeschnitten ist.

Glossar „Liste von Begriffen mit Erklärungen“ (Drachenfels, 2013, S. 3-4).

Graph Eine Struktur die aus einer Menge von Knoten und Kanten besteht. Die Graphentheorie ist ein Gebiet der diskreten Mathematik.

Konsistenz Ein (wissenschaftliches) Dokument ist dann konsistent, wenn seine Inhalte insich stimmig bzw. widerspruchsfrei sind. Wenn ein Text Bezug zu einem Detail eines Domänenmodell nimmt, sich dieses Detail jedoch ändert, aber im Text noch immer zum ursprünglichen Detail verwiesen wird, dann hat das Dokument eine Inkonsistenz.

LaTeX Sammlung an TeX-Makros zum Textsatz von wiss. Dokumenten, insbesondere im Bereich Mathematik, Informatik oder Physik.

Metamodell Ein Modell über Modelle.

Modell Reduziert ein komplexes Universum auf eine überschaubare Welt.

Ontologie „Glossar mit Axiomen und beliebigen Relationen“ (Drachenfels, 2013, S. 3-4).

PDF Portable Document Format. Ein standardisiertes Format für (paginierte) Dokumente.

Projektionseditor Editiert Programmcode direkt auf der Ebene des abstrakten Syntaxbaums eines Programms.

Refactoring Wenn ein Programmierer einen Programm-Quellcode überarbeitet, damit dieser verständlicher oder leichter wartbar wird. Der Ablauf des Programms wird dadruch nicht verändert.

Reichhaltige Verweisungen Gehen über normale Verweisungen hinaus. Sie sind ausführbarer Programmcode und greifen auf (u.U. berechnete) Attribute eines Dokumentelements zu.

Semantic Web Auch Web 3.0 genannt: Vision das Web um semantische Beziehungen anzureichern, so dass die Bedeutung von Informationen auch von Computern genutzt werden kann.

Semiotik Wissenschaft die sich mit den Gesetzmäßigkeiten von Sprache beschäftigt.

SVG Scalable Vector Graphics. Ein standardisiertes Bildformat für Vektorgrafiken.

Taxonomie „Glossar mit hierarchischer Einordnung der Begriffe in Kategorien“ (Drachenfels, 2013, S. 3-4).

Template Eine programmierbare Vorlage die mit statischen und dynamischen Bestandteilen ausgestattet ist. Die dymanischen Inhale werden aus einer Datenstruktur eines Programms bezogen und sind daher variabel.

Thesaurus „Taxonomie mit zusätzlichen Synonym-/Ähnlichkeits-Beziehungen“ (Drachenfels, 2013, S. 3-4).

Vorwort

Naturwissenschaftliche Dokumentation hat mich persönlich schon immer fasziniert. Wie sich jedoch herausstellt, steckt hinter den Textwerkzeugen wie z.B. Microsoft Word nicht sonderlich viel „Magie“. Nahezu alles muss immer noch von Hand eingetippt bzw. Schaubilder eingefügt werden – das ist für meinen Geschmack zu manuell und fehlerträchtig.

Daher wollte ich mich in meiner Bachelorarbeit (Hodapp u. a., 2013) mit diesem Thema auseinandersetzen. Das die Evolution solcher Werkzeuge noch nicht abgeschlossen ist, wurde mir am Fraunhofer-Institut für Solare Energiesysteme ISE während meines Praxissemsters klar: Es herrscht Unzufriedenheit mit den momentanen Werkzeugen zur Generierung des Jahresberichts der Abteilung. In meiner Freizeit habe ich Ideen gesammelt und diese hielten sich hartnäckig bis zur Abschlussarbeit. Am Fraunhofer-Institut für Algorithmen und Wissenschaftliches Rechnen SCAI stieß ich auf offene Ohren; Marko Boger (HTWG) und Marc Zimmermann (SCAI) betreuten die Abschlussarbeit.

Die Bachelorarbeit machte klar, dass es noch Potenzial gibt – ich wollte daran weiterarbeiten. Ich war also auf der Suche nach neuen Impulsen und Ideen, um daraus eine Masterarbeit zu machen. Prof. Boger zeigte mir Markus Völters *mbeddr*, eine IDE mit DSLs für eingebettete Systeme. Mbeddr kann auch LaTeX-Dokumentation generieren; ähnlich der Methodik meiner Arbeit. Der Clou ist jedoch, dass die IDE ein *projectional editor* ist – der fehlende Impuls! Im Master-Seminar durfte ich mich damit auseinandersetzen, was mich ermuntert hat, projectional editing für meine Masterarbeit aufzugreifen.

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, wie z.B. Verweise im Text auf die korrekten Benummerungen von Abbildungen, im Dokument zu pflegen und konsistent zu halten. Hierfür hat insbesondere LaTeX eine solide Lösung, jedoch stößt man auf der Metaebene, beim LaTeX-Code selbst, wieder auf das Problem: Der Autor definiert ein Label und verweist an anderen Stellen darauf. Wird das Label nun irgendwann umbenannt, z.B. weil die Überschrift angepasst wurde, muss es an allen referenzierten Stellen auch umbenannt werden – manuelles Refactoring des LaTeX-Quelltextes ist notwendig.

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 vom 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 sauberes *Metamodell*. Das heißt ein Modell, welches eine Struktur vorschreibt, wann welche Dokumentelemente zulässig sind. Eine solche Struktur kann jedoch einem Autor dabei helfen, ein Dokument (wie z.B. ein Bericht der Fraunhofer Gesellschaft) gemäß definierter Vereinbarungen[1] aufzubauen. Er braucht also kein Wissen über die genaue Vereinbarung und kann dennoch ein formell korrektes Dokument erstellen. Solche Vereinbarungen können z.B. ein Corporate Design oder eine erlaubte Aneinanderreihung von Dokumentelementen sein. Weiterhin können aus einem solchen Metamodell ganze Dokumentklassen abgeleitet werden: All jene Dokumente, die dem gleichen Metamodell übergeordnet sind, gehören somit ganz formal zur gleichen Dokumentklasse. Beispiele für solche Klassen sind: Abschlussarbeit, 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, Inkostistenzen ins Dokument einzubringen.

(ii) Mangel an Domänenwissen: Dokumente „verstehen“ die Konventionen, Konzepte oder Modelle einer Wissenschaft nicht.

[1] Man kann argumentieren, dass Formatvorlagen von z.B. Microsoft Word auch eine Art Metamodell für ein Dokument sind. Jedoch formalisieren diese nicht, wie eine insgesamte Dokumentklasse korrekt aufgebaut sein soll. Zudem können Autoren sehr leicht, durch manuelle Änderungen, unbeabsichtigt an den Formatvorlagen vorbei arbeiten.

(iii) Mangel an Semantik: Bedeutungen einzelner Bestandteile eines Dokuments werden oft nicht explizit sichtbar bzw. überhaupt erst verfügbar gemacht.

(iv) Mangel an Metamodellierung: Es fehlen[2] formale Vereinbarungen, die eine Dokumentenklasse ausmachen und dem Benutzer bei der Strukturierung helfen.

Das wirft die folgende Frage auf: Kann man eine Textverarbeitung erschaffen, welche die o.g. Defizite aufhebt?

1.3 Stand der Technik

Es folgt eine kurze Vorstellung einiger Softwaresysteme die zum Erstellen von wissenschaftlichen Dokumenten eingesetzt werden, oder Forschungsprojekte die bestimmte Aspekte der Wissenskonservierung verbessern wollen.

1. Textverarbeitungen,
2. Textsatzsysteme,
3. SageTeX,
4. SALT,
5. OCA,
6. Wolfram CDF.

(1) Sehr weit verbreitet sind Microsoft Word, Apache OpenOffice oder Google Docs. Dabei handelt es sich um WYSIWYG-Editoren für Dokumente. Sie sind für gewöhnlich als Projektionsedtioren für Markup realisiert.

[2] Oder gehen nicht weit genug, wie z.B. im Falle von LaTeX. Denn eine Dokumentklasse von LaTeX überprüft nicht, ob eine bestimmte Aneinanderreihung von Dokumentelementen zulässig ist. Beispiel: Sinnloserweise dürfte das Inhaltsverzeichnis auch mitten im Dokument vorkommen. Das heißt, dass die Semantik einer Dokumentklasse von LaTeX nicht unterstützt wird.

(2) Weit verbreitet in den Naturwissenschaften ist TeX bzw. LaTeX (Sammlung an TeX-Makros). Ein Dokument wird in Form von TeX-Quellcode geschrieben und kann zu einer PDF-Datei kompiliert werden. Zur Klasse der Textsatzysteme gehören auch Softwaresysteme wie Adobe InDesign, welche sich stärker den gestalterischen Details widmen, als der inhaltlichen Strukturierung von wissenschaftlichen Texten.

(3) SageTeX[3] ist Teil der Open-Source-Mathematik-Software Sage. Sage ist eine freie Alternatve zu Magma, Maple, Mathematica und Matlab. Das SageTeX-Paket ermöglicht dem Autor, direkt aus dem LaTeX-Code heraus Programmbefehle an Sage zu schicken. Das ermöglicht das Einbetten von Sage-Berechnunen und Sage-Plots.

(4) „Semantically Annotated LaTeX for Scientific Publications“ ist ein Forschungsprojekt (Groza u. a., 2007), mit dem Ziel LaTeX-Code um semantische Annotationen zu erweitern. Es werden Meta-Informationen in die generierten PDFs eingebettet, zur Nutzung durch Semantic-Web-Technologien.

(5) „One Click Annotator“ ist ein Forschungsprojekt (Heese u. a., 2010), welches einen WYSIWYG-Web-Editor entwickelt hat, um Texte mit RDFa-Annotationen anzureichern.

(6) Computable-Document-Format[4] ist ein Dokumentformat für dynamisch generierte, interaktive Inhalte basierend auf der wissensbasierten Wolfram-Language. Es handelt sich um ein proprietäres Format, welches zum Betrachten von Dokumenten zumindest den Wolfram-Player (oder ein spezielles Browser-Plugin) benötigt.

1.4 Methode

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

[3] Deutsche Dokumentation von SageTeX auf der Sage-Projektseite: http://www.sagemath.org/de/html/tutorial/sagetex.html

[4] Wolfram CDF Produktwebseite: http://www.wolfram.com/cdf/

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 und die Praxistauglichkeit der Konzepte zu prüfen. Die Entwicklung eines Protoypen macht bei dieser Masterarbeit also Sinn.

- Es können theoretische Konzepte aufgezeigt und empirisch validiert[5] werden.
- Es ist leicht möglich Anwendungsfälle auszuprobieren, um die praktische Tauglichkeit der Konzepte zu verifizieren[6].

1.5 Anforderungen

Bei den Anforderungen an den Prototypen kann man schon konkreter in Richtung Implementierung blicken. Hier die Basisanforderungen, um die Problemstellungen zu lösen:

Die kleinste Einheit im System ist das Dokumentelement. Beispiele für Dokumentelemente sind: Abschnitte, Absätze, Tabellen, Abbildungen, Fußnoten, etc. Mehr dazu in Abschnitt 2.1.2.

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.

[5] Funktioniert das Konzept richtig?

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

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, gibt es folgende Nebenbedingungen:

- 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 Problem des manuellen Refactoring abgefedert 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.

Einige dieser Anforderungen zielen implizit auf die Implementierung eines Projektionseditors (siehe Abschnitt 2.3) und haben sogar Potenzial für ein Dokument-Query-Interface. Das heißt das Dokument im Ganzen oder ein Dokumentelement im Speziellen, kann befragt werden, einen bestimmten Sachverhalt herauszufinden. Beispielsweise können so komfortabel alle Abschnitte nach ihrer Benummerung abgefragt werden, um ein Inhaltsverzeichnis zu bauen.

1.6 Idee

Im Vordergrund stand die Idee, meine Bachelorarbeit (Hodapp u. a., 2013) so weiter zu entwickeln, dass Dokumente in einem Projektionseditor (mehr dazu

in Abschnitt 2.3) verfasst werden können und mit domänenspezifischen Dokumentelementen interagiert werden kann. Die Bachelorarbeit hat sich damit beschäftigt, (1) ein Dokument im Browser zu layouten und zu paginieren und (2) eine interne DSL für Dokumente zu entwickeln, welche eine Datenstruktur zur Verarbeitung für den Layouter ausgibt.

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

Grundsatz der Idee ist, dass jedes Dokumentelement auf einen 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 Referenz[7] 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 Projektionseditor (siehe Abschnitt 2.3) umgesetzt werden.

Der hier mit den Aktoren umgesetzte AST könnte als „reaktiver AST“[8] bezeichnet werden, da er dank der Aktoren (quasi gratis) besondere Eigenschaften erhält:

- Verteilbarkeit, Fehlertoleranz und massive Parallelität. Der 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 Datanbankzugriffe oder schnelle Reaktionszeiten auf Änderungen.
- Möglichkeit zum Query-Interface ausgebaut zu werden. Das Dokument

[7] Über eine solche Aktor Referenz können Nachrichten an andere Aktoren übermittelt werden.

[8] „Merriam-Webster definiert reaktiv als ‚immer bereit auf ein Stimulus zu reagieren‘, z.B. sind Komponenten immer ‚aktiv‘ und jederzeit bereit auf Ereignisse zu reagieren.“ Frei übersetzt aus dem Reactive Manifesto: http://www.reactivemanifesto.org/

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.

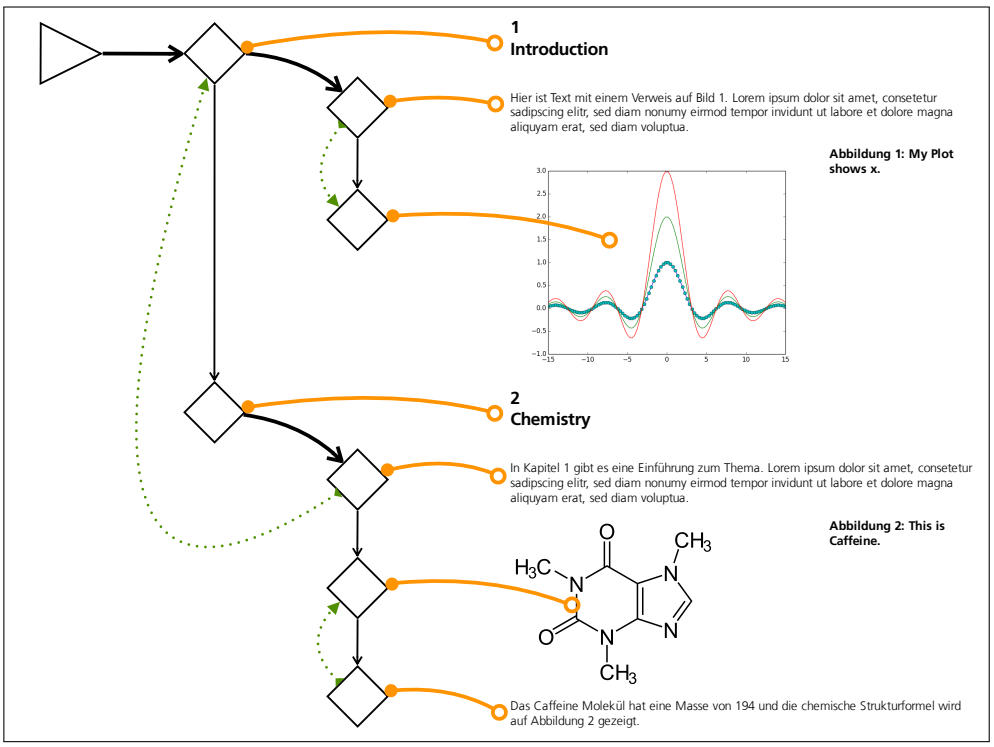


Abbildung 1.1: Idee eines Modells für Dokumente. Ein Dokument fängt an seiner Wurzel (Dreieck) an. Jeder Aktor (Raute) repräsentiert ein Dokumentelement, dargestellt durch die orangene Linie. Die Aktoren kennen ihr erstes Kind (fetter Pfeil) und ihren unmittelbaren Nachfolger (dünner Pfeil) und spannen damit einen Baum auf. Die Aktoren können Nachrichten (grüne Pfeile) austauschen, um Verweise, wie z.B. Benummerungen, aufzulösen. Werden alle Pfeile als Kanten verstanden, so handelt es sich um einen gerichteten Graphen.

1.7 Szenario

Ein motivierendes Szenario ist auf Abbildung 1.2 visualisiert. Hierbei kann der Autor zunächst sein Dokument in einer abstrakteren Ansicht editieren. Dadurch werden die Beziehungen, und deren Bedeutungen, zwischen den einzelnen Dokumentelementen klar. Zudem kann er im Falle des Moleküls auch

auf weitere Informationen, die das Molekül-Dokumentelement anbietet, zugreifen. Das hilft den Autor dabei, sein Dokument konsistent zu halten.

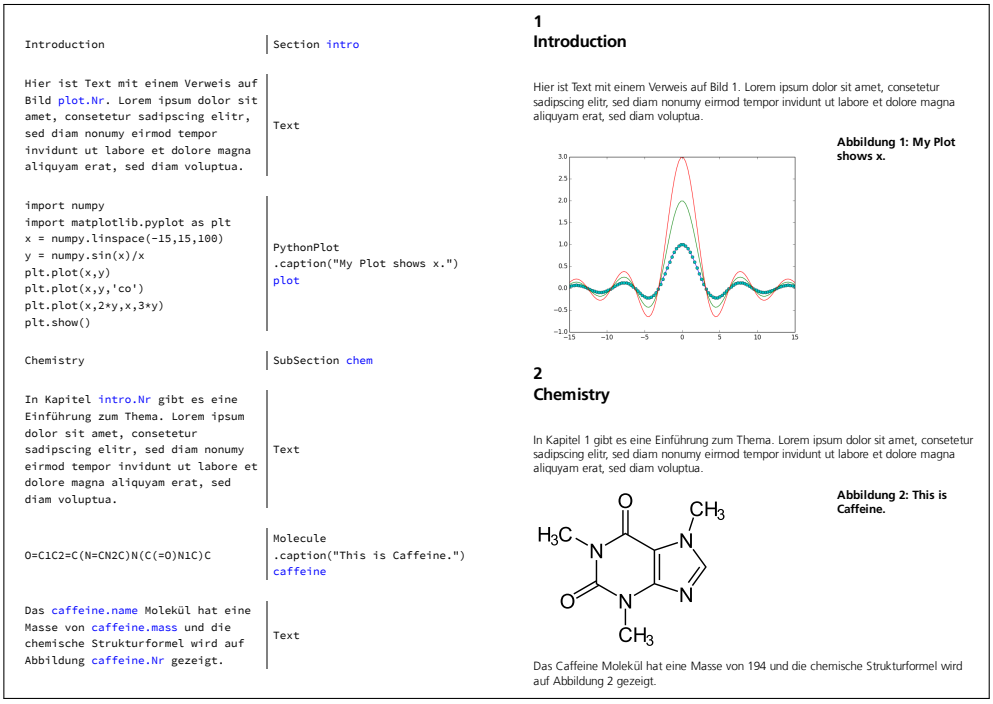


Abbildung 1.2: Motivierendes Szenario. Auf der linken Seite ist die Editier-Ansicht zu sehen, dort werden die einzelnen Dokumentelemente und ihre Beziehungen (Verweise untereinander) ersichtlich. Die Verweise zwischen den Dokumentelementen sind blau markiert. Auf der rechten Seite ist die durch ein Satzsystem gesetzte Ansicht, welche aus der Editier-Ansicht hervorgeht.

1.8 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 verifizieren[9] und damit konsistenter machen?
- Bibliotheks- und Informationswissenschaften: Gibt es eine allgemeingültige Taxonomie für wiss. Publikationen?

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

Kapitel 2

Theorie der Dokumente

In diesem Kapitel wird überlegt, was der modelltheoretische Hintergrund von (wiss.) Dokumenten im Allgemeinen ist und wie sich domänenspezifische Inhalte darauf auswirken. Die hier erarbeiteten Theorien sind das Destillat dessen, was aus dem Prototyp gelernt wurde.

Zunächst wird der Modellbegriff erläutert, um anhand dessen ein geeignetes Modell für Dokumente abzuleiten. Daraus folgt, dass eine natürliche Repräsentation für Dokumente der Graph aus der Graphentheorie ist (siehe Abschnitt 2.1). Eine spezielle Form von Graphen ist der abstrakte Syntaxbaum, welcher in der Informatik als abstrakte Repräsentation für Programmcode dient (siehe Abschnitt 2.2). Auf diese Weise kann ein Dokument auch als Programm aufgefasst und verarbeitet werden. Projektionseditoren geben einem Programmierer (oder hier Autor) die Möglichkeit, direkt einen abstrakten Syntaxbaum (grafisch) zu editieren (siehe Abschnitt 2.3).

Um ein Modell formal besser fassbar zu machen, ist es dienlich ein Metamodell darüber zu stellen (siehe Abschnitt 2.4). Das Metamodell beschreibt aus welchen Entitäten und Beziehungen ein konformes Modell besteht. Begriffe aus der Semiotik, der allgemeinen Lehre der Sprachen, lassen erkennen, dass jedes einzelne Dokumentelement wiederum ein Modell einer spezifischen Domäne enthält (siehe Abschnitt 2.5). Die einzelnen Elemente des inneren Aufbaus von wissenschaftlichen Dokumenten, können in eine Taxonomie ein-

geordnet werden. So werden verschiedene Typen von Dokumentelementen hierarchisch angeordnet und klassifiziert. Erlaubte Verschachtelungen bzw. Aneinanderreihungen von Dokumentelementen kann formal spezifiziert werden. Diese Spezifikation beschreibt, zusammen mit einer entsprechenden Taxonomie, sehr prägnant eine spezifische Dokumentklasse (z.B. wiss. Bericht, EU-Patent). Mehr dazu in Abschnitt 2.6.

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? Warum sollte eine Wissenschaftlerin bzw. ein Wissenschaftler seine Modelle nicht direkt im Dokument verwenden können? Es folgt zunächst die genaue Definition eines Modells.

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.

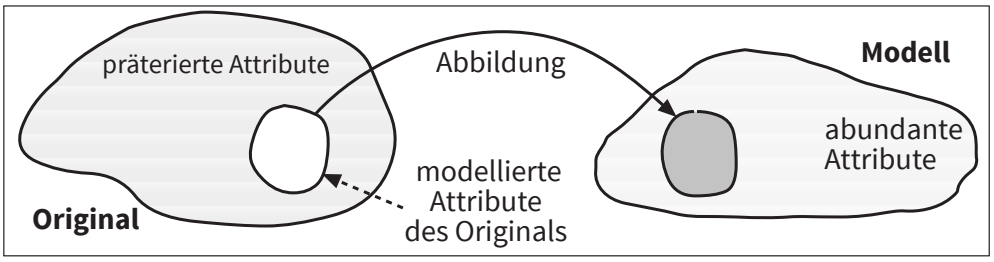


Abbildung 2.1: Schematische Zeichnung der Original-Modell-Beziehung. Entnommen aus (Ludewig, 2002, S. 9).

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 abudantes Attribut, welches erst innerhalb des Modells hinzugefügt 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, bei dem er sich ausschließlich auf die Dokumentelemente konzentrieren kann. Das Modell kann präskriptiv sein, d.h. aus dem Modell kann wieder ein Original entspringen.

Im Allgemeinen ist ein natürliches Modell für ein Dokument ein gerichteter Graph. Die Knoten des Graphen entsprechen den Dokumentelementen. Die Topologie des Dokuments entspricht einer baumartig gerichteten Graph-Struktur, denn das innere eines Dokuments oder einer Rede ist hierarchisch aufgebaut. Jedoch gibt es in Dokumenten immer wieder Sprünge die auf andere Stellen verweisen – ein klassisches Beispiel dafür ist „... siehe Abbildung x für ...“. Das bedeutet es gibt zwei Arten von Kanten: Kanten die die Topologie des Dokuments aufspannen, diese bilden einen gerichteten Baum. Kanten die Verweise zwischen Dokumentelementen darstellen, diese bilden einen gerichteten Graph. Wie man sich eine Abbildung von einem Dokument auf einen Baum bzw. Graph vorstellen kann, ist auf Abbildung 1.1 visualisiert. Die beiden vorkommenden Kanten-Arten werden in der Grafik vereint dargestellt.

Hier wird der abstrakte Syntaxbaum als Modell für das Dokument dienen, indem jedes Dokumentelement als ein Knoten des abstrakten Syntaxbaums aufgefasst wird. Mehr zum abstrakten Syntaxbaum in Abschnitt 2.2. Der abstrakte Syntaxbaum repräsentiert also die Hierarchie bzw. Gliederung des Dokuments. Zudem wird angenommen, dass die Knoten des abstrakten Syntaxbaumes miteinander kommunizieren können. Auf diese Weise können die Verweisungen unter den einzelnen Dokumentelementen aufgelöst werden. Dies wird auf Abbildung 1.1 veranschaulicht.

Beweis

- Abbildungsmerkmal: Der abstrakte Syntaxbaum („das Modell“) ist z.B. ein Abbild eines gedruckten Buches, da Dokumentelemente als Knoten des Syntaxbaums abgebildet werden. Die Hierarchie des inneren Aufbaus des Buches wird beibehalten, was wichtig ist, um die argumentative Kette des Textes zu erhalten.
- 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.
- Pragmatisches 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 (Compiler) als Zwischenrepräsentation eines Quellcodes generiert wird. Er repräsentiert die hierarchische syntaktische Struktur eines Programms. Aus einer solchen Zwischenrepräsentation wird schlussendlich das Zielprogramm (z.B. Maschinencode) generiert.

Während der Syntaxanalyse (parsing) werden Syntaxbaum-Knoten erstellt, welche wiederum signifikante Programmkonstrukte repräsentieren. (ebd.) 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 nur 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.)

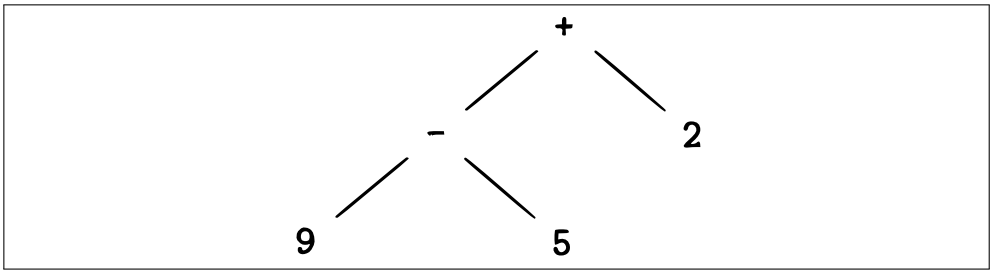


Abbildung 2.2: Abstrakter Syntaxbaum für den Ausdruck 9-5+2. Entnommen aus (Aho, 2007, S. 70).

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. Dort kann jeder Knoten einen ersten Kind-Knoten und oder ein direkt nachfolgenden Geschwister-Knoten besitzen. Diese Struktur ist notwenig, um in der Datenstruktur eine explizite Reihenfolge der Knoten vorzugeben. In der reinen mathematischen Notation ist keine explizite Reihenfolge kodiert, dort werden die Kanten als Menge von Tupeln angegeben – die Reihenfolge spielt dort keine Rolle. Für Dokumente ist es aber wichtig, dass die hierarchische Gliederung erhalten bleibt, somit muss die Knoten-Reihenfolge ein first-class-citizen sein, d.h. nicht nur implizit vorkommen. Und genau das gilt auch für abstrakte Syntaxbäume, denn diese repräsentieren Quellcode eines Programms, welcher als eine Art von Computer-Literatur angesehen werden kann, vgl. die Idee von Literate-Programming (Knuth, 1984). Programmquellcode unterliegt also gewissermaßen den gleichen Gesetzmäßigkeiten, wie auch Dokumente die in natürlicher Sprache verfasst sind.

Wird dieser Gedankte weitergedacht, wird klar, dass der Umgang mit Pro-

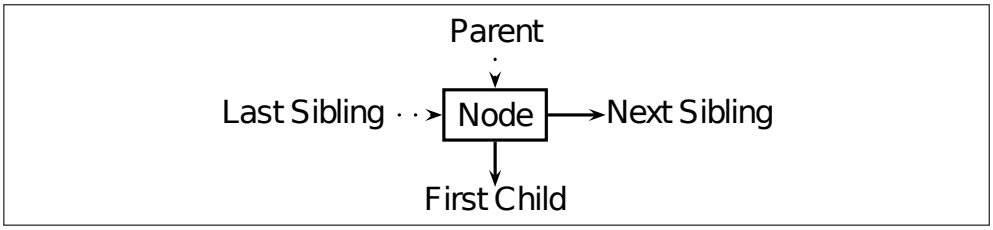


Abbildung 2.3: Generellste Implementierung eines AST. Entnommen aus (Edwards, 2003).

grammcode eine sehr ähnliche Evolution durchschritten hat, wie es die „natürlichen“ Dokumente getan haben. In der Antike wurden Reden in „einer Wurst“ aufgeschrieben; das ist Vergleichbar mit der Goto-Programmierung (Assembler). Irgendwann wurden Abstände zwischen einzelnen Gedankengängen eingeführt, die Absätze; das ist Vergleichbar mit der Einführung von Prozeduren oder Funktionen in Programmiersprachen bzw. strukturierte Programmierung. Später kamen noch Kapitel hinzu; das ist Vergleichbar mit der objektorientierten Programmierung, in der mehrere zusammengehörige Funktionen in Objekten zusammengefasst werden.

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 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, das 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 also 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 gesamte 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 eine Baumstruktur aufbauen, um die syntaktische Korrektheit des Programms zu prüfen. Das heißt der Übersetzer kann anhand des 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 parserbasierten Editoren. Eine kurze schematische Grafik (Abb. 2.4) verdeutlicht den Unterschied zwischen Quellcode-Editoren und Projektionseditoren.

Im englischen[1] Sprachraum gibt es mehrere Bezeichnungen: structure editor, structured editor oder projectional editor. Man kann diese mit Strukturedi tor oder Projetkionseditor übersetzten.

Die Idee von Projektionseditoren ist nicht neu, bereits 1971 hat Wilfred J. Hansen ein System entwickelt, welches auf solch einem Ansatz beruht. (Gomolka u. Humm, 2013, S. 91) Jedoch haben sich zur damaligen Zeit diese Editor-Formen nicht durchgesetzt, wegen der unzureichenden Benutzerfreundlichkeit. (ebd.) 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 sogar erhöht.

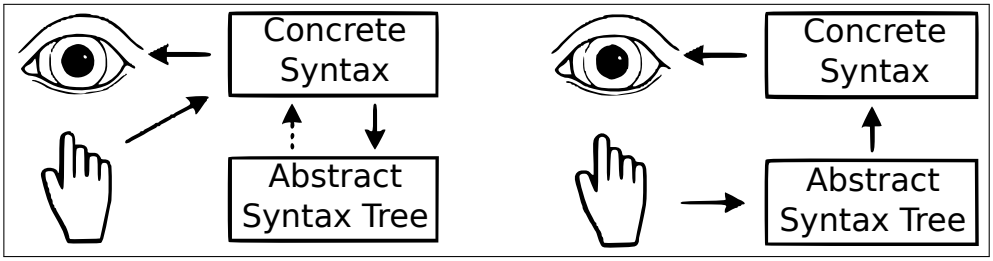


Abbildung 2.4: Parserbasierte Editierung / Quellcode-Edior (links) verglichen mit projektionsbasierter Editierung / Projektionseditor (rechts). Im parserbasierten Ansatz sieht und bearbeitet der Programmierer die konkrete Syntax eines Programms. Ein Übersetzer gewinnt aus der konkreten Syntax einen AST. Beim projektionsbasierten Ansatz sieht der Programmierer eine (von vielen) konkrete Syntax, und bearbeitet jedoch unmittelbar den AST. Im Allgemeinen kann aus dem AST eine konkrete Syntax gewonnen werden, das kann z.B. Quellcode einer Programmiersprache sein, aber auch direkt Maschinencode. Grafiken aus (Voelter, 2013, S. 68) entnommen.

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

2.3.1 Übertragung auf den Prototypen

Der hier vorgestellte Prototyp ist augenscheinlich ein Projektionseditor. Sein Basiskonzept ist, Dokumente als ein Modell wahrzunehmen. Und das Modell eines 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, ermöglicht das ganz andere Repräsentationen bzw. konkrete Syntaxen des vorliegenden abstrakten Syntaxbaumes. 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. (Strahringer, 1998, S. 1)

(Strahringer, 1998) hat versucht den Metamodellbegriff anhand der Sprachstufentheorie der Logik, durch Übertragung auf die Modellierungswelt, zu prägen. (Strahringer, 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 OMG[2] 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.

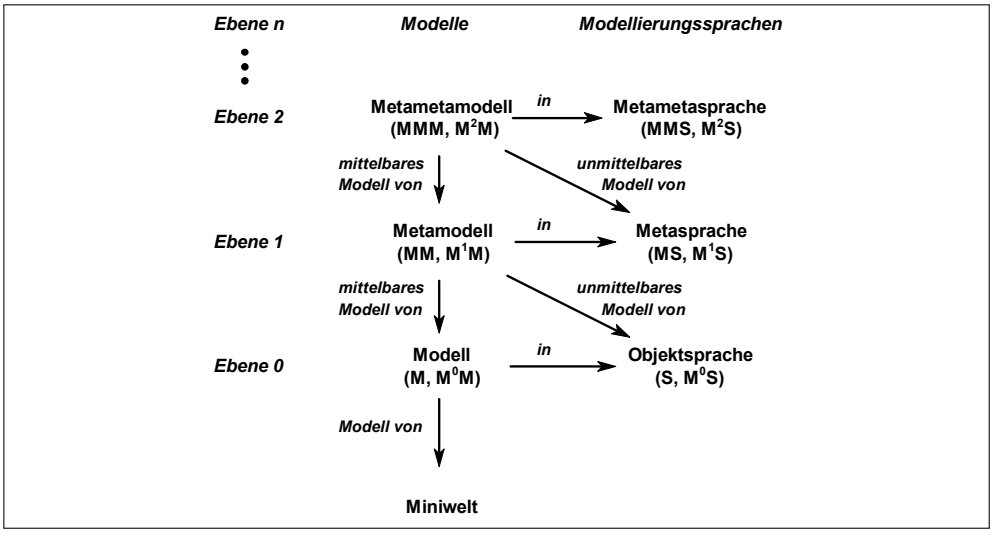


Abbildung 2.5: Der sprachbasierte Metamodellbegriff. Die „Miniwelt“ entspricht dem Original. Grafik entnommen aus (Strahringer, 1998, S. 3).

[2] Die OMG (Object Management Group) hat MOF (Meta Object Facility) als Standard (ISO/IEC 19508) veröffentlicht.

2.4.1 Metamodell des Dokumentmodells

Das Metamodell schafft auf einer abstrakteren Ebene Klarheit über das Modell, denn es beschreibt aus welchen Entitäten und Beziehungen ein konformes Modell besteht. Auf Abbildung 2.6 ist das eigentliche Modell visualisiert. Dieses soll vom Metamodell beschrieben werden.

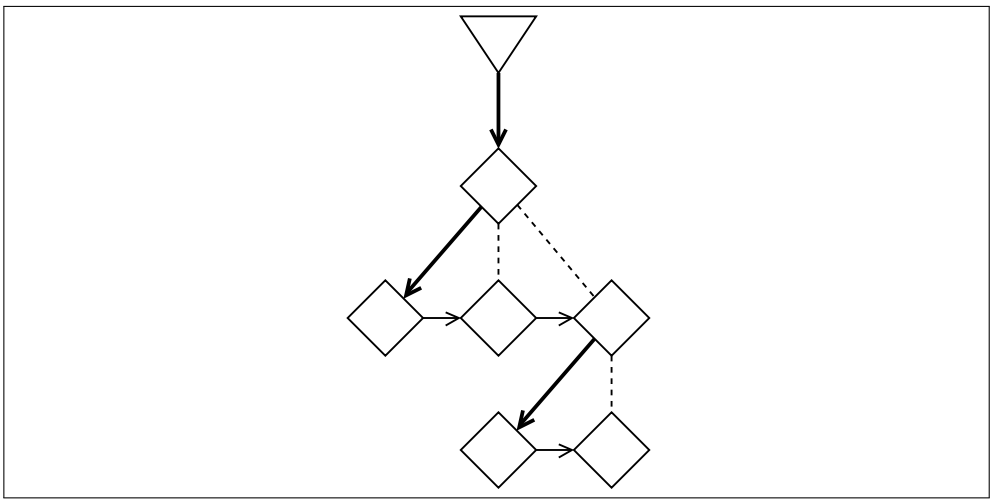


Abbildung 2.6: Das hier verwendete allgemeine Dokumentmodell. Das Dreieck symbolisiert die Wurzel. Die Rauten symbolisieren jeweils einen Aktor bzw. ein Dokumentelement. Die stark gezeichneten Pfeile symbolisieren das erste Kind, die gestrichelte Linie symbolisiert die zugehörigen anderen Kinder. Die schwach gezeichneten Pfeile symbolisieren die nächsten Geschwister. Dadurch wird die Hierarchie des Dokuments aufgespannt.

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 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 Topologie-Informationen, daher kennt er alle im Dokument vorhandenen Basis-Aktoren. Jeder Basis-Aktor kennt auch seine zugehörige Wurzel, welche er bei ggf. anstehenden lokalen Topologie-Veränderungen benachrichtigen muss.

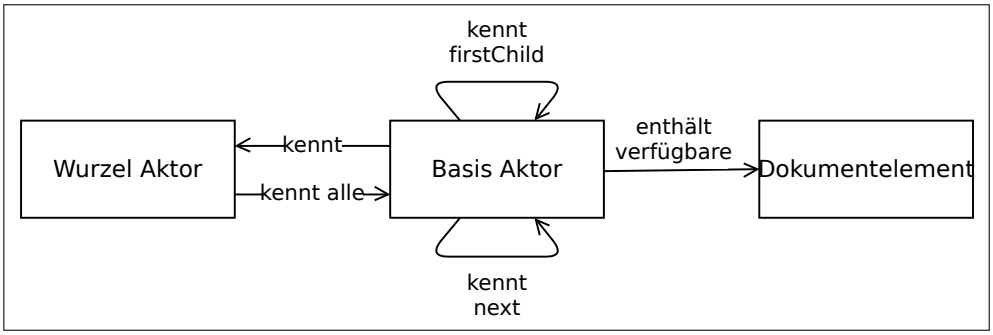


Abbildung 2.7: Das Metamodell des Dokumentmodells.

Beweis

Eine konkrete Modellinstanz wird in der zugehörigen Objektsprache beschrieben, siehe Abbildung 2.5. Beispiel: Mit Scala als Objektsprache wird ein Modell, das konkrete Scala-Programm, beschrieben. 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 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- bzw. 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. ▪

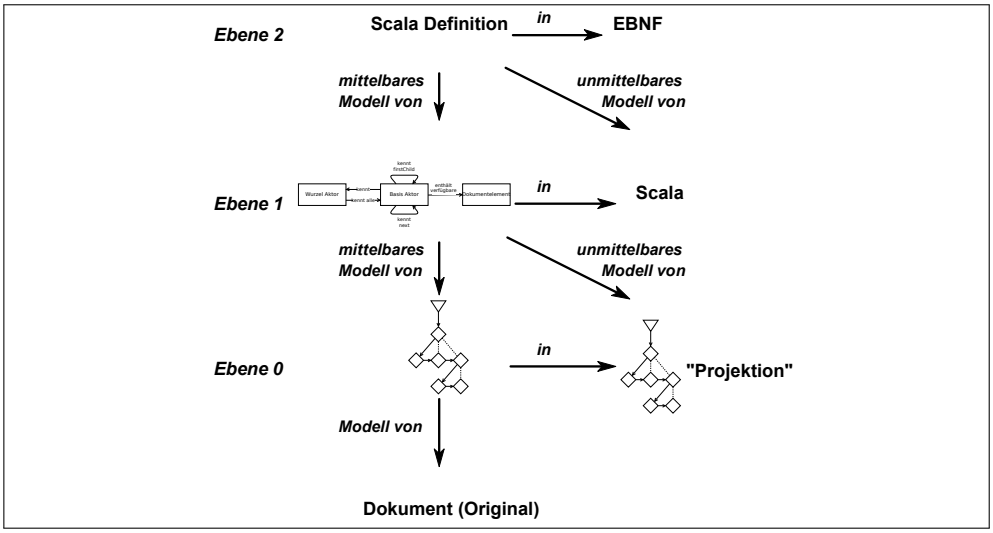


Abbildung 2.8: Die Modellebenen des Dokumentmodells, verdeutlichen die Zusammenhänge zwischen Dokumentmodell und Dokumentmetamodell. Grafik nach (Strahringer, 1998). Die erweiterte Backus-Naur-Form (EBNF) kann als Metametamodell dienen – denn mit EBNF können Programmiersprachen beschrieben werden und 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.

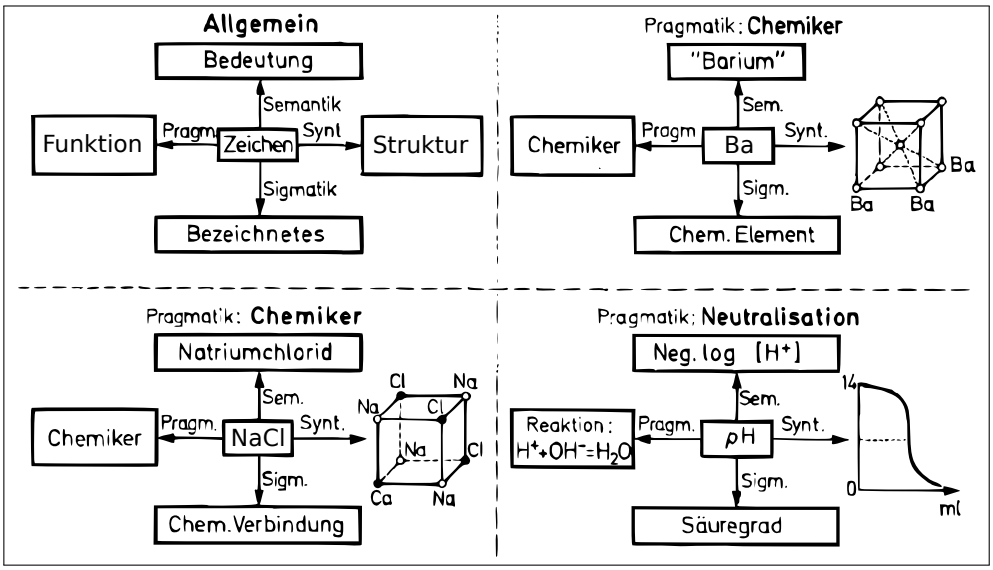


Abbildung 2.9: Die vier Hauptteile der Semiotik grafisch dargestellt mit Beispielen aus der analytischen Chemie. Allgemein: Visualisierung der Semiotik eines Zeichens. Beispiel „Ba": Ba ist ein Zeichen, dessen Bedeutung Barium ist und ein chemisches Molekül bezeichnet. Das ist zumindest dann der Fall, wenn dies von einem Chemiker gelesen wird. Strukturell ordnen sich Barium Moleküle als ein geordneter Kristall an. Analog ist es bei der chemischen Verbindung dargestellt als NaCl. Beispiel „pH": Im Kontext einer Neutralisierungsreaktion, hat das Symbol pH die Bedeutung der umgekehrten Logarithmusfunktion über die Konzentration der positiv geladenen Wasserstoffionen. Dies bezeichnet den Säuregrad einer Flüssigkeit. Strukturell kann es als Plot dargestellt werden. Visualisierung mit einigen Anpassungen aus (Malissa, 1971, S. 9).

2.5.1 Anwendung auf (wiss.) Dokumente

Diese vier Aspekte wirken auch implizit in jedem Text, da insbesondere wissenschaftliche 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, kann man beobachten, wie sich die Konzepte innerhalb eines Dokuments auswirken. Auf Abbildung 2.10 wird ein einzelnes Dokumentelement herausgegriffen. Das Dokumentelement ist spezialisiert auf den Umgang mit chemischen Strukturformeln. Die konkrete Instanz des Dokumentelements zeigt die chemische Verbindung Aspirin an – dies entspricht dem Zeichen. Das Zeichen ist also eine Projektion des abstrakten Syntaxbaumes. Das Aspirin wird mit einer DSL beschrieben – dies entspricht der Syntax. Der so vorgestellte Syntaxbegriff vertritt hier also das Domänenmodell. In der Chemie-Domäne hat das Zeichen die Bedeutung „Aspirin" und bezeichnet eine Strukturformel.

Hinter jedem Dokumentelement, sei es ein Abschnitt, Absatz, Tabelle oder eine chemische Strukturformel, steckt also ein entsprechendes Domänenmodell. Wird das Konzept weitergedacht, könnten die einzelnen Modell-Attribute der Dokumentelemente mit passenden Semantic-Web-Ontologien verknüpft werden. Vorteil hierbei wäre, dass kuriertes Wissen, welches im Dokument steckt, explizit sichtbar und benutzbar gemacht wird.

Das heißt, dass bei der Erstellung eines Dokuments, Dokumentelemente vom Autor entworfen werden können, welche Modelle der jeweiligen Wissenschaft enthalten. Das ermöglicht dem Autor deutlich strukturierter, formaler und konsistenter an die Dokumentation seins Projektes heranzugehen. Zudem

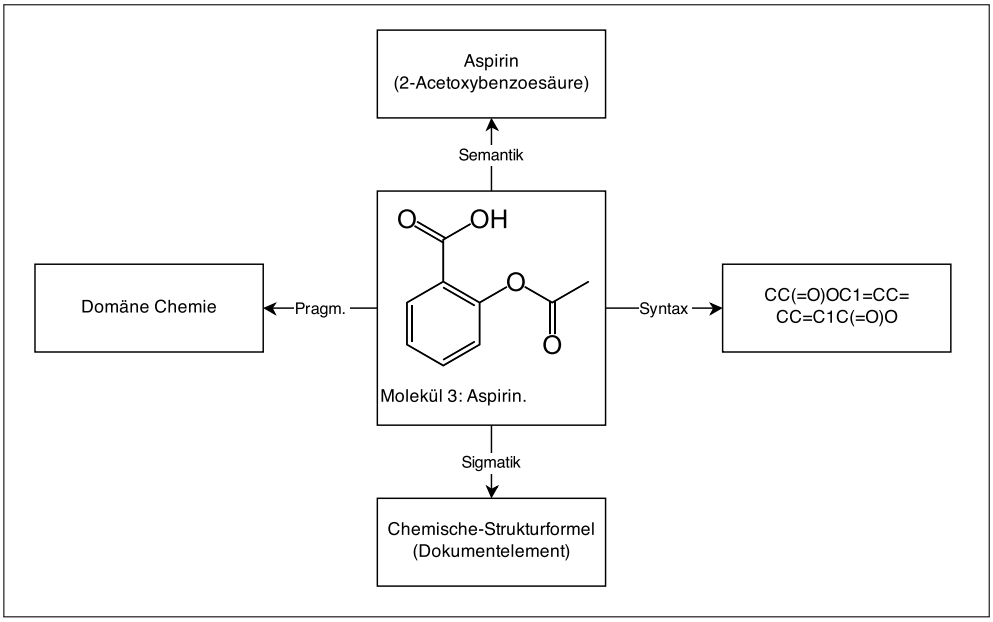


Abbildung 2.10: Semiotik am Beispiel der Instanz eines Chemischen-Strukturformel-Dokumentelements. Das ist analog übertragbar auf Dokumentelemente beliebigen Typs. Hier im Beispiel wird das Dokumentelement in seiner Gänze angezeigt, man könnte von einer Makro-Ansicht des Dokumentelements sprechen.

werden dem Leser dadurch explizite semantische Informationen bereitgestellt, welche direkt vom dahinterstehenden Modell stammen. Der Leser kann ggf. das Modell eines Autors in eigenen Dokumenten wiederverwenden. 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. Beispiel: Ein Absatz-Dokumentelement möchte auf die atomare Masse einer chemischen Verbindung, die im Strukturformel-Dokumentelement angezeigt wird, verweisen. Auf Abbildung 2.11 ist dieses Beispiel veranschaulicht. Im Beispiel: Der Autor schreibt einen Absatz z.B. „Die Masse beträgt (aspirinReference.mass) …“, dann wertet das System den Absatz aus, und es wird „Die Masse beträgt 180,16 u …“ angezeigt. Als „Zeichen“ dient also das Ergebnis des Programmausdrucks, welcher direkt auf die Modell-Attribute des chem. Strukturformel-Dokumentelements zugreifen kann. Der Programmausdruck ist Syntax, um das gewünschte Zeichen zu genieren. Hier bedeutet das Zeichen „Atomare Masseneinheit“. Die chemische Verbindung Aspirin ist das

Bezeichnete. Im Falle des Verweises ist es möglich die Semantik z.B. aus den Informationen des Typsystems des Programmausdrucks zu ziehen – also „aspirinReference.mass“ evaluiert zum Datentyp „AtomicMassUnit“. Schaut man auf Abbildung 2.10, fällt auf, dass die Sigmatik gleich der Semantik des Objekts, auf welches referenziert wird, ist.

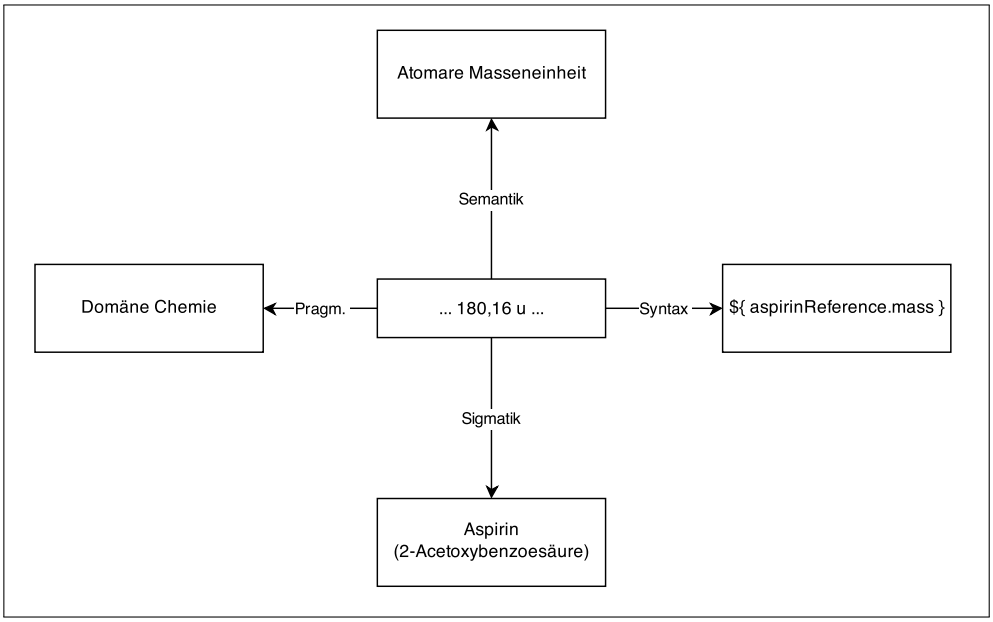


Abbildung 2.11: Semiotik am Beispiel eines Verweises auf ein chem. Strukturformel-Dokumentelement innerhalb eines Absatzes. Die Auslassungspunkte entsprechen den Rest des Absatzes, im Fokus steht der Verweis in Form der Maßeinheit „Atomare Masseneinheit“. Es wird lediglich auf ein spezifisches Attribut des Strukturformel-Dokumentelements zugegriffen, daher könnte man von der Mikro-Ansicht des Dokumentelements sprechen.

Wird dieses Konzept weitergedacht, scheint es sinnvoll Typsystem der Programmiersprache, die zur Auflösung der Verweisungen genutzt wird, mit passenden Onotologien zu verweben. Das ist nützlich, wenn mehrere Programmanweisungen kombiniert werden, so dass sie zu anderen Datentypen führen. Beispiel: Der Programmausdruck „Newton(10) / Meter(3)“ hat als Resultat den Typ NewtonMeter. Damit wäre es möglich, Verweisungen mit Semantik auszustatten, obwohl diese so nicht durch das entsprechende Dokumentelement vorgesehen bzw. modelliert ist. Das erspart dem Autor manuelle Annotationen!

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. Heutige Textverarbeitungen 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 ein Glossar 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 andere Publikationen ihre Gültigkeit, wie z.B. Briefe, Leitartikel oder Buchkritiken. Es handelt sich also um ein formal gestaltetes Glossar.

(Shotton u. Peroni, 2014) stellt eine die Ontologie (im formalen OWL 2 Format) über einzelnen Komponenten, die in einem Dokument vorkommen können, zur Verfügung. Diese Ontologie heißt DoCO: Document Component Onology. 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.

In der Tabelle auf Abbildung 2.12 werden die verschiedenen Systematiken miteinander verglichen. Die erste Spalte enthält die selbst benannten Dokumentelement-Namen, die in quasi allen Quellen so oder so ähnlich vorkommen. Dabei handelt es sich um die wichtigsten Elemente, die eine wissenschaftliche Arbeit im Allgemeinen ausmachen. Je nach Dokumentklasse können Elemente hinzukommen oder wegfallen. Viele Elemente sind in den jeweiligen Systematiken noch spezifiziert, die hier jedoch der Einfachheit halber nicht aufgeführt sind, oder die dem hier vorliegenden Dokumentelement-Begriff nicht genügen; siehe dafür Abschnitt 2.6.1. Das NISO-Article-Authoring-Tag-Set besteht aus insgesamt 197 Tags. DoCO besteht aus insgesamt 53 Klassen. Da es sich bei der DIN-Systematik um eine nicht-formale Beschreibung handelt, gibt es einen sehr großen Interpretationsspielraum, ob es sich um ein Dokumentelement handeln könnte oder nicht. Die LaTeX-Klasse report.cls[3] ist eine generierte LaTeX-Quellcode-Datei. Dadurch, dass dort auch viele Hilfs-Makros dort definiert sind, kann nicht so einfach herausgelesen werden, ob es sich um ein Dokumentelement oder eine Hilfsfunktion handelt. Daher wurde hier der eigene Erfahrungsschatz mit dem Umgang der LaTeX-Report-Klasse genutzt.

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

2.6.1 Dokumentelemente

Dokumentelemente bilden das Grundgerüst eines jeden Dokuments. Eine Tabelle über die wichtigsten Dokumentelemente, die in wissenschaftlichen Texten vorkommen, ist auf Abbildung 2.12 zu sehen. 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.

[3] Quellcode der LaTeX-Report-Klasse: http://tug.ctan.org/macros/latex/unpacked/report.cls

Dokumentelement	DIN	NISO	DoCO	LaTeX report.cls
Root		<article>		\begin{document}
Front Matter	Titelei	<front>	front matter	
Body Matter		<body>	body matter	
Back Matter		<back>	back matter	
Part			part	\part
Chapter			chapter	\chapter
Section	Abschnitt	<sec>	section	\section
Subsection				\subsection
Subsubsection				\subsubsection
Paragraph	Absatz	<p>	paragraph	\paragraph
List	Aufzählung	<list>	list	\begin{itemize}
Quotation	Zitat	<disp-quote>	block quotation	\begin{quotation}
Figure	Bild	<figure>	figure	\begin{figure}
Table	Tabelle	<table>	table	\begin{table}
Formula	Formel	<disp-formula>	formula	\[...\]
Code		<code>		\begin{verbatim}
Document Title	Sachtitel	<article-title>	title	\title
Author	Autor	<name>, <bio>	~ list of authors	\author
Abstract	Kurzreferat	<abstract>	abstract	\begin{abstract}
Table of Contents	Inhaltsverzeichnis		table of contents	\tableofcontents
Bibliography	Literaturverzeichnis	<ref-list>	bibliography	\bibliography
List of Figures			list of figures	\listoffigures
List of Tables			list of tables	\listoftables
Emphasize	Auszeichnung	~ <italic>		\emph
Footnote	Fußnote	<fn>	footnote	\footnote
Marginal Note				
Reference Item		<ref>		~ BibTeX
...	...	...	...	...

Abbildung 2.12: Dokumentelemente im Vergleich. In der ersten Spalte sind die wichtigsten (selbst benannten) Dokumentelemente herausgegriffen, welche auch z.T. im Prototypen implementiert sind (daher englische Bezeichnungen). Die Dokumentelemente sind aus den verschiedenen Systematiken von DIN, NISO, DoCO und LaTeX (Dokumentklasse Report) entstanden und werden hier nochmals übersichtlich dargestellt. Die mit Tilde markierten Einträge treffen nur ungefähr das gemeinte Dokumentelement. Wenn das Dokumentelement in einer Systematik nicht vorkommt, wir das Feld leer gelassen. Es gibt in der jeweiligen Systematik u.U. noch mehr Dokumentelemente bzw. Bestandteile aus denen Dokumentelemente zusammengesetzt sind, welche hier der Einfachheit halber nicht modelliert wurden, aber in die theoretischen Überlegungen eingeflossen sind.

- 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. Container-Elemente geben somit die Gliederung vor. 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 computergestützte 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 ein 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) gemacht. 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 ansässig. 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 Dokumentelement) 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.

Was natürlich für die Vorgehensweise von (Shotton u. Peroni, 2014) und (ANSI/NISO, 2012) spricht, ist dass alles was in einer (Dokumenten) Programmiersprache ausgedrückt würde, auch als Dokumentelement dargestellt wird. Beispiel: Ein Abschnitt hat die (Modell-)Attribute Nummer und Titel. So kann Nummer und Titel jeweils als extra Dokumentelement auftreten, aus denen schlussendlich der Abschnitt besteht. Das macht hier aber nur begrenzt Sinn, da jedes Dokumentelement wiederum eine eigene (mini-)DSL kapselt, aus welcher die (Modell) Attribute gewonnen werden. Es ist also nicht unbedingt nötig, jedes dieser Attribute als extra Dokumentelement zu modellieren. Beispiel: Der Autor hat eine kleine DSL für Abschnitte, die es ihm ermöglicht den Titel zu definieren. Das Nummer-Attribut wird vom Abschnitt-Dokumentelement selbstständig bestimmt. Bei komplexeren Dokumentelementen bzw. Modellen ist die DSL entsprechend mächtiger.

2.6.2 Taxonomie

In der Tabelle auf Abb. 2.13 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. 2.13) 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, Softwaredokumentation, etc.), die möglicherweise einen (leicht) anderen Aufbau haben und somit ggf. andere Dokuementelemente 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 Textverarbeitungssystem welches die Taxonomie umsetzt, formalisiert.

2.6.3 Dokumentstruktur Matrix

Damit das Textverarbeitungssystem 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

abstract to concreter elements			attributes
Container	Start Point	Root	
	Matter	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
		Author	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	autor, year, title, ...

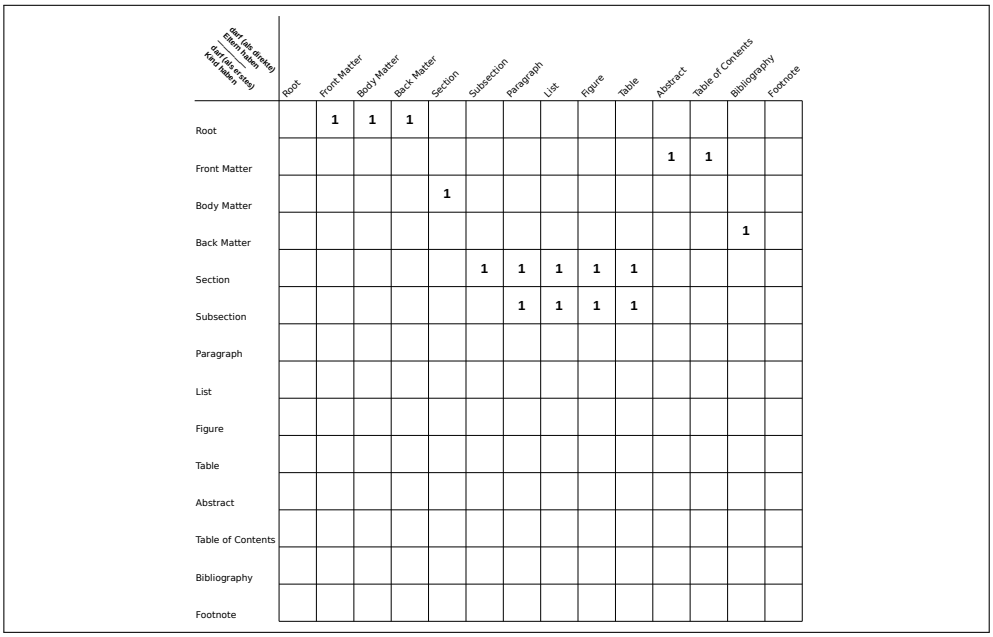
Abbildung 2.13: Taxonomie des inneren Aufbaus von wissenschaftlichen Dokumenten. Sortiert von abstrakt zu konkret. Anmerkung: Die Tabelle ist in englischer Sprache gehalten, da die Namen den Implementierungsnamen im Prototyp entsprechen.

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 2.14 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. 2.13. Diese Matrix definiert die erlaube Schachtelung der vorliegenden Dokumentklasse.

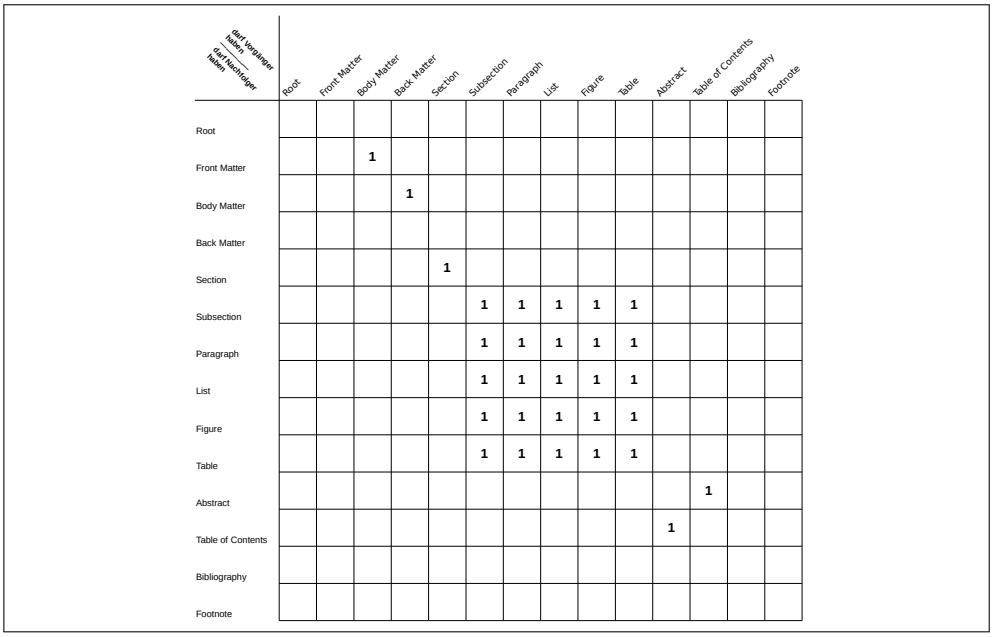


darf (als direkte) Eltern haben / darf (als direkte) Kind haben	Root	Front Matter	Body Matter	Back Matter	Section	Subsection	Paragraph	List	Figure	Table	Abstract	Table of Contents	Bibliography	Footnote
Root		1	1	1										
Front Matter											1	1		
Body Matter					1									
Back Matter									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

Abbildung 2.14: Stellt die erlaubten Eltern-Kind-Beziehungen zwischen den Dokumentelementen dar. Hier wird also definiert mit welchen Elementen die Hierarchie des Dokuments aufgespannt werden darf. Gelesen wird es z.B. von links nach oben: Body-Matter darf als Kind Section haben. Oder von oben nach rechts: Section darf als Eltern Body-Matter haben. Es fällt auf, dass die Matrix unterhalb der Determinanten leer ist. Das liegt daran, dass Container-Elemente keine Struktur-Elemente als Eltern haben dürfen.

Erlaube Abfolgen

Auf Abbildung 2.15 sind die erlauben Nachfolger-Vorgänger-Bezeihungen der hier beispielhaft vorliegenden Dokumentklasse als Matrix dargestellt. Hier wird also die Frage geklärt, welche Dokumentelemente aneinander gereiht werden dürfen, um ein wohlgeformtes Dokument der vorliegenden Dokumentklasse zu bilden.



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

Abbildung 2.15: Stellt die erlaubten Geschwister-Beziehungen (Nachfolger/Vorgänger) zwischen den Dokumentelementen dar. Hier wird also definiert welche Elementen aneinandergereiht werden dürfen. Es fällt auf, dass Struktur-Elemente eine Gruppe bilden, da sie nur als Blätter vorkommen.

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. Diese sollen einer Dokumentklasse zugeordnet werden, um sie in eine saubere oder aktuelle Vorlage zu überführen. Wenn nun ein Klassifikator die aufbereiteten PDFs in ihre Dokumentelemente zerlegt, kann über die Gesamtheit der im PDF gefunden 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.

Kapitel 3

Anwendungsfälle

Hier werden vier verschiedene Anwendungsfälle vorgestellt. Es gelingt dem Prototypen – ohne Modifikation – all diese Anwendungsfälle umzusetzen. Damit beweist der Prototyp die Praxistauglichkeit der hier vorgestellten theoretischen Konzepte und gibt Anwendern gleichzeitig ein Gefühl dafür, wie praktisch mit dem System gearbeitet werden kann.

Erster Anwendungsfall ist die Erstellung dieser Masterarbeit, d.h. der Prototyp wurde dazu verwendet diese Masterarbeit zu verfassen. Das zeigt, dass der Prototyp in der Lage ist, Texte zu verfassen, die wissenschaftlichen Ansprüchen genügen.

Zweiter Anwendungsfall ist die Nutzung des Prototypen als UIMA-CAS-Editor, d.h. der Prototyp wird dazu verwendet, Texte mit Annotationen zu versehen und diese darzustellen. Das zeigt, dass der Prototyp in der Lage ist auch mit dokument-spezifischen Metadaten umzugehen und eine sehr spezialisierte Form von Editoren umsetzen kann.

Dritter Anwendungsfall betrifft das Spray-Dokumentelement. Dieses kann formale Spray Softwaremodelle darstellen und bearbeiten. Im Text kann direkt auf einschlägige Attribute des Modells zugegriffen werden (z.B. Name einer Entität). Das zeigt, dass der Prototyp in der Lage ist ein wertvolles Werkzeug zur (technischen) Dokumentation von Softwaremodellen zu werden.

Vierter Anwendungsfall ist die Fähigkeit das Dokument in andere Formate zu transformieren. Konkret wird hier LaTeX-Code generiert, um die Masterarbeit aus dem ersten Anwendungsfall als paginiertes PDF zu erhalten. Durch hinzufügen von entsprechenden Templates können also beliebige andere Formate produziert werden. Das zeigt, dass der Prototyp in der Lage ist, durch einfache Anpassung von Templates, quasi beliebige Dokumenten-Formate zu produzieren.

3.1 Erstellung dieser Masterarbeit

In erster Linie soll das System Autoren bei der Erstellung hochwertiger wissenschaftlicher Berichte helfen. Das heißt im Allgemeinen sollen wissenschaftliche Dokumente diesem System verfasst werden können. Im Speziellen wurde diese Masterarbeit selbst mit dem, im Zuge der Arbeit entstandenen, Prototyp verfasst. Dies veranschaulicht die Tragfähigkeit der Konzepte, bzw. die Stabilität der momentanen Prototypen-Implementierung.

Diese Masterarbeit basiert also auf der hier entwickelten Theorie von (wiss.) Dokumenten. Das „Original“ (Dokument), welches Sie in Händen halten, ist also aus einem (formalen) Modell entstanden. Modelle können recht leicht auf bestimmte Aspekte hin untersucht werden: Die Masterarbeit besteht aus 479 Dokumentelementen, jedes Dokumentelement wurde im Durchschnitt 47 mal aktualisiert. Dazu zählen auch automatische Aktualisierungen, wie Benummerung von Kapiteln. Wird nur die neuste Version behalten, belegt das Dokument 1.8 MB in der Datenbank (ohne Bilder); werden alle Versionen behalten, belegt es knapp 113 MB. Es gibt 27 Fußnoten im Dokument, wobei diese z.T. an mehreren Stellen wiederverwendet werden. Die Aktoren senden untereinander rund 20000 Nachrichten, um dieses Dokument aufzubauen. Dabei werden Verweise aufgelöst, Benummerungen ausgerechnet, mit dem Browser kommuniziert und andere interne Abläufe abgewickelt – wobei manche Aktionen mehrfachen wechselseitigen Nachrichtentausch erfordern.

Damit das Dokument später auch der Hochschule bzw. Fakultät vorgelegt werden kann, muss es auch noch in einem printfähigen Format (insb. mit Pa-

ginierung) vorliegen. Da zur Zeit noch keine Paginierung in der Web-Ansicht erfolgt, muss auf ein anderes Werkzeug zurückgegriffen werden. Dank des abstrakten Syntaxbaumes ist das System fähig, das Dokument in verschiedenen Ansichten zu präsentieren – so kann auch LaTeX-Code generiert werden. Nähere Erklärungen zu solchen Transformationen in Abschnitt 3.4. Das heißt, die Print-Version der Masterarbeit wird via LaTeX erzeugt. In Zukunft sollte es 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 für Texte und FrontMatter (Titelei), BodyMatter, BackMatter zur Trennung der großen Bereiche. In der Masterarbeit wurde ausschließlich die BodyMatter, also der Hauptteil, und die BackMatter, also insb. Literaturverzeichnis, mit dem Prototyp geschrieben. Die Titelei (Deckblätter, Inhaltsverzeichnis, Glossar, Vorwort) wurde komplett mit LaTeX erstellt. Der Hauptteil und sowie die BibTeX-Literatur wurde komplett vom Prototyp generiert.

3.1.1 Szenario: Dokumente mit chemischen Strukturformeln

Der Gewinn ein solches System einzusetzen liegt in der Fähigkeit, domänenspezifische Inhalte zu verwenden. Beispielsweise in der Domäne Chemie werden immer wieder Strukturformeln von chemischen Verbindungen gezeichnet. Die Chemiker haben entsprechende Modelle von chemischen Verbindungen ausgearbeitet, so dass damit auch formal gerechnet werden kann. Und genau das soll das System: Auf Attribute und Funktionen eines formalen Modells einer spezifischen (Teil-)Domäne zugreifen und nutzen.

Zur Demonstration wurde ein Strukturformel-Dokumentelement entwickelt,

welches den in Strukturformel-Editor Ketcher[1] der Firma GGA Software enthält. Damit ist der Autor in der Lage in seinem Dokument direkt chemische Strukturformeln zu zeichnen, und dessen semantische Informationen zu nutzen. Der Moleküleditor produziert Molfiles[2] als Ausgabe- bzw. Austauschformat, dieses Molfile dient als ein (persistentes) Modell für eine chemische Verbindung. Das chem. Strukturformel-Dokumentelement kann zudem noch an eine Chemie-Softwarebibliothek, die Molfiles verarbeiten kann, angebunden werden. Eine solche Programmbibliothek kann weitere Informationen zur jeweiligen chem. Verbindung bereitstellen, und damit einen erheblichen Beitrag zur expliziten Semantik und Konsistenz des Dokuments beitragen.

In Abbildung 3.1 ist ein durch das Strukturformel-Dokumentelement generiertes Molekül (Modell) abgebildet. Wenn man darauf klickt, wird der Domäneneditor Ketcher geöffnet. Das funktioniert im gedruckten Dokument oder im statischen PDF nicht, daher ist es auch auf Abb. 3.2, als Bildschirmaufnahme dargestellt. Das Strukturformel-Dokumentelement, ist lediglich ein Wrapper[3] für Ketcher. Es verwendet die Ketcher-API, um ein SVG-Bild aus dem Molekül-Modell für die Visualisierung zu generieren, sowie das Molfile-Format, welches eine portable Modell-Repräsentation für chem. Verbindungen ist. Es handelt sich hierbei also nicht nur um ein simples Zeichenprogramm für chemische Verbindungen, sondern das Dokumentelement hat tatsächlich semantisches Wissen über das chemische Objekt. Daher kann der Autor auch davor bewahrt werden, nicht-konforme Modelle zu zeichnen. Zudem kann der Autor dieses Wissen nutzen, um sein Dokument konsistent zu halten. So kann ein Text, der die Verbindung beschreibt, direkt auf das dahinterliegende Modell der theoretischen Chemie zugreifen. Wird die chemische Verbindung bearbeitet, passen sich entsprechend markierte Textstellen unmittelbar an.

[1] Webseite des Ketcher Projekts: http://ggasoftware.com/opensource/ketcher

[2] Das Molfile der Firma MDL repräsentiert chemische Strukturen. Siehe „Repräsentation chemischer Strukturen“ unter http://de.wikipedia.org/wiki/Chemoinformatik

[3] „Wrapper (Software), ein Programm, das als Schnittstelle zwischen zwei anderen Programmen dient“. Aus Wikipedia http://de.wikipedia.org/wiki/Wrapper

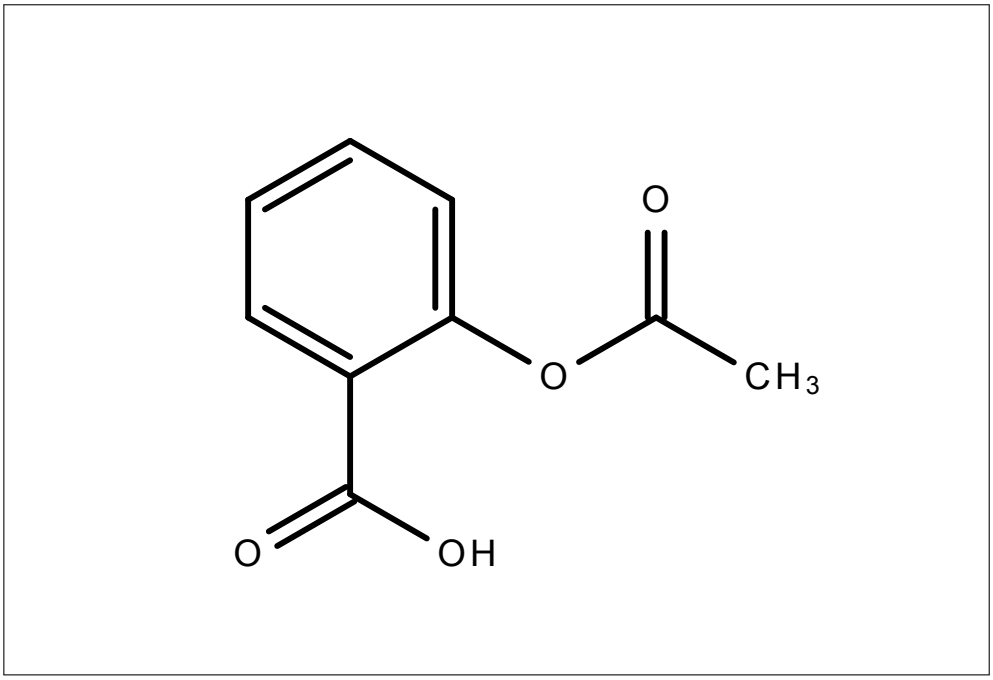


Abbildung 3.1: Generiertes Molekül Aspirin.

3.1.2 Erhoffter Nutzen

Die vorgestellten Konzepte ermöglichen es also, einem Dokument mehr explizite Semantik zu verleihen. Das ist nützlich für den Leser, da ihm zusätzliche Informationen bei Bedarf eingeblendet werden können – das beugt Verständnisprobleme vor. Nützlich ist es auch für Dienste die automatisiert entsprechende Dokumente analysieren – Wissensnetze wie das Semantic-Web, können die Informationen nutzen. Auch für den Autor hat es einen Sinn sich mit diesen Konzepten auseinanderzusetzen, denn es erspart ihm u.U. Schreibarbeit bzw. ärgerliche Inkonsistenzen. All das führt hoffentlich zur Anhebung der Qualität bei der wissenschaftlichen Dokumentation.

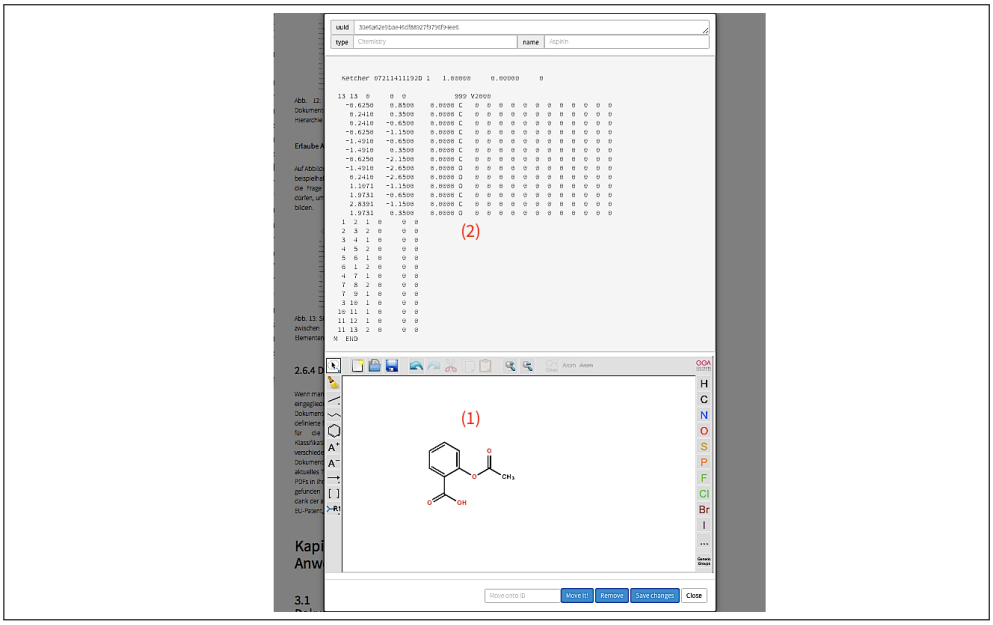


Abbildung 3.2: Bildschirmaufnahme der Editier-Ansicht des Strukturformel-Dokumentelements. Es ist der Domäneneditor „Ketcher“ (1) zu sehen, der gerade das Aspirin Molekül anzeigt und zum editieren einlädt. Der Editor ist nicht nur ein einfaches Zeichenprogramm, sondern er kennt seine Domäne und bietet daher entsprechende Semantik (siehe dazu die unterschiedlichen Farben für die chemischen Elemente die auch so im Molekül vorkommen). Zudem wird das Modell (2) des Aspirin Moleküls im Molfile-Format angezeigt. Das alles ermöglicht dem Autor, direkt mit dem chemischen Objekt zu interagieren.

3.2 UIMA-CAS-Editor

Apache UIMA[4] 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, sogenannte Annotationen, angereichert werden. Diese Annotationen sind dafür zuständig, einzelne Entitäten des Texts zu identifizieren, wie z.B. Personen, Orte, Organisationen etc. oder auch Relationen wie „arbeitet für“.

CAS[5] (Common Analysis System) Objekte bieten Zugriff auf das Typsystem, In-

[4] Webseite des Apache UIMA (Unstructured Information Management Architecture) Projekts: http://uima.apache.org

[5] UIMA CAS Objekt Dokumentation: http://uima.apache.org/downloads/releaseDocs/2.3.0-incubating/docs/api/org/apache/uima/cas/CAS.html

dexe, Iteratoren und Filter von UIMA, um auf (gewonnene) Informationen programmatisch zuzugreifen. Insbesondere ist es möglich, Annotationen mit dem CAS-Objekt zu erstellen. Vereinfacht ausgedrückt: 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 CAS-Editor würde ein Autor also gerne ein Dokument betrachten, und darin UIMA-Annotationen anzeigen lassen, diese bearbeiten oder neue hinzufügen.

3.2.1 Konzeptuelle Umsetzung

Für jeden Annotations-Typ (z.B. Person, Ort, etc.) muss 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 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 Metamodell ist quasi ein komplett eigenständiges Dokument, denn es hat eine eigene Wurzel. Daher bedient es sich der gleichen Infrastruktur, die auch das Hauptdokument nutzt. Weil das Metadokument und Hauptdokument in Kontakt stehen, haben sie 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 auf Attribute der Annotation kann via Verweis zugegriffen 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*[6] 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.

Die größte Patentschrift die mit dem Prototyp bisher geöffnet wurde hat 515 Dokumentelemente, wovon 146 Annotationen sind. Das für Demonstrationszwecke entwickelte Annotation-Dokumentelement enthält zusätzliche Informationen („das Modell“) die aus UIMA stammen. Konkret sind das, neben dem eigentlichen Inhalt der Annotation, die Attribute identifier, source und tool. So könnte ein Leser angezeigt bekommen, aus welcher Quelle die Information stammt bzw. welches Werkzeug die Annotation generiert hat.

[6] Webseite des UIMA-HPC Projekts: http://uima-hpc.org

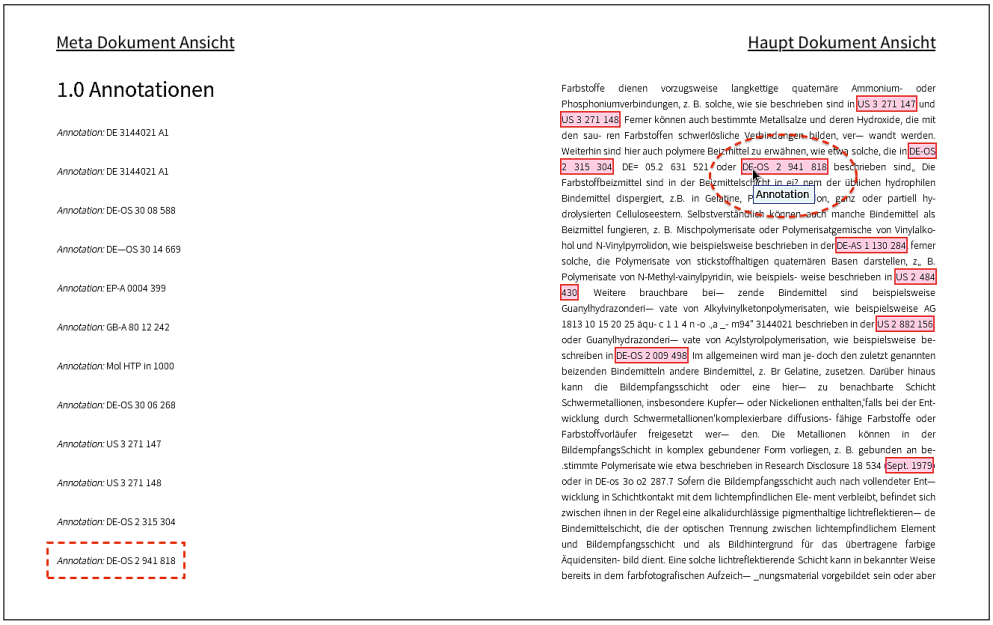


Abbildung 3.3: Entnommen aus Screenshots des Prototypen. Es wird dargestellt, wie Annotationen vom Metadokument in das Hauptdokument (über Verweisung) fließen können. Im Haupt Dokument werden bei „mouse over“ alle Annotationen vom gleichen Annotations-Typ „Annotation“ farbig hervorgehoben. Es ist möglich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

3.2.2 Erhoffter Nutzen

In erster Linie kann es sich für SCAI.BIO zu einem nützlichen Werkzeug entwickeln, um automatisch erzeuge Annotationen zu kurieren. Wenn der Prototyp ausgebaut wird, kann ein komfortabler CAS-Editor entstehen, welcher in der UIMA-Gemeinschaft sicher anklang finden wird. Gerade dadurch, dass das System auf einem allgemeinen Modell bzw. Theorie von Dokumenten fußt, bieten die Theorien einen wohl strukturierten Rahmen, um z.B. das Typsystem von UIMA sinnvoll zu erweitern. CAS-Objekten wird so ermöglicht, Dokumentelemente zu abstrahieren bzw. diese mit Attributen zu füllen. Anhand der Dokumentelemente die im Dokument vorkommen, kann eine Dokumentklasse geschlussfolgert werden; ein Dokument kann so auch automatisch verschlagwortet (für die Bibliografie) werden. Alte Dokumente können so automatisch zu semantischen Dokumenten aufgewertet werden.

3.3 Dokumentation von Spray-Modellen

Spray[7] 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, 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

[7] Webseite des Spray-Projekts: http://eclipselabs.org/p/spray/

Modell kann die Dokumentation das Gedankengebäude des Entwicklers sinnvoll unterstützen.

Ich persönlich denke, dass sich durch ein solches Dokumentationswerkzeug ein System wie Spray von anderen Lösungen absetzt und daher ein großes Erfolgspotenzial entwickeln kann. Oder um es etwas drastischer mit den Worten von Dick Brandon zu formulieren:

> Documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing. (Dick Brandon)

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 Webanwendung[8] mit REST-Schnittstelle vor. Der Prototyp selbst ist auch mit Web-Standards gebaut und kann daher recht mühelos mit „Spray Online“ interagieren.

Es wurde in Wrapper[9] entwickelt, der es ermöglicht, den Spray-Online-Editor in Form eines Dokumentelements anzubieten. 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

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

[9] „Wrapper (Software), ein Programm, das als Schnittstelle zwischen zwei anderen Programmen dient“. Aus Wikipedia http://de.wikipedia.org/wiki/Wrapper

Iframe 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.

In 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.

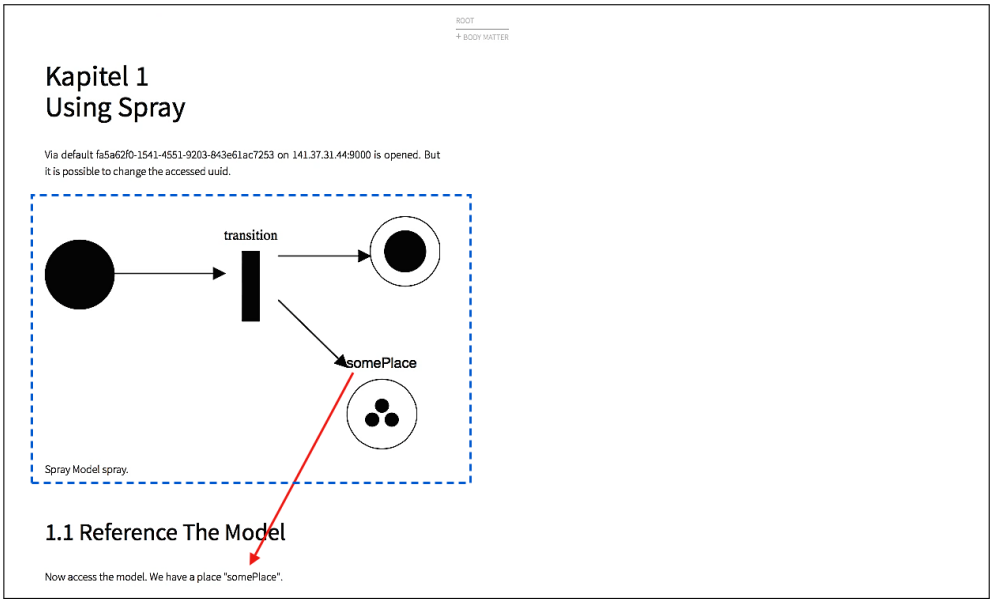


Abbildung 3.4: Im blau gestrichelten Kasten ist das Spray-Dokumentelement in seiner Anzeigeansicht zu sehen. Szenario: Der Name einer Stelle des Petrinetz-Diagramms soll im Text erwähnt werden (roter Pfeil).

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).

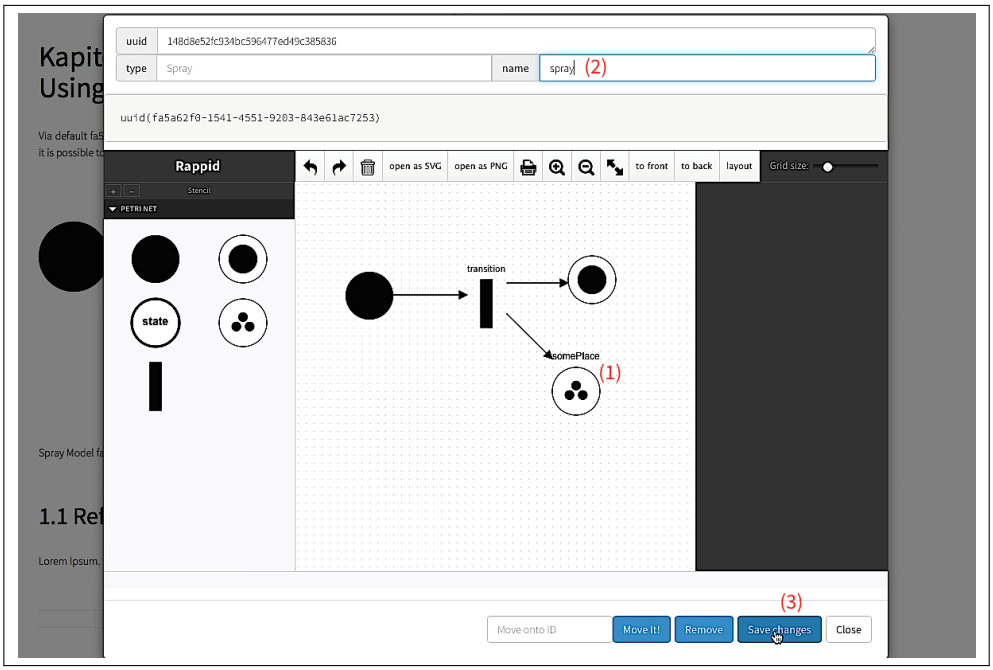


Abbildung 3.5: Ansicht zum Bearbeiten des Spray-Dokumentelement. In (1) ist der Domäneneditor, welcher das Modell in Diagrammform anzeigt und durch den Benutzer manipulierbar macht. Das Modell kann benannt werden für die spätere Verweisung im Text (2). Speichern der Veränderungen (3) veranlassen das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

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 her-

auspicken und ihren (aktuellen) Namen abfragen. Abbildung 3.4 zeigt das Resultat.

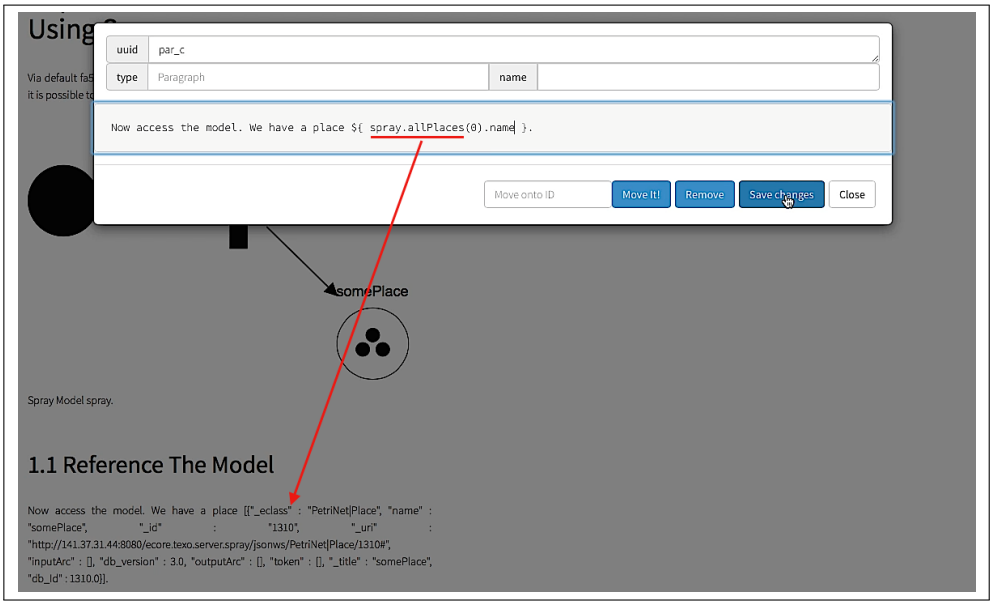


Abbildung 3.6: Hier wird auf das Spray-Dokumentelement verwiesen. Mit einem Programmausdruck kann der aktuelle Zustand des Modells abgefragt werden und bei Bedarf auf einzelne Attribute verwiesen werden. Bei Modelländerungen (z.B. Umbenennungen eines Diagrammknotens) bleibt der Text konsistent.

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.

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). Es gibt auch noch andere Szenarien, in denen Transformatio-

nen in u.U. vollkommen unterschiedliche Formate nützlich sein kann:

- Format zur Speicherung von Dokumente in digitalen Langzeitarchiven.
- Überführung von Dokumenten mit alten Layout-Konventionen in die Neuen.
- Dokumente gleichzeitig sowohl für Legacy-Software als auch für aktuelle Software zugänglich halten.
- Ein Dokument für andere Arbeitsgebiete aufbrechen: wie z.B. Knowledge-Discovery, Semantic-Web.
- Ein Dokument auf ein bestimmtes Publikum spezialisieren oder in eine andere Dokumentklasse überführen (z.B. Journal-Artikel in eine Präsentation transformieren).

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 somit 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 beantworten, 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 Impulse für neue Ideen bzw. Theorien gefunden werden.

Zunächst wird das allgemeine Vorgehen erläutert. Hierbei stehen die für den Prototyp aufgestellten Test-Spezifikationen im Mittelpunkt. Dann folgen Basiskonzepte, 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 (Hodapp, 2014) 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. Die Unit-Tests werden dort in Form von Spezifikationen verfasst. Jede Spezifikation soll ein bestimmtes Verhalten der Software dokumentieren und verifizieren. Das Framework ScalaTest[1] 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 ScalaTest[2] einrichtet. Es werden automatisch die benötigen Scala-Abhängigkeiten (Softwarebibliotheken) herunterladen 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 wurden 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/themerius/scaltex-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 (secundary) 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 1.5 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 instanziieren 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 die Textverarbeitung über den Browser. Diese stellt den abstrakten Syntaxbaum des Dokuments übersichtlich dar. 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 zu sehen. 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 jeder 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 (d.h. es ist komplett rekonstruierbar) 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 es, Websockets für die Kommunikation mit dem Browser bereitzustellen und 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.

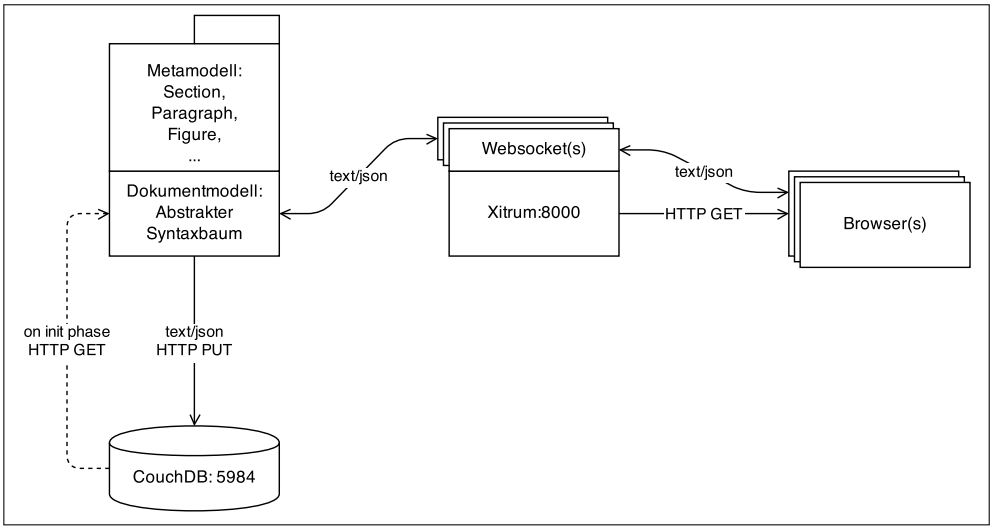


Abbildung 4.1: Die Architektur des Prototypen. Die Pfeile stellen die Kommunikation zwischen den einzelnen Komponenten dar. Die Kommunikation läuft zum einen über das Websocket-Protokoll und zum anderen über HTTP-Calls. Die Daten werden stets als JSON-String übertragen, außer bei der Auslieferung der statischen Daten durch Xitrum. Der gestrichelte Pfeil ist eine einmalige Kommunikation, die nur zum Initialisieren des Aktor 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. Das Dokumentmodell ist in Scala bzw. Akka implementiert, Xitrum ist in Scala implementiert.

4.3.1 Wurzel-Aktor

Der Wurzel- oder auch Root-Aktor enthält die Topologie des Dokuments. Hier werden also alle Aufgaben, die mit der Topologie zusammenhängen, verarbeitet. Der Root-Aktor hat als einziger Aktor die Gesamtübersicht über das Dokument. 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 Topologie-Informationen in der Datenbank ab. Wenn Versionen des Root-Aktors wiederherstellt 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. Er enthält 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 eine Analysephase des gesamten abstrakten Syntaxbaumes auslösen. Dort werden die Dokumentelemente aufgerufen, um zu überprüfen, ob noch alle Attribute aktuell sind. Beispielsweise kann ein Abschnitt-Dokumentelement seine Benummerung nochmals überdenken. Nochmals in Stichworten; ein Basis-Aktor:

- 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).
- Führt bei Änderungen eine Analysephase aus, welche 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 ist (1) ein Menge an Attributen festgelegt, die sich z.B. während der Analysephase ändern können. Wenn der Basis-Aktor über Änderungen benachrichtigt wird, teilt er das über die in (2) definierte Methode 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 direkt an das betreffende Dokumentelement gerichtet sind. Diese werden in (3) verarbeitet. Beispielsweise kann Methode (2) eine Nachricht an alle anderen Abbildungen schicken, um ihre 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 Analysephase

Die Analysephase des AST wird ausgelöst, wenn eine Update-Nachricht durch den Baum, der aus Aktoren besteht, geschickt wird. Visualisiert ist das Verhalten als Petrinetz auf Abbildung 4.2. Anmerkung: Das Petrinetz liegt als lebendiges Spray-Modell vor, d.h. alle Verweise auf das Modell werden auch direkt aus diesem bezogen!

Eine Update-Nachricht entsteht hauptsächlich dann, wenn der Benutzer ein Dokumentelement modifiziert. Vom Wurzel-Aktor aus wird die Nachricht an die Kinder und Geschwister weitergeleitet. Wenn eine Update-Nachricht eingeht landet sie in der Stelle ”onUpdate”. Die Variable contentSrc, also die textuelle Eingabe des Autors, wird aus dem aktuellen Zustand des Aktors geholt und es wird analysiert, ob darin Referenzen zu anderen Aktoren enthalten sind oder nicht. Je nach dem wird eine Marke in ”noRef” oder ”hasRef” gelegt. Wenn Verweisungen aufzulösen sind, wird die Code-Generierung getriggert (”triggerCodeGen”), um die angefragten Attribute aus dem Dokumentelement

zu holen. Der genaue Ablauf ist in Abschnitt 4.3.5 beschrieben. Wenn keine Verweisungen im contentSrc aufzufinden sind, wird das Dokumentelement direkt aufgefordert sich zu aktualisieren (”gotUpdate”). Hier kann es beispielsweise seine aktuelle Benummerung herausfinden. Dann wird der aktuelle Zustand des Aktors an die Anzeige geschickt (”sendState”).

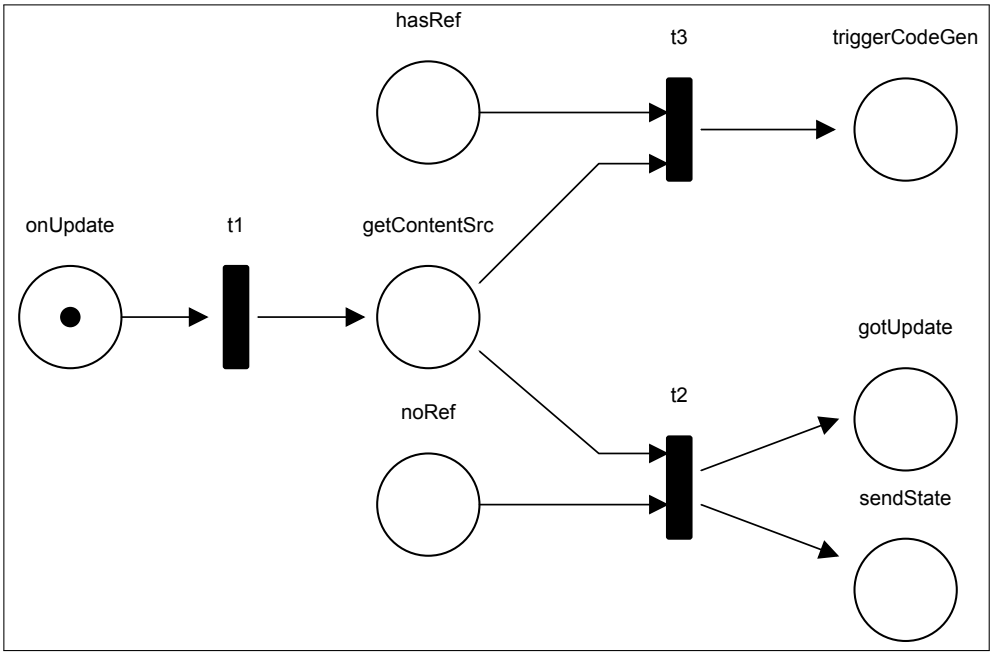


Abbildung 4.2: Spray-Modell Analysephase.

4.3.5 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 direkt auf diese zu. Innerhalb eines Textes wird mit einem Scala-Programmausdruck notiert, auf welches Attribut oder welche Methode der Dokumentelement-Scala-Klasse zugegriffen werden soll. Beispiel: „Siehe Abbildung (actor.nummer).“ wird aufgelöst zu „Siehe Abbildung 3.“ Prinzipieller Ablauf, um solche Verweise aufzulösen:

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-Aktoren schicken ihren jeweils aktuellen Zustand, in Form von generiertem Scala-Code, an den anfragenden Aktor zurück.
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 jeweils an die Stelle des Programmausdrucks kopiert. Beispiel: contentSrc ist „siehe Abbildung (actor.nummer)“, der Interpreter wertes es aus und speichert in contentRepr „siehe Abbildung 3“ ab.
7. Wenn alle Programmausdrücke aufgelöst sind, wird der neue String als contentRepr abgespeichert.

Die Anzeige im Browser kann contentSrc, contentRepr, contentUnified sowie alle anderen Attribute des Dokumentelements nach belieben verwenden. Details zu contentSrc, contentRepr und contentUnified sind in Abschnitt 4.4.6 zu finden.

Zyklische Verweise

Zyklische Verweise sind für die gegenwärtige Implementierung des Prototypen kein Problem. Die Nachrichten werden nach dem Schema „fire and forget“ gesendet. Das vermeidet, dass ein Aktor unendlich lange auf eine Antwort wartet. Am Beispiel von Aktor a und Aktor b soll erklärt werden, warum das Auflösen von Verweisen immer terminiert. Jeder der Aktoren nimmt eine einfache DSL entgegen, woraus die Attribute gewonnen werden. Die DSL sowie

die Attribute werden im Zustand des jeweiligen Aktors gespeichert. Die Aktoren verweisen in der DSL auf sich gegenseitig.

```
a {
  x: "",
  y: "",
  contentSrc: "x = foo, y = {b.z}"
}
```

```
b {
  z: "",
  contentSrc: "z = {a.y}"
}
```

Jeder Aktor hat Attribute die mit einem Vorgabewert initialisiert sind. Wird durch ein Ereignis, wie Benutzereingabe, die Analysephase ausgelöst, so soll die DSL in die Attribute übersetzt werden. Wenn Verweise zu anderen Aktoren in der DSL enthalten sind, wird der andere Aktor nach seinem aktuellen Zustand befragt, um diesen in die DSL einzubauen. Angenommen Aktor a fragt Aktor b nach seinem aktuellen Zustand und gleichzeitig fragt Aktor b auch Aktor a. Dann ist in den Antworten jeweils, der aktuelle Zustand zu finden, also jeweils nur die Vorgabewerte der Attribute. Es folgen keine weiteren Nachrichten, da sich die Aktoren jeweils mit dem Vorgabewert zufrieden geben. Somit entsteht keine endlose Nachrichtenkette; es terminiert also wie bei jedem anderen Verweis.

Selbst wenn die Aktoren gegenseitig auf das rohe contentSrc (die DSL) gegenseitig zugreifen wollen, ist das kein Problem. Das liegt daran, dass der ausgewertete contentSrc (d.h. die Attribute des anderen Aktors wurden dort eingebaut) nicht wieder in das contentSrc-Attribut zurückgespeichert wird, sondern in dem gesonderten contentRepr-Attribut abgespeichert wird. Das heißt ein Programmcode (hier in Klammern) kann nicht doppelt ausgewertet werden, da nur contentSrc nach entsprechenden Aktor-Referenzen durchsucht und interpretiert wird.

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 Sprachsyntax. 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 geschriebenen 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, ist der Aktor. Die ursprüngliche Idee des Aktorenmodells 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 eines Aktors, also wie auf Nachrichten reagiert wird, 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 Betriebsystem (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 (Typesafe Inc., 2014, S. 69). Laut Akka-Projektwebseite http://akka.io/ setzen viele Firmen das Toolkit produktiv ein, was die Reife des Projekts verdeutlicht.

(Typesafe Inc., 2014, S. 7) nennt einige Anwendungsszenarien von Akka. Der Einsatz als Fundament für einen abstrakten Syntaxbaum ist dort jedoch nicht aufgeführt. Das Aktorenmodell 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, auch bekannt als abstrakter Syntaxbaum oder 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. Es unterstützt keine Websockets; ist jedoch für die Zukunft geplant. Es wird seit 2011 entwickelt. (Entnommen aus Changelog)
- Play-Framework v.2.2.2: Sehr vielversprechend und stabil mit großer Entwicklergemeinde. Play selbst nutzt jedoch ein spezielles sbt-Plugin, welches sich nicht mit dem Scala-Interpreter verträgt. Der Interpreter findet u.a. keinen passenden Java-Classpath. Es 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. Es wird seit ca. 2012 entwickelt. (Entnommen aus den GitHub-Commit-Logs des Projekts) Als Webserver wird Netty eingesetzt.
- Xitrum v.3.13: Mit Abstand die beste und schönste API und wird schon seit 2010 aktiv entwickelt. Es wird laut Autor 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

Websockets[3] sind noch ein relativ junger Standard. Die technischen Anforderungen stehen schon fest, jedoch fehlt noch die endgültige Ratifizierung (die Candidate Recommendation wird zur 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 von 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 – somit muss der Client kein Polling betreiben, um über serverseitige Änderungen benachrichtig 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 bedeutet, dass der Programmierer von

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

client-side-JavaScript oft in Konflikt mit Namespace-Pollution[4] gerät. CommonJS versucht Empfehlungen für JavaScript-APIs zu geben. Dazu gehört auch Asynchronous-Module-Definition (AMD)[5]. Damit ist es möglich Module in JavaScript zu definieren und deren Abhängigkeiten zu managen – dies vermeidet Namespace-Pollution. Require.js[6] 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-Modul-Definitionen;
- ketcher: Strukturediteor für chemische Verbindungen;
- bootstrap: Layout-Framework;
- at.js: Autovervollständigung;
- jquery: Komfortable API zur XML-Baummodifikation (DOM);
- mustache.js: Template-Engine.

4.4.5 Dokumentelemente und Domäneneditoren

Um dem Autor eine übersichtliche Repräsentation bzw. die Möglichkeit zur Bearbeitung eines Modells einer Domäne zu geben, sind die hier sogenannten

[4] Global-Namespace-Pollution: Variablen können 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.

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

[6] Require.js Projektwebseite: http://requirejs.org/

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 zwischen Autor und Modell. Im einfachsten Fall bietet der Domäneneditor eine DSL an. Für komplexere Modelle, wie z.B. 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. Eine beliebige andere Repräsentation wird also durch Anpassen des Templates erreicht.

Ketcher

Ketcher[7] ist ein für sich allein stehender, in JavaScript geschriebener, Editor für chemische Verbindungen. Damit lassen sich chemische Verbindungen darstellen und zeichnen. Ein chemische Verbindung kann als Molfile[8] abgespeichert werden. Das Molfile kann als Modell einer chemischen Verbindung betrachtet werden und wird hier als Attribut im Dokumentelement abgespeichert.

Ein Wrapper[9] 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.

[7] Webseite des Ketcher Projekts: http://ggasoftware.com/opensource/ketcher

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

[9] „Wrapper (Software), ein Programm, das als Schnittstelle zwischen zwei anderen Programmen dient“. Aus Wikipedia http://de.wikipedia.org/wiki/Wrapper

Spray

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

Anders 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 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-Softwarebibliothek 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 es 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. Es wird eine Datenstruktur zur Speicherung der Dokumenten-Topologie vorgestellt. Dazu gehört ein Algorithmus zur Auswertung der Reihenfolge der einzelnen Elemente innerhalb der Topologie. Es wird die Funktionsweise der Code-Generierung zur Auflösung der „reichhaltigen“ Verweisungen erklärt. Zuletzt wird die Speicherung der Benutzereingabe (contentSrc, contentRepr, contentUnified) besprochen.

Benummern im Syntaxbaum

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 der baumartigen Hierarchie des Dokuments Benummerungen zuweisen kann.

Gegeben ist ein gerichteter kreisfreier Graph $G = (V, E)$. Wobei V die Menge der Knoten (Dokumentelemente) darstellt und E die Kanten (Referenzen zum ersten Kind bzw. nächsten Geschwister). Es werden nur Knoten benummert, die als Kapitel bzw. Abschnitt markiert sind. sind die Abschnitte die benummert werden sollen.

```
Jeder Abschnittsknoten hat das Attribut ‘Nummer‘,
welches mit dem Wert 1 initialisiert ist.

Wird die Analysephase angestoßen wird von der Wurzel aus
die Nachricht ”Update” an die Knoten geschickt.
Das erfolgt hierarchisch, d.h. jeder Knoten reicht
diese Nachricht nur an sein erstes Kind bzw. seinen
direkt nachfolgenden Geschwisterknoten weiter.
Jeder Knoten bekommt pro Analysephase genau einmal
die Nachricht ”Update”.

Wird ”Update” von einem Abschnittsknoten erhalten,
schickt dieser die Nachricht ”Durchzählen(Nummer)”
an seinen direkt nachfolgenden Geschwisterknoten.

Erhält ein Knoten die Nachricht ”Durchzählen(NummerVorgänger)”,
setzt er sein Attribut Nummer := NummerVorgänger + 1.

Erhält ein anderer Knoten-Typ die ”Durchzählen” Nachricht,
leitet er sie unverändert weiter.
```

Damit werden maximal $|E|$ Nachrichten benötigt, um für jeden Abschnitts-Knoten die korrekte Benummerung zu bestimmen. Die Komplexität bezüglich der Laufzeit beträgt also $O(|E|)$. Notiz: Der Algorithmus benummert also jede Hierarchieebene individuell, das heißt, dass z.B. in der Titelei die Kapitel von 1 bis x gezählt werden und ab dem Hauptteil werden die Kapitel erneut von 1 bis y gezählt. Für die Zukunft ist es bestimmt sinnvoll, wenn Kapitel und Abschnitte selbst Teil der Hierarchie sind.

Dadurch, dass die Update-Nachricht jeden Knoten im Baum erreicht, werden die betreffenden Knoten ihre Durchzählen-Nachricht mehrfach verbreiten. Das hat zur Folge, dass insgesamt deutlich mehr Nachrichten versendet werden. Es werden exakt so viele Nachrichten versendet – wobei $|E|$ die zu benummernden Abschnitts-Knoten sind – wie in der nachfolgenden Formel angegeben. Dies entspricht einer Komplexität bezüglich der Laufzeit von $O(|E|^2)$. Hier gibt es also noch Optimierungspotenzial für die Analysephase.

$$\sum_{i=1}^{|E|} (|E| - i) = \frac{1}{2} (|E| - 1) |E|$$

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 im Idealfall bei $|E|$ Nachrichten.

Topologie speichern und verarbeiten

In einer JSON-Datenstruktur wird die Topologie des Dokuments festgehalten. Als Schlüssel dient die ID des Dokumentelements 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 in (Hodapp, 2014) unter misc/topology verfügbar. Der Root-Aktor enthält eine entsprechende Scala-Implementierung dieses Algorithmus.

Code-Generierung für Verweise

In Abschnitt 4.3.5 wurde bereits das Vorgehen angesprochen, um Verweise, die zwischen Dokumentelementen bestehen, aufzulösen.

Speicherung der Benutzereingabe

Wenn der Autor Dokumentelemente erstellt, möchte er diese auch mit entsprechendem Inhalt füllen. Beispielsweise mit einem Fließtext oder einer DSL, die ein konkretes Modell beschreibt. Diese Eingaben werden innerhalb des Zustands des Basis-Aktors abgespeichert, bekannt als contentSrc. Wenn der Autor auf Attribute eines anderen Dokumentelements zugreifen will, schreibt er einen entsprechenden Scala-Programmausdruck. Jeder Aktor kann sein contentSrc auf Programmausdrücke hin untersuchen, so dass das entsprechende Dokumentelement nach dem verlangten Attribut befragt werden kann.

Wenn die Scala-Programmausdrucke erfolgreich ausgewertet werden konnten, werden diese mit dem jeweiligen Auswertungsergebnis ersetzt. Das wird in contentRepr abgespeichert. Beispiel: „Auf Abbildung (pinguin.nr) ist ein Pinguin.“ → „Auf Abbildung 3 ist ein Pinguin.“

Eine alternative Repräsentation ist contentUnified. Hier werden contentSrc und contentRepr vereinigt. Das heißt, an jeder Stelle, an der ein Verweis vorkommt, wird die komplette Information in einer JSON-Datenstruktur gehalten. Der Programmausdruck und sein Ergebnis werden festgehalten. Das ist wichtig für die Anzeige, um die UUIDs der einzelnen Aktoren unter einem Kurznamen zu verstecken, aber dennoch den gesamten Programmausdruck übermitteln zu können. Nachfolgend ein kleines Codebeispiel dafür:

```
[
  {
    str: ”Auf Abbildung ”,
    expression: [
      ”{ ”,
    {uuid: ”id_ 80b4_id”, shortName: ”pinguin”, documentElement: ”Figure”},
      ”.nr }”
    ]
  },
  {
    str: ” ist ein Pinguin”
```

```
  }
]
```

4.5 Erreichter Funktionsumfang

Mit dem Prototyp ist es möglich, eine Masterarbeit zu schreiben. Das heißt, dass von der prinzipiellen Funktionalität alles vorhanden 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 Inhalte 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 ein 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 Potenzial 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. Mit Hilfe des zweiten Prototyps wurden die Konzepte immer klarer, so dass daraus eine gut strukturierte Theorie gefolgert und

nachvollzogen werden konnte. Die neuen Erkenntnisse zeigen aber auch, dass der Prototyp nicht an allen Stellen die Theorien konsequent umsetzt. Jetzt ist folgendes deutlich klarer:

- Jedes Dokumentelement 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.

Kritik: Der Prototyp stößt 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[10] ist – im Fokus stand zunächst eine Bibliothek mit einer übersichtlichen und flexiblen API zu verwenden. Zudem ist es bei großen Dokumenten sehr aufwändig, wenn der DOM über JavaScript in der Browser-Benutzeroberfläche aufgebaut wird. 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 sein. Außerdem ist es

[10] Ein Sampling über das Zeitverhalten von Methodenaufrufen mit VisualVM ergab, dass die Aktoren oft schlafen gelegt werden, und das recht viel Rechenzeit für das Parsing von JSON und das Interpretieren des (generierten) Scala-Codes (Verweise) aufgewendet wird. Dieser Scala-Code selbst setzt wiederum auch auf diese JSON-Bibliothek.

für einen Autor sehr bequem, wenn häufig verwendete Modelle als fertige Dokumentelemente vorliegen – es vereinfacht die Visualisierung, die sogar direkt in Veröffentlichungsqualität vorliegt.

4.5.1 Codestatistik

Es wurden etwa 4 Monate intensive Entwicklungsarbeit geleistet. Kleinere Bugfixes oder Einführungen neuer Dokumentelemente haben sich angeschlossen. 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 Code-Zeilen gehen an die in JavaScript geschriebene Benutzeroberfläche (ohne Template- oder Style-Code). Aus zirka 1000 Code-Zeilen besteht das Aktorsystem, ohne Dokumentelement-Definitionen und ohne Server-Code. Es wurden insgesamt sechs Releases erstellt, welche in diversen (internen) Vorträgen vorgeführt wurden.

Kapitel 5

Analyse der Ergebnisse

In diesem Kapitel werden zunächst die Ergebnisse und die daraus gefolgerten Erkenntnisse zusammengetragen. Danach wird der Versuch unternommen die wissenschaftlichen Fragen zu beantworten, jeweils mit einer kurzen Begründung. Anschließend wird der Prototyp bzw. die Konzepte mit ähnlichen Arbeiten verglichen. Zum Schluss erfolgt eine Diskussion zum Prototypen.

5.1 Ergebnisse und Erkenntnisse

Aus den vier Mängeln (siehe Abschnitt 1.2) Mangel an Konsistenz, Mangel an Domänenwissen, Mangel an Semantik und Mangel an Metamodellierung ist ein Prototyp entstanden, welcher versucht diese Mängel zu entschärfen. Hieraus wurden entsprechende Theorien entwickelt:

Es gibt ein sinnvolles Modell für Dokumente. Als natürliche Realisierung eignet sich der abstrakte Syntaxbaum, bekannt aus den Theorien zum Bau von Übersetzern (Compiler). Jedes Dokumentelement, wie z.B. Kapitel, Absätze, Abbildungen, etc., wird als Knoten des Syntaxbaumes betrachtet. Es gibt zwei Arten von Kanten: Zum einen Kanten, die die Topologie des Dokuments vorgeben und zum anderen, Kanten, die Verweise zwischen Dokumentelementen ermöglichen.

Der abstrakte Syntaxbaum wird mit dem Akka-Aktoren-Toolkit umgesetzt. Jeder Knoten bzw. Dokumentelement wird in Form eines Aktors abgebildet. Die Aktoren versenden untereinander Nachrichten, um das Dokument aufzubauen. Dazu gehört es auch Verweisungen zwischen Dokumentelementen auflösen (z.B. ein Absatz verweist auf eine bestimmte Abbildung) oder topologische Informationen zu verarbeiten (Herausfinden der korrekten Benummerung eines Abschnitts).

Mit Projektionseditoren (projectional editing) wird direkt der abstrakte Syntaxbaum manipuliert. Der Autor bearbeitet also direkt das Modell des Dokuments. Das ermöglicht quasi beliebig viele konkrete Syntaxen oder Repräsentationen eines Dokuments bzw. eines Dokumentelements. Ein Dokument kann dadurch in beliebige Formate transformiert werden. Das heißt, hier werden Inhalt, Struktur und Layout/Präsentation sauber voneinander getrennt. Der intellektuelle Inhalt des Dokuments liegt in den Attributen der Dokumentelemente. Die Struktur entspricht der Topologie des Syntaxbaumes. Die Präsentation wird von verschiedenen Templates, die vom Projektionseditor gerendert werden, beigesteuert.

Die Analyse der Dokumentelement-Semiotik hat gezeigt, dass jedes Dokumentelement selbst ein Modell einer spezifische Domäne enthält. Das Modell wird in Form von Attributen im Dokumentelement gespeichert. Autoren können eine textuelle DSL an das Dokumentelement übermitteln, daraus werden dann die Attribute des Modells gewonnen. Oder das Dokumentelement bietet einen grafischen Editor zur Modellierung an; der Editor kann dann die Attribute des Modells extrahieren. Daher verfügt das Dokumentelement über explizite Semantik der Domäne und kann verschiedene Repräsentationen des Modells anbieten oder zusätzliche Informationen (durch Berechnung) bereitstellen.

Es wurde eine allgemeingültige Taxonomie für den inneren Aufbau von (wissenschaftlichen) Dokumenten ausgearbeitet. Sie ordnet die Dokumentelemente in drei Kategorien (mit beliebigen Unterkategorien) ein: Container-Elemente, Struktur-Elemente und Meta-Elemente. Container-Elemente enthalten andere Dokumentelemente, Struktur-Elemente enthalten den intellektuellen Inhalt und Meta-Elemente beschreiben wiederum andere Dokumentelemente oder Daten. Aus der Taxonomie können zwei

Matrizen abgeleitet werden, diese deklarieren die erlaubten Verschachtelungen bzw. Reihenfolgen der Dokumentelemente. Die Taxonomie bzw. Matrizen spezifizieren formal den Aufbau einer Dokumentklasse, wie z.B. EU-Patent, Masterarbeit, Jahresbericht, etc.

Aus der Taxonomie kann ein Metamodell für das Dokumentmodell (AST) abgeleitet werden. In einem solchen Metamodell sind die einzelnen Dokumentelemente, die in einem Dokument vorkommen können, definiert. In Verbindung mit den vorgestellten Matrizen kann der Projektionseditor, den Autor durch die Erstellung seines Dokuments, in einer speziellen Dokumentklasse, führen. Daraus resultiert, dass der Autor mutmaßlich weniger Fehler, durch diesen Wegweiser, macht. Zudem können Dokumentelemente, die ihre Domäne verstehen, sich zu einem gewissen Grad selbst verifizieren. Das ist möglich, da eine DSL, auf syntaktische oder sogar semantische Korrektheit, geprüft werden kann.

Wie sich herausgestellt hat, ist das Konzept sehr flexibel, denn dem Prototypen gelingt es bereits sehr unterschiedliche Anwendungsfälle zu lösen: Erstellung *dieser* Masterarbeit als Beispiel für ein wissenschaftliches Dokument, ein UIMA-CAS-Editor zur Annotation von Texten, die Dokumentation von Spray-Softwaremodellen und die Transformation in andere Formate am Beispiel von LaTeX-Code.

5.2 Analyse im Detail

Zunächst sollen die wissenschaftlichen Fragen aus Abschnitt 1.8 aufgegriffen und ein Versuch der Beantwortung unternommen werden:

- Ist es sinnvoll Dokumente als Modell aufzufassen? Ja, dadurch ergibt sich eine bessere Verarbeitung durch formale Methoden.
- Wie könnte eine sinnvolle Modellierung von Dokumenten aussehen? Eine natürliche und mächtige Realisierung ist als abstrakter Syntaxbaum/Graph, denn ein Dokument entspricht vom prinzipiellen Aufbau her einem (hierarchischen) gerichteten Graphen.

- Gibt es ein gemeinsames Metamodell? Ja, die Dokumentelemente sind die kleinste Einheit im Dokument, die sich weiter in einer Taxonomie kategorisieren lassen.
- Kann ein Aktorsystem verwendet werden, um einen abstrakten Syntaxbaum zu implementieren? Ja es ist möglich, z.B. mit dem Akka-Toolkit. Knoten sind Aktoren und Kanten die Nachrichten zwischen den Aktoren.
- Taugt dieser abstrakte Syntaxbaum als sinnvolle Architekturgrundlage für einen Projektionseditor, und damit auch als Code-Generator? Ja, der Code für die Benutzeroberfläche ist sehr kompakt/generisch und trotz Prototypenstadium recht benutzerfreundlich. Ein einfaches Template pro Dokumentelement genügt, um eine LaTeX-Code-Projektion zu generieren.
- Brauchen wir mehr explizite Semantik innerhalb von Dokumenten? Ja, damit Dokumente sich selbst verifizieren können und Autor, Leser oder Dienste (Stichwort Semantic-Web) zusätzliche Informationen zum betrachteten Objekt erhalten.
- Ergeben sich Vorteile, wenn diese Semantik direkt vom Autor transportiert wird? Ja, der Autor kann sein Dokument leichter konsistent halten und Dienste erhalten Objekte mit kurierten Informationen und deren Bedeutung.
- Ist es möglich, dass sich Dokumente zu einem gewissen Grad selbst verifizieren und damit konsistenter machen? Ja, jedes Dokumentelement beherbergt Wissen aus seiner Domäne und kann Eingaben des Autors mit einem formalen Modell (z.B. vorliegend als DSL, Softwarebibliothek oder XML-Schema) auf Korrektheit prüfen.
- Gibt es eine allgemeingültige Taxonomie für wiss. Publikationen? Ja, sie kennt drei Kategorien von Dokumentelementen: Container-Elemente, Struktur-Elemente und Meta-Elemente.

5.2.1 Ähnliche Arbeiten

In Abschnitt 1.3 wurden verschiedene Systeme vorgestellt, die im Bereich der Dokumenterstellung etabliert sind oder Verbesserungsansätze liefern. Hier werden diese Systeme mit dem hier entstanden Prototyp bzw. dessen konzeptuellen Überlegungen kurz verglichen.

Textverarbeitungen wie Microsoft Word haben Formatvorlagen, welche als primitive Dokumentelemente bezeichnet werden könnten. Diese Formatvorlagen bringen jedoch keinerlei Wissen über deren Domäne mit, und somit auch keine ausgefeilten Repräsentationen oder Editoren, wie z.B. das hier gezeigte (chem.) Strukturformel-Dokumentelement es tut. Eine Spezifikation, welche Elemente aneinander gereiht werden dürfen gibt es nicht. Dies ist eine Fehlerquelle für den Autor, sowohl was die Konsistenz des Dokuments betrifft, als auch den formell korrekten Aufbau der Dokumentklasse.

Das Textsatzsystem LaTeX spezifiziert explizit Dokumentklassen, welche passende TeX-Makros anbietet, um ein Dokument in der vorliegenden Klasse zu bauen. Diese Makros entsprechen quasi den Dokumentelementen. Jedoch prüft keine Taxonomie bzw. Metamodell die korrekte Aneinanderreihung der Dokumentklasse. Zudem wird direkt mit dem Quellcode gearbeitet, was Kopilierzeiten zum generieren des PDFs erfordert. Interaktive grafische Domäneneditoren sind damit also nicht ohne weiteres möglich. SALT (Groza u. a., 2007) erweitert LaTeX, so dass semantische Annotationen zum PDF hinzugefügt werden können. Um dies zu tun, erhält der Autor eine zusätzliche Menge an TeX-Makros, speziell für die Annotation. Diese Annotationen sind für die spätere Verarbeitung durch Semantic-Web-Dienste nützlich, bringen aber dem Autor selbst noch keinen Vorteil. Beim hier vorgestellten Prototyp hat der Autor auch etwas davon, diese Informationen zu übermitteln; denn er kann direkt auf die Domänenmodelle zurückgreifen und sich so Schreibarbeit und Fehler ersparen.

SageTeX[1] ist eine Erweiterung zu LaTeX, die es erlaubt ausführbare mathematische Ausdrücke oder Funktionsgraphen (Plots) direkt in LaTeX-Code

[1] Deutsche Dokumentation von SageTeX auf der Sage-Projektseite: http://www.sagemath.org/de/html/tutorial/sagetex.html

einzubinden. Dies ist also sehr ähnlich zur Funktionsweise bzw. Idee des hier vorgestellten Systems. Jedoch ist Sage sehr stark auf die Domäne Mathematik zugeschnitten. Die Konzepte des hier vorgestellten Systems jedoch weiter; es ist so flexibel, dass beliebige Domänenmodelle mit ihren Visualisierungen dargestellt werden können – und diese Modelle liegen sogar als semantische Objekte vor, mit denen z.B. der Text des Dokuments direkt interagieren kann. Sage hat gewiss Potenzial als z.B. ein Dokumentelement in das hier vorgestellte System integriert zu werden. Dies wäre mit Sicherheit ein großer Gewinn.

Der „One Click Annotator“ (Heese u. a., 2010) soll es z.B. Journalisten einfach machen einen Text für das Semantic-Web aufzubereiten. Sinn ist, dass auch nicht-Ontologie-Experten auf einfache weise eine Wissensbasis aufbauen und durchsuchbar machen. Das primäre Ziel sind hier nicht zwangsweise wissenschaftliche Texte. Auch mit dem hier vorgestellten Konzepten ist es möglich Annotationen zu Texten hinzuzufügen. Es steht jedoch noch aus, diese Annotationen an RDF-Systeme anzukoppeln. Durch Projektion bzw. Transformation in andere Formate sollte es jedoch durchaus möglich sein, mit solchen Systemen zu interagieren. Jedoch ist es mit dem Prototypen noch relativ umständlich eine Annotation hinzuzufügen. Anzeigen und bearbeiten von Annotationen funktioniert schon recht zuverlässig.

Das Wolfram-CDF ist wohl die stärkste Konkurrenz für das hier vorgestellte System. Denn es bietet auch eine Art von interaktiven Dokumentelementen, auch wenn Wolfram diese nicht so nennt, die mit einer Programmiersprache (Wolfram-Language) generiert werden. Zudem bedient sich diese Programmiersprache an der Wissensbasis der semantischen Suchmaschine Wolfram-Alpha – ist jedoch zugleich sehr stark an diese Plattform gebunden. Zudem benötigt das Format zwingend eine Wolfram-Runtime; es basiert also nicht auf offenen Standards. Die Geschlossenheit des proprietären Wolfram-CDF verhindert vermutlich den richtigen Durchbruch bzw. eine breite Anwenderbasis. Weiterhin gibt es zu kritisieren, dass es bei diesem System keine (strikten) Vorgaben zur Beschaffenheit eines Dokuments gibt. Dokumentelemente entsprechen hier also mehr oder weniger den Word-Formatvorlagen, d.h. einem Dokumentelement ist auch keine spezifische Domäne, mit einem spezifischen Modell, explizit zugeordnet.

5.3 Diskussion zum Prototyp

Am Anfang der Masterarbeit stand die Idee Dokumentelemente auf Aktoren abzubilden. Es sollte geprüft werden, ob diese Idee sinnvoll und umsetzbar ist. Daraus hat sich der hier vorgestellte Prototyp entwickelt. Von Anfang an war es klar, dass es ein Projektionseditor werden soll, was bedingt, dass das Softwaresystem einen abstrakten Syntaxbaum in irgendeiner Form vorweisen muss. Die Benutzerschnittstelle ist mit Web-Standards umgesetzt, so dass über einen Browser das Dokument betrachtet sowie bearbeitet werden kann.

Der Prototyp hat das gezeigt, was er zeigen sollte: Es ist möglich einen Projektionseditor auf Basis von Web-Standards und ein abstrakten Syntaxbaum im Aktorenmodell umzusetzen. Der Prototyp läuft bereits recht stabil, so dass auch diese Masterarbeit damit, ohne große Hürden, verfasst werden konnte. Zudem ist es recht leicht, neue Dokumentelemente hinzuzufügen. Der Prototyp hat zudem Licht ins Dunkel gebracht: Nur so konnten die Theorien weiterentwickelt werden, so dass sie eine solide Basis für eine produktiv einsetzbare Neuentwicklung bilden.

Jedoch gibt es auch noch Verbesserungspotenzial für das Softwaresystem: (1) Die Architektur ist zu monolithisch, d.h. insbesondere der Basis-Aktor ist nicht modular genug aufgebaut. (2) Große Dokumente bringen das System an die Performanzgrenzen, da der DOM im Browser rein durch JavaScript aufgebaut wird und (vermutlich) eine ineffiziente Implementierung der eingesetzten JSON-Bibliothek[2] die Aktoren bremst. (3) Es gibt weitere Limitierungen, wie z.B. keine Benutzerverwaltung, die jedoch zum Zeigen der Anwendungsfälle nicht nötig sind.

[2] Ein Sampling über das Zeitverhalten von Methodenaufrufen mit VisualVM ergab, dass die Aktoren oft schlafen gelegt werden, und das recht viel Rechenzeit für das Parsing von JSON und das Interpretieren des (generierten) Scala-Codes (Verweise) aufgewendet wird. Dieser Scala-Code selbst setzt wiederum auch auf diese JSON-Bibliothek.

Kapitel 6

Zusammenfassung

Heutige Textverarbeitungen die zum Erstellen von wissenschaftlichen Dokumenten verwendet werden, weisen noch einige Defizite auf. Im Dokument schleichen sich leicht Fehler, durch Inkonsistenten zwischen Text und z.B. einem darin beschriebenen Modell, ein. Das beschriebene Modell bringt Domänenwissen mit, welches jedoch dem Autor, in traditionellen Systemen, nicht zur direkten Nutzung vorliegt. Das System kann also dadurch nicht explizit mit der Semantik der Domäne umgehen. Zudem liegen oftmals keine formal spezifizierten Dokumentklassen vor, welche es dem Autor ermöglichen sich stärker auf den eigentlichen Inhalt zu konzentrieren.

Ziel dieser Arbeit ist es, herauszufinden, ob und wie diese Defizite abgefedert werden können. Dazu wird ein Prototyp entwickelt, anhand dessen theoretische Konzepte zur Beseitigung der Defizite erarbeitet und auf praktische Tauglichkeit hin untersucht werden. Die praktische Tauglichkeit wird anhand verschiedenartiger Anwendungsfälle überprüft.

Die prinzipielle Idee ist es den prototypischen Dokumenteditor als Projektionseditor (projectional editor) zu gestalten. Dabei wird zur Darstellung verstärkt auf Web-Standards gesetzt und zur serverseitigen Programmierung wird Scala eingesetzt. Der Autor bearbeitet das Dokument, indem er direkt dessen abstrakten Syntaxbaum manipuliert, wobei die Benutzeroberfläche dem Autor eine konkrete und übersichtliche Repräsentation des Dokuments

anzeigt. Ein Dokument ist aus einzelnen Dokumentelementen (z.B. Abschnitt, Absatz, Abbildung, etc.) aufgebaut, ein jedes dieser Dokumentelemente wird in einen Knoten des Syntaxbaumes abgebildet. Die konkrete Implementierung des abstrakten Syntaxbaums wird mit dem Aktorenmodell realisiert, wobei jeder Syntaxbaum-Knoten als Aktor abgebildet wird. Es können spezialisierte Dokumentelemente (z.B. für chemische Strukturformeln) entwickelt werden, welche dem Autor domänenspezifischen Inhalte (z.B. Attribute, wie Masse oder Name einer chemischen Verbindung) zur Nutzung bereitstellen.

6.1 Fazit der Ergebnisse

Erstes Ergebnis ist der Prototyp, der in der Lage ist verschiedenartige Anwendungsfälle zu realisieren. Damit beweist er die Tauglichkeit und den praktischen Nutzen der hier vorgestellten Konzepte. Bei den beispielhaft umgesetzten Anwendungsfällen handelt es sich um (1) die Erstellung *dieses* wissenschaftlichen Dokuments, (2) die Anzeige, Bearbeitung von Annotationen in wissenschaftlichen Texten, (3) die Dokumentation von Spray-Softwaremodellen und (4) die Transformation *dieser* Masterarbeit in ein anderes Format, namentlich LaTeX-Code.

Zweites Ergebnis ist eine Reihe von Theorien, die das Verfassen von wissenschaftlichen Dokumenten geordneter und formaler gestalten. Ein abstrakter Syntaxbaum bzw. Graph dient als natürliches Modell für ein Dokument. Die einzelnen Bauteile eines Dokuments, die Dokumentelemente, werden auf Graph-Knoten abgebildet. Für formale Modelle können auch Metamodelle (Modell eines Modells) gefunden werden. Dort enthalten sind u.a. Definitionen zu den einzelnen Dokumentelementen. Dieses Dokumentmodell passt zudem wunderbar in die Gesetzmäßigkeiten der Semiotik. Daraus wird ersichtlich, dass jedes Dokumentelement ein Modell einer spezifischen Domäne repräsentiert.

Drittes Ergebnis ist die Erkenntnis, dass es eine gewisse, allgemein gültige Gesetzmäßigkeit für den inneren Aufbau von Dokumenten gibt. Diese können als eine Taxonomie aufgeschrieben werden. Eine Dokumentklasse, wie z.B. natur-

wiss. Aufsatz, Abschlussarbeit oder EU-Patent etc., kann genau spezifiziert werden, indem Matrizen über die erlaubten Abfolgen bzw. Verschachtelungen der Dokumentelemente aufgestellt werden. Dies kann als ein Teil des Metamodells verstanden werden. Zudem ermöglicht dies dem Softwaresystem, einem Autor bei der Erstellung eines korrekt aufgebauten Dokuments zu unterstützen.

6.2 Ausblick

Im Ausblick werden noch einige Ideen oder Visionen für die Zukunft des Projekts vorgestellt. Zunächst werden technische Verbesserungen bzw. konzeptuelle Erweiterungen vorgestellt. Gefolgt von Visionen, die zum Ausbau des Systems zu einer vollwertigen Platform zur Erstellung und Verwaltung von akademischen Dokumenten anstiften. Damit könnte der vorgestellte Prototyp bzw. seine Konzepte zum Kern eines nützlichen Werkzeuges für die Wissenschaft werden.

6.2.1 Technische Verbesserungen

Überlegungen und Ideen zur Verbesserung der Architektur: XML statt JSON, da XML durch DTDs etc. validiert werden kann? Oder gar intern auf mehr native Datenstrukturen setzen und pickling verwenden? Eine andere Programmiersprache für die reichhaltigen Verweise, mit besserer Möglichkeit zur Metaprogrammierung? Modularere Architektur durch Aufspaltung in mehrere kleine Aktoren? Ecore bzw. MOF als Metamodell-Implementierung, denn es gibt hierfür mehr Werkzeuge, dadurch sind diese Formate besser austauschbar? Neue bzw. veränderte Dokumentelemente zur Laufzeit injizieren, z.B. als Plugin? Eine DOM-Implementierung als Topologie-Datenstruktur? Oder gar eine DOM-Implementierung statt Aktoren?

Wenn Templates auf Serverseite gerendert werden und die Dokumentelemente via REST-Schnittstelle direkt abrufbar sind, ermöglicht das z.B. generierten LaTeX-Code direkt von einem Shell-Programm abholen zu lassen. Einzelne Do-

kumentelemente können zudem in verschiedenen Formaten abgerufen werden, beispielsweise das Chemiemolekül als SVG-Bild. Zudem muss der Browser den DOM nicht mehr ausschließlich mit JavaScript aufbauen, sondern muss dieses nur noch bei erfolgten Editier-Operationen tun.

Realisierung der Annotationen. Beim Verfassen der Masterarbeit ist aufgefallen, dass es neben einfachen Inline-Annotationen auch so etwas wie schwebende Annotationen geben sollte. Ein Beispiel dafür sind die Fußnoten, denn diese werden im Text, den sie ergänzen, verkürzt dargestellt, aber an einer anderen Stelle im Dokument vollständig angezeigt. Solche schwebenden Annotationen können auch für anderen Szenerien nützlich sein: Beispielsweise eine Annotation die einen Teil einer Grafik vergrößert (Lupe) und diese Hervorhebung entsprechend erklärt. Momentan wird Scala-Programmcode zum Verweisen auf Inhalte anderer Dokumentelemente (betrifft insbesondere Inline-Annotationen) verwendet. Dabei wird von der entsprechenden Scala-Methode des Dokumentelements ein statischer String zurückgegeben. Jedoch möchte man in anderen Projektionen des Dokuments diesen String eventuell mit z.B. einem speziellen Markup umhüllen. Das ist im hiesigen Prototypen noch nicht sauber umgesetzt.

Schnelle Erstellung von Annotationen. Beispielsweise entscheidet sich der Autor nun doch, einen Textteil als Fußnote auszulagern (d.h. ins Metadokument aufnehmen). Verbesserter Arbeitsablauf: Der Autor markiert den betreffenden Text und erhält eine Auswahlbox. Dort werden passende Dokumentelemente vorgeschlagen. Er möchte eine Fußnote aus dem markierten Text machen und wählt diese in der Auswahlbox. Danach wird der Text automatisch ausgeschnitten und in das neue Fußnoten-Dokumentelement verschoben. Im ursprünglichen Text wird ein Verweis zur so entstandenen Fußnote hinerlassen. Momentan muss dieser Vorgang noch von Hand ausgeführt werden und ist daher etwas umständlich.

Weiterentwicklung der semantischen Ansicht für Dokumentelemente, siehe Abb. 6.1. Das Modell eines Dokumentelements kann auf verschiedene Weise eingegeben werden. Zum einen als textuelle DSL, zum anderen durch einen grafischen Editor. Aus diesen Eingaben können die Attribute extrahiert werden, welche für die Weiterverarbeitung gut geeignet sind. Die Attribute können

z.B. durch Templates eine bestimmte Repräsentation annehmen. Es ist wohl nützlich, wenn zwischen Makro-Repräsentation und Mikro-Repräsentation des Dokumentelements unterschieden wird. Beispiel Chemiemolekül: Die Mikro-Repräsentation ist z.B. der Verweis auf die Masse des Moleküls innerhalb eines anderen Dokumentelements und die Makro-Repräsentation ist eine Darstellung des gesamten Dokumentelements, also z.B. die Zeichnung der chemischen Struktur. Weiteres Beispiel Fußnote: die Mikro-Repräsentation ist die hochgestellte Zahl im Text, d.h. ein einzelnes Attribut wird inline innerhalb eines anderen Dokumentelements dargestellt; die Makro-Repräsentation ist eine Abbildung des gesamten Dokumentelements, also die Fußnote im Ganzen. Diese Unterscheidung könnte ein weiterführendes Konzept sein.

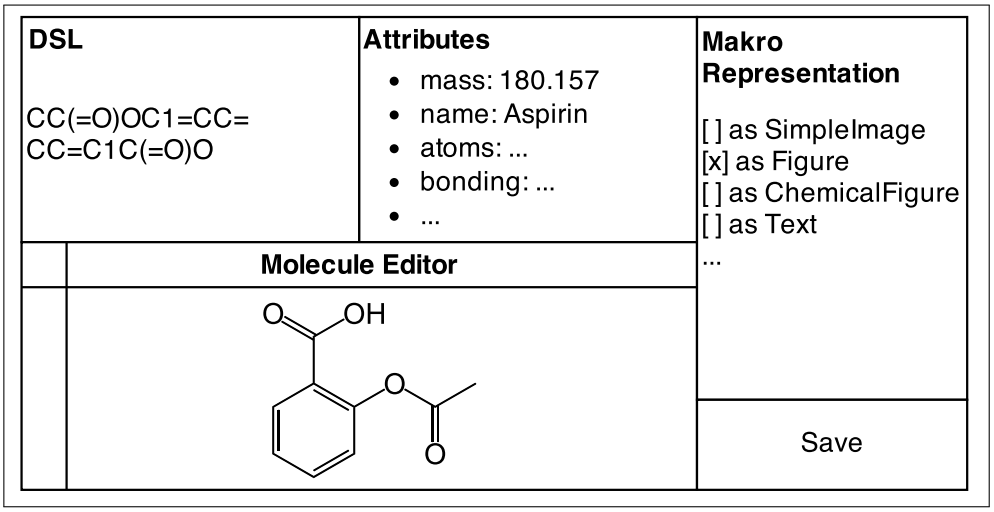


Abbildung 6.1: Konzept für eine Weiterentwicklung des semantischen Editors für Dokumentelemente, anhand des Beispiels chem. Strukturformel-Dokumentelement.

6.2.2 Visionen neuer Ausbaustufen und Anwendungsfälle

Ein Marktplatz für Dokumentelemente. Dort können Dokumentelemente z.B. als Software-Bibliotheken, oder direkt als SaaS via REST-Schnittstelle, zur Verfügung gestellt werden. Zur Konservierung und Wiederauffindbarkeit von Dokumentelementen, kann der Markplatz diese mit einem DOI (digital object identifier) versehen. Das kann auch als Geschäftsmodell dienlich sein: z.B. indem SaaS-Dokumentelemente, bei Laufzeit, Geld kosten (insbesondere interessant, wenn Dokumentelemente aufwändige Simulation durchführen).

Oder Autoren können wie in einem App-Store Dokumentelemente durch einmaliges bezahlen zukaufen (Account/Firmagebunden), oder mit Bezahlung-pro-Dokument.

Positionierung als Open-Access-Platform, zum schreiben sowie publizieren von akademischen Werken. Dieser Dienst kann z.B. Dokumentelemente aus dem Marktplatz anbieten. Dokumentelemente können auch assistieren, z.B. ein Abschnitt kann abhängig von seinem Kontext selbständig nach Zitaten suchen und Werke, die mutmaßlich darin vorkommen, vorschlagen und diese auf Wunsch automatisch ins Literaturverzeichnis aufnehmen. Oder eine Art Typographie-Lint, welches Warnungen herausgibt, falls bestimmte typografische Konventionen der gewählten Sprache mutmaßlich missachtet werden (z.B. falsche Anführungszeichen). Zudem können Autoren gegenseitig ihre Werke begutachten oder lektorieren. Ein Reputationssystem aus Zitaten und Begutachtungen kann besonders gute Gutachter herausheben. Ein Geschäftsmodell kann sein, dass Autoren bezahlte Gutachter oder Lektoren suchen bzw. beauftragen können und die Plattform erhält eine entsprechende Transaktionsgebühr pro Gutachten.

Erweiterte Autorenfunktionen. Ein Autor kann um das Dokument zugehörige Metadokumente aufbauen, wie beispielsweise Exzerpte. Aus diesen Exzerpten kann er direkt entsprechende Dokumentelemente in das Hauptdokument übernehmen. Ein kollaboratives Lektorat wird es vereinfachen, dass verschiedene Gutachter gleichzeitig zum Dokument Korrekturvorschläge und Kommentare einreichen.

Projektion von und in andere Formate, eine Format-Zentrale. Vorstellbar als eine Art „Reißwolf“ für Dokumente, es werden digitalisierte Handaufschriebe, Textdateien, PDFs etc. eingegeben; ein Klassifikator (der eventuell vom Dokumentelement direkt mitgeliefert wird) erkennt die verschiedenen Dokumentelemente die darin vorkommen. Der Autor bekommt vom System einen Gliederungsüberblick, der die aufgefunden Dokumentelemente anzeigt – dadurch kann er sie bequem in sein Dokument einbauen. Beispiel: Ein Chemiker kann seine im Labor von Hand aufgeschriebenen Tabellen automatisch in ein Tabellen-Dokumentelement übersetzen lassen und dieses weiterverarbeiten. Auf diese Weise können auch historische Dokumente aufbereitet werden. Da-

durch, dass das Dokument als abstrakter Syntaxbaum vorliegt, kann es in beliebige andere Formate transformiert werden. So könnte auch die Lektoratfunktion davon profitieren: Das Dokument wird z.B. in eine Word-Datei transformiert, der Lektor kann in seinem gewohnten Format Korrekturvorschläge machen; danach wird die annotierte Word-Datei in den Reißwolf gesteckt, dieser fusioniert die Korrekturvorschläge mit denen aus dem eigenen Lektorat. Ein Geschäftsmodell könnte sein: Die Benutzung des Reißwolfs kostet jeweils ein paar Cent, oder aufgefundene Dokumentelemente in das Dokument einzubauen kostet ein paar Cent.

Durch präzise definierte Dokumentklassen, kann der Editor dem Autor assistierend zur Seite stehen. Der Editor kann den Autor wie ein Sherpa durch die verschiedenen Vorgehensmodelle, die es bei der Erstellung von wissenschaftlicher Dokumentation zu beachten gibt, führen. So kann der Editor als ein Lernprogramm fungieren, welches die einzelnen Dokumententeile und ihre Bedeutungen erklärt und Tipps dazu gibt, welche Aussagen jeweils darin vorkommen sollten.

Durch Projektionen kann ein Dokument auf beliebige Art dargestellt werden. Es müssen auch nicht alle Bestandteile darin vorkommen oder können verkürzt dargestellt werden, oder mit zusätzlichen Anmerkungen versehen werden. Dies ermöglicht, dass der Dokumenteditor Strukturhilfen anbietet. Beispielsweise eine Mindmap-Ansicht, zum schnellen Umstrukturieren des Dokuments. Oder auch die Anzeige aller im Dokument vorkommenden Fußnoten, zur schnellen Überprüfung, ob diese gut ausformuliert sind.

Modellierung von Dokumentelementen, durch schreiben eines einfachen Dokuments. Ein Autor verfasst ein (formales) Dokument, welches wiederum ein spezielles Dokumentelement beschreibt. Dieses Dokument dient quasi direkt als Modell für das Dokumentelement. Darin wird beschrieben welche Attribute vorkommen oder wie die Repräsentationen gestaltet sein sollen. Danach kann das dort beschriebene Dokumentelement in einem anderen Dokument verwendet werden. Eventuell kann dieser Vorgang mit Knowledge-Discovery und durch Natural-Language-Programming verbessert werden, so dass sogar aus dem Text in natürlicher Sprache weitere Erkenntnisse über das Dokumentelement, bzw. dessen Modell, extrahiert werden können. Auf diese Art könnten

Wissenschaftler ihre Modelle immer weiter verfeinern. Beispiel: Atommodelle. Es gäbe ein Dokument, welches formal das Teilchenmodell von Demokrit beschreibt. Resultat ist ein entsprechendes Dokumentelement. Dalton bezieht sich nun darauf bzw. verwendet in seinem Dokument das Demokrit-Modell und entwickelt es entsprechend weiter. Resultat seines Dokuments ist ein Dokumentelement mit dem Dalton-Modell. Es basiert nun sogar direkt auf dem Demokrit-Modell. Das ist mutmaßlich bis zum Orbitalmodell ausbaubar. Damit gäbe es eine formale semantische Verknüpfung zwischen den Modellen. Ob dies umsetzbar oder überhaupt nützlich ist, müsste jedoch erst noch geklärt werden.

(Segaran u. a., 2009, S. 3) trifft es sehr präzise, wenn er behauptet, dass natürliche Sprache wohl die beste API-für-Wissen ist, die wir Menschen kennen. (Knuth, 1984, S. 1) hat dies in etwa auch so erkannt und daraus das programmiersprachenübergreifende Programmierparadigma des Literate-Programming entwickelt. Dort soll Software wie ein Literaturwerk behandelt werden, d.h. Dokumentation und Programmcode werden in einer Datei simultan untergebracht. Die hier vorgestellten Konzepte eigenen sich mutmaßlich hervorragend, um Literate-Programming umzusehen. Beispielsweise können einzelne größere Programmkomponenten wie z.B. ein For-Schleifen-Block auf spezielle Dokumentelemente abgebildet werden. Eine Funktionsdekleration wäre mit einem Kapitel vergleichbar, etc. Gemischt mit normalen Dokumentelementen, die beschreibend zur Seite stehen, könnte somit auch mit diesem System Literate-Programming umgesetzt werden. Zudem kann jedes Programm-Dokumentelement eine passende Visualisierung von Datenstrukturen oder Programmflüssen darstellen. Durch den Modellcharakter der Dokumentelemente wäre es somit möglich, ein Programm sogar in verschiedene Programmiersprachen zu transformieren. Anwendungsfall: Ein DSL-Skript kann von einem Wissenschaftler mit diesem System programmiert, visualisiert und gleichzeitig dokumentiert werden.

Wird das Konzept von Literate-Programming weitergedacht, ist es möglich einen solchen Dokumenteditor auch zur Generierung von Software zu nutzen. Beispielsweise kann aus Spray-Softwaremodellen automatisch Code generiert werden, direkt aus der Dokumentation heraus. In der Dokumentation sind zudem manueller Code und Erklärungen enthalten. Am Ende drückt der

Entwickler auf den generieren-Button und der Gesamt-Quellcode des Softwaresystem wird zusammengesetzt – ganz nach den Vorstellungen der modellgetriebenen Softwareentwicklung. Die Informatik baut immer Systeme für bestimmte Domänen und mit diesem System ist es leicht möglich verschiedenste Dokumentelemente mit verschiedensten Modellen zu mischen; also nicht nur für die Generierung von Software, sondern auch zu deren Dokumentation. Oder: Man stelle sich ein Buch über den Linux-Betriebssystem-Kernel vor. Darin sind Diagramme und Grafiken enthalten, welche die Funktionsweise des Kernels erklären. Was wäre, wenn diese Diagramme direkt in ausführbaren Code umgewandelt werden können? Wenn die erklärten Algorithmen nicht nur Code-Beispiele sind? Dann wäre das Buch selbst der ausführbare Kernel! Das würde jedoch bedeuten, dass das Dokument ein Dateisystem benötigt, um dort generierte Quellcode-Dateien ablegen zu können.

6.3 Schlussbemerkung

Zu Beginn der Masterarbeit war es gar nicht absehbar, welch gewaltiges Potenzial sich hinter der anfänglichen Idee verbarg. Genauer analysiert, ist aus der Idee eine umfassende Theorie über (wiss.) Dokumente im Allgemeinen entstanden. Diese Theorie kann die Sicht auf Textverarbeitungssysteme bzw. Dokumenteneditoren nachhaltig verändern. Der hier entstandene Prototyp läuft bereits so stabil, dass diese Abschlussarbeit – ohne größere Hürden – damit verfasst werden konnte. Mit dem Spray-Dokumentelement wurde ein domänenspezifischer Editor eingesetzt, mit dessen Hilfe ein Aspekt der Architektur grafisch dargestellt wird. Im erklärenden Text werden Attribute des Spray-Softwaremodells verwendet, welche Modell und Text semantisch miteinander verknüpfen. Das zeigt, dass Modell und erklärender Text so leichter konsistent gehalten werden können. Das System ist sehr flexibel, d.h. es kann problemlos um beliebige verschiedenartige domänenspezifische Inhalte erweitert werden. Diese Konzepte werden die nächste Evolutionsstufe der Textverarbeitungssoftware einläuten und die Vision des Semantic-Web maßgeblich vorantreiben, dessen bin ich mir sicher.

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

[Drachenfels 2013] Drachenfels, Heiko: *Modellgetriebene Softwareentwicklung. Teil 3: Domänenspezifische Sprachen*. http://www-home.htwg-konstanz.de/~drachen/mgse/Teil-3.pdf. Version: August 2013, Abruf: 2014-07-31. – Vorlesungsskript

[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

[Groza u. a. 2007] Groza, Tudor ; Handschuh, Siegfried ; Möller, Knud ; Decker, Stefan: SALT – Semantically Annotated LaTeX for Scientific Publications. Version: 2007. http://dx.doi.org/10.1007/978-3-540-72667-8_37. In: *The Semantic Web: Research and Applications*. Springer Science + Business Media, 2007. – DOI 10.1007/978–3–540–72667–8_37, 518–532

[Heese u. a. 2010] Heese, Ralf ; Luczak-Rösch, Markus ; Paschke, Adrian ; Oldakowski, Radoslaw ; Streibel, Olga: One Click Annotation. In: *6th Workshop on Scripting and Development for the Semantic Web* (2010), Mai

[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 2014] Hodapp, Sven: Final Prototype of Scaltex. (2014), Jul. http://dx.doi.org/10.5281/zenodo.11064. – DOI 10.5281/zenodo.11064

[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

[Knuth 1984] Knuth, Donald E.: Literate Programming. In: *The Computer Journal* 27 (1984), Feb, Nr. 2, 97-111. http://dx.doi.org/10.1093/comjnl/27.2.97. – DOI 10.1093/comjnl/27.2.97

[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

[Segaran u. a. 2009] Segaran, Toby ; Evans, Colin ; Taylor, Jamie: *Programming the Semantic Web*. Sebastopol : O'Reilly Media, Inc., 2009. – ISBN 978–1–449–37917–9

[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

[Strahringer 1998] Strahringer, 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.))

[Typesafe Inc. 2014] Typesafe 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]