Bachelorarbeit

Implementierung einer domänenspezifischen Sprache zur Spezifikation paralleler und nebenläufiger Prozesse in Scala

von Andreas K. Breer

Matrikelnummer:	5024552
Eingereicht am:	12. August 2013
Erstprüfer:	Prof. Dr. G. J. Veltink
Zweitprüfer:	Dipl.-Inform. Robert Bozic
Studiengang:	Medieninformatik

Zusammenfassung

Moderne Computerarchitekturen bieten eine immer bessere Unterstützung für die parallele Ausführung von Computerprogrammen. Die Entwicklung derartiger Programme erweist sich jedoch immer noch als sehr schwierig. Die Prozessalgebra stellt eine Möglichkeit zur Spezifikation nebenläufiger, paralleler und distributiver Systeme dar. Das Ziel dieser Arbeit ist es, die mathematische Theorie der Prozessalgebra in eine maschinenlesbare, ausführbare Form zu überführen. Hierzu soll die objektfunktionale Programmiersprache Scala zum Einsatz kommen. Im Verlauf dieser Bachelorarbeit wurden mehrere Ansätze ausprobiert. Es wurde sowohl eine interne DSL als auch eine externe DSL implementiert. Zur Simulation müssen die Terme in die sogenannte Head-Normal-Form überführt werden. Zu diesem Zweck kam das Pattern Matching zum Einsatz. Das im Rahmen dieser Bachelorarbeit entstandene Programm könnte beispielsweise in der Lehre eingesetzt werden, um Studierenden die Konzepte der Prozessalgebra näher zu bringen.

Danksagung

An dieser Stelle möchte ich mich bei Prof. Dr. G. J. Veltink und Dipl.-Inform. Robert Bozic für die Möglichkeit, in diesem interessanten Themenfeld zu arbeiten und für die freundliche und engagierte Betreuung während der Bachelorarbeit, bedanken. Ich danke auch meinem Arbeitgeber, der Johannesburg GmbH, die mir durch eine Verringerung meiner wöchentlichen Arbeitszeit den notwendigen Freiraum für die Erstellung dieser Arbeit einräumte. Des Weiteren möchte ich mich bei meiner Familie und meiner Freundin Carina Abeln für das Verständnis bedanken, das mir während der Bearbeitungszeit entgegengebracht wurde.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende wissenschaftliche Projektarbeit bis auf die offizielle Betreuung selbst und ohne fremde Hilfe angefertigt habe und die benutzten Quellen und Hilfsmittel vollständig angegeben sind.

Erklärung

Soweit meine Rechte berührt sind, erkläre ich mich einverstanden, dass die Bachelor-Arbeit Angehörigen der Hochschule Emden/Leer für Studium / Lehre / Forschung uneingeschränkt zugänglich gemacht werden kann.

_____

Datum, Unterschrift

Inhaltsverzeichnis

1. Einleitung

Im Verlauf dieser Bachelorarbeit soll eine domänenspezifische Sprache zur Spezifikation paralleler und nebenläufiger Prozesse entwickelt werden. Hierzu soll die objekt-funktionale Programmiersprache Scala zum Einsatz kommen. Im folgenden Kapitel wird auf die Notwendigkeit derartiger Spezifikationsmethoden im Kontext aktueller Entwicklungen im Hardwarebereich eingegangen.

1.1. Motivation

Seit einigen Jahren sind Prozessorhersteller dazu übergegangen, massiv Hyperthreading- und Multi-Core-Architekturen in ihren Produkten einzusetzen. Der Grund hierfür ist, dass die Taktfrequenz von Prozessoren aus technischen Gründen nicht in gewohnter Geschwindigkeit erhöht werden kann. Der Intel-Gründer Gordon Moore prognostizierte laut [Moo65] 1965, dass sich die Anzahl der Transistoren auf einem Chip alle 24 Monate verdoppeln würde. Die als „Moore's Law“ bekannte Gesetzmäßigkeit hat seine Gültigkeit bis heute nicht verloren. Die Anzahl der Transistoren auf einem Chip steigt immer noch exponentiell an. Bis ca. 2003 ging dies mit einem gleichzeitigen Anstieg der Taktfrequenz der Prozessoren einher. Die seitdem stagnierende Taktfrequenz führte dazu, dass sich die Transistoren heute auf mehrere Prozessorkerne verteilen. Dies stellt Programmierer vor neue Herausforderungen. Wie Herb Sutter in seinem Artikel „The free lunch is over“ [Sut05] darstellt, konnte die Performance von Anwendungsprogrammen früher alleine durch das Aufkommen einer neuen Prozessorgeneration erhöht werden, ohne dass die Programmierer hierzu etwas beitragen mussten. Die Anwendungsentwickler waren dadurch in der Lage, mit jeder neuen Programmversion immer aufwendigere Anwendungsprogramme mit immer mehr Anweisungen zu produzieren, ohne dass der Anwender Leistungseinbußen befürchten musste, da durch die gleichzeitig ansteigende Taktfrequenz der Prozessoren auch immer mehr Anweisungen pro Sekunde verarbeitet werden konnten. Anwendungsprogramme, die nicht für eine parallele Ausführung optimiert sind, profitieren jedoch kaum noch von der erhöhten Leistungsfähigkeit moderner Hyperthreading- und Multi-Core-Prozessoren. Dieser Sachverhalt führt nach Meinung von Herb Sutter zu einem ähnlich weitreichenden Paradigmenwechsel wie das Aufkommen der objektorientierten Programmierung vor über 30 Jahren.

Die Taktfrequenz ist allerdings nicht der einzige Indikator für die Leistungsfähigkeit eines Prozessors. Auch rein sequenzielle Anwendungen profitieren immer noch von größeren Caches und anderen technischen Weiterentwicklungen. Des Weiteren können mehrere, parallel laufende Anwendungsprogramme auf getrennten Prozessorkernen ausgeführt werden. So wird

zum Beispiel das Betriebssystem auf einem und das Anwendungsprogramm auf dem anderen Kern ausgeführt. Dies wirkt sich positiv auf die Gesamtperformance des Systems aus. Das grundsätzliche Problem bleibt jedoch erhalten. Rechenintensive Anwendungen, die nicht für eine parallele Ausführung optimiert sind, können sich nur die Leistungsfähigkeit eines einzigen Prozessorkerns zu Nutze machen.

Damit derartige Programme weiterhin am technischen Fortschritt im Hardwarebereich partizipieren können, müssen deren Algorithmen angepasst werden. Die parallele Ausführung einzelner Programmteile erhöht jedoch die Komplexität eines Softwareproduktes. Um einen sequenziellen Algorithmus auf einfache Weise in seine parallele Entsprechung zu überführen, werden in [Bre09] zwei Strategien vorgestellt. Bei der Task Decomposition wird der Algorithmus auf voneinander unabhängige Teilaufgaben untersucht, welche in beliebiger Reihenfolge ausgeführt werden können. Diese Teilaufgaben können dann jeweils in einem separaten Thread erledigt werden. Bei der Data Decomposition verarbeitet der Algorithmus eine große Menge gleichartiger Daten, wobei jedes Element oder zumindest einzelne Bereiche unabhängig voneinander verarbeitet werden können. Ein gängiges Beispiel ist ein großes zweidimensionales Array, dessen Reihen oder Spalten jeweils von einem eigenen Thread verarbeitet werden könnten.

Werden bei der Analyse der Algorithmen Abhängigkeiten zwischen den Teilaufgaben bzw. den einzelnen Datenelementen übersehen, kann die parallele Ausführung zu Problemen führen. Der gleichzeitige Zugriff auf Datenelemente könnte beispielsweise ohne einen wechselseitigen Ausschluss (engl. Mutual Exclusion) zu inkonsistenten Daten führen. Die Umsetzung eines wechselseitigen Ausschlusses durch Locks, Semaphore oder Monitore birgt jedoch die Gefahr eines Deadlocks. Bei der sequenziellen Ausführung werden die einzelnen Anweisungen in einer prognostizierbaren Reihenfolge ausgeführt. Währenddessen hat der Programmierer bei der parallelen Ausführung nur bedingt Einfluss auf die Reihenfolge, in der die einzelnen Programmteile ausgeführt werden. Diese Entscheidung obliegt dem Scheduler des Betriebssystems. Dadurch entsteht eine völlig neue Klasse an Programmfehlern, die bei einer sequenziellen Ausführung nicht auftreten können. Ein Programm kann somit bei gleicher Eingabe in 100 aufeinander folgenden Ausführungen 99-mal das richtige Ergebnis liefern und einmal fehlschlagen, obwohl sich die Rahmenbedingungen nicht verändert haben. Der Fehler tritt nur auf, wenn der Scheduler eine ungünstige Ausführungsreihenfolge wählt. Derartige Fehler sind schwer zu entdecken und steigern die Komplexität der Softwareentwicklung, da sie nicht vorsätzlich reproduziert werden können.

Dieser Komplexität ist es geschuldet, dass sich das Verhalten nebenläufiger Systeme im Allgemeinen nicht so einfach vorhersagen lässt, wie es bei rein sequenziell ablaufenden Programmen der Fall ist. Wie in [BA06] dargestellt, werden daher formale Methoden zur Spezifikation und Verifikation derartiger Systeme benötigt. Die Prozessalgebra stellt eine solche Methode dar. Dazu werden parallele Systeme als algebraische Terme beschrieben. Um Systeme zu verifizieren, kommt eine auf Gleichungen basierende Beweisführung (engl. Equational Reasoning) zum Einsatz. Dabei werden Prozessalgebra-Terme oder Bestandteile von Termen mittels Substitution durch gleichbedeutende Terme ersetzt. Dieses Verfahren kompensiert laut [BBR09] einige Nachteile gängiger Verifikationsmethoden. Das Model Checking birgt die Gefahr einer Zustandsraumexplosion, während beim interaktiven Theorembeweis nicht auf das Eingreifen des Anwenders verzichtet werden kann. Im Verlauf dieser Bachelorarbeit soll eine domänenspezifische Sprache entwickelt werden, welche die mathematische Theorie der Prozessalgebra in eine maschinenlesbare, ausführbare Form überführt. Dadurch lässt sich ein Programm entwickeln, mit dessen Hilfe mögliche Prozessabläufe simuliert werden können.

1.2. Aufbau der Arbeit

Im Grundlagenkapitel folgt zunächst eine kurze Einführung in die Prozessalgebra, bevor auf das Konzept der domänenspezifischen Sprachen und die verwendete Programmiersprache Scala eingegangen wird. Danach wird das klassische Wasserfallmodell vorgestellt und mit den nichtlinearen Vorgehensmodellen verglichen. Anschließend wird erläutert, warum für dieses Projekt ein inkrementelles Vorgehen gewählt wurde. Im folgenden Kapitel werden die Ergebnisse der Analyse- und Entwurfsphase besprochen. Zur Visualisierung werden eine Reihe von UML-Diagrammen präsentiert. Im Realisierungskapitel werden einige interessante Aspekte der Umsetzung näher beleuchtet, darunter der Aufbau des Domänenmodells durch Case-Klassen, die Erstellung einer internen und einer externen domänenspezifischen Sprache in Scala, sowie die Verarbeitung der Prozessalgebra-Terme mithilfe des Pattern-Matchings. Diese Arbeit schließt mit einem Fazit ab.

2. Grundlagen

Das Ziel dieses Kapitels ist es nicht, die vorgestellten Konzepte und Technologien vollständig und lehrbuchartig zu beschreiben. Es soll vielmehr eine solide Grundlage für das Verständnis dieser Arbeit geschaffen werden. In diesem Kapitel folgt zunächst eine kurze Einführung in die Prozessalgebra, bevor auf das Konzept der domänenspezifischen Sprachen und die Programmiersprache Scala eingegangen wird.

2.1. Prozessalgebra

Die Wurzeln der Algebra lassen sich bis ins 9. Jahrhundert n. Chr. zurückverfolgen. Wie in [SS11] treffend formuliert, nutzt dieses Teilgebiet der Mathematik Gleichungen, um Menschen zum Darstellen und Lösen von Problemen mit unbekannten Ergebnissen zu befähigen. Die Verwendung von Variablen ermöglicht es, Probleme auf einem höheren Abstraktionsniveau und somit allgemeingültig auszudrücken. Des Weiteren besteht in einer Algebra die Möglichkeit, algebraische Ausdrücke regelbasiert in gleichbedeutende Ausdrücke umzuformen. Wie der Turing-Award-Preisträger Tony Hoare im Vorwort zu [BBR09] darstellt, lernen heute bereits Schulkinder das Rechnen mit Variablen und Gleichungen kennen. Ihnen eröffnet sich damit das erste Mal das Potential mathematischer Abstraktion. Genau dieses Potential nutzt die Prozessalgebra, um das Verhalten nebenläufiger Systeme in Form algebraischer Terme auszudrücken.

In der Prozessalgebra werden Systeme als eine Menge gleichzeitig ausgeführter Prozesse beschrieben. Diese kommunizieren über ein nachrichtenbasiertes Modell miteinander und haben somit die Möglichkeit, sich gegenseitig zu beeinflussen. Wan Fokkink vergleicht in [Fok07, S.1] dieses Modell mit Ameisen, die auf der Suche nach Nahrung sind. Jede einzelne Ameise handelt als eigenständige Einheit. Findet eine Ameise einen Haufen Zucker, so sendet sie einen Lockstoff aus, um die anderen Ameisen über ihren Fund zu benachrichtigen. Die Ameisen, die den Geruch wahrnehmen, können dann entscheiden, wie sie auf die Nachricht reagieren.

Wie in [Bae05] dargestellt, entstanden im Verlauf der letzten 25 Jahre unterschiedliche Ansätze, um nebenläufige Prozesse auf eine algebraische Weise zu beschreiben. Die bedeutendsten Vertreter sind CCS (Calculus of Communicating Processes) von Robin Milner [Mil80], CSP (Communicating Sequential Processes) von Tony Hoare [Hoa78] und ACP (Algebra of Communicating Prozesses) von Jan Bergstra. In der 2009 erschienenen Publikation „Process Algebra: Equational Theories of Communicating Processes“ [BBR09] von J. C. M. Baeten, T. Basten und M. A. Reniers werden die wichtigsten Eigenschaften dieser drei Theorien in einem einheitlichen Framework vereint. Die vorliegende Arbeit wird sich mit diesem Frame-

work beschäftigen. Sie hat jedoch nicht den Anspruch einer vollständigen Einführung in die Prozessalgebra. Als Referenzwerk sei daher [BBR09] empfohlen. Im diesem Kapitel sollen nur die für das Verständnis dieser Arbeit wichtigen Prinzipien und Operatoren dargestellt werden.

2.1.1. Minimale Prozessalgebra MPT

Einführend wird in [BBR09, S.61-108] mit der MPT eine minimale Prozessalgebra vorgestellt, mit deren Hilfe sich simple sequenzielle Prozesse mit alternativen Ausführungszweigen spezifizieren lassen. Ein Prozess wird durch eine Menge atomarer Aktionen definiert, die im Folgenden auch nur als Aktionen bezeichnet werden. Sie stellt einen diskreten, nicht weiter unterteilbaren Bestandteil eines Prozesses dar. Die Ausführung einer Aktion markiert einen Zeitpunkt, der sie von anderen Aktionen unterscheidet. Ausgehend von dieser minimalen Prozessalgebra werden im Verlauf dieses Kapitels weitere Operatoren und Konstanten eingeführt. Die Prozessalgebra MPT beinhaltet eine Konstante, sowie eine binäre und eine unäre Funktion. Die Konstante 0, die auch als *Inaction* bezeichnet wird, definiert einen Zustand, in dem keine weitere Aktion ausgeführt werden kann. Mithilfe dieser Konstanten kann demnach ein Deadlock abgebildet werden. Andrew S. Tanenbaum definiert einen Deadlock in [Tan09, S.516] wie folgt:

> „Eine Menge von Prozessen befindet sich in einem Deadlock-Zustand, wenn jeder Prozess aus der Menge auf ein Ereignis wartet, das nur ein anderer Prozess aus dieser Menge auslösen kann."

Der Action Prefix $a.x$ ist eine unäre Operation, die für jede Aktion $a \in A$ existiert, wobei A die Menge aller möglichen Aktionen darstellt. Wenn x ein beliebiger Term der MPT ist, dann bedeutet $a.x$, dass zunächst die Aktion a ausgeführt wird, bevor mit dem Term x fortgefahren werden kann. Der Basisterm $a.0$ würde also die Aktion a ausführen und danach einen Deadlock verursachen. Mithilfe des Operators + kann eine Auswahl zwischen zwei Termen der MPT dargestellt werden. Der Operator wird auch als *Choice*-Operator bezeichnet. Wenn x und y Terme der MPT sind, dann verhält sich $x + y$ entweder wie der Term x oder wie der Term y.

Durch das Ausführen einer Aktion a auf einem Term x wird dieser nach x' abgeleitet. Die Regeln für derartige Ableitungen werden durch ein Term-Deduktionssystem definiert. Dabei bedient sich die Prozessalgebra einer sogenannten strukturierten operationalen Semantik, wie sie in [BBR09] beschrieben wird. Tabelle 1 stellt die Ableitungsregeln für die MPT dar.

$$a.x \xrightarrow{a} x \qquad \frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} \qquad \frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'}$$

Tabelle 1: Term-Deduktionssystem der MPT [BBR09])

Die erste Regel besagt, dass der Term $a.x$ durch Ausführung der Aktion a zum Term x abgeleitet wird, wobei a eine beliebige Aktion und x ein beliebiger Term der MPT ist. Die zweite Regel besagt, dass wenn für den Term x die Möglichkeit besteht, durch Ausführen der Aktion a nach x' abgeleitet zu werden, dann gilt dies auch für die Auswahl $x + y$. Das Gleiche gilt auch für die rechte Alternative, wie die dritte Regel zeigt.

Das Verhalten eines Prozessalgebra-Terms kann durch ein Transitionssystem dargestellt werden. Ein solches Transitionssystem besteht aus einer Menge von Knoten, die durch Kanten miteinander verbunden sind. Die Knoten enthalten Terme der Prozessalgebra. Die Kanten sind mit den ausführbaren Aktionen beschriftet. Besteht die Möglichkeit, dass Term x durch Ausführung der Aktion a nach x' abgeleitet wird, dann besitzt auch das Transitionssystem eine Kante mit der Beschriftung a, die vom Knoten x zum Knoten x' zeigt. Die Abbildung 1 zeigt die Transitionssysteme für die Terme $a.b.0$ und $a.b.0 + a.(b.0 + b.0)$.

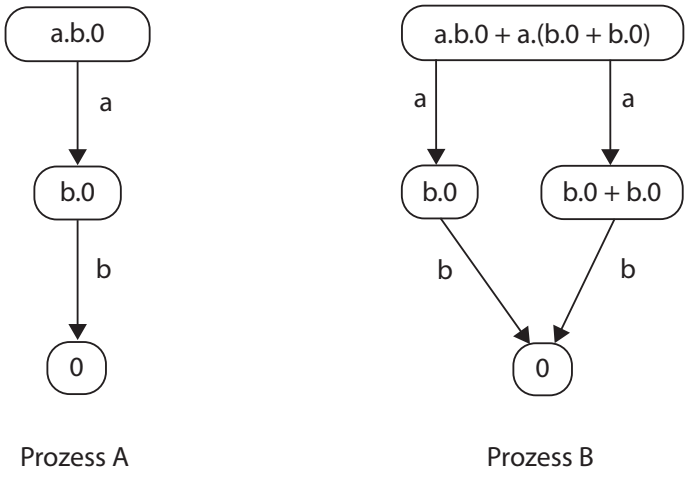

Abbildung 1: Transitionssystem (Eigene Darstellung, in Anlehnung an [BBR09])

Für eine Algebra ist der Begriff der Gleichheit von zentraler Bedeutung. Die Prozessalgebra definiert die Gleichheit zweier Prozesse über Bisimilarität. Während ein endlicher Automat den Zustand eines Systems zusammen mit den möglichen Zustandsübergängen beschreibt, liegt der Fokus der Prozessalgebra auf dessen Verhalten.

Die Geschichte „The Tiger and the Lady“ aus [BBR09, S.38] verdeutlicht diesen Sachverhalt. In der Geschichte muss sich ein Gefangener zwischen zwei Türen entscheiden. Hinter einer der beiden Türen verbirgt sich ein gefährlicher Tiger, der ihn fressen würde, hinter der anderen eine hübsche Frau. Die Frau würde ihn zum Mann nehmen und ihm die Freiheit schenken. Die beiden Prozessgraphen aus Abbildung 2 versuchen diese Situation mithilfe der Prozessalgebra darzustellen. Sie sehen auf den ersten Blick recht ähnlich aus, sind jedoch im Sinne der Prozessalgebra nicht gleich. Die Abbildung 2 zeigt die Transitionssysteme für die Terme $open.(eat.0 + marry.0)$ (Prozess A) und $open.eat.0 + open.marry.0$ (Prozess B).

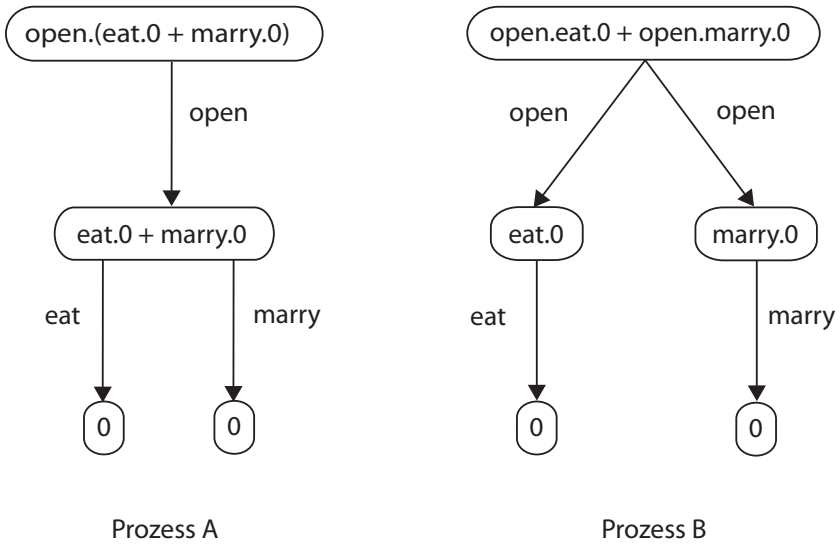

Abbildung 2: The Tiger and the Lady (Eigene Darstellung, in Anlehnung an [BBR09])

Würde man lediglich die möglichen Aktionsverläufe betrachten, so wären die beiden Prozesse gleich. Beide Prozesse bieten die Möglichkeit, entweder *open* und dann *eat* oder *open* und dann *marry* auszuführen. Abweichende Aktionsfolgen können in beiden Prozessen nicht vorkommen. Obwohl die möglichen Aktionsverläufe identisch sind, weisen die Prozesse doch ein voneinander abweichendes Verhalten auf. Der entscheidende Unterschied liegt im Moment der Auswahl. Während der Prozess A zuerst *open* ausführt und danach die Auswahl zwischen *eat* und *marry* trifft, verhält sich der Prozess B völlig abweichend. Hier wird bereits zu Beginn das weitere Verhalten des Prozesses durch die Auswahl zwischen *open* und *open* festgelegt. Der Prozess A würde demnach suggerieren, dass der Gefangene Einfluss darauf hat, ob ihm nach dem Öffnen der Tür der Tiger oder die hübsche Frau gegenübersteht.

Der Prozess B entspricht eher der Situation des Gefangenen. Dieser hat durch seine Wahl keine Möglichkeit, die weitere Entwicklung des Prozesses zu beeinflussen. Das Beispiel zeigt, dass es nicht ausreicht, allein die möglichen Aktionsfolgen zu betrachten, um Prozesse auf

Verhaltensgleichheit zu prüfen. Vielmehr ist es notwendig, zusätzlich zu den ausführbaren Aktionen auch die Verzweigungsstruktur zu berücksichtigen. Diese Form der Gleichheit wird auch als Bisimilarität bezeichnet und wird in [Fok07, S.11] wie folgt definiert:

> „Bisimulation equivalence [...] discriminates more processes than trace equivalence. Namely, if two processes are bisimilar, then not only they can execute exactly the same strings of actions, but also they have the same branching structure."

Der Prozess B veranschaulicht eine weitere Besonderheit der Prozessalgebra. Ob durch die Ausführung der Aktion *open* im Prozess B nun der rechte oder linke Zweig des Prozessgraphen weiterverfolgt wird, lässt sich nicht bestimmen. Diese Eigenschaft wird als Nichtdeterminismus bezeichnet. Ein derartiger Prozess kann bei gleicher Eingabe unterschiedliche Folgezustände aufweisen. Im Gegensatz zu deterministischen Prozessen lässt sich das Ergebnis demnach nicht exakt vorhersagen. Vielmehr existiert eine Menge von möglichen Ergebnissen. Robin Milner bezeichnet im Vorwort zu [BBR09] Nichtdeterminismus als die Regel bei der Spezifikation von Informationssystemen mit vielen interagierenden Komponenten. Verhält sich ein solches Informationssystem deterministisch, so ist dies ein Spezialfall. Wie in [AB02, S.116] erläutert, arbeiten reale Computersysteme immer deterministisch. Dessen ungeachtet ist das Konzept des Nichtdeterminismus sehr hilfreich, um komplexe Computersysteme mit vielen parallelen Komponenten zu entwerfen und zu analysieren. Für die Analyse derartiger Prozesse können entweder alle möglichen Folgezustände betrachtet werden, oder es kann zufällig einer der Folgezustände gewählt und somit nur ein möglicher Prozessverlauf weiterverfolgt werden.

Die Prozessalgebra MPT definiert eine Reihe von Axiomen der Form $s = t$ mit $s, t \in MPT$. In der folgenden Tabelle 2 sind die Axiome der MPT aufgestellt.

$x, y, z :$	
$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$x + 0 = x$	A6

Tabelle 2: Axiome der MPT [BBR09]

Die Axiome A1, A2 und A3 definieren die Eigenschaften Kommutativität, Assoziativität und Idempotenz des *Choice*-Operators. Das Kommutativgesetz (Axiom A1) besagt, dass die Argumente einer Operation vertauscht werden können, ohne dass sich das Ergebnis ändert. Dem-

nach besitzen die Alternativen einer Auswahl keine definierte Reihenfolge. Das Assoziativgesetz (Axiom A2) definiert, dass auch die Klammerung bei mehreren Alternativen beliebig ist. Die Idempotenz (Axiom A3) zeigt, dass eine Wahl zwischen zwei identischen Alternativen keine wirkliche Wahlmöglichkeit bietet. Somit kann diese Auswahl entfallen. Wie in Axiom A6 letztlich ausgedrückt, wird ein Deadlock solange wie möglich vermieden. Bei einer Wahl zwischen einem Term x und einer *Inaction* kann somit nur der Term x gewählt werden.

2.1.2. Erfolgreiche Termination BSP

Der Prozessalgebra MPT fehlt noch eine Möglichkeit zur Spezifikation eines erfolgreich terminierten Prozesses. Für die Analyse von Prozessen ist es von entscheidender Bedeutung, zwischen einer erfolgreichen und einer nicht erfolgreichen Termination differenzieren zu können. Genau diese Lücke wird in [BBR09] durch die Konstante 1 geschlossen, die auch als *Empty Process* bezeichnet wird. Es handelt sich somit um eine Erweiterung der Prozessalgebra MPT. Der Term

$$open.eat.0 + open.marry.1$$

stellt dar, dass ein wünschenswerter Prozessverlauf das Überleben des Gefangenen zur Folge hätte. Die um den *Empty Process* erweiterte Prozessalgebra trägt den Namen BSP. Die Tabelle 3 zeigt die neuen Regeln des Term-Deduktionssystems der BSP.

$$1\downarrow \qquad \frac{x\downarrow}{(x+y)\downarrow} \qquad \frac{y\downarrow}{(x+y)\downarrow}$$

Tabelle 3: Term-Deduktionssystem der BSP [BBR09]

Die BSP erweitert das Term-Deduktionssystem der MPT um drei weitere Regeln. Die erste Regel definiert, dass der *Empty Process* terminiert. Die anderen beiden Regeln besagen, dass wenn eine der beiden Alternativen einer Auswahl terminieren kann, dies auch für die gesamte Auswahl gilt. Die Abbildung 3 stellt die Beziehung zwischen BSP und MPT grafisch dar.
In [BBR09] werden eine Reihe derartiger Erweiterungen vorgenommen, die einer bestehenden Prozessalgebra neue Operatoren und Konstanten hinzufügen. Alle Terme der MPT behalten in der BSP unverändert ihre Gültigkeit. Die Menge aller MPT-Terme stellt demnach eine Teilmenge aller BSP-Terme dar. Wie in [BBR09, S.81] erläutert, kann es für eine solche Erweiterung zwei Gründe geben. Zum einen kann die Spezifikation von Systemen durch neue Konstanten und Operatoren vereinfacht werden. Zum anderen kann die Ausdrucksstärke der

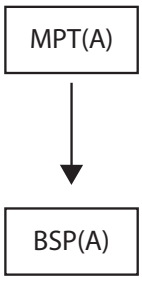

Abbildung 3: Erweiterung der MPT (Eigene Darstellung, in Anlehnung an [BBR09])

Algebra erhöht und somit Sachverhalte dargestellt werden, die in der ursprünglichen Algebra nicht darstellbar waren. Letzteres gilt auch für die Einführung der Konstante 1 in der BSP.

2.1.3. Rekursion

Die Möglichkeiten der BSP beschränken sich auf die Spezifikation von endlichen Prozessen. Mit jeder Ableitung verkürzt sich der Prozessterm. Um auch nicht endliche Prozesse spezifizieren zu können, wird im Folgenden das Konzept der Rekursion eingeführt. In [Ern08, S.523] wird Rekursion als Definition eines Verfahrens, einer Struktur oder einer Funktion durch sich selbst beschrieben.

Das folgende Beispiel ist aus [BBR09] entnommen und veranschaulicht die Spezifikation rekursiver Prozesse. Zunächst wird der Prozess-Term mit einem Namen versehen. Dieser kann dann innerhalb der Definition des Prozess-Terms referenziert werden. Dies geschieht mithilfe einer Rekursionsvariablen, die wie ein beliebiger Term der Prozessalgebra verwendet werden kann. Ein Kaffeeautomat kann mithilfe der BSP wie folgt spezifiziert werden:

$$SCM = quarter.coffee.1$$

Die Aktion *quarter* repräsentiert den Einwurf einer Münze in den Automaten. Danach wird der Kaffee ausgegeben, was durch die Aktion *coffee* dargestellt wird. Der Prozess terminiert danach und der Automat stellt somit seine Arbeit ein. Dies ist nicht das gewünschte Verhalten. Um darzustellen, dass der Automat wieder in den Ursprungszustand zurückkehrt, nachdem der Kunde seinen Kaffee erhalten hat, wird eine rekursive Spezifikation benötigt. Der folgende Term spezifiziert einen solchen Kaffeeautomaten.

$$SCM = quarter.coffee.SCM$$

Der Name des Prozesses wird auf der linken Seite des Gleichheitszeichens festgelegt, auf der rechten Seite kann dieser dann als Rekursionsvariable verwendet werden. Eine Variable kann

als Verweis auf einen bereits definierten Prozess-Term betrachtet werden. Dadurch, dass die Definition des Prozesses SCM auf sich selbst verweist, entsteht eine Endlosschleife. Das Beispiel verdeutlicht, dass durch die Rekursion eine potenziell unendliche Menge an Aktionsschritten mithilfe einer endlichen Notation ausgedrückt werden kann. In Abbildung 4 wird das zugehörige Transitionssystem dargestellt.

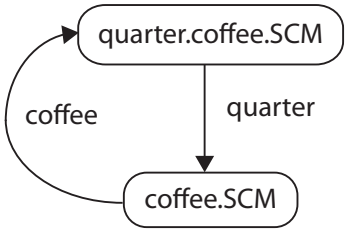

Abbildung 4: Kaffeeautomaten

Die Rekursion dient jedoch nicht nur zur Spezifikation unendlicher Prozesse. Sie kann auch herangezogen werden, um endliche Prozesse darzustellen, die Schleifen enthalten. Hierfür wird durch eine Auswahl die Möglichkeit geschaffen, die rekursive Ausführung des Prozesses zu beenden. Die Abbildung 5 stellt die Erweiterung der Prozessalgebra um Rekursion grafisch dar.

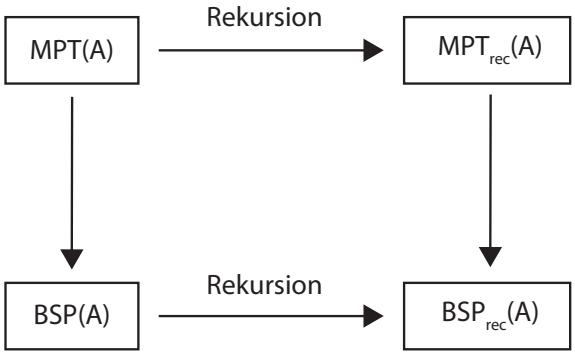

Abbildung 5: Erweiterung um Rekursion (Eigene Darstellung, in Anlehnung an [BBR09])

Um eine große Anzahl ähnlicher Alternativen darzustellen, wird eine verkürzte Notation eingeführt. Mithilfe des Symboles Σ wird jedes Element aus der angegebenen Menge auf den Term angewendet und mit einem $+$ verbunden. Diese Notation wird auch als *Generalized Choice* bezeichnet. Sei $D = \{0, 1, 2\}$, dann gilt:

$$\sum_{d \in D} add(d).1 = add(0).1 + add(1).1 + add(2).1$$

2.1.4. Sequenzielle Komposition

Die BSP enthält nur sehr beschränkte Möglichkeiten für die sequenzielle Komposition von Prozessen. Für die Spezifikation komplexerer Prozesse sind ausdrucksstärkere Mechanismen von großer Bedeutung. Daher wird in [BBR09] mit der TSP ein Operator eingeführt, der es erlaubt, Prozess-Terme hintereinander zu schalten. Somit kann die Reihenfolge, in der Teilprozesse auftreten, bestimmt werden.

Gegeben sind zwei Terme x und y. Der Term $x \cdot y$ führt zunächst den Term x aus. Sofern dieser erfolgreich terminiert, wird mit Term y fortgefahren. Verursacht der Term x einen Deadlock, so endet auch die sequenzielle Komposition $x \cdot y$ in einem Deadlock. Die Theory of Sequential Processes (TSP) erweitert die BSP um einen Operator zur sequenziellen Komposition. Die Tabelle 4 zeigt die neuen Axiome der TSP.

$x, y, z:$			
$(x + y) \cdot z = x \cdot z + y \cdot z$	A4	$0 \cdot x = 0$	A7
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5	$x \cdot 1 = x$	A8
$a.x \cdot y = a.(x \cdot y)$	A10	$1 \cdot x = x$	A9

Tabelle 4: Axiome der TSP [BBR09]

Das erste Axiom A4 definiert die Rechtsdistributivität der Sequenz über die Alternative. Linksdistributivität ist nicht gegeben, da diese den Moment der Auswahl verschieben würde und somit das Verhalten des Terms ändert, ähnlich wie im Beispiel „The Tiger and the Lady“. Das Axiom A5 zeigt die Assoziativität des neuen Operators. Das Axiom A8 stellt die bereits oben erwähnte Eigenschaft dar, dass nach einem Deadlock keine weitere Aktion ausführbar ist. Die sequenzielle Komposition eines Terms mit dem Empty Process hat keine Auswirkungen, wie die Axiome A8 und A9 zeigen. Letztlich zeigt das Axiom A10, dass der unäre Prefix-Operator stärker bindet als die Sequenz. Tabelle 5 zeigt das Deduktionssystem für den sequenziellen Operator.

$$\frac{x\downarrow \quad y\downarrow}{(x \cdot y)\downarrow} \qquad \frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x\downarrow \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'}$$

Tabelle 5: Term-Deduktionssystem der TSP [BBR09]

Die erste Regel zeigt, dass sofern sowohl Term x als auch Term y terminieren können, dies auch für die sequenzielle Komposition der beiden Terme gilt. Die zweite Regel definiert, dass alle Ableitungen des Terms x auch auf den Term $x \cdot y$ angewendet werden können. Sofern Term x terminiert ist, können alle Ableitungen des Terms y gebildet werden.

Mithilfe der TSP lassen sich bereits recht komplexe Systeme auf einfache Weise spezifizieren. Wie in [BBR09] beschrieben, kann zum Beispiel ein Stapelspeicher (engl. Stack) durch den folgenden Term beschrieben werden.

$$Stack = 1 + \sum_{d \in D} push(d).Stack \cdot pop(d).Stack$$

In [Ern08, S.177] wird ein Stapelspeicher als eine in der Informatik oft genutzte Datenstruktur beschrieben. Diese auch Kellerspeicher genannte Datenstruktur ist nach dem LIFO-Prinzip (Last-In-First-Out) organisiert. Demnach ist sie mit einem Stapel Teller vergleichbar. Das Element, welches als letztes auf den Stapel gelegt wurde, ist auch das erste Element, welches den Stapel wieder verlässt. Der Term spezifiziert eine Auswahl zwischen einer *Inaction* und der sequenziellen Komposition zweier Terme. Der erste Term beinhaltet die Aktion *push*, die dem Stapel ein neues Element hinzufügt. Nachdem die *push* Aktion aufgerufen wurde, wird der Prozess mittels Rekursion erneut ausgeführt. Dies bietet die Möglichkeit der Termination des ersten Teilterms oder der wiederholten Ausführung der Aktion *push*. Durch die sequenzielle Komposition der beiden Terme kann im Falle einer Termination des ersten Teilterms mit dem zweiten Teilterm fortgefahren werden. Durch die Aktion *pop* wird das Element wieder vom Stapel entfernt.

2.1.5. Umbenennen von Aktionen

Für die Spezifikation komplexerer Systeme ist es laut [BBR09, S. 189] erstrebenswert, Aktionen umbenennen zu können. Dies geschieht mithilfe des Operators $\rho_f(t)$, wobei f eine Funktion der Form $f : A \to A$ ist. Diese Funktion bildet jedes Element einer Menge von Aktionen auf eine neue Aktion ab. Diese auch als *Renaming* bezeichnete Operation kann auf einen Term der Prozessalgebra angewendet werden, woraufhin die Funktion f auf jede Aktion des Terms angewendet wird. Als Beispiel soll der bereits bekannte Term $a.b.0 + a.(b.0 + b.0)$ dienen. Wendet man den *Renaming*-Operator mit $f : b \to c$ auf diesen Term an, so wird Aktion b in die Aktion c umbenannt.

$$\rho_f(a.b.0 + a.(b.0 + b.0)) = a.c.0 + a.(c.0 + c.0)$$

Eine Spezialform des *Renaming*-Operators stellt der sogenannte *Encapsulation*-Operator $\partial_H(t)$ dar, wobei H eine Menge von Aktionen ist. Die Tabelle 6 stellt die Axiome des *Encapsulation*-Operators dar.

$x, y:$		
$\partial_H(1) = 1$		D1
$\partial_H(0) = 0$		D2
$\partial_H(a.x) = 0$	if $a \in H$	D4
$\partial_H(a.x) = a.\partial_H(x)$	if $a \notin H$	D5
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$		D6

Tabelle 6: Axiome Encapsulation-Operators [BBR09]

Dieser Operator benennt jedes Vorkommen eines Elements aus der Menge H im Term t in eine *Inaction* um. Hierdurch wird die Ausführung dieser Aktionen verhindert. Wie in [Vel95, S.24] dargestellt, ist der hauptsächliche Verwendungszweck dieses Operators, die Kommunikation zwischen zwei Prozessen zu steuern. Dies wird im nächsten Kapitel dargestellt. Ein weiterer verwandter Operator ist der *Skip*-Operator $\varepsilon_I(t)$. Jede Aktion in I wird in den *Empty Process* umbenannt und somit übersprungen. Ein Term $\varepsilon_I(a.x)$ mit $a \in I$, wird somit zu $\varepsilon_I(x)$, ohne Aktion a auszuführen.

2.1.6. Parallele Komposition

Wie Gert Veltink in seiner Dissertation [Vel95] erläutert, begründet sich ein großer Teil der Komplexität paralleler Systeme in der enormen Anzahl möglicher Zustände. Als Beispiel wird eine Kreuzung mit vier Ampeln herangezogen. Jede Ampel besitzt für sich genommen 3 Zustände. Sie kann ein rotes, gelbes oder grünes Lichtsignal abgeben. Das parallele System der Kreuzung besitzt somit theoretisch 3^4 also 81 Zustände. Jedoch ist der größte Teil dieser Zustände nicht zulässig, da diese einen Stau oder einen Autounfall provozieren würden. Durch die Kommunikation zwischen den einzelnen Ampeln muss also sichergestellt werden, dass das Gesamtsystem nur gültige Zustände annimmt.

Um von der tatsächlichen parallelen Ausführung zu abstrahieren, nutzt die Prozessalgebra eine Interleaving-Semantik, wie sie unter anderem in [BA06] beschrieben wird. Diese ist vergleichbar mit der quasiparallelen Ausführung mehrerer Prozesse auf einer Single-Core-CPU. Hierbei werden die sequenziellen Bestandteile der einzelnen Prozesse beliebig in einander verschachtelt. M. Ben-Ari definiert dies in [BA06, S. 8] so:

> „A concurrent program consists of a finite set of (sequential) processes. The processes are written using a finite set of atomic statements. The execution of a concurrent program proceeds by executing a sequence of the atomic statements obtained by arbitrarily interleaving the atomic statement from the processes. A computation is an execution sequence that can occur a result of the interleaving. Computations are also called scenarios.“

Demnach würde die konkrete Ausführung eines parallelen Systems durch ein Element einer Menge von Szenarien repräsentiert. Jedes Szenario bildet die atomaren Bestandteile der parallel ausgeführten sequenziellen Prozesse jeweils in einer unterschiedlichen Reihenfolge ab. Die Tabelle 7 aus [BA06, S. 9] stellt ein Beispiel zur Verdeutlichung dieses Prinzips dar. In diesem Beispiel werden die beiden Prozesse P und Q parallel ausgeführt, ohne dass eine Kommunikation zwischen ihnen stattfindet. Der Prozess P besteht aus den atomaren Aktionen $p1$ gefolgt von $p2$, der Prozess Q besteht aus $q1$ gefolgt von $q2$.

$$p1 \rightarrow q1 \rightarrow p2 \rightarrow q2$$
$$p1 \rightarrow q1 \rightarrow q2 \rightarrow p2$$
$$p1 \rightarrow p2 \rightarrow q1 \rightarrow q2$$
$$p1 \rightarrow q1 \rightarrow q2 \rightarrow p2$$
$$q1 \rightarrow p1 \rightarrow p2 \rightarrow q2$$
$$q1 \rightarrow q2 \rightarrow p1 \rightarrow p2$$

Tabelle 7: Ausführungsszenarien [BA06, S. 9]

Zu beachten ist, dass die Ausführungsreihe $p2 \rightarrow p1 \rightarrow q1 \rightarrow q2$ kein Szenario darstellt, da dies der sequenziellen Komposition des Prozesses P widerspräche. Des Weiteren verwendet die Prozessalgebra ein nachrichtenbasiertes Kommunikationsmodell, um Daten zwischen den Prozessen auszutauschen. Wie unter anderem in [BA06] dargestellt, abstrahiert sie damit von vielen Problemen, die durch die Verwendung eines gemeinsamen Speichers entstehen.

Die Prozessalgebra BCP aus [BBR09] erweitert die BSP um einen neuen Operator ||, der auch als *Merge*-Operator bezeichnet wird. Der Term $x \parallel y$ würde also darstellen, dass die Prozesse x und y parallel zueinander ausgeführt werden. Als Beispiel soll der Term

$$a.1 \parallel b.1$$

herangezogen werden. In der Semantik der Prozessalgebra bedeutet dies, dass entweder zunächst die Aktion a und dann Aktion b ausgeführt werden, bevor der Prozess terminiert, oder

die Aktion a folgt der Aktion b, oder die Aktionen a und b werden gleichzeitig ausgeführt. Dieses Verhalten lässt sich auch durch die folgende Gleichung beschreiben:

$$a.1 \parallel b.1 = a.b.1 + b.a.1 + \gamma(a, b).1$$

Die letzte Alternative zeigt die synchrone Ausführung der Aktionen a und b. Dies wird als $\gamma(a, b)$ ausgedrückt. Während einer solchen, gleichzeitigen Ausführung, haben die Prozesse die Chance, Nachrichten auszutauschen. Daher wird der Ausdruck $\gamma(a, b)$ auch als Kommunikation zwischen a und b bezeichnet. Damit zwei Prozesse kommunizieren können, muss eine Aktion der Form $\gamma(a, b) = c$ definiert sein, wobei die beiden Aktionen a und b in diesem Zusammenhang als Kommunikationsaktionen bezeichnet werden. Ist $\gamma(a, b)$ für zwei bestimmte $a, b \in A$ nicht definiert, so kann auch keine Kommunikation zwischen den Prozessen stattfinden. Die Kommunikation ist eine Funktion der Form $\gamma : a \times b \to c$.

Die synchrone Ausführung von Aktionen wird oft ausschließlich zum Nachrichtenaustausch verwendet. Die Möglichkeit, eine der beiden Kommunikationsaktionen einzeln auszuführen, ist im Prozess also gar nicht vorgesehen. Um dieses Verhalten zu spezifizieren, könnte der *Encapsulation*-Operator[1] verwendet werden. Es entspricht der *Encapsulation* ∂_H mit $H = \{a, b\}$. Da es sich um ein vielgenutztes Standardverhalten handelt, wird mithilfe von Kanälen eine vereinfachte Notation eingeführt. Das Konzept der Kanäle wurde das erste Mal von Tony Hoare in der Prozessalgebra CSP [Hoa78] eingesetzt. Der Term $p!d$ sendet das Datenelement d auf dem Kanal p, während der Term $p?d$ das Datenelement d auf dem Kanal p empfängt. Mithilfe dieser Notation wird verhindert, dass die Aktion $p!d$ bzw. $p?d$ einzeln ausgeführt wird. Daher verbleibt nur die Aktion zur Kommunikation zwischen den beiden Prozessen $\gamma(p!d, p?d) = c$.

Für eine vollständige Axiomatisierung des *Merge*-Operators ist es laut [BBR09] notwendig, noch zwei weitere Hilfsoperatoren einzuführen. Wie oben bereits beschrieben, kann die parallele Ausführung zweier Terme durch 3 Alternativen beschrieben werden. Die Gleichung

$$a.1||b.1 = a.b.1 + b.a.1 + \gamma(a, b).1$$

gilt zwar im Speziellen, jedoch lässt sich davon noch keine generelle Regel ableiten. Hierfür wird die BCP um den *Left-Merge*-Operator $||\lfloor$ und den *Communication-Merge*-Operator $|$ erweitert. Der *Left-Merge*-Operator erzwingt, dass die erste Aktivität aus dem linken Term kommt. Danach verhält sich dieser Operator wie der *Merge*-Operator. Ein *Right-Merge*-Operator wird nicht benötigt, da zu diesem Zweck ebenso gut die Operanden vertauscht werden können. Der *Communication-Merge*-Operator $|$ definiert, dass die erste ausführbare

[1] Der *Encapsulation*-Operator wurde im Kapitel 2.1.5 eingeführt

Aktion eine Kommunikation zwischen dem rechten und linken Term darstellt. Nachdem die Kommunikation stattgefunden hat, verhält sich auch dieser Operator wie der *Merge*-Operator. Die Tabelle 8 zeigt die Axiome der BCP.

$x, y, z:$			
$x \parallel y = x \mid\mid\lfloor y + y \mid\mid\lfloor x + x \mid y$	M	$x \mid y = y \mid x$	SC1
$0 \mid\mid\lfloor x = 0$	LM1		
$1 \mid\mid\lfloor x = 0$	LM2	$x \parallel 1 = x$	SC2
$a.x \mid\mid\lfloor y = a.(x \parallel y)$	LM3	$1 \mid x + 1 = 1$	SC3
$(x + y) \mid\mid\lfloor z = x \mid\mid\lfloor z + y \mid\mid\lfloor z$	LM4		
$0 \mid x = 0$	CM1	$(x \parallel y) \parallel z = x \parallel (y \parallel z)$	SC4
$(x + y) \mid z = x \mid z + y \mid z$	CM2	$(x \mid y) \mid z = x \mid (y \mid z)$	SC5
$1 \mid 1 = 1$	CM3	$(x \mid\mid\lfloor y) \mid\mid\lfloor z = x \mid\mid\lfloor (y \parallel z)$	SC6
$a.x \mid 1 = 0$	CM4	$(x \mid y) \mid\mid\lfloor z = x \mid (y \mid\mid\lfloor z)$	SC7
$a.x \mid a.x = c.(x \parallel y)$	if $\gamma(a, b) = c$		CM5
$a.x \mid a.x = 0$	if $\gamma(a, b) = c$ is not defined		CM6

Tabelle 8: Axiome der BCP [BBR09]

Mithilfe der Axiome M, LM1-4 und CM1-6 lässt sich jeder Term, der eine parallele Komposition beinhaltet, in einen Term ohne diese umformen. Somit ist es möglich, die Parallelität herauszurechnen. Dies ist ein entscheidender Faktor bei der computergestützten Analyse derartiger Prozesse. Die weiteren Axiome SC1–7 definieren Eigenschaften der *Merge*-Operatoren. So stellt zum Beispiel SC4 die Assoziativität des *Merge*-Operators dar.

2.1.7. Abstraktion

Die Abstraktion ist ein Grundprinzip der Softwaretechnik und vieler anderer Ingenieurswissenschaften. Sie wird von Helmut Balzert in [Bal11, S. 26] wie folgt definiert:

> „Unter Abstraktion versteht man Verallgemeinerung, das Absehen vom Besonderen und Einzelnen, das Loslösen vom Dinglichen. Abstraktion ist das Gegenteil von Konkretisierung."

In [Fok07, S. 49] wird als Beispiel eine Kundenanfrage an einen Softwareentwickler angeführt. Im Regelfall ist der Kunde in der Lage zu beschreiben, welche Eingabe das Programm akzeptieren muss und welche Ausgabe er erwartet. Er ist demnach in der Lage, das externe Verhalten des Programms zu beschreiben. Die internen Verarbeitungsschritte spielen für den Kunden keine Rolle, sofern er bei einer konkreten Eingabe die korrekte Ausgabe erhält. Um zu prüfen, ob die implementierte Software das gewünschte Verhalten aufweist, ist es erstrebenswert, die internen Verarbeitungsschritte ausblenden zu können. Genau dies leistet die Abstraktion.

Mit dem *Skip*-Operator[2] wurde bereits eine Möglichkeit eingeführt, Aktionen zu überspringen. In [BBR09, S. 245] wird jedoch dargestellt, dass das Überspringen einer Aktion das externe Verhalten eines Systems verändert. Als Beispiel soll der Term $a.1 + b.0$ dienen. Das Überspringen der Aktion b mithilfe des Skip-Operators würde demnach in

$$\varepsilon_I(a.1 + b.0) = a.1 + 0 = a.1$$

resultieren. Jedoch besaß der ursprüngliche Term die Möglichkeit in einen Deadlock zu verfallen. Diese Möglichkeit ist nach der Anwendung des *Skip*-Operators nicht mehr gegeben. Mithilfe des *Silent Steps* τ lassen sich Aktionen überspringen, die durch Ausführung der Aktion entstehenden Konsequenzen jedoch erhalten. Der *Silent Step* kann als Action Prefix $\tau.x$ in Termen der Prozessalgebra vorkommen. Er kann als eine Art prozessinterne Aktion betrachtet werden, die von außen nicht wahrnehmbar ist. Durch die Einführung des neuen Prefix Operators, wird die Prozessalgebra um folgende Regel erweitert:

$$a.(\tau.(x + y) + x) = a.(x + y)$$

Das neue Axiom stellt dar, dass die Ausführung eines *Silent Steps* in manchen Fällen keine Konsequenzen hat, und deshalb entfallen kann. Alternativen die anstelle eines *Silent Steps* ausgeführt werden könnten, bleiben auch nach Ausführung des *Silent Steps* erhalten. Mithilfe des *Hide*-Operators $\tau_I(x)$ wird jedes Vorkommen eines Aktion Prefix $a.x$ mit $a \in I$ in einen *Silent Step* Prefix $\tau.x$ umbenannt. Die Tabelle 9 stellt das Term-Deduktionssystem dar.

$$\frac{x\downarrow}{\tau_I(x)\downarrow} \qquad \frac{x \xrightarrow{a} x' \quad a \notin I}{\tau_I(x) \xrightarrow{a} \tau_I(x')} \qquad \frac{x \xrightarrow{a} x' \quad a \in I}{\tau_I(x) \xrightarrow{\tau} \tau_I(x')}$$

Tabelle 9: Term-Deduktionssystem des Hide-Operators (τ_I) [BBR09]

[2]Der *Skip*-Operator wurde im Kapitel 2.1.5 eingeführt

Die erste Regel stellt dar, dass die Anwendung der Abstraktion keinen Einfluss auf die Terminierung eines Terms hat. Die andern beiden Regeln zeigen, dass die Abstraktion keine Auswirkungen auf Aktionen hat, die nicht in I enthalten sind, während alle Aktionen in I durch ein τ ersetzt werden. Die Tabelle 10 zeigt die neuen Axiome für den *Hide*-Operator.

$x, y:$		
$\tau_I(1) = 1$		TI1
$\tau_I(0) = 0$		TI2
$\tau_I(a.x) = a.\tau_I(x)$	if $a \notin I$	TI3
$\tau_I(a.x) = \tau.\tau_I(x)$	if $a \in I$	TI4
$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$		TI5

Tabelle 10: Axiome Hide-Operators (τ_I) [BBR09]

Die Regeln entsprechen im Prinzip denen anderer *Renaming*-Operatoren, wie dem *Encapsulation*-Operator.

2.1.8. Zwischenfazit

Das Ziel dieser Arbeit ist es, durch geeignete Werkzeuge den formalen Spezifikationen der in diesem Kapitel vorgestellten Prozessalgebra Leben einzuhauchen. Es soll ein Programm entwickelt werden, welches den Benutzer bei der Spezifikation und Analyse von parallelen und distributiven Systemen mittels Prozessalgebra unterstützt. Die Prozessalgebra kann, da alle Regeln als Gleichungen definiert sind, relativ einfach als Computerprogramm umgesetzt werden. Die einzelnen Terme müssen dabei nicht auf abstraktem Niveau interpretiert werden, sondern können mittels Term-Rewriting-System in einen einfacher zu verarbeitenden Term umgewandelt werden.

2.2. Domänenspezifische Sprachen

In [VAC+09, S. 276] wird eine Domäne als ein entweder fachlich oder technisch motiviertes Interessens- oder Wissensgebiet definiert. Eine domänenspezifische Sprache (englisch: Domain Specific Language, kurz DSL) ist somit eine formale Sprache, die nicht zum Ziel hat, möglichst universell einsetzbar zu sein. Vielmehr handelt es sich um eine Spezialsprache zur Lösung ganz spezifischer Problemstellungen. Das Gegenteil von DSLs sind demnach General-Purpose Programmiersprachen (GPL), wie C, Java oder das im nächsten Kapitel vorgestellte Scala.

2.2.1. Aufbau

Als Ausgangspunkt für die Konzeption einer domänenspezifischen Sprache dient die Identifikation der relevanten Konzepte und Eigenarten der Anwendungsdomäne im Rahmen einer Domänenanalyse, wie sie unter anderem in [Gho11] beschrieben wird. Domänenspezifische Sprachen werden oft mit dem Ziel entwickelt, auch von Nichtprogrammierern genutzt oder zumindest verstanden zu werden. Eine domänenspezifische Sprache weist hierfür eine an die Anwendungsdomäne angepasste Syntax und Semantik auf. Bekannte Vertreter sind unter anderem SQL, HTML, CSS oder Rake (Buildsystem für Ruby). Laut [Fow09] ist der Sinn der meisten domänenspezifischen Sprachen, eine für den Anwender möglichst natürlich wirkende Schnittstelle zur Nutzung einer Programmbibliothek bereitzustellen. Zu diesem Zweck muss die domänenspezifische Sprache ein gemeinsames Vokabular mit der Anwendungsdomäne aufweisen. Durch die Verwendung von vertrauten Vokabeln soll Domänenexperten die Nutzung der Sprache erleichtert werden. Sie kann somit als Kommunikationsinstrument zwischen den Domänenexperten und den Anwendungsentwicklern eingesetzt werden. In [Gho11] lässt sich folgende Definition finden:

> „A DSL is nothing but a layer of abstraction over an underlying implementation model. The implementation model is nothing but an abstraction on top of the problem domain model, using the technology platform of the solution domain.“

Jedes Computerprogramm bildet Anforderungen einer Anwendungsdomäne auf eine Lösungsdomäne ab. Die Realisierung geschieht unter Verwendung geeigneter Werkzeuge und Techniken aus der Lösungsdomäne. Wie Abbildung 6 darstellt, bildet eine domänenspezifische Sprache eine Abstraktionsschicht zwischen den Artefakten der Anwendungsdomäne und den Artefakten der Lösungsdomäne.

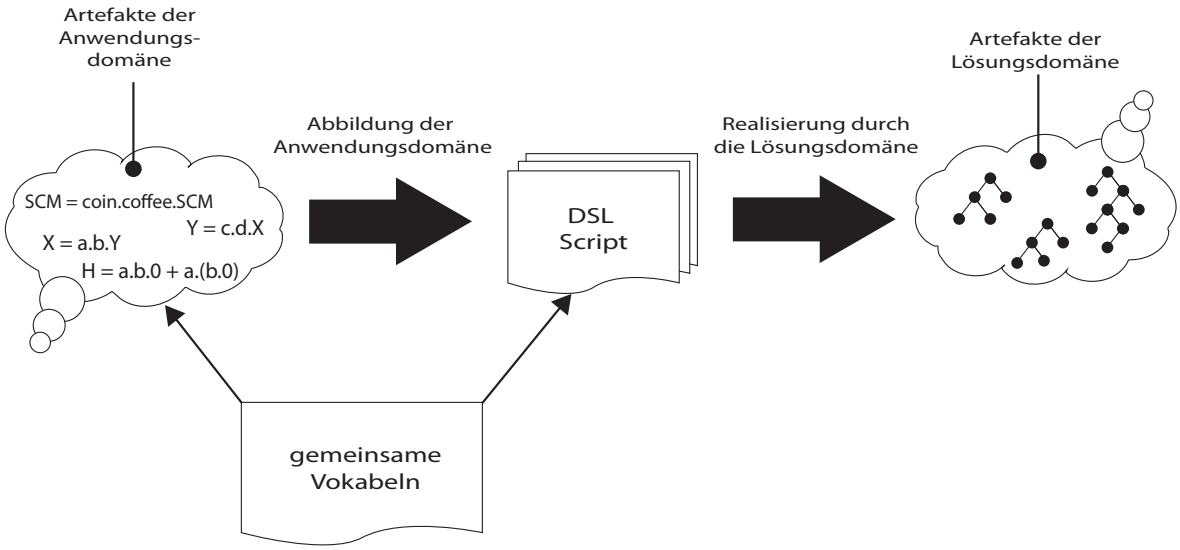

Abbildung 6: Struktur einer DSL (Eigene Darstellung, in Anlehnung an [Gho11])

Hierzu verwendet die domänenspezifische Sprache das Vokabular der Anwendungsdomäne und bildet dieses auf Artefakte der Lösungsdomäne ab. Üblicherweise ist zur Nutzung einer domänenspezifischen Sprache kein Verständnis für die Lösungsdomäne erforderlich. Die Syntax und Semantik liegen auf dem Abstraktionsniveau der Anwendungsdomäne.

Wie Martin Fowler in seinem Vortrag [Fow09] hervorhebt, ist der wichtigste Unterschied zwischen einer General-Purpose Programmiersprache und einer domänenspezifischen Sprache neben der Fokussierung auf einen bestimmten Anwendungsbereich, dass Letztere eine beschränkte Ausdrucksfähigkeit besitzt. Da mit ihr nur Aufgaben aus einer spezifischen Anwendungsdomäne gelöst werden sollen, ist sie in der Regel wesentlich kompakter aufgebaut als eine General-Purpose Programmiersprache. Den Kern einer domänenspezifischen Sprache bilden die für den Anwender wesentlichen Attribute der Anwendungsdomäne. Sie abstrahiert daher von den unwichtigen Details der Implementierung.

Als Beispiel für eine viel genutzte domänenspezifische Sprache soll die Structured Query Language (kurz: SQL) dienen. Sie wurde laut [Ste09] vom American National Standard Institut (ANSI) standardisiert und ermöglicht dem Anwender die Kommunikation mit relationalen Datenbanksystemen, um Daten abzufragen, zu manipulieren oder zu schützen. Datenbankabfragen werden in einem deklarativen Stil formuliert, bei dem nur noch das Ergebnis definiert wird und nicht mehr die Einzelschritte formuliert werden, die zu diesem führen. Der Anwender von SQL braucht demnach keine Kenntnisse über den internen Aufbau der Datenbank. Er braucht zum Beispiel nicht zu wissen, in welcher Datei die gesuchten Informationen liegen, oder wie diese organisiert sind. Dies ist Teil der Lösungsdomäne des Datenbanksystems. Der

Anwender kann mit den Modellen der Anwendungsdomäne, wie Tabellen und Spalten, arbeiten und braucht sich nicht mit den Eigenschaften der Lösungsdomäne auseinanderzusetzen.

Viele Techniken zur Implementierung einer DSL zielen darauf ab, aus dem DSL-Skript Quellcode einer General-Purpose Sprache zu erzeugen. Dieser wird gegebenenfalls noch kompiliert und kann dann ausgeführt werden. Die Erzeugung von Quellcode ist jedoch nicht die einzige Möglichkeit. Wie in [Fow09] erläutert, reicht es oft aus, mithilfe eines DSL-Skriptes direkt eine Menge von Domänenobjekten im Speicher zu erzeugen.

Domänenspezifische Sprachen lassen sich laut [Gho11] auf unterschiedliche Arten klassifizieren. Eine weit verbreitete Taxonomie stellt die von Martin Fowler in [FP10] verwendete Unterteilung in interne und externe DSLs dar. Die Klassifizierung geschieht hierbei anhand der Implementierung der DSL.

2.2.2. Interne DSL

Der Ausgangspunkt einer internen DSL stellt eine bereits existierende Wirtssprache dar. Ausgehend von dieser wird die domänenspezifische Semantik in Form einer Bibliothek implementiert. Die interne DSL kann somit an der bestehenden Infrastruktur der Wirtssprache partizipieren. Zur Implementierung einer internen DSL können laut [Gho11] folgende Techniken verwendet werden.

- Embedded
  - Smart API
  - Syntax-Tree Manipulation
  - Typed embedding
  - Reflective Metaprogramming
- Generative
  - Compile-time Metaprogramming
  - Runtime Metaprogramming

Es existieren demnach ein eingebetteter und ein generativer Ansatz. Der Unterschied besteht darin, dass bei den eingebetteten DSLs der gesamte Code vom Nutzer erstellt wird, während beim generativen Ansatz ein Teil des Codes entweder zur Laufzeit oder während der Kompilierung generiert wird. Beim generativen Ansatz muss die Wirtssprache Konzepte der Metaprogrammierung unterstützen, die es erlauben, die Struktur eines Programms zu verändern. Wie in [VAC+09, S. 164] beschrieben, wird dieses Konzept zurzeit nur von wenigen

Mainstream-Sprachen unterstützt, da es als nur schwer beherrschbar gilt. Daher kommen die Techniken des eingebetteten Ansatzes ohne dieses Konzept aus.

Bei einem Smart API handelt es sich um eine besonders aussagekräftige Programmierschnittstelle. Um dies zu erreichen, wird zum Beispiel das Fluent Interface eingesetzt, welches von Martin Fowler in [Fow05] beschrieben wird. Hierbei werden Methodennamen und Rückgabewerte so gewählt, dass ihre Verwendung einer Art Grammatik folgt. Die Verwendung von Bibliotheken wird durch dieses Vorgehen vereinfacht und die Lesbarkeit des entstehenden Codes wird erhöht. Das Listing 1 illustriert dieses Konzept:

```
customer.newOrder()
        .with(6, "TAL")
        .with(5, "HPK").skippable()
        .with(3, "LGV")
        .priorityRush();
```

Listing 1: API mit Fluent Interface [Fow05]

Martin Fowler führt als Beispiel eine Klassenstruktur zur Verwaltung von Bestellungen an. Bei einem klassischen API-Design werden verschiedene Objekte erzeugt und dann miteinander verbunden. Die in [Mey98] beschriebene Command Query Separation stellt ein oft verwendetes Prinzip im objektorientierten API-Design dar. Die grundlegende Idee ist hierbei, Methoden, die den Zustand eines Objektes verändern und solche die den Zustand abfragen, strikt voneinander zu trennen. Jede einzelne Methode verfolgt einen gesonderten Zweck, der sich aus ihrem Namen ableiten lässt. Bei einem Fluent Interface ist dies nicht der Fall. Methoden wie *with* machen nur im Kontext anderer Methodenaufrufe Sinn. Die Methodenaufrufe können dabei in einer Art Kette hintereinander gesetzt werden. Dies wird laut [Fow05] durch einen geschickt gewählten Rückgabewert erzielt. In diesem Zusammenhang kommt oft das Builder Entwurfsmuster aus [GHJV94] zum Einsatz. Bei einem Smart API wird also versucht, eine Programmierschnittstelle zu erzeugen, die sich an die übliche Notation der Anwendungsdomäne anlehnt und trotzdem den syntaktischen Forderungen der Wirtssprache genügt. Das Ziel ist es also, die Lesbarkeit des Codes für Domänenexperten zu erhöhen.

Bei der Syntax-Baum Manipulation wird mithilfe der Infrastruktur der Wirtssprache ein abstrakter Syntax-Baum erstellt bzw. verändert. Nachdem der Syntax-Baum aufgebaut wurde, kann dieser in eine Textrepräsentation überführt werden. Somit wird modellbasiert Quellcode der Wirtssprache erzeugt. Wie in [Ull11a] beschrieben, bietet zum Beispiel Eclipse eine API [ECLa], mit der sich Java-Programmcode einlesen, aufbauen und manipulieren lässt.

Beim Typed Embedding wird das Typsystem der Wirtssprache genutzt, um domänenspezifische Typen und Operationen zu implementieren. Die Objektorientierung stellt einen geeigneten Rahmen hierfür dar. Ein Objekt ist nach [LR09] ein Container, der semantisch zusammengehörige Daten und Methoden kapselt. Objekte sind dabei selbst Daten. Sie können in anderen Containern enthalten sein, als Argumente an Methoden übergeben werden oder das Resultat einer Methode darstellen. Dieses Konzept kann genutzt werden, um ein Domänen-Modell zu erzeugen und dieses dem Anwender als Programmierschnittstelle bereitzustellen. Die konkrete Implementierung bleibt innerhalb der Klassen verborgen. Wie in [Gho11] dargestellt, kann durch eine typsichere Wirtssprache gewährleistet werden, dass das DSL-Skript bestimmte Regeln befolgt. So ist beispielsweise sichergestellt, dass Methoden immer mit den richtigen Eingaben versorgt werden. Wird einer Methode als Argument ein Objekt des falschen Typs übergeben, so gibt der Compiler eine Fehlermeldung aus und beendet die Übersetzung.

Reflektion stellt laut [VAC+09, S. 164] eine abgeschwächte Form der Metaprogrammierung dar, bei der das Programm lesenden Zugriff auf die eigene Struktur besitzt. Mithilfe der Java-Reflection-API ist es laut [Ind11] möglich, Klassen zu instanziieren, die zum Zeitpunkt der Kompilierung noch unbekannt sind. Es können Metainformationen über Klassen abgerufen werden, jedoch lässt sich ihre statische Struktur nicht verändern. Dafür sind weitergehende Konzepte der Metaprogrammierung erforderlich. Reflection wird oft genutzt, um zur Laufzeit Objekte und Methodenaufrufe aufgrund einer Konfigurationsdatei zu erzeugen.

Wie bereits oben beschrieben, nutzen die beiden generativen Techniken die Metaprogrammierung, um die Programmstruktur zu ändern. In [Gho11] wird sie als eine Möglichkeit beschrieben, Programme zu erstellen, die in der Lage sind, Programme zu schreiben. Der Unterschied zwischen den beiden Techniken liegt im Zeitpunkt der Veränderung. Bei der Compile-time Metaprogrammierung geschieht dies bei der Kompilierung, während bei der Runtime Metaprogrammierung die Veränderungen zur Laufzeit durchgeführt werden. In der Programmiersprache Groovy ist es laut [Sta07] mithilfe der Metaprogrammierung möglich, Objekte um Methoden und Attribute zu erweitern, die sie eigentlich nicht besitzen. Hier kommt das Meta-Object-Protokoll zum Einsatz. Das Ziel des generativen Ansatzes ist es, dass der Anwender der DSL nur noch Code mit semantischer Bedeutung für die Anwendungsdomäne schreibt und die restliche Implementierung durch Metaprogrammierung automatisch erzeugt wird.

2.2.3. Externe DSL

Eine externe DSL wird vom Grund auf neu entwickelt. Während eine interne DSL die Infrastruktur der Wirtssprache nutzt, besitzt eine externe DSL im Normalfall eigene Mechanismen zur lexikalischen Analyse, zum Parsen, zum Kompilieren und zur Code-Generierung. Der Parser ist in der Lage, das DSL-Skript in eine ausführbare Form zu überführen. In [Gho11] werden folgende Techniken zur Erstellung einer externen DSL identifiziert.

- Context-driven string manipulation
- Transforming XML to consumable resource
- DSL workbench
- DSL with embedded foreign code
- DSL design based on parser combinators

Die einfachste Form einer externen DSL stellt die Context-Driven String Manipulation dar. Ein Programm liest einen in einer domänenspezifischen Sprache formulierten Text ein und transformiert diesen mittels String-Manipulation in den korrespondierenden Quellcode einer existierenden Programmiersprache. In [Ull11a] wird hierzu die Verwendung von Templatesystemen empfohlen.

Die Extensible Markup Language (XML) ist laut [SN09] eine Auszeichnungssprache zur Beschreibung von hierarchisch strukturierten Daten. XML definiert kein konkretes Datenformat. Vielmehr handelt es sich um eine Metasprache, welche die Basis für die Entwicklung weiterer anwendungsorientierter Auszeichnungssprachen bildet. Der XML-Standard legt lediglich fest, welche syntaktischen Bestandteile zulässig sind und wie diese miteinander kombiniert werden. In vielen Fällen ist es sinnvoll, die XML-Syntax weiter zu beschränken und um eigene Regeln zu erweitern. Mithilfe von Definitionssprachen wie *DTD* [W3C02a] oder *XML-Schema* [W3C04] ist es möglich, Vorgaben bezüglich der zu verwendenden Grammatik zu machen. Auf diese Weise entstehen domänenspezifische Sprachen, die auf dem XML-Standard fußen. Beispiele hierfür sind *RSS* [RSS09], *MathML* [W3C10] oder *XHTML* [W3C02b].

Zur Implementierung einer externen DSL können auch Parsergeneratoren wie *YACC* oder *ANTLR* [ANT] verwendet werden. Als Eingabe erwartet ein solcher Parsergenerator die Spezifikation der Syntax einer Sprache. Daraus produziert er den Quellcode eines Programms, welches in der Lage ist, diese Sprache zu parsen. Im Regelfall lassen sich Anweisungen hinterlegen, die bei der Verarbeitung ausgeführt werden sollen. Dadurch lässt sich während des Parsens der domänenspezifischen Sprache ein semantisches Domänenmodell erzeugen.

Im Gegensatz zu Parsergeneratoren wird bei einer DSL Workbench nicht nur ein Parser generiert, sondern auch ein Klassenmodell und eine eigene Entwicklungsumgebung für die domänenspezifische Sprache. Eine DSL Workbench stellt somit ein spezielles Werkzeug zur Erstellung einer domänenspezifischen Sprache dar. Bekannte Vertreter sind unter anderem *XText* [ECLb] und *MPS* [MPS].

Mit den Parser Combinators besteht in einigen Programmiersprachen die Möglichkeit auf die Nutzung eines externen Werkzeugs wie *YACC* oder *XText* zu verzichten. Ein *Parser Combinator* ist eine Funktion höherer Ordnung, die eine Menge von Parsern als Eingabe erwartet und einen neuen Parser zurückgibt. Dadurch können die Parser miteinander kombiniert werden. Ausgehend von sehr simplen Parsern für einzelne Artefakte der Anwendungsdomäne können so immer komplexere Strukturen abgebildet werden.

2.2.4. Zwischenfazit

Um die mathematische Theorie der Prozessalgebra in eine maschinenlesbare, ausführbare Form zu überführen, soll eine domänenspezifische Sprache entwickelt werden. Im Verlauf dieser Bachelorarbeit werden mehrere Ansätze ausprobiert. Es soll sowohl eine interne DSL mittels SmartAPI als auch eine externe DSL durch Parser Combinators in Scala implementiert werden.

2.3. Scala

Scala ist eine, mit Java oder C++ vergleichbare General-Purpose Programmiersprache. Eine detaillierte Darstellung des Sprachumfangs ist der Spezifikation [EPF11] zu entnehmen. Der Name steht laut [OSV11] für „Scalable Language“ und soll verdeutlichen, dass Scala sowohl für die Erstellung sehr großer Systeme als auch für die Programmierung kleiner Skripte geeignet ist. Es handelt sich um eine statisch typisierte Sprache, die sowohl objektorientierte als auch funktionale Paradigmen in sich vereint.

2.3.1. Entstehung

Die Programmiersprache Scala kann genutzt werden, um Programme für die Java Virtual Maschine zu erstellen. Unter der Leitung von Professor Dr. Martin Odersky wird sie an der renommierten École polytechnique fédérale de Lausanne (ETFL) in der Schweiz entwickelt. Die erste Version für die Java Virtual Maschine wurde laut [EPF13] im Januar 2004 veröffentlicht. Martin Odersky führt im Interview [VS09a] durch die Entstehungsgeschichte der Programmiersprache Scala. Bereits ein Jahr nachdem die erste Java-Version veröffentlicht wurde, startete er die ersten Versuche, eine neue Sprache für die Java Virtual Maschine zu entwickeln. Das Ergebnis war die Programmiersprache Pizza. Diese erweiterte Java um funktionale Aspekte, wie Generics, Funktionen höherer Ordnung und Pattern Matching. Erst sechs Jahre später wurde das Konzept der Generics auch in Java integriert. Hieran war Martin Odersky maßgeblich beteiligt.

Nach Meinung von Martin Odersky sind der Weiterentwicklung der Programmiersprache Java Grenzen gesetzt, die eine schnelle Integration neuer Features oftmals verhindern. In der Vergangenheit getroffene Fehlentscheidungen im Design von Java lassen sich nicht mehr revidieren, da ansonsten keine vollständige Abwärtskompatibilität gewährleistet werden kann. Dies beeinflusst die weitere Entwicklung nachteilig. Aus diesen Gründen sah er die Notwendigkeit, eine völlig neue Programmiersprache zu entwickeln. Scala ist binärkompatibel zu dem aus Java-Quellcode erzeugten Bytecode, ohne dabei quellcodekompatibel sein zu müssen. Hierdurch kann sie von den bei der Entwicklung von Java und Pizza gewonnenen Erkenntnissen in einem Maße profitieren, in dem es Java selbst nicht möglich ist. Ohne Rücksicht auf Abwärtskompatibilität beim Quellcode nehmen zu müssen, kann diese Sprache völlig neue Konzepte umsetzen und gleichzeitig an der hochoptimierten Laufzeitumgebung und den umfangreichen Klassenbibliotheken partizipieren.

2.3.2. Skalierbarkeit

Dieses Kapitel wird sich mit der namensgebenden Eigenschaft der Programmiersprache Scala beschäftigen, ihrer Skalierbarkeit. In „The cathedral and the bazaar“ [Ray99] vergleicht Eric Raymond mithilfe einer Metapher zwei unterschiedliche Herangehensweisen an die Softwareentwicklung. Dabei steht die Kathedrale für den klassischen Ansatz, bei dem eine Software nach einem „Bauplan“ erstellt wird. Nur ein sehr eingeschränkter Personenkreis hat Einfluss darauf. Der Quellcode der Software bleibt also im Unternehmen und wird der Öffentlichkeit nicht zugänglich gemacht. Nachdem der Plan umgesetzt wurde, ist das Ziel erreicht und das Projekt abgeschlossen. Der Basar dagegen ist wesentlich organischer aufgebaut. Er besteht aus einer Vielzahl an Ständen, die alle gleichberechtigt nebeneinander existieren. Auf diesem Basar kann jeder, der etwas beitragen möchte, seine Waren anbieten. Etablierte Stände bleiben lange bestehen oder vergrößern sich sogar. Andere funktionieren nicht so gut und verschwinden schnell wieder oder werden ersetzt. Der Quellcode wird also veröffentlicht und jeder hat prinzipiell die Möglichkeit, Änderungen daran vorzunehmen, um die Software an die eigenen Anforderungen anzupassen. Fehlerkorrekturen und weitere sinnvolle Änderungen fließen dann in das Projekt zurück, sodass die restlichen Anwender davon profitieren. Man kann nicht bestimmen, wann das Projekt abgeschlossen ist, es erreicht lediglich einen Zustand, in dem es verwendet werden kann. Diese Vorgehensweise entspricht der Open-Source Bewegung, aus der Produkte wie Linux oder der Apache Webserver hervorgegangen sind.

Wie im „Growing a language“ [Ste99] von Guy L. Steele Jr. schlüssig argumentiert wird, kann die gleiche Metapher auch auf das Design von Programmiersprachen angewandt werden. Nach seiner Auffassung können die Entwickler einer Programmiersprache nicht alle möglichen Anwendungsfälle vorhersehen und somit im Design der Sprache berücksichtigen. Daher sollte eine Programmiersprache Raum für Wachstum lassen. Dieser Raum würde dann zwangsläufig von den Anwendern mit sinnvollen Erweiterungen für ihre spezifische Anwendungsdomäne gefüllt. Er bezieht sich dabei auf die Aussagen des amerikanischen Architekten Christopher Alexander, der in [Ale78] darstellte:

> „After all, the very existence of a master plan means, by definition, that the members of the community can have little impact on the future shape of their community, because most of the important decisions have already been made. In a sense, under a master plan people are living with a frozen future, able to affect only relatively trivial details. When people lose the sense of responsibility for the environment they live in, and realize that they are merely cogs in someone else's machine, how can they feel any sense of identification with the community, ...“

Scala könnte als eine Umsetzung des Basar-Stils betrachtet werden. Die Sprache besteht aus relativ wenigen, klaren Konzepten, gibt dem Anwender jedoch die Möglichkeit, den Sprachumfang zum Beispiel mithilfe eigener domänenspezifischer Sprachen selbstständig zu erweitern. Die Anwender haben somit großen Einfluss auf die zukünpftige Entwicklung der Sprache.

2.3.3. Eigene Datentypen

Eine Möglichkeit der Einflussnahme besteht in der Einführung spezieller Datentypen. Die Objektorientierung stellt mit ihrem Klassen-Konzept hierfür einen geeigneten Mechanismus zur Verfügung. Scala setzt das objektorientierte Paradigma wesentlich konsequenter als Java um. In Java existieren neben Objekten laut [Ull11b] auch noch primitive Datentypen für Zahlen, Unicode-Zeichen und Wahrheitswerte. Aus Performancegründen entschied sich Sun Microsystems gegen eine vollständige Objektorientierung. Im Gegensatz zu Objekten können auf den primitiven Datentypen Operationen angewendet werden. Für Ganzzahlen sind zum Beispiel die Operationen Addition (+), Subtraktion (-), Multiplikation (*) und Division (/), sowie der Modulo-Operator (%) definiert.

Die Programmiersprache Scala kennt keine primitiven Datentypen. Hier ist jeder Wert ein Objekt und jede Operation ein Methodenaufruf. So existiert für die Repräsentation von Ganzzahlen die Klasse *Int*. Diese Klasse enthält Methoden zur Addition, Subtraktion usw.. In Java würde dies zu Methodenaufrufen wie im Listing 3 führen.

```
x.add(y)
```

Listing 2: Infix-Operatornotation 1

Diese Notation entspricht jedoch nicht der natürlichen Schreibweise binärer Operationen. Sie entsteht lediglich, um die syntaktischen Forderungen der Programmiersprache zu erfüllen. Um eine natürliche Notation algebraischer Terme zu ermöglichen, definiert Scala neue Syntaxregeln. Im Gegensatz zu Java erlaubt Scala, einzelne Symbole als Methodennamen zu verwenden. Somit existieren in der Klasse *Int* Methoden mit Namen wie $+, -, *$ oder /. Dadurch lässt sich eine Addition jedoch immer noch nicht wie gewohnt durchführen, wie Listing 3 verdeutlicht.

```
x.+(y)
```

Listing 3: Infix-Operatornotation 2

Zur weiteren Verbesserung der Syntax erlaubt Scala, den Punkt als Trennzeichen zwischen Objekt- und Methodennamen wegfallen zu lassen. Ebenfalls sind die Klammern um den Para-

meter fakultativ, sofern eine Methode nur ein Argument akzeptiert. Somit ist der Methodenaufruf in Listing 4 vollkommen äquivalent zu den beiden zuvor gezeigten Methodenaufrufen.

```
x + y
```

Listing 4: Infix-Operatornotation 3

Hierdurch wird die Verwendung der gewohnten Notation bei binären Operatoren ermöglicht, obwohl es sich bei x um eine Instanz einer Klasse und nicht um einen in die Programmiersprache eingebauten Datentyp handelt. Die Semantik der drei alternativen Methodenaufrufe ist dabei vollkommen identisch. Auf dem *Int*-Objekt x wird die Methode + aufgerufen, der als Argument das *Int*-Objekt y übergeben wird. Als Rückgabewert liefert die Methode ein neues *Int*-Objekt, welches die Summe von x und y repräsentiert.

Bill Venners demonstriert in seinem Vortrag „The Feel of Scala“ [Ven09], dass der Nutzen dieser Notation nicht auf einen mathematischen Kontext begrenzt ist. Die beiden folgenden Anweisungen sind semantisch vollkommen identisch, wobei Letztere schon fast wie ein Ausdruck in natürlicher englischer Sprache anmutet.

```
map.containsKey(123);
map containsKey 123
```

Listing 5: Infix-Operatornotation 2

Es fällt auf, dass der Quellcode durch die sogenannte Infix-Operatornotation an Lesbarkeit gewinnt. Für die Entwickler einer Programmiersprache ist es kaum möglich, sämtliche Bedürfnisse der Anwender an Datentypen zu befriedigen. Die Programmiersprache Scala setzt diese Konzepte der Objektorientierung konsequent um, sodass sich selbsterstellte Klassen nahtlos in die bestehende Sprache integrieren lassen. Auf diese Weise erstellte Datentypen unterscheiden sich nicht von denen, die bereits im Sprachumfang von Scala enthalten sind. Sollte also der Bedarf an einem Datentyp bestehen, den Scala nicht erfüllt, so kann der Anwender selbst eine entsprechende Klasse erstellen.

2.3.4. Eigene Kontrollstrukturen

Im Sprachumfang von Scala sind nur sehr wenige Kontrollstrukturen, wie Schleifen, bedingte Anweisungen und der *try-catch*-Block enthalten. Deshalb lassen sich in Scala eigene Kontrollstrukturen implementieren, wie Martin Odersky in [OSV11] demonstriert. Hierzu nutzt Scala Konzepte der funktionalen Programmierung. Wie in [Pie10] dargestellt, wird die funktionale Programmierung durch folgende Attribute charakterisiert:

- Jedes Programm ist eine Funktion
- Jede Funktion kann weitere Funktionen aufrufen
- Funktionen werden wie Daten behandelt.

Eine Funktion ist dabei frei von Seiteneffekten. Demnach liefert sie immer dasselbe Ergebnis, sofern ihr dieselben Argumente übergeben werden. Das Ergebnis einer Funktion hängt ausschließlich von seinen Argumenten ab. Hierin unterscheiden sich Funktionen von den Methoden eines Objektes, deren Rückgabewert oft vom internen Zustand des Objektes abhängt. Des Weiteren hat der Aufruf einer Funktion neben dem Rückgabewert keine weiteren Auswirkungen. Der Aufruf einer Methode verändert oft den internen Zustand des Objektes. Das Listing 6 zeigt die Verwendung einer Funktion in Scala.

```
val inc = (x: Int) => x + 1
inc(1) // = 2
inc(9) // = 10
```

Listing 6: Funktionsobjekte in Scala

Die Funktion `(x: Int) => x + 1` kann wie ein ganz normales Objekt behandelt werden. In der Tat nehmen Funktionen in Scala keine Sonderstellung ein, sondern sind Objekte vom Typ Funktion$N$, wobei das N für die Anzahl der Parameter steht, die der Funktion übergeben werden können. Im Beispiel wird die Funktion der Variablen *inc* zugewiesen. Das Funktionsliteral in Zeile 1 ist dabei wie folgt zu interpretieren: Die Funktion *inc* akzeptiert einen Parameter x vom Typ *Int* und bildet diesen auf $x + 1$ ab. Das Funktionsobjekt kann mithilfe seines Namens ausgeführt werden, ohne dass eine spezielle Methode verwendet werden muss. Durch Funktionen höherer Ordnung kann Scala ein sehr hohes Abstraktionsniveau erreichen. Laut [See11] handelt es sich dabei um Funktionen, die eine weitere Funktion als Argument erwarten. Als Beispiel soll die Collection-Methode *map* dienen. Die Methode bildet eine Collection auf eine neue Collection ab. Listing 7 zeigt deren Anwendung.

```
newNumbers = Numbers.map(x => x+1)
```

Listing 7: Funktionen höherer Ordnung in Scala

Der Methode wird als Argument ein Funktionsobjekt übergeben, welches die Abbildungsregel darstellt. Der konkrete Nutzen der Methode *map* offenbart sich erst durch die als Argument übergebene Funktion.

Ein weiteres wichtiges Konzept für die Erstellung eigener Kontrollstrukturen ist das nach dem Mathematiker Haskell Curry benannte Currying. In Scala können Funktionen und Me-

thoden mehrere Parameterlisten aufweisen. Dadurch kann einer Funktion nur ein Teil der benötigten Parameter übergeben werden. Ein solcher Methodenaufruf liefert kein Ergebnis, sondern eine neue Funktion zurück, welche die bereits übergebenen Argumente konserviert und nur noch die nicht belegten Parameter als Eingabe erwartet. Durch die partielle Übergabe von Argumenten lässt sich eine Funktion mit n Parametern in eine neue Funktion überführen, die nur noch $n-1$ Argumente verlangt. Das folgende Listing 8 soll das Konzept verdeutlichen.

```
def add(x: Int)(y: Int) = x + y
add(1)(2) // = 3

val addOne = add(1) _
addOne(2) // = 3
addOne(5) // = 6
```

Listing 8: Currying in Scala

Die Methode *add* besitzt zwei Parameterlisten, die jeweils ein Argument fassen. Sie gibt als Ergebnis die Summe der beiden übergebenen Ganzzahlen zurück. Der Variablen *addOne* wird mithilfe des Curryings eine Funktion zugewiesen, welche nur noch einen Parameter erwartet. Dieser wird bei jedem Aufruf mit dem bereits übergebenen Wert Eins addiert.

Auf dieser Grundlage lassen sich eigene Kontrollstrukturen entwickeln. Scala erlaubt die Verwendung von geschweiften anstelle runder Klammern, sofern die Parameterliste nur ein Argument besitzt. In dem die erste Parameterliste runde Klammern und die zweite Parameterliste geschweifte Klammern verwendet, muten Methodenaufrufe wie Kontrollstrukturen an. Martin Odersky demonstriert dies in [OSV11] anhand einer Kontrollstruktur, die eine Datei automatisch nach ihrer Verarbeitung schließt. Somit ist sichergestellt, dass das Programm die verwendeten Ressourcen auch wieder frei gibt. Der Algorithmus zur Verarbeitung der Datei wird der Funktion als Argument übergeben. Das Listing 9 zeigt die Implementierung der Kontrollstruktur und deren Verwendung.

```
// Erstellung von Kontrollstrukturen
def using(file: File)(op: PrinterWriter => Unit) {
        val writer = new PrinterWriter(file)
        try {
                op(writer)
        } finally {
                writer.close()
        }
}
```

```
// Verwendung der Kontrollstruktur
using(file) {
        writer => writer.println(new java.util.Date)
}
```

Listing 9: Eigene Kontrollstrukturen in Scala

Der Quellcode zeigt, wie die Kontrollstruktur verwendet wird, um das aktuelle Datum in eine Datei zu schreiben.

2.3.5. Zwischenfazit

In Scala ist es möglich, den Sprachumfang um eigene Operatoren oder zusätzliche Kontrollstrukturen zu erweitern. Daher ist diese Sprache besonders gut geeignet, um eigene domänenspezifische Sprachen zu entwickeln. Auf diese Weise wächst die Sprache durch das Engagement ihrer Nutzer, wie von Guy L. Steele Jr. vorhergesagt.

3. Vorgehensmodell

Ein Vorgehensmodell bildet die Grundlage eines jeden professionell durchgeführten Softwareprojektes. Dieses definiert die Aktivitäten und Abläufe innerhalb des Projektes und bringt sie in eine logische Reihenfolge. In diesem Kapitel soll zunächst das klassische Wasserfallmodell vorgestellt und mit den nichtlinearen Vorgehensmodellen verglichen werden. Anschließend soll erläutert werden, warum für dieses Projekt ein inkrementelles Vorgehen gewählt wurde.

3.1. Klassische Vorgehensmodelle

Jedes Softwareprojekt lässt sich in mehrere Phasen unterteilen, die jeweils durch eine konkrete Aktivität im Softwareentwicklungsprozess geprägt werden. Die einzelnen Phasen werden beim Wasserfallmodell in der festgelegten Reihenfolge linear durchlaufen. Sofern dies notwendig erscheint, besteht jedoch die Möglichkeit, in frühere Phasen zurückzukehren, um Fehler zu beheben oder Änderungen vorzunehmen. Dieses Modell wurde erstmalig im Artikel „Managing the Development of Large Software Systems“ [Roy70] von Winston Royce vorgestellt. Die Abbildung 7 stellt das Wasserfallmodell und dessen Phasen dar.

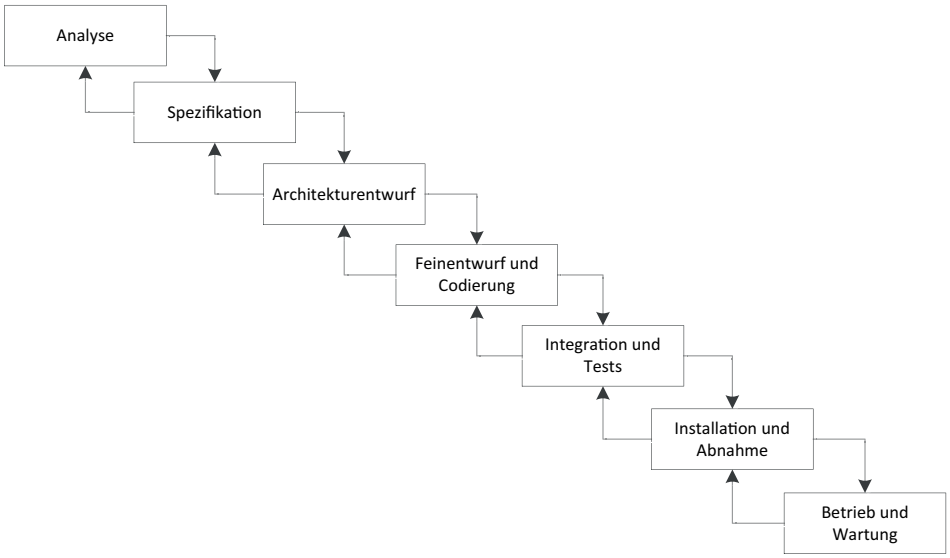

Abbildung 7: Wasserfallmodell (Eigene Darstellung, in Anlehnung an [LL10])

Das Wasserfallmodell gilt als dokumentengetriebenes Vorgehensmodell. In jeder Phase wird ein Artefakt erzeugt, welches die Arbeitsgrundlage der nachfolgenden Phase darstellt. So entsteht als Resultat der Analyse-Phase ein Lastenheft, welches die Anforderungen an das zu erstellende Produkt definiert. Während der Softwarespezifikation werden aufgrund dieser Anforderungen konkrete Realisierungsvorgaben erarbeitet und im Pflichtenheft festgehalten.

Die Vorgaben aus dem Pflichtenhelft münden in einen Softwarearchitekturentwurf. Dieser legt eine sinnvolle Struktur für die zu entwickelnde Software fest und regelt den Aufbau und das Zusammenspiel der einzelnen Komponenten. Der Entwurf wird im Verlauf der nächsten Phase immer weiter verfeinert und anschließend realisiert. Um die Funktionstüchtigkeit des Systems sicherzustellen, muss der entstandene Quellcode getestet werden. Wird kein Fehler festgestellt und das System vom Kunden abgenommen, kann mit der Integration des Systems in die bestehende Umgebung und dessen Inbetriebnahme begonnen werden.

Ludewig und Lichter erläutern in [LL10], dass der Entwicklungsprozess beim Wasserfallmodell bis zur Codierung top-down abläuft, also von der abstrakten Spezifikation zur spezifischen Umsetzung. Danach verläuft er genau umgekehrt, also buttom-up. Die Badewannenkurve in Abbildung 8 soll darstellen, dass Fehler meist nur auf gleicher Abstraktionsebene gefunden werden.

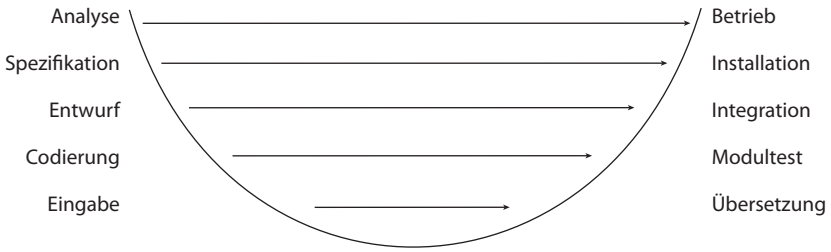

Abbildung 8: Badewannenkurve (Eigene Darstellung, in Anlehnung an [LL10])

Ein Fehler in der Analyse ist somit besonders teuer, da er wahrscheinlich erst auffällt, wenn die Software bereits eingesetzt wird. Da die Resultate der einzelnen Phasen aufeinander aufbauen, müssen, nachdem der eigentliche Fehler beseitigt wurde, alle Phasen des Wasserfallmodells erneut durchlaufen und deren Artefakte überarbeitet werden. Aufgrund dieses Umstandes wird das Wasserfallmodell in der aktuellen Fachliteratur heftig kritisiert, so auch in [LL10]. Die Schwierigkeit besteht darin, bereits zu Beginn eines Projektes alle Anforderungen an das zu erstellende System vollständig zu erkennen. Einige Aspekte oder Probleme ergeben sich erst während der Entwicklung oder verändern sich im Laufe der Zeit. Wie bereits oben angedeutet, sind die dann erforderlichen Rücksprünge in frühere Phasen mit einem hohen Aufwand an Arbeit und Zeit verbunden. Da die theoretisch vorgesehenen Rücksprünge in der Praxis nur schwer durchführbar sind, vergleichen Ludewig und Lichter in [LL10] das Wasserfallmodell mit einer Einbahnstraße. Somit können zu spät erkannte Anforderungen oder geänderte Rahmenbedingungen oft nicht berücksichtigt werden. Dies führt dazu, dass Programme erstellt werden, welche die an sie gestellten Erwartungen nicht vollständig erfüllen können.

3.2. Nichtlineare Vorgehensmodelle

Die Schwächen des Wasserfallmodells führten zum Aufkommen nichtlinearer Vorgehensmodelle. In diesem Zusammenhang wird unter anderem zwischen der iterativen und der inkrementellen Softwareentwicklung unterschieden. Bei der iterativen Softwareentwicklung werden die oben genannten Phasen des Entwicklungsprozesses mehrfach durchlaufen. Die Abbildung 9 stellt dar, wie sich das Programm mit jedem Iterationsschritt der gewünschten Lösung annähert.

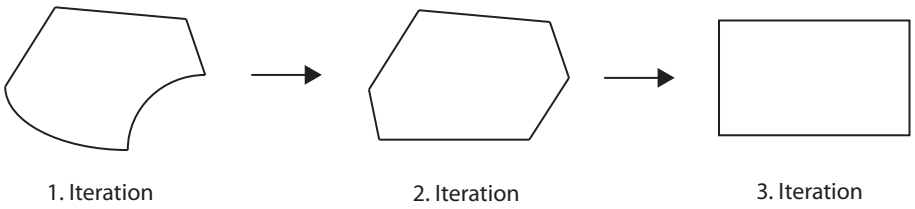

Abbildung 9: Iteratives Vorgehensmodell (Eigene Darstellung, in Anlehnung an [LL10])

Die Erkenntnisse aus den vorangegangenen Iterationsschritten bilden die Grundlage für die Verbesserungen und Anpassungen, die in der aktuellen Iteration am System vorgenommen werden. Neue oder geänderte Anforderungen werden gesammelt und können dann in der nächsten Iteration berücksichtigt werden. Jeder Schritt setzt somit die gesamte, zu diesem Zeitpunkt gültige Spezifikation um. Im Gegensatz dazu wird bei der inkrementellen Softwareentwicklung ein Kernsystem erstellt, das während der nachfolgenden Zyklen immer weiter ausgebaut wird. Abbildung 10 stellt dieses Prinzip grafisch dar.

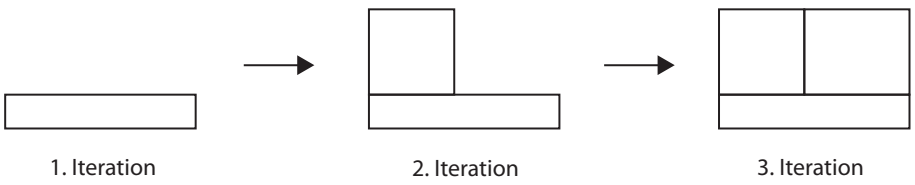

Abbildung 10: Inkrementelles Vorgehensmodell

Als Ergebnis liefert jeder Zyklus ein weiteres Inkrement des zu erstellenden Systems und erweitert damit das Vorhergehende. Die Funktionalität wird ausgehend von einem Kernsystem so lange erweitert, bis es alle Anforderungen vollständig erfüllt. Da das Kernsystem sehr früh eingesetzt werden kann, können die Erfahrungen der Anwender in die weitere Entwicklung einfließen. Dies verhindert, dass an den Anforderungen der Anwender vorbei entwickelt wird.

Für die Umsetzung der Prozessalgebra scheint ein inkrementelles Vorgehensmodel als vorteilhaft. Zum einen umgeht dieses die Schwächen des Wasserfallmodels, zum anderen ent-

spricht es der Natur einer Algebra. Auch die Prozessalgebra wird ausgehend von einer minimalen Algebra durch das Hinzufügen neuer Operatoren und Regeln um neue Möglichkeiten erweitert. Dabei wird zunächst die in [BBR09] beschriebene MPT-Algebra umgesetzt. Dieses Teilprojekt ist abgeschlossen, wenn das Programm Terme dieser Algebra parsen, die nächsten möglichen Aktionen bestimmen und Aktionen ausführen könnte.

Danach wird das Programm analog zum Aufbau des Buches [BBR09] um weitere Teilaspekte, wie Rekursion, sequenzielle Komposition, parallele und kommunizierende Prozesse sowie Abstraktion ergänzt. Jede Ausbaustufe erweitert das vorhandene System, wobei jede Erweiterung in der Regel auch eine Verbesserung der bestehenden Komponenten nach sich zieht. Die Abbildung 11 stellt das Vorgehen beim vorliegenden Projekt dar.

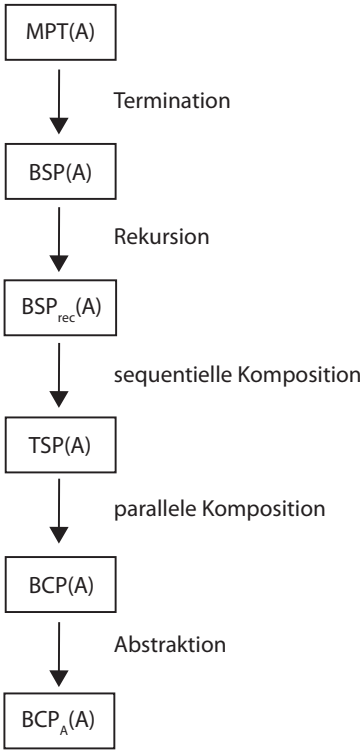

Abbildung 11: Vorgehen

4. Analyse und Entwurf

In diesem Kapitel soll die Analyse und Entwurfsphase des Projektes dargestellt und deren Ergebnisse präsentiert werden. Zur Visualisierung, Dokumentation und Kommunikation des Systementwurfs wird häufig die Unified Modelling Language (UML) verwendet. Sie enthält laut [Kec05] 13 Diagrammtypen, welche jeweils unterschiedliche Teilaspekte des Systems adressieren. Es können zwei Gruppen von Diagrammen identifiziert werden. Die dynamischen Diagramme, auch Verhaltensdiagramme genannt, dienen zur Beschreibung der Verhaltensaspekte eines Systems, einer Funktion oder eines Geschäftsvorgangs. Im Wesentlichen werden die einzelnen Elemente zur Laufzeit und deren Wechselwirkung untereinander beschrieben. Beispiele für derartige Diagramme sind unter anderem Aktivitäts- oder Sequenzdiagramme. Die zweite Gruppe umfasst die statischen Strukturdiagramme. Sie definieren die Systemstrukturen mit allen darin vorhandenen Elementen, sowie deren Beziehungen und Abhängigkeiten zueinander. Zu ihnen zählen zum Beispiel die Paket- und Klassendiagramme.

4.1. Analyse

Der Zweck der Analysephase ist laut [LL10] die Beschreibung der Ausgangssituation, die Definition der Ziele für das zu erstellende System sowie eine personelle, finanzielle, zeitliche und technische Ressourcenplanung. Aber auch das Festlegen von Projektstandards und eine Machbarkeitsanalyse und Risikoanalyse gehören in diese Phase. Um diese Ziele zu erreichen, werden verschiedene Aktivitäten durchgeführt, die je nach Größe und Komplexität des Projektes variieren.

Das Anwendungsfalldiagramm wird häufig bereits während der Analysephase eines Softwareprojektes erstellt. Es besitzt ein sehr hohes Abstraktionsniveau und stellt die Black-Box-Sicht eines Anwenders auf das zu erstellende System dar. Dazu werden alle ausführbaren Anwendungsfälle, sowie deren Akteure geschildert. Auch die Systemgrenzen werden in diesem Diagrammtyp berücksichtigt. Das Ziel ist es, auf verständliche Weise zu zeigen, was das entstehende System leisten kann bzw. muss. Dieses Diagramm dient demnach der Beschreibung der funktionalen Anforderungen und eignet sich, auch aufgrund seines einfach zu verstehenden Aufbaus, sehr gut zur Kommunikation mit Fachvertretern und Projektauftraggebern. Die folgende Abbildung 12 stellt das Anwendungsfalldiagramm für die im Rahmen dieser Arbeit entstandene Software dar.

Die Funktionalität des Systems lässt sich durch vier einfache Anwendungsfälle beschreiben. Der Benutzer hat die Möglichkeit, Prozesse in Form von Termen der Prozessalgebra zu definieren. Diese Terme werden vom System geparst und somit in eine ausführbare Form überführt.

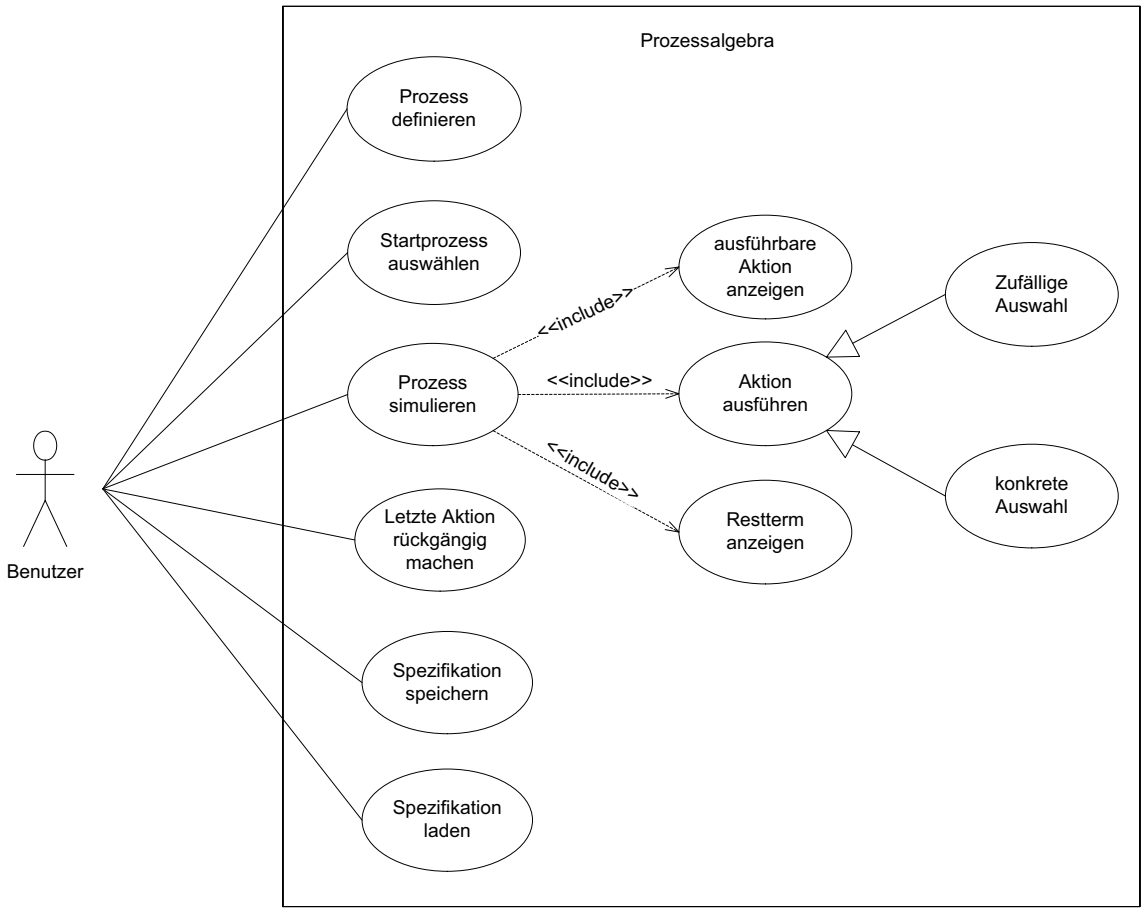

Abbildung 12: Anwendungsfalldiagramm

Dadurch lassen sich die eingegebenen Prozesse analysieren, indem der Prozessablauf simuliert wird. Der zweite Anwendungsfall lässt sich in drei Unteraufgaben unterteilen. Zunächst werden dem Benutzer die im aktuellen Zustand des Prozesses ausführbaren Aktionen angeboten. Entweder wählt er bewusst eine dieser Aktionen aus oder lässt den Zufall entscheiden. Durch die Ausführung einer neuen Aktion ändert sich der Zustand des Prozesses, der durch den noch auszuführenden Restterm repräsentiert wird. Diese Schritte wiederholen sich solange, bis ein möglicher Prozessablauf vollständig simuliert wurde. Dies ist der Fall, wenn der Prozess entweder terminiert oder in einen Zustand gerät, in dem keine weitere Aktion möglich ist.

Dem Benutzer wird außerdem die Möglichkeit eingeräumt, die ausgewählte Aktion rückgängig zu machen und somit in den vorherigen Zustand zurückzukehren. Diese auch oft als *Undo* bezeichnete Funktion fördert die Gebrauchstauglichkeit (Usability) der Anwendung. Ben Shneiderman erkannte bereits in den 1980er Jahren die Wichtigkeit einer solchen Funktionalität. Der Vorreiter der Mensch-Computer-Kommunikation hat acht Regeln zur Gestal-

tung interaktiver Benutzeroberflächen aufgestellt und diese in seinem Werk „Designing the User Interface“ [SP09] festgehalten. Eine Rücksetzungsmöglichkeit fördert das Erlernen eines Systems, da der Anwender die unangenehmen Konsequenzen einer unbedacht ausgeführten Aktion nicht fürchten muss. Der Anwender kann einfach eine Aktion ausprobieren und im Bedarfsfall einzelne Aktionen wieder rückgängig machen.

Des Weiteren besteht die Möglichkeit, Spezifikationen auf der Festplatte abzuspeichern. Die auf diese Weise persistierten Spezifikationen lassen sich zu einem späteren Zeitpunkt wieder in das Programm laden. Dem Benutzer wird so ermöglicht, auch umfangreichere Systeme zu spezifizieren, ohne die Terme nach jedem Programmstart neu eingeben zu müssen. Ebenfalls wird so der Austausch von Spezifikationen unter den Anwendern ermöglicht.

4.2. Grobentwurf

Ein Paketdiagramm ist eine erste, recht grobe Darstellung der technischen Struktur der Anwendung. Pakete oder Namensräume werden in vielen gängigen Programmiersprachen als strukturgebendes Element eingesetzt. So auch in der in diesem Projekt verwendeten Programmiersprache Scala. Sie können Klassen oder weitere Pakete enthalten. Die Abbildung 13 zeigt das Paketdiagramm für die hier entwickelte Anwendung.

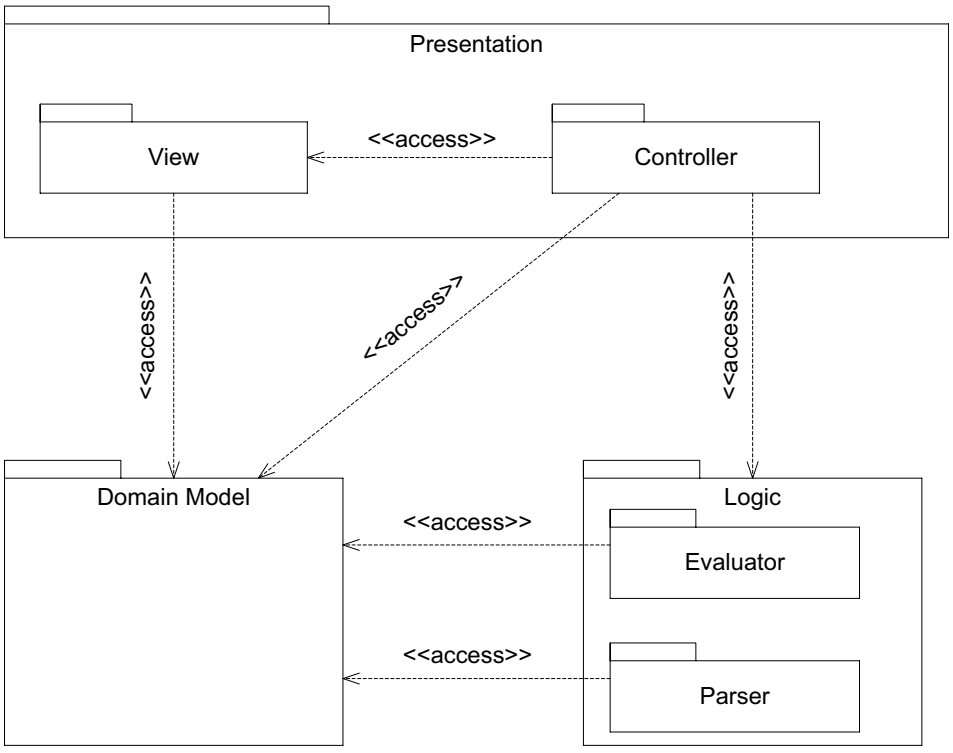

Abbildung 13: Paketdiagramm

Es stellt die Struktur der Anwendung aus der Vogelperspektive dar. Aus dieser Höhe lassen sich noch nicht alle Details erkennen, jedoch können bereits einige Muster wahrgenommen werden. So kann man bereits erkennen, dass das Model-View-Controller-Konzept verwendet wird. Durch das in [GHJV94] beschriebene Entwurfsmuster wird die Interaktion mit dem Anwender in drei Teilaufgaben unterteilt.

Die Funktion der passiven Datenrepräsentation übernimmt das Model. Die Verarbeitung von Eingaben des Benutzers und die damit verbundene Abfrage oder Änderung des Models werden im Controller implementiert. Als darstellende Komponente kommt dann der View zum Einsatz. Dabei reagiert der View aktiv auf Änderung des Models. Hierzu wird oft auf das Observer-Entwurfsmuster aus [GHJV94] zurückgegriffen. Wie in [LR09] hervorgehoben wird, handelt es sich beim Model-View-Controller nicht um ein komplettes Architekturmodell, sondern um ein Interaktionsmuster in der Präsentationsschicht eines Softwareprogramms.

Es wird also eine Trennung der Zuständigkeiten (Separation of Concerns) angestrebt. Der Ursprung dieses Prinzips liegt im dem bereits 1974 von Edsger W. Dijkstra veröffentlichten Artikel „On the role of scientific thought“ [Dij82]. Jede Komponente sollte demnach für eine ganz spezifische Aufgabe zuständig sein. Komponenten, die mehrere Aufgaben gleichzeitig erfüllen, sind oft unnötig komplex und erschweren somit die Wartung und Weiterentwicklung der Software. Die Trennung zwischen Model, View und Controller ermöglicht es zum Beispiel durch Hinzufügen einer neuen View-Komponente, dem Benutzer eine völlig andere Darstellung zu präsentieren, ohne dass die Model- oder Controllerkomponenten angepasst werden müssten.

Des Weiteren lässt sich erkennen, dass der Controller zur Erfüllung seiner Aufgabe auf die Dienste einer Logikschicht zurückgreift. Diese enthält zwei Unterpakete. Im Paket Parser sind Klassen enthalten, die in der Lage sind, aus einer als Text verfassten Spezifikation eine Menge von Domänenobjekten im Speicher zu erzeugen. Auf der Grundlage dieser Objekte kann dann mithilfe der im Paket Evaluator enthaltenen Klassen eine Simulation der Spezifikation durchgeführt werden.

4.3. Feinentwurf

Ein Klassendiagramm dient zur detaillierten Darstellung der in der Anwendung enthaltenen Klassen und Schnittstellen. Das Diagramm zeigt die einzelnen Klassen mit deren Attributen und Methoden. Des Weiteren werden Beziehungen zwischen den Klassen thematisiert, wie Assoziationen, Kompositionen oder Vererbungshierarchien. Sogar die Sichtbarkeit von Operationen und Attributen kann beschrieben werden. Dieser Diagrammtyp orientiert sich dabei so stark an den Paradigmen der Objektorientierung, dass es möglich ist, automatisiert

Code aus ihm erzeugen zu lassen. Die entstehenden Klassenrümpfe werden dann in die entsprechende Programmierumgebung geladen und müssen nur noch mit Leben gefüllt werden. Von dieser Möglichkeit wird im vorliegenden Projekt jedoch abgesehen, da zum einen der Umfang der Programmierarbeiten dies nicht notwendig erscheinen lässt und zum anderen eine derartige Werkzeugunterstützung für die verwendete Programmiersprache Scala noch nicht verfügbar ist. Die Abbildung 14 zeigt das Klassendiagramm des Domänenmodells für die Prozessalgebra.

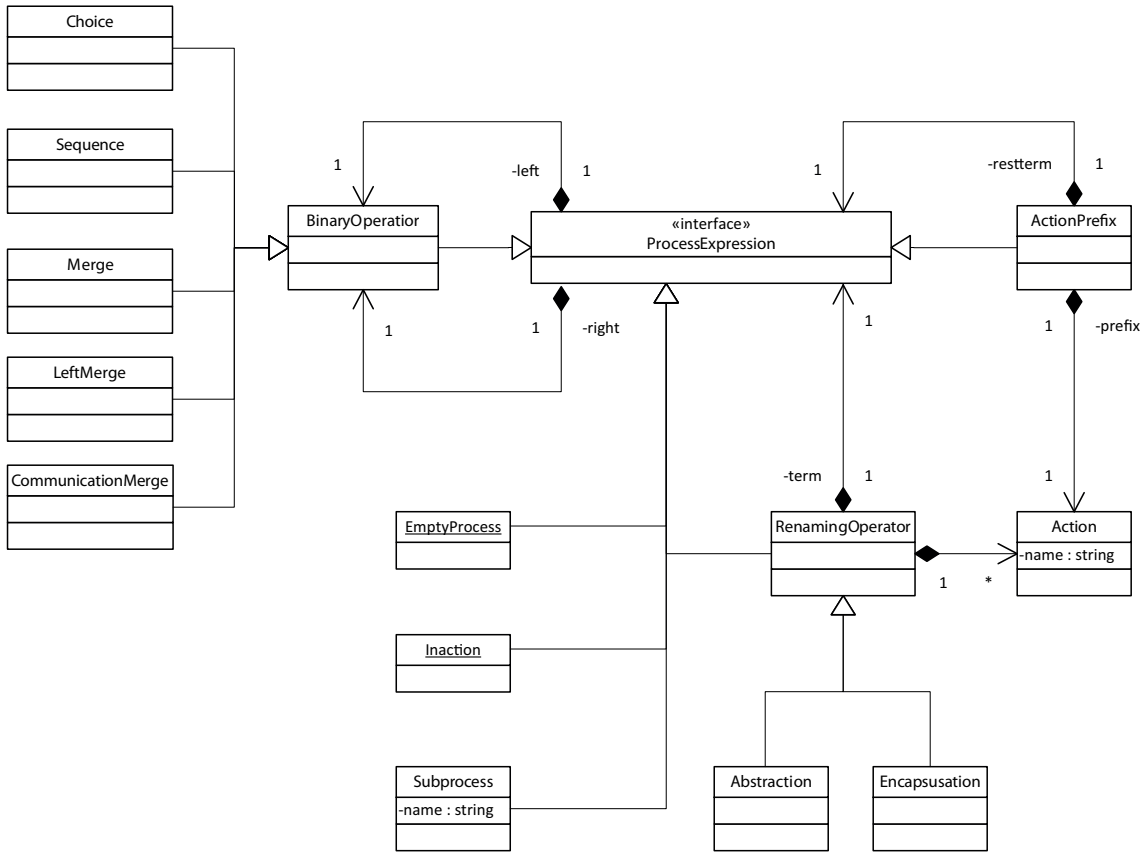

Abbildung 14: Domänenmodell

Die umgesetzte Form der Prozessalgebra besitzt ein Repertoire an zwei Konstanten, sowie fünf binären, einem unären und zwei *Renaming*-Operatoren. Das hier dargestellte Domänenmodell bildet das Herzstück der entwickelten Software. Es wird eine hierarchische Datenstruktur in Anlehnung an das *Composite*-Entwurfsmuster aus [GHJV94] verwendet. Dieses Muster kann zur Abbildung einer Teil-Ganzes-Hierarchie verwendet werden. Es wird eine einheitliche Behandlung von Einzelobjekten und der Komposition aus ihnen angestrebt. Zu diesem Zweck wird mithilfe eines Interfaces oder einer abstrakten Klasse eine Verallgemeinerung geschaffen, von der sowohl das Einzelobjekt als auch die Komposition erben. Im vorliegenden

Fall wird diese Verallgemeinerung durch das Interface *ProcessExpression* umgesetzt. Dieses Interface wird von allen weiteren Klassen implementiert, die einen Prozessalgebra-Ausdruck repräsentieren sollen. Als Einzelobjekte können die Prozessalgebra-Konstanten *Inaction* und *EmptyProcess*[3] angesehen werden. Eine Komposition kann weitere Objekte des abstrakten Typs *ProcessExpression* aufnehmen, dessen konkrete Ausprägungen entweder durch eine weitere Komposition oder ein Einzelobjekt auftreten können. Auf diese Weise entstehen beliebig tief ineinander verschachtelte Strukturen.

Die Klasse *ActionPrefix*[4] kann durch das Attribut *restterm* ein Objekt vom Typ *ProcessExpression* aufnehmen. Dieses könnte eine *Inaction*, ein weiterer *ActionPrefix* oder ein Objekt der Klasse *Choice* sein. Zusätzlich besitzt die Klasse *ActionPrefix* ein Attribut *prefix* vom Typ *Action*. Die Klasse *Action* repräsentiert die atomaren Aktionen eines Prozessterms.

Für die binären Operationen *Choice*, *Sequence*[5], *Merge*[6], *LeftMerge* und *CommunicationMerge* wird eine abstrakte Basisklasse gebildet. In ihr wird die gemeinsame Funktionalität der oben genannten Klassen an zentraler Stelle definiert. Jeder binären Operation ist gemein, dass sie zwei Operanden besitzt, einen linken und einen rechten Term. Da diese Eigenschaft bereits durch die Basisklasse bestimmt wird, muss sie nicht in jeder Subklasse einzeln implementiert werden. Im Rahmen dieser Generalisierung entsteht eine Basisklasse, die ein so allgemeines Konzept realisiert, dass aus ihr keine sinnvollen Instanzen gebildet werden können. In einer derartigen Klasse können zum Beispiel bewusst Details offen gelassen und deren konkrete Implementierung somit in die erbenden Klassen ausgelagert werden. Mit der Kennzeichnung als abstrakte Klasse wird ihr daher die Möglichkeit genommen, ein Objekt zu instanziieren. Die Unterklassen hingegen können instanziiert werden und partizipieren an den bereits in der Oberklasse definierten Methoden und Attributen.

Bei den *Renaming*-Operatoren[7] wird ähnlich vorgegangen. Sie können eine Menge von *Action*-Objekten und ein Objekt vom Typ *ProcessExpression* aufnehmen. Jedes Vorkommen einer *Action* aus dieser Menge wird dann in eine durch den Subtyp bestimmte Entsprechung umbenannt.

Die benannten *ProcessExpression*-Objekte werden an einer zentralen Stelle registriert und können dann mittels *SubProcess*-Objekt referenziert werden. Dieses enthält lediglich den Namen des referenzierten Prozesses. Dadurch wird unter anderem die rekursive Spezifikation von Prozessen ermöglicht.

[3] Der *EmptyProcess* wurde im Kapitel 2.1.2 und die *Inaction* im Kapitel 2.1.1 eingeführt
[4] Der *ActionPrefix* wurde im Kapitel 2.1.1 eingeführt
[5] Die *Sequence*-Operator wurde im Kapitel 2.1.4 eingeführt
[6] Die *Merge*-Operator wurde im Kapitel 2.1.6 eingeführt
[7] Die *Renaming*-Operatoren wurde im Kapitel 2.1.5 eingeführt

Das in Abbildung 15 dargestellte Objektdiagramm zeigt, wie der aus dem Grundlagenkapitel bekannte Prozessalgebra-Term $open.eat.0 + open.marry.1$ über die zuvor besprochene Klassenhierarchie abgebildet werden kann.

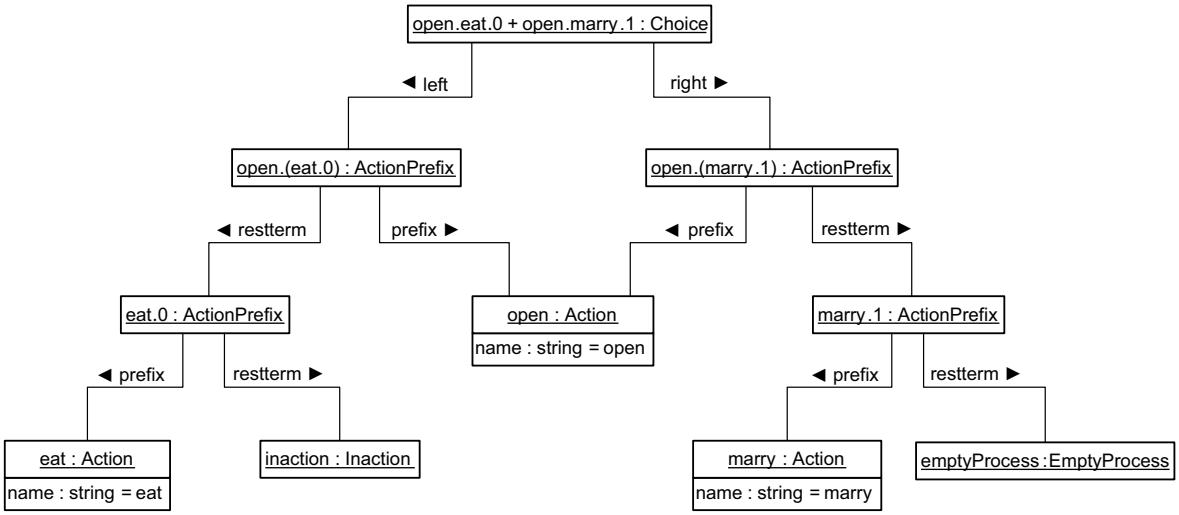

Abbildung 15: Objektdiagramm

Der Term wird durch ein Objekt vom Typ *Choice* repräsentiert. Dieses Objekt verweist mithilfe seiner Attribute *left* und *right* auf Objekte vom Typ *ActionPrefix*. Beide besitzen einen Verweis auf das *Action*-Objekt mit dem Namen *open*, sowie einen Verweis auf einen weiteren *ActionPrefix*.

5. Realisierung

Nachdem im letzten Kapitel näher auf den Entwurf der erstellten Software eingegangen wurde, sollen nun einige ausgewählte Aspekte der Realisierung beleuchtet werden.

5.1. Domänenmodell

Für die Umsetzung des im vorherigen Kapitels vorgestellten Entwurfs des Domänenmodells in Scala werden Traits, Singleton-Objekte und Case-Klassen verwendet. Den Ausgangspunkt stellt das Trait *ProcessExpression* dar. Ein Trait ist laut [See11] vergleichbar mit einem Interface aus Java. Optional können Methoden bereits im Trait implementiert werden. Die Prozessalgebra-Konstanten *Inaction* und *Empty Process*[8] werden als Singleton-Objekt definiert. Diese werden durch das Schlüsselwort *object* deklariert. Es stellt in gewissermaßen eine in die Programmiersprache Scala integrierte Umsetzung des Singleton-Entwurfsmusters aus [GHJV94] dar. Scala setzt das objektorientierte Paradigma wesentlich konsequenter um als Java. Im Zuge dessen verzichtet es komplett auf statische Klassenbestandteile. Diese werden durch die Singleton-Objekte ersetzt. Eine Case-Klasse hat viel Ähnlichkeit mit den *Immutable Objects*, wie sie von Joschua Bloch in seinem Buch „Effective Java“ [Blo08] beschrieben werden. Dieses Entwurfsmuster eignet sich hervorragend um Klassen zu erstellen, welche vorrangig als Datencontainer fungieren sollen. Während jedoch der Java-Code zur Umsetzung eines *Immutable Objects* etwa eine Din-A4 Seite umfasst, genügt in Scala der Modifier *case*, wie in [See11, S.13-14] demonstriert wird. Hinter dem Namen der Case-Klasse können in runden Klammern deren Attribute als kommagetrennte Liste angegeben werden. Die auf diese Weise definierten Attribute sind unveränderlich, öffentlich sichtbar und müssen bei der Konstruktion eines Objektes angegeben werden. Die sogenannten Klassenparameter definieren demnach gleichzeitig Konstruktorargumente, mit deren Hilfe die Attribute initialisiert werden.

Das Überschreiben der Methoden *toString*, *equals* und *hashCode* ist bei Case-Klassen nicht notwendig. Der Compiler ergänzt selbstständig eine für diese Klasse sinnvolle Implementierung. Die Identität eines Objektes wird somit standartmäßig durch seine Attribute und nicht durch seine Position im Speicher bestimmt. Da in Scala der Aufruf des Operators == an die Methode *equals* delegiert wird, kann auch mithilfe der Operatornotation die Gleichheit zweier Objekte geprüft werden. Wie in [OSV11] dargestellt, wird so ein häufiger Anfängerfehler in Java-Quellcode vermieden. Das Listing 10 zeigt einen Teil der Realisierung des Domänenmodells.

[8] Der *EmptyProcess* wurde im Kapitel 2.1.2 und die *Inaction* im Kapitel 2.1.1 eingeführt

```
sealed trait ProcessExpression
object Inaction extends ProcessExpression
object EmptyProcess extends ProcessExpression

case class ActionPrefix(prefix: Action, term: ProcessExpression)
  extends ProcessExpression

abstract class BinaryOperation(left: ProcessExpression,
  right: ProcessExpression) extends ProcessExpression

case class Alternative(left: ProcessExpression,
  right: ProcessExpression) extends BinaryOperation(left, right)
case class Sequence(left: ProcessExpression,
  right: ProcessExpression) extends BinaryOperation(left, right)
...
```

Listing 10: Domänenmodell in Scala

Des Weiteren wird automatisch eine Factory-Methode bereitgestellt, aufgrund welcher bei der Konstruktion neuer Objekte auf das Schlüsselwort *new* verzichtet werden kann. Dies birgt gerade für die verschachtelten Strukturen, des in diesem Abschnitt vorgestellten Domänenmodells Vorteile. Als Beispiel soll der folgende Prozess-Term dienen:

```
new Choice(new ActionPrefix(new Action("a"), new Inaction()),
           new ActionPrefix(new Action("b"), new EmptyProcess())
```

Listing 11: Prozess-Term mit new-Operator

Durch Verzicht auf den *new*-Operator kann der Term in verkürzter Form notiert werden:

```
Choice(ActionPrefix(Action("a"), Inaction),
       ActionPrefix(Action("b"), EmptyProcess)
```

Listing 12: Prozess-Term ohne new-Operator

Mithilfe des Schlüsselworts *sealed* wird ausgeschlossen, dass Subklassen des Traits *Process-Expression* außerhalb der eigenen Quellcodedatei gebildet werden. Somit entsteht ein algebraischer Datentyp. Dieser wird in [OSV11] als abgeschlossene Menge von alternativen Typen beschrieben. Durch die Verwendung eines algebraischen Datentyps wird eine bessere Compiler-Unterstützung erreicht, welche vor allem im Zusammenhang mit dem für die Verarbeitung eingesetzten Pattern-Matching zum Tragen kommt.

5.2. Interne DSL

Mithilfe des im letzten Kapitel vorgestellten Domänenmodells lassen sich Prozessalgebra-Terme repräsentieren. Ein Term wird mithilfe einer Menge ineinander verschachtelter Objekte dargestellt. So kann eine Repräsentation des Prozess-Terms $a.1 + b.0$ wie im Listing 12 aus den vorherigen Kapitel erzeugt werden. Dies entspricht jedoch nicht der üblichen Notation für Prozessalgebra-Terme, wie sie im Grundlagenkapitel beschrieben wurde. In diesem Kapitel soll die Erstellung einer internen domänenspezifischen Sprache thematisiert werden, die es erlaubt, Prozessalgebra-Terme in einer natürlicheren Schreibweise zu deklarieren.

Die Infix-Operatornotation stellt hierfür ein probates Mittel dar. Weitere Spracheigenschaften, wie Semicolon und Type Inference, tragen dazu bei, eine leichtgewichtige Syntax zu erzeugen. Hierdurch kann auf das Semikolon am Zeilenende und auf Typangaben verzichtet werden, sofern der Compiler diese erraten kann. Somit treten die syntaktischen Eigenarten der Programmiersprache in den Hintergrund und die Prozessalgebra-Terme können in gewohnter Notation formuliert werden.

Die Klassen des Domänenmodells könnten demnach einfach mit Operatoren, wie +, ausgestattet werden. Um das Domänenmodell jedoch nicht mit Methoden zu verschmutzen, die nur in einem bestimmten Kontext genutzt werden, wird ein anderer Ansatz gewählt. Mithilfe von Implicit Conversion ist laut [See11] eine non-invasive Erweiterung bestehender Klassen erreichbar. Somit ist es möglich, auf Objekten Methoden aufzurufen, die sie eigentlich gar nicht besitzen. Da es sich bei Scala um eine statisch typisierte Sprache handelt, kann die Struktur einer Klasse zur Laufzeit jedoch nicht verändert werden. Bei einer Implicit Conversion wird das ursprüngliche Objekt von einer Wrapper-Klasse umschlossen, welche dieses Objekt kapselt und die fehlende Methode anbietet. Zu diesem Zweck wird eine Konvertierungsmethode definiert, welche den ursprünglichen Ausgangstyp entgegennimmt und die benötigte Wrapper-Klasse zurückgibt. Wie das Listing 13 zeigt, wird eine derartige Methode mit dem Schlüsselwort *implicit* eingeleitet.

```
implicit def expressionToHelper(expression: ProcessExpression): ExpressionHelper = {
        ExpressionHelper(expression)
}
```

Listing 13: Implizite Methode

Diese Methode konvertiert ein Objekt der Klasse *ProcessExpression* in ein Objekt vom Typ *ExpressionHelper*. Das Schlüsselwort *return* ist in Scala fakultativ. Der Rückgabewert einer Methode wird immer durch das Ergebnis der zuletzt durchgeführten Anweisung bestimmt. Die

Wrapper-Klasse nimmt als Konstruktorargument die zu kapselnde ProcessExpression entgegen und stellt die fehlenden Methoden und Operatoren zu Verfügung, wie Listing 14 darstellt.

```
case class ExpressionHelper(expression: ProcessExpression) {
       def +(right: ProcessExpression): ProcessExpression = {
         Alternative(expression, right)
       }

       def ||(right: ProcessExpression): ProcessExpression = {
         Merge(expression, right)
       }
       ...
}
```

Listing 14: Wrapper-Klasse

Wird nun auf einem *ProcessExpression*-Objekt die Methode + aufgerufen, sorgt der Scala-Compiler selbstständig dafür, dass dieses zuvor mithilfe der Methode *expressionToHelper* in ein *ExpressionHelper*-Objekt konvertiert wird. Der Operator + kann somit auf jedem Objekt des Typs *ProcessExpression* aufgerufen werden und nimmt eine weitere *ProcessExpression* als Argument entgegen. Die Methode gibt ein neues *Choice*-Objekt zurück, welches eine Auswahl zwischen dem ausführenden und dem als Argument übergebenen Objekt darstellt. Da Scala den Punkt zur Trennung von Objekten und Methodenaufrufen nutzt, kann dieser nicht als Methodenname verwendet werden. Alternativ wurde im vorliegenden Projekt das Symbol $*$ für den *ActionPrefix*[9] genutzt.

Damit der Compiler bei Bedarf eine Konvertierung durchführt, muss die implizite Methode in den lexikalischen Scope geladen werden. Laut [See11] gibt es hierfür drei Möglichkeiten:

- Single Identifier
- Companion Objekt des Ausgangtyps
- Companion Objekt des Zieltyps

Im vorliegenden Projekt wird die erste Variante gewählt, da hier eine Konvertierung nur durchgeführt wird, sofern ein bestimmtes Objekt importiert wurde, welches die impliziten Konvertierungsmethoden enthält. Dies geschieht über die Import-Anweisung

```
import de.andreasbreer.processalgebra.dsl.ProcessExpressionImplicits._.
```

[9]Der *ActionPrefix* wurde im Kapitel 2.1.1 eingeführt

Nun lässt sich der Prozessalgebra-Term $a.1+b.0$ in wesentlich natürlicherer Weise deklarieren, wie die Abbildung 16 darstellt.

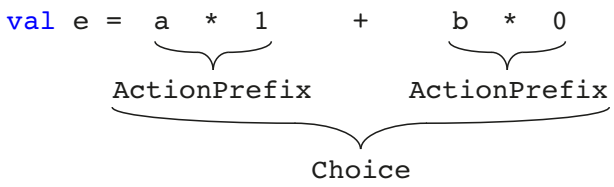

Abbildung 16: Interne DSL

Zunächst wird mithilfe des Operators $*$ ein *ActionPrefix* erzeugt, welcher über den Operator $+$ mit einem zweiten *ActionPrefix* verknüpft wird. Das Resultat ist ein Objekt vom Typ *ProcessExpression*, welches einer Variablen zugewiesen wird. Es fällt auf, dass die Operationen nicht in angegebener Reihenfolge, sondern entsprechend ihrer Bindungsstärke ausgeführt werden. Der Grund hierfür ist, dass Scala für einstellige Operatoren wie $+$ oder $*$ eine Ausführungsreihenfolge definiert. Die Tabelle 11 stellt diese dar.

(alle Buchstaben)
|
^
&
< >
= !
:
+ −
* / %
(alle anderen Sonderzeichen)

Tabelle 11: Bindungsstärke von Operatoren in Scala, entnommen aus [EPF11]

Die Bindungsstärke nimmt von oben nach unten zu. Somit bindet der Operator $*$ stärker als der Operator $+$. Dieser Umstand führt für die Umsetzung der Prozessalgebra zu Problemen, da der *Merge*-Operator schwächer bindet als der *ActionPrefix*, aber stärker als die Auswahl. In Scala bindet jedoch der Operator || schwächer als der Operator $+$. Um diese Probleme zu umgehen, müssen alternative Operator-Symbole gewählt werden, was jedoch Domänenexperten den Zugang zur DSL erschweren wird.

5.3. Externe DSL

Um die im vorherigen Kapitel dargestellten Probleme zu umgehen, soll mit den Parser Combinators ein alternatives Vorgehen erprobt werden. Die Grundlage für die Erstellung eines Parsers ist eine kontextfreie Grammatik, welche alle gültigen Prozessalgebra-Terme beschreibt. Die Regeln einer solchen Grammatik enthalten laut [Sch08] Variablen, die als Platzhalter für syntaktische Einheiten fungieren und Symbole, aus denen die zulässigen Wörter der Sprache letztlich gebildet werden. Diese Symbole werden auch als Terminalsymbole bezeichnet. Die Ersetzungsregeln einer Grammatik bilden eine Variable auf eine Menge von Variablen und/ oder Terminalsymbolen ab. Findet eine dieser Regeln Anwendung, so spricht man von einer Ableitung. Ausgehend von einer Startvariablen werden die Ableitungsregeln der Grammatik solange angewendet, bis ein Wort entsteht, dass nur noch Terminalsymbole enthält. Jedes Wort, welches sich auf diese Weise bilden lässt, ist Teil der durch die kontextfreie Grammatik beschriebenen Sprache. Die Backus-Naur-Form stellt dabei einen Formalismus dar, mit dessen Hilfe sich kontextfreie Grammatiken in kompakter und präziser Weise darstellen lassen. Die Tabelle 12 zeigt die Backus-Naur-Form der Grammatik, die für das Parsen von Prozessalgebra-Termen im vorliegenden Projekt verwendet wird.

<alternative>	::=	<choice> | <parallel>
<choice>	::=	<parallel> + <alternative>
<parallel>	::=	<merge> | <part>
<merge>	::=	<part> || <parallel>
<part>	::=	<seq> | <term>
<seq>	::=	<term> · <part>
<term>	::=	<prefix> | <inaction> | <emptyProcess> | (<alternative>) | <subProcess> | <encaps> | <hide>
<prefix>	::=	<action> . <term>
<encaps>	::=	encaps(<setOfActions> , <alternative>)
<hide>	::=	hide(<setOfActions> , <alternative>)
<inaction>	::=	0
<emptyProcess>	::=	1
<subProcess>	::=	X | Y | Z ...
<action>	::=	a | b | c ...

Tabelle 12: BNF der Prozessalgebra

Ausgehend von dieser Grammatik ist beispielsweise $a.1$ ein gültiger Prozessalgebra-Term, da hierfür folgende Ableitung beginnend mit der Startvariablen *<prefix>* gebildet werden kann.

<prefix>

↓

<action>.<term>

↓

a.<term>

↓

a.<emtpyProcess>

↓

a.1

Tabelle 13: BNF-Ableitung eines Prozessalgebra-Terms

Gleichermaßen gilt, dass $a.b + c.1$ kein gültiger Prozessalgebra-Term ist, da hierfür keine Ableitung ermittelt werden kann.

Scala setzt das Konzept der Parser Combinators folglich [OSV11] als Bibliothek um. Die entsprechenden Klassen befinden sich im Paket *scala.util.parsing.combinator*. Jeder Parser ist dabei eine Implementierung des *Traits Parser[T]*. Der Typparameter T steht hierbei für den Typ des gewünschten Resultats. In [OSV11] wird ein Parser vereinfacht als eine Funktion beschrieben, welche eine beliebige Eingabe auf ein Ergebnis vom Typ *ParseResult[T]* abbildet. Die zuvor definierten Ersetzungsregeln werden dann mithilfe der Parser Combinators umgesetzt. Jede Regel wird durch genau einen Parser realisiert. Im einfachsten Fall setzt ein solcher Parser eine Zeichenkette direkt in ein Domänenobjekt um. Das Listing 15 zeigt die Umsetzung derartiger Parser in Scala.

```
def inaction: Parser[ProcessExpression] = "0" ^^ (x => Inaction)
def emptyProcess: Parser[ProcessExpression] = "1" ^^ (x => EmptyProcess)
def action: Parser[Action] = """[a-z]\w*""".r ^^ (x => Action(x))
```

Listing 15: Einfache Parser in Scala

Mithilfe des Operators ^^ wird das Parsingergebnis definiert. Die ersten beiden Parser setzen das Zeichen 1 bzw. 0 in einen *Empty Process* bzw. eine *Inaction* um. Manchmal soll ein Parser jedoch eine große oder sogar unendliche Anzahl von Zeichenketten verarbeiten. So auch im Fall des Parsers *action*. Zu diesem Zweck kommen dann reguläre Ausdrücke zum Einsatz. Bei regulären Ausdrücken handelt es sich laut [Fri08] um ein Mittel zur Beschreibung von

Textmustern. Der vorliegende Ausdruck beschreibt alle möglichen Namen von Aktionen. Diese beginnen laut Konvention mit einem Kleinbuchstaben, gefolgt von beliebig vielen weiteren Buchstaben, Ziffern und Sonderzeichen. Durch die drei Anführungszeichen interpretiert der String keine Metazeichen, wie dem *Backslash*. Diese werden somit wie ganz normale Zeichen behandelt und müssen nicht *escaped* werden. Die String-Klasse in Scala besitzt die Methode *r*, welche ein Objekt vom Typ *Regex* zurückgibt. Hierbei handelt es sich laut [See11] um eine Klasse, die einen regulären Ausdruck kapselt. Das Parsingergebnis ist ein Objekt der Klasse *Action*, dessen Attribut *name* mit dem geparsten String belegt wird.

Durch die Kombination einzelner simpler Parser können komplexere Eingaben verarbeitet werden. Hierdurch entsteht ein neuer Parser, der wiederum innerhalb anderer Parser verwendet werden kann. Die Abbildung 17 zeigt exemplarisch, wie der Prozessalgebra-Term $a.1 + b.0$ mithilfe des Parsers *choice* verarbeitet wird. Im Verlauf der Verarbeitung wird das Parsen einzelner Bestandteile des Prozessalgebra-Terms an untergeordnete Parser delegiert.

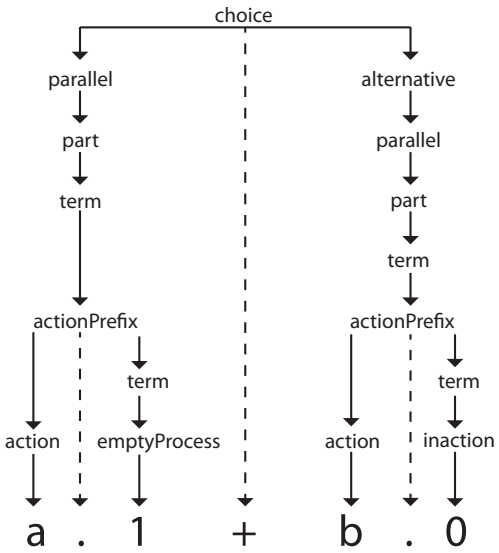

Abbildung 17: Parserbaum

Dabei können verschiedene Kombinationsarten identifiziert werden. Zwei Parser P und Q lassen sich über den Operator $\sim$ sequenziell kombinieren. Der neu entstandene Parser ist in der Lage, Eingaben unter den folgenden Bedingungen zu verarbeiten. Der Parser P konsumiert einen Teil der Eingabe und ist in der Lage, diesen zu parsen. Der Parser Q konsumiert den verbleibenden Teil der Eingabe, der noch nicht von P verarbeitet wurde und parst diesen. Es existieren zwei Spezialformen der sequenziellen Kombination. Die Operatoren $<\sim$ und $\sim>$ liefern als Resultat nur das Ergebnis des linken bzw. rechten Parsers und verwerfen das restliche Ergebnis. Das Listing 16 zeigt den Parser für einen *ActionPrefix*.

```
def prefix: Parser[ProcessExpression] = action ~ ("." ~> term)
                        ^^ { case a ~ b => ActionPrefix(a, b) }
```

Listing 16: sequenzielle Kombination von Parsern

Der dargestellte Parser verweist auf den zuvor angesprochenen Parser *action*, dessen Ergebnis ein *Action*-Objekt ist. Es folgt eine Kombination aus dem Punktzeichen und einem zweiten Parsers *term*, der als Ergebnis ein Objekt vom Typ *ProcessExpression* liefert. Für das Parsingergebnis ist das Punktzeichen irrelevant, daher wird es durch die Verwendung des Operators $\sim>$, nicht in die Ergebnismenge aufgenommen. Die beiden verbleibenden Ergebnisse werden dann zu einem *ActionPrefix* zusammengesetzt.

Bei der alternativen Kombination zweier Parser P und Q, welche durch den Operator | ausgedrückt wird, muss entweder P oder Q in der Lage sein, die vollständige Eingabe zu parsen. Zunächst wird versucht mithilfe von P die Eingabe zu parsen. Sofern P die Eingabe nicht verarbeiten kann, wird diese an Q übergeben. Nachfolgend wird versucht, die vollständige Eingabe mit Q zu parsen. Das Listing 17 zeigt den Parser *term*. Dieser ist lediglich eine Kombination aus mehreren alternativen Parsern.

```
def term: Parser[ProcessExpression] =
                    prefix | inaction | emptyProcess | "(" ~> alternative <~ ")" |
                    recursion | encaps | hide
```

Listing 17: Alternative Kombination von Parsern

Des Weiteren existieren Kombinationsmöglichkeiten, mit deren Hilfe sich wiederholende oder optionale Bestandteile der Syntax verarbeiten lassen. Das Listing 18 zeigt exemplarisch den Parser *setOfActions*, welcher eine beliebig lange, mit runden Klammern umschlossene, kommagetrennte Liste von Aktionen parst.

```
def setOfActions: Parser[List[Action]] =
                    "(" ~> repsep(action, ",") <~ ")"
```

Listing 18: Wiederholende Parser

Durch die Funktion *repsep* wird eine Wiederholung mit einem Separator dargestellt. Der Funktion wird als erstes Argument ein Parser übergeben, welcher die einzelnen Elemente der Liste weiterverarbeitet. Der Separator wird mithilfe des zweiten Arguments bestimmt. Die umschließenden runden Klammern werden nicht in die Ergebnismenge aufgenommen. Somit liefert dieser Parser eine Collection von *Action*-Objekten.

Der Parsingvorgang wird durch die Methode *parseAll* angestoßen. Dieser werden als Argumente der zu verwendende Parser und die zu parsende Zeichenkette übergeben. Da der Parsingvorgang auch ergebnislos verlaufen kann, besitzt der Typ *ParseResult* konsequenterweise die beiden Subklassen *Success[T]* und *NoSuccess*. Während die Klasse *Success* als Attribut das Parsingergebnis beinhaltet, enthält die Klasse *NoSuccess* eine Fehlermeldung mit einem Hinweis, warum das Parsen der übergebenen Eingabe fehlgeschlagen ist. Sofern der Prozessalgebra-Term erfolgreich geparst werden konnte, wird dieser an zentraler Stelle registriert, andernfalls wird eine Exception geworfen.

```
def parseExpression(expression: String) = parseAll(process, expression) match {
    case Success(process, _) => ProcessRegistry.registerProcess(process)
    case f: NoSuccess => throw new ParserException(f.msg)
}
```

Listing 19: Prozess-Term parsen

Mithilfe der Parser Combinators lassen sich die im vorherigen Kapitel beschriebenen Beschränkungen bezüglich der Bindungsstärke von Operatoren überwinden. Die Bindungsstärke wird durch die Reihenfolge bestimmt, in der die verschiedenen Parser kombiniert werden.

5.4. Verarbeitung

Zur einfacheren Auswertung müssen die Prozessalgebra-Terme eine bestimmte Struktur aufweisen. Diese, als Head-Normal-Form bezeichnete Struktur, wird in [BBR09, S.140] wie folgt definiert. Die Konstanten *EmptyProcess (1)* und *Inaction (0)* befinden sich in Head-Normal-Form. Für jede Aktion $a \in A$ und jeden beliebigen Prozess-Term t ist $a.t$ eine Head-Normal-Form. Sofern die Prozess-Terme s und t in Head-Normal-Form sind, ist auch die Auswahl $s+t$ eine Head-Normal-Form.

Für die Umformung ist ein einfaches Term-Rewriting-System zuständig, wie es in [BBR09] beschrieben wird. Mithilfe der im Anhang A dargestellten Umformungsregeln kann ein beliebiger Prozessalgebra-Term in die gleichbedeutende Head-Normal-Form überführt werden. Zu diesem Zweck werden die Gleichungen mit einer Richtung versehen. Die Gleichungen werden dabei nur von links nach rechts angewendet, die entgegengesetzte Umformung ist hingegen nicht zulässig. Die Umformungsregeln werden solange auf den Prozess-Term angewendet, bis keine weitere Umformung mehr möglich ist. Auf diese Weise entsteht ein endlicher Prozess, dessen Ergebnis ein Term in Head-Normal-Form ist. So wird beispielsweise der Prozess-Term $a.1||b.1||c.1$ mit $\gamma(a,b) = x$, $\gamma(a,c) = y$ und $\gamma(b,c) = z$ mithilfe des Term-Rewriting-Systems in die äquivalente Head-Normal-Form

$$a.(b.1||c.1) + b.(c.1||a.1) + c.(b.1||a.1) + x.c.1 + y.b.1 + z.a.1$$

überführt. Hier zeigt sich auch der Nachteil dieses Vorgehens. Selbst bei einer recht überschaubaren Anzahl an parallelen Prozessen entsteht ein sehr großer Prozess-Term. Der Vorteil hingegen liegt in der einfacheren Auswertbarkeit der Head-Normal-Form. So muss zur Bestimmung der ausführbaren Aktionen, sowie des Nachfolgezustands nur der *EmptyProcess*, die *Inaction*, der *ActionPrefix* und eine Auswahl zwischen ihnen berücksichtig werden.

Eine derartige Umformung lässt sich in Scala mithilfe des Pattern Matching erreichen. Das kann laut [OSV11] als eine verbesserte Form der Switch-Anweisung aus Java angesehen werden. Es ermöglicht eine zuvor aufgebaute Abstraktion zu zerlegen, um Subklassen auf unterschiedliche Weise zu behandeln. Wie Martin Odersky in [VS09b] darlegt, wäre dieses in Java durch eine Reihe von *instanceof*-Vergleichen und Typumwandungen möglich. Dieses Vorgehen führt jedoch zu schlecht wartbaren und unsicheren Code. Eine weitere Möglichkeit wäre die Verwendung des Visitor-Entwurfsmusters aus [GHJV94]. Die eigentliche Verarbeitung wird dabei in eine *Visitor*-Klasse ausgelagert. Diese enthält eine mehrfach überladene Methode *visit*, welche jeweils einen Subtyp von *ProcessExpression* entgegennehmen und verarbeiten kann. Des Weiteren wird jedes Domänenobjekt mit einer Methode *accept(Visitor)* ausgestattet. Innerhalb des Methodenrumpfes von *accept* wird der Methode *visit* des Visitors eine Selbstreferenz übergeben. Dadurch wird dann für jeden Subtyp von *ProcessExpression* eine differenzierte Methode aufgerufen. Dieses Vorgehen ist jedoch wesentlich aufwendiger, als das von Martin Odersky präferierte Pattern Matching.

Beim Pattern Matching wird ein Ausdruck mit einer Reihe von Mustern verglichen. Der Ausdruck wird durch das Schlüsselwort *match* von den Mustern getrennt. Jedes Muster wird mit einem *case* eingeleitet. Hinter dem => stehen Anweisungen, die im Fall einer Übereinstimmung ausgeführt werden. Die Muster werden der Reihe nach durchprobiert. Wird eine Übereinstimmung festgestellt, werden die hinterlegten Anweisungen ausgeführt und keine weiteren Muster geprüft. Eine *break*-Anweisung wie in Java ist nicht notwendig. Trifft keines der Muster zu, wird eine Exception geworfen. Laut [See11] lassen sich sechs verschiedene Pattern-Arten identifizieren, die im folgenden aufgezählt werden:

- Wildcard Pattern
- Constant Pattern
- Variable Pattern und Typed Pattern
- Tuple Pattern
- Constructor Pattern
- Sequence Pattern

Für die Umformung von Process-Algebra-Termen in die Head-Normal-Form kommt ausschließlich das Constructor Pattern zum Einsatz. Dieses kann im Zusammenhang mit Case-Klassen verwendet werden, um diese zu dekonstruieren. Die Domänenobjekte werden dabei in ihre Bestandteile zerlegt und wieder neu zusammengesetzt. Das Listing 20 zeigt exemplarisch die Umsetzung der Umformungsregeln.

```
def simplify(expression: ProcessExpression): ProcessExpression =
 expression match {
  //..
  case Sequence(ActionPrefix(a,x),y)        => ActionPrefix(a,Sequence(x,y)) //A10
  //..
  case Alternative(x,Inaction)              => simplify(x) //A6
  case Alternative(Inaction,x)              => simplify(x) //A6
  case Alternative(x,y) if x == y           => simplify(x) //A3
  //..
  case Alternative(left,right)              => Alternative(simplify(left),
                                                           simplify(right))
}
```

Listing 20: Umformung von Prozess-Termen

Die erste *case*-Anweisung überführt einen Term der Form $a.x \cdot y$ in die äquivalente Head-Normal-Form $a.(x \cdot y)$. Die Variablen a, x und y binden dabei die Klassenparameter des übergebenen Objektes. Diese können dann auf der rechten Seite zu einem neuen Objekt zusammengesetzt werden. In der darunterliegenden Zeile wird die Regel $x + 0 \rightarrow x$ umgesetzt. Trifft die Umformungsroutine auf eine Auswahl, deren rechter Term eine Inaction ist, wird als Ergebnis nur der linke Term zurückgegeben. Auf dem Term x wird dann die Umformungsroutine erneut rekursiv aufgerufen, um auf diesem gegebenenfalls weitere Vereinfachungen durchzuführen. Da das Constructor Pattern nicht kommutativ ist, muss eine zweite Anweisung mit vertauschten Klassenparametern die Möglichkeit abdecken, dass der linke Term eine Inaction ist. Mithilfe der sogenannten Pattern Guards lassen sich weitere Bedingungen an das Muster knüpfen. Für die Umformungsregel $x + x \rightarrow x$ muss geprüft werden, ob die beiden Alternativen x und y identisch sind. Durch die letzte Anweisung werden, sofern keines der vorherigen Muster zutrifft, die einzelnen Alternativen der Auswahl vereinfacht. Die Kontextregel einer Algebra besagt, dass die Vereinfachungen auch auf Teilterme angewendet werden können, ohne dass sich die Bedeutung des Gesamtausdrucks verändert.

Das Listing 21 zeigt, wie mithilfe dieser Umformungen die Ableitung eines Terms gebildet werden kann. Bei der Methode *exec* handelt es sich um eine sogenannte innere Funktion. Sie wird innerhalb des Methodenrumpfes definiert und kann auch nur dort verwendet werden. Zunächst wird der Term der oben beschriebenen Methode *simplify* übergeben, welche ihn in eine Head-Normal-Form überführt. Hierdurch brauchen bei der anschließenden Auswertung nur die *Inaction*, der *EmptyProcess*, der *ActionPrefix* und eine Auswahl zwischen diesen betrachtet werden.

```
def execute(action: Executable, expression: ProcessExpression): ProcessExpression = {
  // Inner Function for recursive invocation
  def exec(action: Executable, expression: ProcessExpression): List[ProcessExpression] =
        simplify(expression) match {
               case Inaction                              => List()
               case EmptyProcess                          => List()
               case ActionPrefix(prefix, restTerm)        => if (prefix == action)
                                                               List(restTerm) else List()
               case SilentStepPrefix(prefix, restTerm)    => if (prefix == action)
                                                               List(restTerm) else List()
               case Alternative(left, right)              => exec(action, left) :::
                                                               exec(action, right)
        }

        val terms = exec(action, expression)

        if (terms.isEmpty) expression else terms(Random.nextInt(terms.length))
}
```

Listing 21: Ableitung von Prozess-Termen

Sofern es sich bei dem übergebenen Term um eine Auswahl handelt, wird die Verarbeitungsroutine rekursiv auf den beiden Teiltermen aufgerufen und die Ergebnisse mittels *:::* in einer neuen Collection vereint. Wird die Methode mit einem *ActionPrefix* aufgerufen, wird anschließend geprüft, ob die ausgeführte Aktion mit der vorgelagerten Aktion aus dem ActionPrefix übereinstimmt. Trifft dies zu, wird der Restterm in die Menge der möglichen Folgezustände übernommen. Durch den nichtdeterministischen Charakter der Prozessalgebra kann die Ausführung einer Aktion unterschiedliche Folgezustände hervorrufen. Die innere Funktion *exec* gibt eine Collection zurück, welche alle möglichen Folgezustände enthält. Aus dieser Collection wird dann zufällig ein Element gewählt und zurückgegeben.

6. Fazit

Dieses Kapitel hat zum Ziel, den Verlauf des Projektes zu umreißen. Dabei wird an dieser Stelle auf eine Bewertung des Resultates verzichtet. Stattdessen soll hier auf die Schwierigkeiten, Fehler und Risiken und den daraus resultierenden Erkenntnisgewinn eingegangen werden. Anschließend soll ein kurzer Ausblick auf mögliche Weiterentwicklungen gegeben werden.

6.1. Projektbewertung

Die objekt-funktionale Programmiersprache Scala bietet eine gute Unterstützung für die Erstellung domänenspezifischer Sprachen. Durch Spracheigenschaften, wie Infix-Operatornotation, Implicit-Conversion sowie Semicolon und Type Inference lässt sich auf einfache Weise eine interne DSL umsetzen. Leider lässt sich in diesen Kontext die Bindungsstärke von Operatoren nicht beeinflussen. Aus diesem Grund wurde im Rahmen dieser Bachelorarbeit eine zweite, externe DSL implementiert. Hierfür kamen Paser Combinators zum Einsatz. Dieses Konzept ermöglicht den einfachen Aufbau von Parsern, um beliebige Zeichenketten zu verarbeiten. Laut [Gho11] handelt es sich um eine interne DSL zur Erstellung einer externen DSL. Dieses Verfahren erwies sich als äußerst flexibel. Bei einer externen DSL müssen jedoch sämtliche Werkzeuge selbst erstellt werden, während bei einer internen DSL auf die Infrastruktur der Wirtssprache zurückgegriffen werden kann. Die Umsetzung als Scala-Bibliothek ermöglicht zum Beispiel die Nutzung eines Editors, wie Eclipse. Merkmale, wie Autovervollständigung und Fehlerkorrektur stehen dann von Haus aus zur Verfügung. Währenddessen muss, sofern auf diese Hilfestellungen nicht verzichtet werden kann, für eine externe DSL ein eigener Editor bzw. ein Eclipse-Plugin erstellt werden. Alternativ kann auch ein einfacher Texteditor wie der *vi* genutzt werden.

Mithilfe des Pattern Matching konnte eine regelbasierte Transformation von Prozessalgebra-Termen umgesetzt werden. Hierdurch konnten die Terme vor der Verarbeitung in die Head-Normal-Form überführt werden. Dies vereinfachte die Berechnung der ausführbaren Aktionen, sowie des Folgezustands enorm.

Auffällig ist, dass Scala signifikant weniger Quellcode als andere etablierte Programmiersprachen benötigt, um ein Programm mit identischer Funktionalität zu erstellen. Zusammenfassend lässt sich feststellen, dass die Programmiersprache Scala ausgereift und stabil ist, sowie eine Reihe interessanter neuer Konzepte einführt, mit denen die Produktivität bei der Softwareentwicklung deutlich gesteigert werden kann.

6.2. Ausblick

Viele Experten sind heute der Ansicht, dass die Herausforderungen der parallelen Programmierung zu einem tiefgreifenden Paradigmenwechsel führen werden, wie zuletzt durch das Aufkommen der Objektorientierung verursacht. In seinem Artikel [Vel10] formuliert Gert Veltink die Fragestellung „How are we going to use object-oriented technology to help and tap into the parallel resources that modern hardware has to offer?". Die Antwort des Autors folgt bereits im nächsten Satz „We have to start to think in parallel terms!". In der Prozessalgebra manifestiert sich diese parallele Denkweise. Gegenwärtige Programmiersprachen begegnen parallelen Systemen oft auf einem sehr geringen Abstraktionsniveau. Unter der Annahme, dass jedes Objekt prinzipiell parallel zu jedem anderen Objekt aktiv sein kann, nähert sich die Objektorientierung konzeptionell der Prozessalgebra. Führt man diesen Gedanken weiter, so könnten Methodenaufrufe mit atomaren Aktionen gleichgesetzt werden. Die Konzepte der objektorientierten Programmierung und der Prozessalgebra könnten so im Laufe der Zeit miteinander verschmelzen und Programmiersprachen hervorbringen, die dem Anwendungsentwickler bei der Erstellung paralleler und distributiver Systeme entlasten.

Einige Ansätze lassen sich bereits in modernen Programmiersprachen erkennen. So setzt Scala laut [Hal12] das Actor-Model als Klassen-Bibliothek um. In diesem, erstmals 1973 in [HBS73] von Carl Hewitt, Peter Bishop und Richard Steiger beschriebenen Modell, existieren sogenannte Aktoren als unabhängige Aktivitätsträger. Beim Actor-Model wird zugunsten eines asynchronen, nachrichtenbasierten Kommunikations-Modells auf einen gemeinsamen Speicher verzichtet. Hierdurch ist keine Synchronisierung der einzelnen Aktoren mittels Locks, Semaphoren oder Monitoren erforderlich. Die parallel laufenden Aktoren sind in der Lage, sich gegenseitig Nachrichten zuzustellen. Diese werden dann zunächst in einer Art Postfach abgelegt. Somit hat jeder Aktor die Möglichkeit, die einkommenden Nachrichten sequenziell abzuarbeiten oder zu verwerfen.

Das im Rahmen dieser Bachelorarbeit entstandene Programm konzentriert sich jedoch auf die Spezifikation von Systemen und nicht auf deren Erstellung. Das hier entstandene Anwendungsprogramm könnte beispielsweise in der Lehre eingesetzt werden, um Studierenden die Konzepte der Prozessalgebra näher zu bringen. Durch das Programm können den formalen Spezifikationen der Prozessalgebra Leben eingehaucht werden. Die Studierenden könnten so am Beispiel nachvollziehen, welche Wirkung die einzelnen Operatoren und Terme haben.

A. Umformungsregeln

$(x + y) + z$	$=>$	$x + (y + z)$	A2
$x + x$	$=>$	x	A3
$(x + y) \cdot z$	$=>$	$x \cdot z + y \cdot z$	A4
$(x \cdot y) \cdot z$	$=>$	$x \cdot (y \cdot z)$	A5
$x + 0$	$=>$	x	A6
$0 \cdot x$	$=>$	0	A7
$x \cdot 1$	$=>$	x	A8
$1 \cdot x$	$=>$	x	A9
$a.x \cdot y$	$=>$	$a.(x \cdot y)$	A10
$\partial_H(1)$	$=>$	1	D1
$\partial_H(0)$	$=>$	0	D2
$\partial_H(a.x)$ if $a \in H$	$=>$	0	D3
$\partial_H(a.x)$ if $a \notin H$	$=>$	$a.\partial_H(x)$	D4
$\partial_H(x + y)$	$=>$	$\partial_H(x) + \partial_H(y)$	D5
$x \parallel 1$	$=>$	x	SC2
$1 \mid x + 1$	$=>$	1	SC3
$(x \parallel y) \parallel z$	$=>$	$x \parallel (y \parallel z)$	SC4
$(x \mid y) \mid z$	$=>$	$x \mid (y \mid z)$	SC5
$(x \mid\_ y) \mid\_ z$	$=>$	$x \mid\_ (y \parallel z)$	SC6
$(x \mid y) \mid\_ z$	$=>$	$x \mid (y \mid\_ z)$	SC7
$x \parallel y$	$=>$	$x \mid\_ y + y \mid\_ x + x \mid y$	M
$0 \mid\_ x$	$=>$	0	LM1
$1 \mid\_ x$	$=>$	0	LM2
$a.x \mid\_ y$	$=>$	$a.(x \parallel y)$	LM3
$(x + y) \mid\_ z$	$=>$	$x \mid\_ z + y \mid\_ z$	LM4
$0 \mid x$	$=>$	0	CM1
$(x + y) \mid z$	$=>$	$x \mid z + y \mid z$	CM2
$1 \mid 1$	$=>$	1	CM3
$a.x \mid 1$	$=>$	0	CM4
$a.x \mid a.x$ if $\gamma(a, b) = c$	$=>$	$c.(x \parallel y)$	CM5
$a.x \mid a.x$ if not $\gamma(a, b) = c$	$=>$	0	CM6
$a.(\tau.(x + y) + x)$	$=>$	$a.(x + y)$	B
$\tau_I(1)$	$=>$	1	TI1
$\tau_I(0)$	$=>$	0	TI2
$\tau_I(a.x)$ if $a \notin I$	$=>$	$a.\tau_I(x)$	TI3
$\tau_I(a.x)$ if $a \in I$	$=>$	$\tau.\tau_I(x)$	TI4
$\tau_I(x + y)$	$=>$	$\tau_I(x) + \tau_I(y)$	TI5

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literatur

[AB02] Asteroth, Alexander ; Baier, Christel: *Theoretische Informatik - Eine Einführung in Berechenbarkeit, Komplexität und formale Sprachen mit 101 Beispielen.* Pearson, 2002

[Ale78] Alexander, Christopher: *The Oregon Experiment.* Oxford University Press, 1978

[ANT] *ANTLR.* `http://www.antlr.org`, Abruf: 10.08.13

[BA06] Ben-Ari, M.: *Principles of cuncurrent and distributed programming.* 2. Addison-Wesley, 2006

[Bae05] Baeten, J.C.M.: *A brief history of process algebra.* http://www.win.tue.nl/fm/0402history.pdf : Technische Universiteit Eindhoven, 2005

[Bal11] Balzert, Helmut: *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb.* 3. Springer Verlag, 2011

[BBR09] Baeten, J. C. M. ; Basten, T. ; Reniers, M. A.: *Process Algebra: Equational Theories of Communicating Processes.* 1. Cambridge University Press, 2009

[Blo08] Bloch, Joshua: *Effective Java: A Programming Language Guide.* Addison-Wesley, 2008

[Bre09] Breshears, Clay: *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications.* 1. O'Reilly, 2009

[Dij82] Dijkstra, Edsger W.: *On the role of scientific thought.* 1982

[ECLa] *Eclipse AST API.* `http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html`, Abruf: 10.08.13

[ECLb] *XText.* `http://www.eclipse.org/Xtext/`, Abruf: 10.08.13

[EPF11] EPFL: *The Scala Language Specification Version 2.9.* `www.scala-lang.org/docu/files/ScalaReference.pdf`. Version: 2011, Abruf: 22.05.2013

[EPF13] EPFL: *An Overview of the Scala Programming Language.* `http://www.scala-lang.org/docu/files/ScalaOverview.pdf`. Version: 2013, Abruf: 21.05.2013

[Ern08] Ernst, Hartmut: *Grundkurs Informatik*. 4. Springer, 2008

[Fok07] Fokkink, Wan: *Introduction to Process Algebra*. Springer Verlag, 2007

[Fow05] Fowler, Martin: *FluentInterface*. `http://martinfowler.com/bliki/Fluent-Interface.html`. Version: 12 2005, Abruf: 21.05.2013

[Fow09] Fowler, Martin: *Introducing Domain-Specific Languages*. `http://channel9.msdn.com/Series/DSL-DevCon-2009/Martin-Fowler-Introducing-Domain-Specific-Languages`. Version: 2009, Abruf: 17.06.2013

[FP10] Fowler, Martin ; Parsons, Rebecca: *Domain Specific Languages*. Addison-Wesley, 2010

[Fri08] Friedl, Jeffrey E. F.: *Reguläre Ausdrücke*. 3. O'Reilly, 2008

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design patterns: elements of reusable object-oriented software*. 1. Addison-Wesley Longman, Amsterdam, 1994

[Gho11] Ghosh, Debasish: *DSLs in Action*. Manning, 2011

[Hal12] Haller, Philipp: *Actors in Scala*. 1. Artima, 2012

[HBS73] Hewitt, Carl ; Bishop, Peter ; Steiger, Richard: A Universal Modular Actor Formalism for Artificial Intelligence. In: *Proceeding IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence* (1973), S. 235–245

[Hoa78] Hoare, C. A. R.: *Communicating Sequential Processes*. http://www.usingcsp.com/cspbook.pdf : Prentice Hall, 1978

[Ind11] Inden, Michael: *Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung*. 1. dpunkt.verlag, 2011

[Kec05] Kecher, Christoph: *UML 2: Das umfassende Handbuch*. Galileo Computing, 2005

[LL10] Ludewig, Jochen ; Lichter, Horst: *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, 2010

[LR09] Lahres, Bernhard ; Rayman, Gregor: *Objektorientierte Programmierung*. 2. Galileo Computing, 2009

[Mey98] Meyer, Bertrand: *Object-Oriented Software Construction*. 2. Prentice Hall, 1998

[Mil80] Milner, Robin: *A Calculus of Communicating Systems*. Springer, 1980

[Moo65] Moore, Gordon E.: Cramming More Components onto Integrated Circuits. In: *Electronics* (1965), Nr. 8, 114–117. `http://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf`, Abruf: 04.08.2013

[MPS] *MPS*. `http://www.jetbrains.com/mps/`, Abruf: 10.08.13

[OSV11] Odersky, Martin ; Spoon, Lex ; Venners, Bill: *Programming in Scala*. Artima, 2011

[Pie10] Piepmeyer, Lothar: *Grundkurs funktionale Programmierung mit Scala*. Hanser Fachbuchverlag, 2010

[Ray99] Raymond, Eric: *The Cathedral and the Bazaar*. O'Reilly, 1999

[Roy70] Royce, Winston: *Managing the Development of Large Software Systems*. 1970

[RSS09] *RSS Specification*. `http://www.rssboard.org/rss-specification`. Version: 2009, Abruf: 10.08.13

[Sch08] Schöning, Uwe: *Theoretische Informatik - kurz gefasst*. 5. Spektrum Akademischer Verlag, 2008

[See11] Seeberger, Heiko: *Durchstarten mit Scala*. entwickler.press, 2011

[SN09] Scholz, Michael ; Niedermeier, Stephan: *Java und XML: Alles zu DOM, SAX, JAXP, StAX. JAXB und Webservices sowie den Grundlagen des XML-Publishing-Prozesses*. 2. Galileo Computing, 2009

[SP09] Shneiderman, Ben ; Plaisant, Catherine: *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 5. Pearson, 2009

[SS11] Sterlingand, Mary J. ; Steffen, Eva: *Algebra für Dummies*. 2. Wiley-VCH, 2011

[Sta07] Staudemeyer, Jörg: *Groovy für Java-Entwickler*. 1. O'Reilly, 2007

[Ste99] Steele, Guy: *Growing a Language*. `http://www.cs.virginia.edu/~evans/cs655/readings/steele.pdf`. Version: 1999, Abruf: 21.05.2013

[Ste09] Steiner, René: *Grundkurs Relationale Datenbanken*. 7. Springer Verlag, 2009

[Sut05] Sutter, Herb: *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. www.gotw.ca/publications/concurrency-ddj.htm. Version: 02 2005, Abruf: 08.06.2013

[Tan09] Tanenbaum, Andrew S.: *Moderne Betriebssysteme*. Pearson, 2009

[Ull11a] Ullenboom, Christian: *Java 7 - Mehr als eine Insel*. 1. Galileo Computing, 2011

[Ull11b] Ullenboom, Christian: *Java ist auch eine Insel*. 10. Galileo Computing, 2011

[VAC+09] Vogel, Oliver ; Arnold, Ingo ; Chughtai, Arif ; Ihler, Edmund ; Kehrer, Timo ; Mehlig, Uwe ; Zdun, Uwe: *Software-Architektur: Grundlagen Konzepte Praxis*. 2. Springer Verlag, 2009

[Vel95] Veltink, G. J.: *Tools for PSF*. University of Amsterdam, 1995

[Vel10] Veltink, G.J.: PSF - A Retrospective. In: *Fundamenta Informaticae XXI* (2010), S. 1001–1047

[Ven09] Venners, Bill: *The Feel of Scala*. http://parleys.com/play/514892250364bc17fc56b9a5. Version: 2009, Abruf: 21.05.2013

[VS09a] Venners, Bill ; Sommers, Frank: *The Origins of Scala A Conversation with Martin Odersky, Part I*. www.artima.com/scalazine/articles/origins_of_scala.html. Version: 05 2009, Abruf: 21.05.2013

[VS09b] Venners, Bill ; Sommers, Frank: *The Point of Pattern Matching in Scala A Conversation with Martin Odersky, Part IV*. http://www.artima.com/scalazine/articles/pattern_matching.html. Version: 05 2009, Abruf: 21.07.2013

[W3C02a] *Doctype Declarations*. http://www.w3.org/QA/2002/04/valid-dtd-list.html. Version: 2002, Abruf: 10.08.13

[W3C02b] *XHTML Specification*. http://www.w3.org/TR/xhtml1/. Version: 2002, Abruf: 10.08.13

[W3C04] *XML Schema Specification*. http://www.w3.org/TR/xmlschema-0/. Version: 2004, Abruf: 10.08.13

[W3C10] *MathML 3.0 Specification*. http://www.w3.org/TR/MathML3/. Version: 2010, Abruf: 10.08.13