KoProServ
Komponenten • Produktivität • Services

Michael Becker

Formales Metamodell für Dienstleistungskomponenten

Formalisierung unter besonderer Berücksichtigung der Produktivität

Abstract: Die aus dem Industrial und Software Engineering bekannten Konzepte der Modularisierung lassen sich auch in der Dienstleistungsdomäne nutzen, um deren Komplexität beherrschbar zu machen. Dazu werden in diesem Artikel formale Konzepte spezifiziert, mit denen sich Dienstleistungskomponenten definieren lassen. Darauf aufbauend wird auf die kundenindividuelle Konfiguration der Komponenten eingegangen, bei der sich an kundenspezifische Anforderungen angepasste Dienstleistungen aus Komponenten erstellen lassen.

1 Einleitung

Anbieter von Dienstleistungen stehen heute vor der Herausforderung, einerseits individuell auf Anforderungen von Kunden reagieren zu können, andererseits die Dienstleistungen möglichst effizient zu erbringen, um den gestiegenen Marktanforderungen gerecht zu werden (Böttcher & Klingner, 2011) . Durch die steigende Komplexität von Dienstleistungen und

deren zunehmende wirtschaftliche Relevanz für Unternehmen existiert bereits seit einiger Zeit die Forderung, Konzepte der Modularisierung, wie sie auch in den Domänen des Industrial und des Software Engineering bekannt sind, in die Dienstleistungsdomäne zu übertragen (Akkermans, et al., 2004). Die Modularisierung fokussiert Dienstleistungen mit den folgenden Eigenschaften:

- Business-to-Business: Dienstleistungen, die sich an Endkunden richten, stehen nicht im Fokus. Stattdessen werden Dienstleistungen betrachtet, die anderen Geschäftsbereichen angeboten werden, sei es unternehmensintern oder -übergreifend.
- Produktorientiert: Dienstleistungen werden in Kombination mit Produkten angeboten und sind mit diesen verknüpft. Sie zielen daher nicht auf Menschen ab (wie z.B. Unterhaltungs- oder medizinische Leistungen).
- Komplex: Die Modularisierung unterstützt die Reduzierung der Komplexität von Dienstleistungen. Das Kosten-Nutzen-Verhältnis der Modularisierung einfacher Dienstleistungen ist eher schlecht.
- Konfigurierbar: Dienstleistungen sollen anhand kundenspezifischer Anforderungen konfiguriert (angepasst) werden.

Im Folgenden wird für die Bestandteile modularisierter Dienstleistungen der Begriff der Komponenten verwendet. Die Gesamtheit aller Dienstleistungen, die durch ein Unternehmen angeboten werden, ist im Portfolio gesammelt. Bei entsprechender Strukturierung des Portfolios wird eine individuelle Reaktion auf Kundenwünsche ermöglicht. Weiterhin wird die Analyse der Produktivität komplexer Dienstleistungen vereinfacht, da diese anhand von kleinteiligeren Komponenten sowie deren Abhängigkeiten untereinander vorgenommen werden kann.

Die vorliegende Arbeit stellt ein ganzheitliches, formal spezifiziertes Modell für das Konzept der Modularisierung im Bereich der Dienstleistungen vor. Die Modularisierung wird dabei durch die Zerlegung von Dienstleistungen in Komponenten erreicht. Dazu ist die Arbeit wie folgt gegliedert. Im nachfolgenden zweiten Abschnitt werden die Konzepte vorgestellt, welche notwendig sind, um eine komponentenbasierte Modellierung von Dienstleistungen vorzunehmen. Darauf aufbauend liefert der dritte Abschnitt die Grundlagen zur kundenindividuellen Konfiguration von Dienstleistungen. Um die formal definierten Konzepte anschaulich darzustellen wird anhand der textuellen Beschreibung eines Beispiels in Kapitel 4 das Modell dieses Portfolios entwickelt. Daran an schließt sich die Vorstellung eines UML-Modells, welches auf den formalen Konzepten basiert. Dieses dient als Grundlage zur Entwicklung eines Werkzeugs, welches die Modellierung und Konfiguration von Dienstleistungen ermöglicht. Zum Schluss gibt es ein Fazit und es folgt ein Ausblick auf weitere notwendige Arbeiten zur Umsetzung der Idee der komponentenbasierten Entwicklung von Dienstleistungen.

2 Konzepte für Dienstleistungskomponenten

Im folgenden Abschnitt werden die grundlegend notwendigen Konzepte zur Spezifikation von Dienstleistungen unter Nutzung aussagen- und prädikatenlogischer Formeln vorgestellt. Diese

Formalisierung bietet trotz ihres initialen Aufwands zwei wesentliche Vorteile. Einerseits ermöglicht sie eine eindeutige und unmissverständliche Beschreibung der verwendeten Konzepte, da die Logik im Gegensatz zur natürlichen Sprache eine formal definierte Semantik besitzt. Andererseits ermöglicht diese Form der Spezifikation aber auch eine einfache Erweiterbarkeit und Anpassbarkeit. Dadurch können neue Konzepte, die in der aktuellen Arbeit nicht von Relevanz sind leicht in das Modell integriert werden und existierende Konzepte können je nach Anwendungsgebiet leicht verändert werden.

Die notwendigen Konzepte zur Modellierung von Dienstleistungen aus Komponenten lassen sich wie folgt kurz zusammenfassen

- Komponenten zur Definition der Funktionalität einer Dienstleistung,
- Hierarchische, nicht-hierarchische und zeitliche Beziehungen zwischen Komponenten, um deren Abhängigkeiten untereinander darzustellen,
- Kennzahlen zur Definition der Produktivität von Komponenten und der gesamten Dienstleistung,
- Variablen, um die Umgebung, in der Dienstleistungen erbracht werden, flexibel abzubilden,
- Attribute, um die Leistungsfähigkeit einzelner Komponenten genauer zu definieren,
- Constraints, um die Auswahl einzelner Komponenten anhand kundenspezifischer Vorgaben einzuschränken.

2.1 Dienstleistungskomponenten

Eine Dienstleistungskomponente stellt eine wohldefinierte, abgrenzbare Funktionalität dar, die Ressourcen benötigt und verändert. Die Funktionalität selber wird dabei über präzise spezifizierte Schnittstellen angeboten. Komponenten sind damit Teilprozesse einer Gesamtleistung (Böttcher & Fähnrich, 2009) und repräsentieren logisch zusammengehörige Aktivitäten.

Die kundenspezifische Konfiguration von Dienstleistungen wird dadurch ermöglicht, dass eine Menge von Komponenten zu einer neuen Komponente (bzw. einer Gesamtleistung) zusammengesetzt werden kann. Andererseits ist auch eine Zerlegung von Komponenten in kleinere Einheiten möglich. Die Funktionalität einer Komponente, welche sich aus mehreren Komponenten zusammensetzt, ist die Summe der Funktionalitäten der zusammengesetzten Komponenten (Böttcher & Klingner, 2011).

Komponenten lassen sich durch funktionale Eigenschaften (zur Beschreibung der tatsächlichen Funktionalität) sowie nichtfunktionale Eigenschaften (zur Beschreibung von Restriktionen an die Funktionalität) beschreiben. Nichtfunktionale Eigenschaften sind z.B. die Identität, der Anbieter oder auch die zeitliche und lokale Verfügbarkeit einer Komponente. Diese werden in Abschnitt 2.3 durch das Konzept der Attribute detailliert vorgestellt. Funktionale Eigenschaften lassen sich durch verschiedene Konzepte darstellen. Die Angabe von Zielen beschreibt Zustandsänderungen, die durch eine Komponente erfüllt werden.

Mittels einer Klassifikation lässt sich die Funktionalität implizit beschreiben. Schließlich kann die Funktionalität von Komponenten auch durch die Beschreibung der Änderungen beteiligter Ressourcen beschrieben werden (Böttcher & Fähnrich, 2009).

Zur Entwicklung des Metamodells soll an dieser Stelle nur die Identität von Komponenten betrachtet werden. Diese ist für die spätere Konfiguration unerlässlich. Dazu wird die Menge *Components* verwendet, die alle verfügbaren Komponenten eines Portfolios enthält. Die Abbildung *ServiceIdentity* verbindet eine Komponente mit einem eindeutigen Identifikator aus der Menge *Identities*, welche alle verfügbaren Identifikatoren enthält.

- $ServiceIdentity: Components \rightarrow Identities$
- $\forall c_1, c_2 \in Components: c_1 \neq c_2 \leftrightarrow ServiceIdentity(c_1) \neq ServiceIdentity(c_2)$

2.2 Hierarchische Beziehungen zwischen Komponenten

Die Komponentisierung hat zum Ziel, komplexe Gesamtleistungen anhand einzelner, weniger komplexer Komponenten zu beschreiben. Um Dienstleistungen in Komponenten aufzuteilen ist es notwendig, die Beziehungen zwischen den Komponenten festzulegen. Eine grundlegende Angabe darüber wird durch die Definition hierarchischer Beziehungen ermöglicht. Diese sind ähnlich aufzufassen wie Featuremodelle zur Modellierung von Softwareproduktlinien (Mendonca, et al., 2009) oder Gozintographen zur Modellierung von Produkthierarchien (Vazsonyi, 1954). Im Gegensatz zu Produkten bzw. Features werden Dienstleistungskomponenten als Prozessbestandteile angesehen. Demzufolge lässt sich die Zerlegung von Komponenten in Teilkomponenten als Verfeinerung von Prozessen interpretieren. Die Definition hierarchischer Beziehungen zwischen Komponenten ist grundlegend zur Reduzierung der Komplexität von Dienstleistungen. Die formale Beschreibung hierarchischer Beziehungen zwischen Komponenten lässt sich auf Grundlage eines gerichteten, zyklenfreien Graphen G (im Folgenden als Konfigurationsgraph bezeichnet) definieren. Die Menge der Knoten enthält dabei sowohl die Komponenten als auch die Konnektoren. Konnektoren sind Hilfskonstrukte, mit denen Komponenten verbunden werden. Wie später gezeigt, ermöglichen sie die Definition einer detaillierten Konfigurationssemantik.

- $G = (V, E)$
- $V = Components \cup Connectors$
- $E \subseteq \{(v_1, v_2) \mid v_1, v_2 \in V\}$

Die Zyklenfreiheit des Graphen ist notwendig, da eine Komponente sich nicht selber enthalten kann (transformiert in die Prozessbeschreibung würde sich dadurch die Schachtelung eines Prozesses in sich selber ergeben). Daneben gibt es noch weitere Bedingungen, denen die Struktur des Graphen genügen muss. So dürfen Komponenten nur durch Konnektoren miteinander verbunden werden aber niemals direkt.

Aus Gründen der Konfiguration darf ein Konnektor maximal einen Oberknoten besitzen. Dies ist notwendig, um die Konfigurationssemantik zu erhalten. Im Gegensatz dazu dürfen

Komponenten mehr als einen Oberknoten besitzen – dadurch entstehen sogenannte Klone. Der Hintergrund hinter dieser Konvention wird in Abschnitt 2.8 gegeben.

Um die Anforderungen zu spezifizieren werden die Abbildungen $prenodes$ zur Identifikation vorhergehender Knoten sowie $postnodes$ zur Identifikation nachfolgender Knoten herangezogen.

- $prenodes: V \rightarrow P(V)$
  $$prenodes(v_1) = \{v_2 \in V: \exists e \in E: e = (v_2, v_1)\}$$
- $postnodes: V \rightarrow P(V)$
  $$postnodes(v_1) = \{v_2 \in V: \exists e \in E: e = (v_1, v_2)\}$$
- Zyklenfreiheit:
  $\forall (v_1, v_2) \in E: \exists v_1 P v_2 \rightarrow \nexists v_2 Q v_1$ mit P, Q … Pfade aus G
- Direkte Verbindung von Komponenten verbieten
  $$\forall (v_1, v_2) \in E: \{v_1, v_2\} \nsubseteq Components$$
- Anzahl der Elternknoten eines Konnektors auf 1 beschränken
  $$\forall c \in Connectors: |prenodes(c)| = 1$$

2.2.1 Kardinalitäten

Mit den bisher definierten Konzepten der Komponenten und Konnektoren lassen sich bereits einfache hierarchische Beziehungen zwischen den Komponenten einer Dienstleistung ausdrücken. Dies ist allerdings nicht ausreichend, um komplexe Zusammenhänge (z.B. alternative oder optionale Auswahlmöglichkeiten) zwischen Komponenten darzustellen. Deshalb wird im Folgenden das Konzept der Kardinalitäten vorgestellt.

Durch die Verwendung von Kardinalitäten ist es möglich, genau zu spezifizieren, wie viele nachfolgende Knoten bei der Konfiguration gewählt werden müssen. Einem Konnektor lässt sich mit der Abbildung $Cardinalities$ eine beliebige Menge von Kardinalitäten zuweisen. Jede Kardinalität wird als Tupel (min, max) dargestellt, wobei min festlegt, wie viele nachfolgende Knoten mindestens gewählt werden müssen. Entsprechend legt max fest, wie viele nachfolgende Knotenen maximal gewählt werden dürfen. Durch Gleichheit beider Werte lässt sich spezifizieren, dass eine feste Anzahl nachfolgender Knoten gewählt werden muss. Zur Vereinfachung der Konfiguration kann durch die Abbildung $DefaultCardinality$ eine Standardkardinalität angegeben werden. Mit der Standardkardinalität ist es möglich, einen Vorschlag zu geben, welche Anzahl an nachfolgenden Knoten standardmäßig gewählt werden soll. Die Standardkardinalität ist dabei immer ein fester Wert, wodurch die beiden Werte des Tupels identisch sind.

- $Cardinalities: Connectors \rightarrow P(\mathbb{N} \times \mathbb{N})$
- $DefaultCardinality: Connectors \rightarrow \mathbb{N} \times \mathbb{N}$
- $\forall c \in Connectors, \forall (n_1, n_2) \in DefaultCardinality(c): n_1 = n_2$

Die definierten Kardinalitäten müssen verschiedenen Bedingungen genügen. Der Mindestwert einer Kardinalität darf den Maximalwert nicht übersteigen und der Maximalwert darf nicht größer als die Anzahl nachfolgender Knoten sein. Weiterhin darf die Menge der Kardinalitäten eines Konnektors keine Überlappungen enthalten. Überlappen sich zwei Kardinalitäten sind diese redundant und müssen zu einer zusammengefasst werden. Die letzte Bedingung erfordert, dass die Standardkardinalität eines Konnektors einer der Kardinalitäten zugeordnet werden kann, da ansonsten die Konfiguration unter Nutzung der Standardkardinalität invalide wird.

- Maximalwert der Kardinalitäten darf nicht größer als die Anzahl nachfolgender Knoten sein:
  $$\forall c \in Connectors, \forall (m,n) \in Cardinalities(c): (m \leq n) \wedge (n \leq |postnodes(c)|$$
- Keine Überlappungen
  $$\forall c \in Connectors \forall (m_1, n_1) \in Cardinalities(c):$$
  $$\nexists (m_2, n_2) \in Cardinalities(c): (m_1 \leq m_2 \leq n_1) \vee (m_1 \leq n_2 \leq n_1)$$
- Standardkardinalitäten müssen gültig sein
  $$\forall (m_1, n_1) \in DefaultCardinality(c):$$
  $$\exists (m_2, n_2) \in Cardinalities(c): m_2 \leq m_1 = n_1 \leq n_2$$

2.2.2 Vordefinierte Konnektortypen

In der Praxis werden oftmals nur bestimmte Typen von Konnektoren benötigt. Diese sind in Tabelle 1 mit ihrer Beschreibung in natürlicher Sprache sowie ihrer Formalisierung vorgestellt. Die vordefinierten Konnektortypen erlauben eine vereinfachte Spezifikation von Komponentenmodellen, da sie die Formulierung der entsprechenden aussagenlogischen Formel kapseln. Es ist zu beachten, dass je vordefinierten Typen nur genau eine Angabe zur Kardinalität erlaubt ist.

Tabelle 1 Vordefinierte Konnektortypen und ihre Formalisierung

Name	Beschreibung	Formalisierung
Konjunktivknoten (AND)	Bei der Konfiguration müssen alle nachfolgenden Knoten gewählt werden.	$\forall c \in C_{AND}: Cardinalities(c) = \{(\|postnodes(c)\|, \|postnodes(c)\|)\}$
Exklusiv-Oder-Knoten (XOR)	Bei der Konfiguration muss genau ein nachfolgender Knoten gewählt werden	$\forall c \in C_{XOR}: Cardinalities(c) = \{(1,1)\}$
Optional-Exklusiv-Oder-Knoten (XOR^*)	Bei der Konfiguration darf höchstens ein nachfolgender Knoten gewählt werden	$\forall c \in C_{XOR^*}: Cardinalities(c) = \{(0,1)\}$
Disjunktiv-Obligatorischer-Knoten (OR^+)	Bei der Konfiguration muss mindestens ein nachfolgender Knoten gewählt werden	$\forall c \in C_{OR^+}: Cardinalities(c) = \{(1, \|postnodes(c)\|)\}$
Disjunktivknoten (OR)	Bei der Konfiguration kann eine beliebige Menge nachfolgender Knoten	$\forall c \in C_{OR}: Cardinalities(c) = \{(0, \|postnodes(c)\|)\}$

	gewählt werden	

2.3 Attribute

Die Definition nichtfunktionaler Eigenschaften ist notwendig, um die Möglichkeiten und Grenzen von Dienstleistungskomponenten zu modellieren. Verschiedene Komponenten, die ähnliche Prozesse repräsentieren, unterscheiden sich oftmals in ihrer Leistungsfähigkeit voneinander, da sie z.B. durch unterschiedliche Anbieter erbracht werden. Obwohl dies keine Einschränkung der eigentlichen Funktionalität ist, ist die Angabe der Leistungsfähigkeit zur Auswahl passender Komponenten für ein bestimmtes Einsatzgebiet von großer Bedeutung. Eine Reihe von nichtfunktionalen Eigenschaften wird in (O'Sullivan, 2008) vorgestellt. Diese beinhalten unter anderem die zeitliche und örtliche Verfügbarkeit, mögliche Bezahlmethoden, den Preis sowie Rechte und Pflichten der Vertragspartner. Nichtfunktionale Eigenschaften einer Komponente werden durch das Konzept der Attribute dargestellt.

Die Namen und möglichen Werte aller verwendeten Attribute werden in den Mengen *AttributeNames* und *AttributeValues* definiert. Um anzugeben, dass eine Komponente ein bestimmtes Attribut besitzt wird die Abbildung *AttributeMapping* verwendet. Zusätzlich wird die Abbildung *AttributeValue* definiert, um den Wert eines Attributs einer Komponente zu ermitteln.

$$AttributeMapping: Components \rightarrow P(AttributeNames \times AttributeValues)$$
$$AttributeValue: Components \times AttributeNames \rightarrow AttributeValues$$

Attribute sind spezifische Eigenschaften von Komponenten und unabhängig von der Auswahl anderer Komponenten. Sie haben bereits während der Modellierung feste Werte, die sich auch zur Konfiguration nicht ändern, wodurch Referenzierungen anderer Attribute nicht erlaubt sind.

2.4 Produktivitätskennzahlen

Wie auch in anderen Domänen wird bei Dienstleistungen die Produktivität durch Kennzahlen (Key Performance Identifier, KPI) erfasst. Ein besonderes Problem ist hierbei allerdings die große Komplexität monolithischer Dienstleistungen, die zu einer Vielzahl von Faktoren führt, die Auswirkungen auf die Produktivität haben (Grönroos & Ojasalo, 2004). Neben der reinen Anzahl ist auch die Zuordnung von Kennzahlen zu bestimmten Teilen einer komplexen Dienstleistung problematisch.

Die Bildung von Dienstleistungskomponenten schafft hierbei Abhilfe, da eine Komponente aufgrund ihres geringeren Funktionsumfangs im Vergleich zur Gesamtleistung nur eine geringere Anzahl notwendiger Kennzahlen benötigt. Die Kernfunktionalität einer Komponente lässt sich einfacher bestimmen, wodurch an diese Funktionalität angepasste Kennzahlen verwendet werden können. Durch Aggregation von Kennzahlen verschiedener Komponenten wird eine Aussage über die Produktivität der Gesamtleistung ermöglicht. Die Definition von Kennzahlen im Rahmen des Metamodells basiert auf der Beschreibung in (**Fehler! Verweisquelle konnte nicht gefunden werden.**).

Demnach lassen sich Kennzahlen auf zwei verschiedene Arten erzeugen. Einerseits können sie als fester Wert definiert, andererseits anhand einer Formel berechnet werden. Einzelne Kennzahlen werden Komponenten zugeordnet und bei der Konfiguration von Dienstleistungen aggregiert. Dabei sind verschiedene Arten der Nutzung möglich, die in Abbildung 1 dargestellt sind. Bei der *Descendent Propagation* wird der Wert einer Kennzahl einer Komponente direkt an die darüber liegenden Elternkomponenten weitergegeben und charakterisiert damit auch die jeweiligen Elternkomponenten. Die *Descendent Calculation* ermöglicht die Verwendung der Kennzahl einer Unterkomponente in einer Formel zur Berechnung der Kennzahl in einer Oberkomponente. Diese beiden Varianten sind geeignet, um Kennzahlen einer komplexen Komponente aus deren Unterkomponenten zu aggregieren. Bei der *Cross-Tree Calculation* werden Kennzahlen hingegen aus Komponenten berechnet, die nicht in hierarchischer Beziehung zueinander stehen. Dadurch lassen sich Auswirkungen nicht-hierarchischer Komponenten auf die Produktivität abbilden.

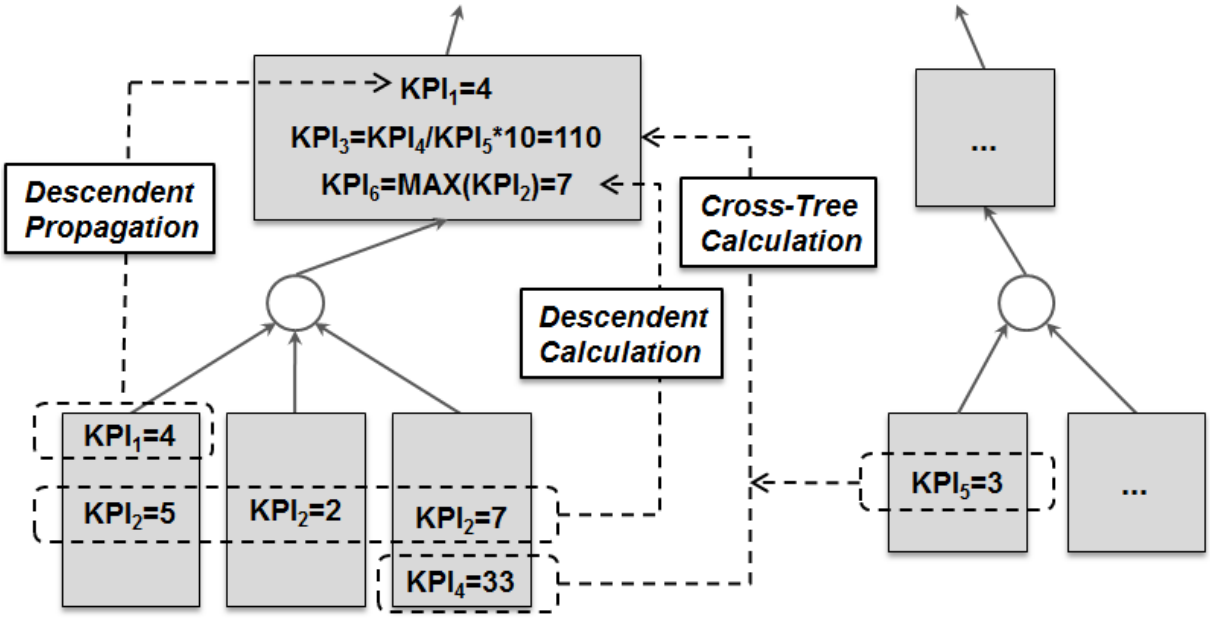

Abbildung 1 Nutzung von KPIs im Konfigurationsgraphen

Zur Repräsentation der Kennzahlen werden die Mengen $KPINames$ und $KPIValues$ verwendet, welche die Namen und Werte (bzw. die Formeln zur Berechnung des Wertes) der Kennzahlen enthalten. Durch die Abbildung $KPIMapping$ werden Komponenten Kennzahlen zugewiesen, die Abbildung *KPIValue* gibt den Wert einer KPI einer bestimmten Komponente zurück.

$$KPIMapping: Components \rightarrow P(KPINames \times KPIValues)$$
$$KPIValue: Components \times KPINames \rightarrow KPIValues$$

Die Analyse komplexer Dienstleistungen ist in der Regel abhängig von einer Reihe externer Einflussfaktoren. Diese sind bei der Modellierung von Dienstleistungen nicht bekannt und können daher erst während der Konfiguration in Zusammenarbeit mit dem Kunden festgelegt werden, da sie von dem geplanten Einsatzgebiet der Dienstleistung abhängig sind. Mittels Variablen ist es möglich, externe Einflussfaktoren abzubilden. Diese werden in Formeln zur Berechnung von KPIs verwendet und während der Konfiguration mit festen Werten versehen.

Die Nutzung von Variablen erhöht die Wiederverwendbarkeit von Komponenten, da KPIs nicht statisch definiert werden müssen sondern dies in Abhängigkeit kundenspezifischer Einflussfaktoren möglich ist. Die in einem Modell verwendeten Variablen werden analog zu KPIs formal in der Menge *VariableNames*, ihre möglichen Werte in der Menge *VariableValues* repräsentiert.

Der Wert einer Kennzahl wird durch eine Formel repräsentiert. Diese Formel kann analog zu den oben definierten Möglichkeiten der Kennzahlenbildung Konstanten sowie Referenzen auf Kennzahlen anderer Komponenten enthalten. Die Verwendung einer Variablen wird durch das Voranstellen eines @-Symbols angezeigt. Neben Variablen und KPIs anderer Komponenten lassen sich weiterhin Attribute (siehe Abschnitt 2.3) von Komponenten zur Berechnung einer KPI heranziehen. Eine gültige Formel $Formula$ zur Berechnung einer Kennzahl wird durch folgende EBNF spezifiziert.

$$
\begin{aligned}
&Formula \rightarrow Operand \mid Operand\ Operator\ Operand\\
&Operand \rightarrow Constant \mid Formula \mid KPIRef \mid AttributeRef \mid VariableRef\\
&Constant \rightarrow \mathbb{R}\\
&Operator \rightarrow + \mid - \mid \times \mid \div\\
&KPIRef \rightarrow (n, c), \qquad n \in KPINames \wedge c \in Components\\
&AttributeRef \rightarrow (a, c), \qquad a \in AttributeNames \wedge c \in Components\\
&VariableRef \rightarrow @v, \qquad v \in VariableNames
\end{aligned}
$$

Die Referenzierung von Kennzahlen und Attributen anderer Komponenten sowie den externen Variablen erfolgt mittels *KPIRef*, *AttributeRef* und *VariableRef*. Um valide Formeln zu erzeugen muss bei der Referenzierung von Kennzahlen und von Attributen weiterhin sichergestellt werden, dass diese in den angegebenen Komponenten existieren, da die Formel sonst nicht aufgelöst werden kann. Diese Bedingung lässt sich wie folgt formalisieren.

$$
\begin{aligned}
\forall (n, c) \in KPIRef: \exists \left(n_{orig}, v\right) \in KPIMapping(c): n = n_{orig}\\
\forall (a, c) \in AttributeRef: \exists \left(a_{orig}, v\right) \in AttributeMapping(c): n = n_{orig}
\end{aligned}
$$

2.5 Nichthierarchische Beziehungen zwischen Komponenten

Hierarchische Beziehungen zwischen Komponenten legen fest, aus welchen Teilprozessen sich eine Gesamtleistung zusammensetzt. Neben diesen Aspekten der Komposition sind bei der Spezifikation weitere Anforderungen an die Erbringung von Teilleistungen zu beachten. Um die durch verschiedene Komponenten repräsentierten Prozesse zu erfüllen, kann es z.B. notwendig sein, andere Teilprozesse auszuführen, die nicht in direkter hierarchischer Relation zueinander stehen. Diese lassen sich daher auch nicht mit Regeln definieren, die auf der hierarchischen Baumstruktur basieren.

Die Spezifikation nichthierarchischer Abhängigkeiten wird ermöglicht durch die Verwendung aussagenlogischer Formeln. Dieser Ansatz wird auch bei der Modellierung von Features im Software Product Line Engineering verfolgt (**Fehler! Verweisquelle konnte nicht gefunden werden.**). Neben der reinen Darstellung nichthierarchischer Abhängigkeiten werden diese Bedingungen auch dazu genutzt, anhand des Modells Schlussfolgerungen zu ziehen sowie die

Konfiguration zu automatisieren. Im Folgenden wird die Menge *Dependencies* genutzt, die alle nichthierarchischen Abhängigkeiten des Dienstleistungsportfolios enthält.

Die Nutzung aussagenlogischer Formeln ermöglicht einen sehr hohen Freiheitsgrad bei der Spezifikation von Abhängigkeiten zwischen Komponenten. Dem steht allerdings das Problem gegenüber, dass die Formulierung der Formeln sehr zeitaufwändig und arbeitsintensiv ist. Da die Definition von Komponenten auch und vor allem durch Fachabteilungen erfolgen soll und nicht durch IT-Abteilungen werden den mit mathematischer Logik nicht vertrauten Anwendern verschiedene Hilfestellungen bereitgestellt. Dazu wurden in (**Fehler! Verweisquelle konnte nicht gefunden werden.**) vier verschiedene, häufig gebrauchte Bedingungen als leicht anwendbare Regeln vorgestellt. Diese Regeln sind in Tabelle 2 noch einmal zusammenfassend dargestellt.

Tabelle 2 Regeln für nichthierarchische Beziehungen zwischen Komponenten

Regel	Aussagenlogische Darstellung	Beschreibung
a führt zu b	$a \rightarrow b$	Wenn die Komponente a gewählt wird, muss auch die Komponente b gewählt werden.
Nicht a führt zu nicht b	$\neg a \rightarrow \neg b$	Wenn die Komponente a nicht gewählt wird, darf auch die Komponente b nicht gewählt werden.
a und b schließen sich aus	$(a \rightarrow \neg b) \wedge (b \rightarrow \neg a)$	Wenn die Komponente a gewählt wird, darf Komponente b nicht gewählt werden und umgekehrt.
Nicht a führt zu b	$\neg a \rightarrow b$	Wenn die Komponente a nicht gewählt wird, muss die Komponente b gewählt werden.

Auch wenn es durch die Verwendung vordefinierter Regeln relativ einfach ist, Bedingungen über Hierarchiegrenzen hinweg zu definieren, sollte diese Möglichkeit nicht im Übermaß genutzt werden. Die Bedingungen verringern sowohl die Lesbarkeit als auch die Verständlichkeit von Dienstleistungsmodellen und erhöhen die Komplexität bei der Konfiguration. Für die Featuremodelle der SPL-Entwicklung haben (**Fehler! Verweisquelle konnte nicht gefunden werden.**) gezeigt, dass besonders das Bearbeiten existierender Modelle schwieriger wird. Da dies eines der Hauptanwendungsgebiete ist, sollte also die Anzahl nichthierarchischer Bedingungen begrenzt sein.

2.6 Temporale Beziehungen zwischen Komponenten

Komponenten sind dadurch charakterisiert, dass sie Prozesse repräsentieren und damit verschiedene Aktivitäten darstellen. Um die Erbringung von Dienstleistungen genauer zu spezifizieren ist es daher unerlässlich, genauer auf zeitliche Abhängigkeiten zwischen den einzelnen Prozessschritten einzugehen. Zu diesem Zweck werden Komponenten als Black-Box-Einheiten betrachtet, die in zeitliche Relation miteinander gestellt werden. Über diese

Einheiten lassen sich dann verschiedene Aussagen treffen, wie z.B. Komponente A muss vor Komponente B ausgeführt werden.

In Analogie zu nichthierarchischen Abhängigkeiten, die auf Konstrukte der Aussagenlogik abgebildet werden, lässt sich für temporale Beziehungen die Lineare Temporale Logik (LTL) nach (**Fehler! Verweisquelle konnte nicht gefunden werden.**) verwenden. Die Menge *TemporalDependencies* enthält die temporalen Abhängigkeiten, die im Modell verwendet werden.

Im Vergleich zur Aussagenlogik besitzt die LTL eine noch höhere Komplexität. Daher sind in Tabelle 3 eine Reihe oftmals genutzter Konstrukte in natürlicher Sprache dargestellt. Diese können direkt auf das Modell einer Dienstleistung angewendet werden und definieren damit zeitliche Anforderungen an dieses Modell.

Tabelle 3 Regeln für temporale Beziehungen zwischen Komponenten

Regel	LTL-Darstellung	Beschreibung
a benötigt nachfolgend b	$\Box(a \rightarrow \diamond\, b)$	Wenn die Komponente *a* ausgeführt wird, muss irgendwann später die Komponente *b* ausgeführt werden.
a benötigt sofort nachfolgend b	$\Box(a \rightarrow (\circ\, a \vee \circ\, b))$	Sofort nachdem Komponente *a* ausgeführt wurde muss Komponente *b* ausgeführt werden (oder nochmals Komponente *a*).
a benötigt vorhergehend b	$\neg(\neg b U a)$	Bevor Komponente *a* ausgeführt werden kann, muss Komponente *b* ausgeführt werden. Die Definition der temporalen Logik legt hierbei fest, dass die Komponente *a* solange nicht ausgeführt werden darf, wie *b* nicht ausgeführt wurde.
a benötigt sofort vorhergehend b	$\neg \diamond (\neg b \wedge \circ\, a)$	Direkt vor der Ausführung der Komponente *a* muss die Komponente *b* ausgeführt werden. Hier definiert als solange *b* nicht ausgeführt wurde, darf *a* nicht ausgeführt werden.
a darf niemals vor b ausgeführt werden	$\neg \diamond (a \wedge \diamond\, b)$	Zu keinem Zeitpunkt darf die Komponente *b* ausgeführt werden, wenn die Komponente *a* bereits ausgeführt wurde.
a darf nicht direkt vor b ausgeführt werden	$\neg \diamond (a \wedge \circ\, b)$	Zu keinem Zeitpunkt darf *b* direkt nach *a* ausgeführt werden.

Um die Validität der Dienstleistungsspezifikation zu erhalten, muss die Definition zeitlicher Abhängigkeiten einigen Anforderungen genügen. Durch den prozessualen Charakter der Komponenten dürfen zeitliche Abhängigkeiten nur zwischen Komponenten definiert werden, die nicht über einen gerichteten Pfad miteinander verbunden sind. Dies ist insofern notwendig, da ein Teilprozess (die Unterkomponente) immer innerhalt eines übergeordneten

Gesamtprozesses (die Oberkomponente) stattfindet und daher die Definition zeitlicher Restriktionen nicht möglich ist. Eine Vorgabe an die zeitliche Abfolge der einzelnen Prozessschritte lässt sich erreichen, indem das der Oberkomponente zugrunde liegende Prozessmodell angepasst wird. Im Beispiel der Textilreinigung von oben würde demnach die Vorwäsche stets vor dem Hauptwaschgang und dieser stets vor dem Schleudern durchgeführt werden.

$$\forall (a,b) \in TemporalDependencies: \nexists Pfad\ P: aPb \vee bPa$$

Neben dieser logischen Anforderung gibt es noch einige technische Anforderungen. Diese sind notwendig, um redundante bzw. unnötige Definitionen temporaler Abhängigkeiten zu vermeiden. So ist es nicht erlaubt, Komponenten, die sich gegenseitig ausschließen, in temporale Abhängigkeitsbeziehung zu setzen. Dies ist nicht nur im Falle der direkten nichthierarchischen Angabe des gegenseitigen Ausschlusses der Fall sondern auch, wenn zwei Komponenten Kinder eines XOR bzw. XOR*-Konnektors mit einer maximalen Kardinalität von 1 sind. Dadurch ergibt sich ein impliziter gegenseitiger Ausschluss.

$$\begin{gathered}\forall (a,b) \in TemporalDependencies: \\ \big((a \rightarrow \neg b) \wedge (b \rightarrow \neg a)\big) \notin Dependencies \wedge \\ \forall c \in prenodes(a) \cap prenodes(b): c \notin C_{XOR} \wedge c \notin C_{XOR^*}\end{gathered}$$

2.7 Constraints zur Auswahl von Komponenten

Basierend auf den nichtfunktionalen Eigenschaften (dargestellt als Attribute) sowie den KPIs eines Modells lassen sich kundenspezifische Einschränkungen an die Auswahl von Komponenten definieren. Diese werden im Weiteren als Constraints bezeichnet und dienen als eine Art Filter, um nur passende Komponenten auszuwählen und nicht passende Komponenten zu verbieten. Constraints lassen sich sowohl auf Attributen definieren und legen damit fest, welche Eigenschaften eine bestimmte Komponente besitzen muss um den Anforderungen der Kunden zu genügen. Sie lassen sich aber auch über Variablen definieren und definieren damit Randbedingungen, die für die gesamte Konfiguration gelten.

Um eine größtmögliche Freiheit bei der Definition von Constraints zu gewährleisten, werden diese in Prädikatenlogik formuliert. Mit Hilfe der oben definierten Abbildungen *KPIValue* und *AttributeValue* lässt sich einfach auf die Werte von KPIs bzw. Attribute zugreifen. Eine Beispielconstraint über die Mindestkapazität einer Komponente sowie den Gesamtpreis der Dienstleistung lässt sich wie folgt spezifizieren.

$$\begin{gathered}C_1 = \forall c \in Components: \\ AttributeValue(c, capacity) \geq 7000 \wedge KPIValue(c, price) \leq 1100\end{gathered}$$

Bei der Definition der Constraint C_1 wurde der Bezeichner *Components* genutzt. Dessen Verwendung zeigt an, dass eine Constraint über das gesamte Modell geprüft werden soll, d.h alle im Konfigurationsgraph vorhandenen Komponenten müssen die Constraint erfüllen. Es ist allerdings oftmals der Fall, dass bei großen und komplexen Portfolios mehrere Komponenten existieren, die Attribute mit gleichem Namen aber unterschiedlicher Bedeutung haben. Diese Problematik lässt sich umgehen, indem statt des Bezeichners *Components* der

Name der jeweiligen Komponente genutzt wird. Ist im Modell eine Komponente *CallCenter* vorhanden, kann die Constraint wie folgt angepasst werden.

$$C_2 = \forall c \in CallCenter:$$
$$AttributeValue(c, capacity) \geq 7000 \wedge KPIValue(c, price) \leq 1100$$

In diesem Fall wird die Constraint nur für Komponenten geprüft, die sich in der transitiven Hülle der angegebenen Komponente besitzen. Die transitive Hülle einer Komponente enthält alle direkt und indirekt nachfolgenden Knoten. Da der Konfigurationsgraph einen Baum bildet, lässt sich die Hülle mit bekannten Algorithmen berechnen, z.B. gezeigt in (**Fehler! Verweisquelle konnte nicht gefunden werden.**).

Neben der Angabe fester Werte lassen sich auch die oben angeführten Variablen in Constraints verwenden, wodurch die Wiederverwendbarkeit von Constraints erhöht wird, da sie nicht mehr für alle Kunden angepasst sondern nur noch die Werte der Variablen festgelegt werden müssen. Die Constraint C_2 lässt sich mit den Variablen *incoming*, die die Anzahl erwarteter Anrufe repräsentiert, und der Variablen *maxprice* zur Definition des Maximalpreises wie folgt darstellen.

$$C_3 = \forall c \in CallCenter:$$
$$AttributeValue(c, capacity) \geq @incoming \wedge KPIValue(c, price) \leq @maxprice$$

2.8 Klone

In Abschnitt 2.2 wurde bereits kurz darauf eingegangen, dass Konnektoren maximal einen Oberknoten besitzen dürfen, diese Restriktion aber nicht für Komponenten gilt. An dieser Stelle soll noch kurz die Begründung für diese Restriktion dargelegt werden. In Abbildung 2 findet sich die skizzenhafte Darstellung eines Portfolios mit Klonen – die genaue Bedeutung der unterschiedlichen Notationselemente erfolgt in Abschnitt 4.2.

Auf der linken Seite der Abbildung befindet sich ein Konfigurationsgraph, der einen Konnektor mit mehreren Oberelementen enthält. Die verschiedenen Kardinalitäten besagen, dass genau zwei der Komponenten der mittleren Ebene und genau eine Komponente aus der unteren Ebene gewählt werden müssen. D.h. die Komponente A kann in der gesamten Konfiguration maximal einmal gewählt werden.

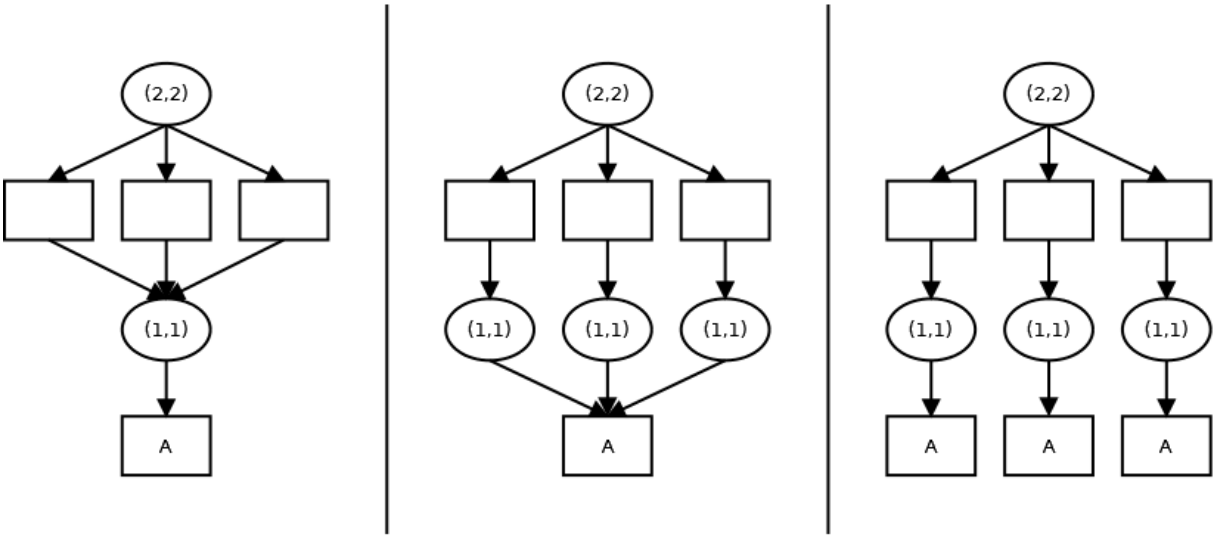

Abbildung 2 Klone im Modell

Der geklonte Konnektor ist im mittleren Teil von Abbildung 2 aufgeteilt dargestellt. Hierbei ergibt sich nun allerdings das Phänomen, dass Komponente A jetzt zweimal gewählt werden kann, da sie von beiden gewählten mittleren Komponenten aus wählbar ist. Dadurch lässt sich erkennen, dass Konnektoren nur einen Oberknoten besitzen dürfen, da das Modell anderenfalls inkonsistent werden kann.

Dem steht die Komponente A gegenüber, die als Klon unter den drei Konnektoren angebracht ist und im Modell die drei Konnektoren als Oberelemente besitzt. Zur Konfiguration wird die Komponente allerdings aufgeteilt und durch eine eigene, eindeutig bestimmte Komponente dargestellt. Hierbei treten keine Probleme mehr auf. Klone lassen sich also nutzen, um die gleiche Komponente an mehreren Stellen im Modell zu repräsentieren. Dadurch erhöht sich die Konsistenz innerhalb des Modells, da nicht mehr unterschiedliche Komponenten, die das gleiche bedeuten, gepflegt werden müssen, sondern dies an einer zentralen Stelle passieren kann.

2.9 Integration der Konzepte

Nachdem in den vorangegangen Abschnitten die einzelnen Bestandteile des Metamodells für komplexe Dienstleistungen vorgestellt wurden, wird in diesem Abschnitt gezeigt, wie sich die verschiedenen Konzepte miteinander integrieren lassen, um ein ganzheitliches und in sich abgeschlossenes Modell zu erzeugen. Dazu wird ein Dienstleistungsportfolio aufgefasst als Tupel $P = (C, I, G, D, T, AN, AV, KN, KV, VN)$ mit den Komponenten C, Identitäten I, dem Konfigurationsgraphen G, nichthierarchischen Abhängigkeiten D, temporalen Abhängigkeiten T, Attributnamen AN und –werten AV, Namen von KPIs KN und Werte von KPIs KV sowie Variablennamen VN. Da Constraints und die Werte von Variablen erst zur Konfiguration von Dienstleistungen relevant werden, sind diese nicht in der Spezifikation des Portfolios enthalten.

Basierend auf dem Portfolio lässt sich eine Konfiguration beschreiben als Abbildung *Con* bestehend aus einer Menge von Komponenten C, einer Zuweisung von Variablen VN zu ihren Werten VV und zusätzlichen Constraints Co. Mit der Abbildung *Con* wird festgelegt, ob die Konfiguration gültig ist (*true*) oder nicht (*false*).

$$Con: P(C) \times P(VN \times VV) \times P(Co) \rightarrow \{true, false\}$$

3 Konfiguration komplexer Dienstleistungen aus Komponenten

Bei der Modellierung von Dienstleistungen werden Komponenten sowie ihre Abhängigkeiten untereinander definiert. Das dadurch entstehende formale Modell einer Dienstleistung lässt sich auf zwei Arten nutzen. Aus der Sicht der Anbieter lassen sich vorgefertigte Angebote erstellen, die aus verschiedenen Komponenten bestehen. Durch den Formalitätsgrad des Modells sind diese Angebote in sich konsistent und valide. Aus Kundensicht ist es möglich, zusammen mit dem Anbieter kundenspezifische Konfigurationen der Komponenten eines Portfolios zu erstellen. Dadurch können Anbieter komplexer Dienstleistungen viel besser auf die Wünsche und Anforderungen ihrer Kunden reagieren. Die Konfiguration von Dienstleistungen ist eine Möglichkeit, auf den Gegensatz zwischen Standardisierung zur Verringerung von Kosten sowie Anpassung zur Erhöhung der Flexibilität zu reagieren. Die Produktivität einer Gesamtleistung lässt sich bei der Konfiguration durch die Aggregation der KPIs einzelner Komponenten analysieren.

Die Konfiguration von Dienstleistungen besteht aus drei einzelnen Schritten.

- Festlegen der Werte der Variablen, die im Modell verwendet werden.
- Auswerten der Constraints über Attributen, um Komponenten explizit zu verbieten.
- Auswahl von Komponenten anhand kundenspezifischer Anforderungen und Auswertung der Constraints über KPIs.

3.1 Festlegen der Variablenwerte

Variablen sind im gesamten Modell gleich, wodurch die Definition ihrer Werte im Rahmen der Konfiguration direkt durch die Abbildung *VariableValue* erfolgt.

$$VariableValue: VariableNames \rightarrow VariableValues$$

Nachdem die Werte aller Variablen festgelegt sind, werden KPIs, werden die Platzhalter zur Berechnung von KPIs aufgelöst und in die entsprechenden fixen Werte umgewandelt.

3.2 Auswerten der Constraints über Attributen

Um eine valide Konfiguration zu erhalten, ist es notwendig, die Constraints auszuwerten und diejenigen Komponenten, welche den Constraints nicht genügen gesondert zu behandeln. Hierbei lassen sich während der Konfiguration zwei Typen von Constraints unterscheiden. *Hard Constraints* müssen von allen Komponenten erfüllt werden, da die Erbringung der Dienstleistung sonst nicht möglich ist. Demgegenüber stehen *Soft Constraints*, deren Nichterfüllung ein potentielles Risiko währen der Erbringung einer Dienstleistung darstellt, die aber nicht zwangsweise erfüllt werden müssen, da sie z.B. durch geeignete Maßnahmen umgangen werden können. Der Typ einer Constraints lässt sich nicht im Voraus vor der eigentlichen Konfiguration bestimmen, da dies von den jeweiligen Anforderungen der Kunden abhängt, d.h. einige Kunden bewerten eine Constraint als Muss-Anforderung während andere diese als Soll-Anforderung ansehen.

Bei der Auswertung von Constraints muss darin unterschieden werden, ob diese sich auf Attribute oder auf KPIs beziehen. Während Constraints über Attributen einzelne Komponenten adressieren, betreffen KPI-Constraints die gesamte Konfiguration. Daher lassen sich vor der Auswahl spezifischer Komponenten nur Constraints über Attributen auswerten.

Demzufolge wird zuerst für alle referenzierten Komponenten geprüft, ob sie das entsprechende Attribut besitzen, worauf sich die eigentliche Auswertung der Constraint anhand der prädikatenlogischen Formel anschließt. Dazu wird die Abbildung *satisfiedAtt* genutzt, die 1 zurückgibt, wenn eine Komponente den Attributteil einer Constraint erfüllt ist und 0, wenn dies nicht der Fall ist:

$$satiesfiedAtt: Constraints \times Components \rightarrow \{0,1\}$$

Verletzt eine Komponente eine Hard Constraint, ist sie im weiteren Verlauf der Konfiguration gesperrt und kann nicht mehr ausgewählt werden. Zusätzlich müssen alle Unterkomponenten dieser Komponente gesperrt werden, damit die Konfiguration konsistent bleibt. Dazu ist es wiederum notwendig, die transitive Hülle der jeweiligen Komponente zu ermitteln. Um festzuhalten, welche Komponenten gesperrt sind, wird die Abbildung *Excluded* verwendet. Diese ist anfangs für alle Komponenten auf 0 gesetzt (d.h. die Komponente kann ausgewählt werden). Komponenten, die Hard Constraints verletzen, wird der Wert 1 zugewiesen.

$$Excluded: Components \rightarrow \{0,1\}$$
$$\forall c \in Components: Excluded(c) = 0$$

$$\forall c \in Components, h \in HardConstraints:$$
$$satisfiedAtt(h, c) = 0 \rightarrow (Excluded(c) = 1)$$

Im Gegensatz zu Hard Constraints können Komponenten, die Soft Constraints verletzen, noch ausgewählt werden. Sie sind allerdings mit einer Warnung versehen, so dass schon während der Konfiguration eventuelle Risiken bekannt sind. An dieser Stelle ist es nicht notwendig, Komponenten in der transitiven Hülle ebenfalls mit einer Warnung zu versehen. Komponenten mit Warnung erhalten von der Abbildung *Warned* den Wert 1 zugewiesen.

$$Warned: Components \rightarrow \{0,1\}$$
$$\forall c \in Components: Warned(c) = 0$$

$$\forall c \in Components, s \in SoftConstraints:$$
$$satisfiedAtt(s, c) = 0 \rightarrow Warned(c) = 1$$

3.3 Auswahl der Komponenten und Auswertung von Constraints über KPIs

Nachdem die Menge der zur Auswahl stehenden Komponenten durch die Auswertung der Constraints über Attributen eingeschränkt wurde, ist es möglich aus den restlichen Komponenten eine kundenspezifische Konfiguration zu erstellen. Dabei werden Komponenten ausgewählt und basierend auf der Struktur des Modells zu einer validen Gesamtleistung zusammengefügt.

Für jede Komponente wird mit der Abbildung *NodeChosen* festgehalten, ob sie im Rahmen der Konfiguration gewählt wurde oder nicht. Alle gewählten Komponenten werden in der Menge *Configuration* gespeichert.

$$NodeChosen: Components \rightarrow \{0,1\}$$
$$Configuration = \{a | \forall a: a \in Components \wedge NodeChosen(a) = 1\}$$

Um invalide Konfigurationen zu verbieten werden im Folgenden noch eine Reihe von Anforderungen an die Menge $Configuration$ definiert. Dazu wird zunächst festgelegt, dass Komponenten, die eine Hard Constraint verletzen, nicht in der Konfiguration vorhanden sein dürfen.

$$\forall c \in Components: Excluded(c) = 1 \rightarrow NodeChosen(c) = 0$$

Weiterhin sind alle nachfolgenden Knoten eines nicht gewählten Knotens ebenfalls nicht gewählt. Dadurch wird verhindert, dass Unterkomponenten ausgewählt werden, ohne die jeweilige Oberkomponente zu wählen.

$$\forall n \in V, \forall n' \in postnodes(n):$$
$$NodeChosen(n) = 0 \rightarrow NodeChosen(n') = 0$$

Auf der anderen Seite werden alle Konnektoren, die Kinder einer Komponente sind, gewählt, wenn die Komponente gewählt wird. Dadurch wird sichergestellt, dass auch notwendige nachfolgende Unterkomponenten ausgewählt (um die Kardinalitäten der Konnektoren zu erfüllen) und somit keine Unterkomponenten vergessen werden. An dieser Stelle kann die Funktion *postnodes* ohne weitere Einschränkungen verwendet werden, da aufgrund der Definition des Graphen Nachfolger von Komponenten immer Konnektoren sind.

$$\forall c \in Components, \forall (n) \in postnodes(c):$$
$$NodeChosen(c) = 1 \rightarrow NodeChosen(n) = 1$$

Auf der Grundlage dieser Bedingungen lässt sich nun die Semantik von Konnektoren während der Konfiguration definieren. Dazu wird die Menge $ChosenPostNodes$ verwendet, die alle nachfolgenden gewählten Knoten eines Konnektors enthält. Die Anzahl gewählter nachfolgender Knoten muss dann eine der Kardinalitäten des Konnektors erfüllen.

$$\forall (c) \in Connectors: ChosenPostNodes \subseteq postnodes(c)$$
$$\forall (c) \in Connectors, \forall (n) \in ChosenPostNodes(c): NodeChosen(n) = 1$$
$$\forall (c) \in Connectors: \exists (m, n) \in Cardinalities(c): m \leq |ChosenPostNodes| \leq n$$

Gleichzeitig mit der Auswahl der Komponenten werden die Constraints über KPIs ausgewertet und eventuelle Verletzungen dieser Constraints angezeigt. Im Gegensatz zu Constraints über Attributen werden Constraints über KPIs an der obersten Komponente ausgewertet, in der die entsprechende KPI verwendet wird. Dies erfolgt auf der Grundlage einer Breitensuche im Konfigurationsbaum, wobei die KPIs anhand ihres Namens verglichen werden. Da KPIs in der Regel eine Kombination anderer KPIs darstellen, lässt sich nicht eindeutig bestimmen, welche Komponente eine Constraint verletzt, wodurch es nur möglich ist, anzugeben, dass die gesamte Konfiguration eine Constraint verletzt. Zur Auswertung des

KPI-Teils einer Constraint wird die Abbildung *satisfiedKPI* verwendet, die diese Auswertung anhand der gesamten Konfiguration vornimmt.

$$satisfiedKPI: Constraints \times Configuration \rightarrow \{0,1\}$$

4 Spezifikation und Konfiguration eines Beispielportfolios

Um die Spezifikation eines Portfolios bestehend aus Dienstleistungskomponenten nachvollziehbar zu beschreiben, wird an dieser Stelle ein Beispielportfolio eines Webshop-Entwicklers vorgestellt und dessen Bestandteile formal definiert. Das Portfolio besteht aus den beiden Hauptkomponenten Entwicklung und Wartung. Der Dienstleiter bietet bei der Entwicklung an, diese Inhouse vorzunehmen oder aber die Entwicklung auszulagern und das Management der Auslagerung zu übernehmen. Bei der Inhouse-Entwicklung wird zunächst ein Entwicklungsplan erstellt, woraufhin entweder das Gesamtsystem oder ein erster Prototyp entwickelt wird. Dieses System wird schließlich an den Kunden ausgeliefert. Die Wartung umfasst mehrere optionale Komponenten. Dies sind Search Engine Optimisation, Search Engine Analysis, Lokalisierung und iterative Weiterentwicklung des Webshops. Bei der Lokalisierung ist zu beachten, dass diese nur auf der Grundlage eines vollständig entwickelten Systems vorgenommen werden kann, die iterative Weiterentwicklung hingegen wird nur für Prototypen angeboten.

Auf der Grundlage der Beschreibung lassen sich Prozessmodelle erstellen, die auf abstrakter Ebene die einzelnen durchzuführenden Aktivitäten zur Erbringung der Dienstleistung kapseln. Dies ist exemplarisch in Abbildung 3 für die Inhouse-Entwicklung in Form einer ereignisgesteuerten Prozesskette (EPK) dargestellt.

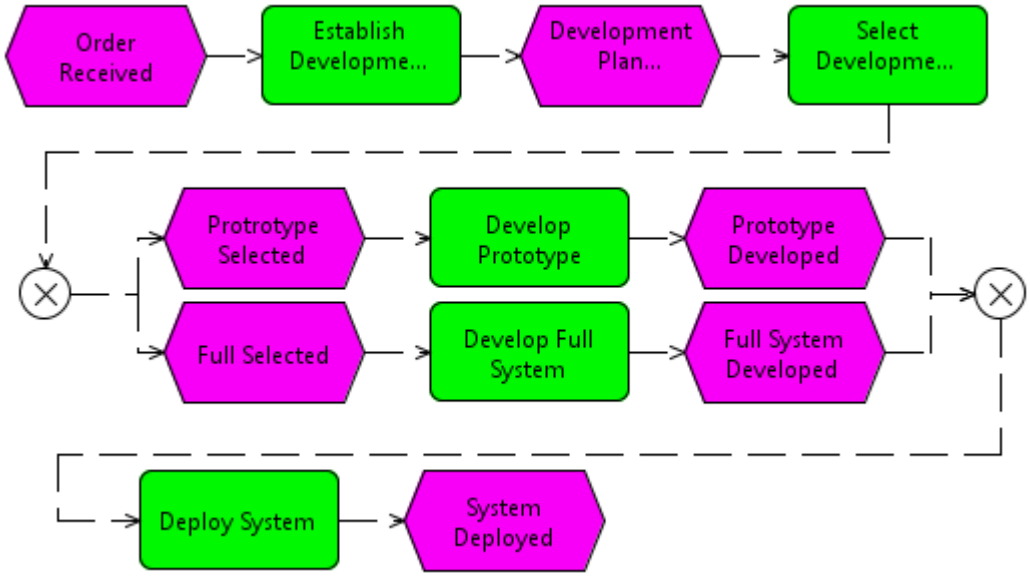

Abbildung 3 Vollständiger Entwicklungsprozess

4.1 Formalisierte Darstellung des Portfolios

Anhand der Beschreibung sowie unter Zuhilfenahme des darauf aufbauenden Prozessmodells lassen sich einzelne Dienstleistungskomponenten sowie deren Abhängigkeiten untereinander

identifizieren. Die Komponenten werden in der Menge $Components$ wie folgt spezifiziert und mit einem eindeutigen Identifikator versehen.

- $Components = \{S = Webshop\ Development, S_{1-1} = Inhouse\ Development, S_{1-2} = Outsourced\ Development, S_{11-1} = Establish\ Development\ Plan, S_{111-1} = Develop\ Prototype, S_{111-2} = Develop\ Full\ System, S_{11-2} = Deploy\ System, S_{2-1} = SEO, S_{2-2} = SEA, S_{2-3} = Localisation, S_{2-4} = Further\ Development\}$

Zur Modellierung der Abhängigkeiten zwischen Komponenten ist nach der Definition des Metamodells die Nutzung von Komponenten notwendig. Die Komponenten lassen sich daher durch die Verwendung der Konnektoren aus der Menge $Connectors$ in hierarchische Beziehung zueinander setzen. Dazu werden Komponenten und Konnektoren in der Graphstruktur miteinander verbunden.

- $Connectors = \{C_1 = Development, C_2 = Maintenance, C_{11} = Inhouse\ Development, C_{111} = System\ Development\}$

- $G = (V, E)$

  $V = Components \cup Connectors$

  $E = \{(S, C_1), (S, C_2), (C_1, S_{11}), (C_1, S_{1-2}), (S_{1-1}, C_{11}), (C_{11}, S_{11-1}), (C_{11}, S_{11-2}), (C_{11}, C_{111}), (C_{111}, S_{111-1}), (C_{111}, S_{111-2}), (C_2, S_{2-1}), (C_2, S_{2-2}), (C_2, S_{2-3}), (C_2, S_{2-4})\}$

Neben der Verbindung der einzelnen Komponenten untereinander lassen sich anhand der Beschreibung auch zwei Constraints an das Portfolio identifizieren. Die Lokalisierung wird demnach nur angeboten, wenn ein vollständiges System entwickelt wurde. Die iterative Weiterentwicklung hingegen wird nur für Prototypen angeboten. Dies bedeutet, dass bei der Wahl der entsprechenden Wartungskomponente die jeweilige Entwicklungskomponente gewählt werden muss. Die Constraints werden in der Menge $Constraints$ zusammengefasst.

- $Constraints = \{(S_{2-3} \rightarrow S_{111-2}), (S_{2-4} \rightarrow S_{111-1})\}$

Abschließend werden die Kardinalitäten der Konnektoren definiert. Diese lassen sich auch anhand der Beschreibung des Portfolios ablesen. Die Entwicklung passiert entweder Inhouse oder ausgelagert, so dass nur eine der beiden Komponenten gewählt werden darf. Bei der Inhouse-Entwicklung müssen alle Schritte durchgeführt werden, wobei die Systementwicklung entweder in einem Prototyp oder im vollständigen System endet. Die Wartung wiederum ist vollständig optional. Dadurch ergeben sich folgende Kardinalitäten, die mit der Abbildung $Cardinalities$ dargestellt werden.

- $Cardinalities(C_1) = \{(1,1)\}$

- $Cardinalities(C_{11}) = \{(3,3)\}$

- $Cardinalities(C_{111}) = \{(1,1)\}$

- $Cardinalities(C_2) = \{(0,4)\}$

An dieser Stelle soll der Prozess zur Bildung von Komponenten sowie deren Verbindung untereinander noch einmal etwas genauer betrachtet werden. Im Beispiel wurde die exklusive Auswahl zwischen der Entwicklung des vollständigen Systems und des Prototypen in Form eines Konnektors modelliert, der festlegt, dass nur eine der beiden Optionen gewählt werden darf. Im Gegensatz dazu ist es allerdings auch möglich, die beiden Komponenten als direkte Kinder des Inhouse-Entwicklungs-Konnektors zu modellieren. In diesem Fall müsste eine nichthierarchische Abhängigkeit zwischen beiden Komponenten erstellt werden, die deren gleichzeitige Auswahl verbietet.

Das Beispiel soll verdeutlichen, dass es in der Regel mehrere Möglichkeiten gibt, einen bestimmten Sachverhalt zu modellieren. Es lässt sich kein Patentrezept angeben, welche Variante zu bevorzugen ist, da dies vom Verwendungszweck des Modells sowie der Integration von Kennzahlen abhängig ist. So kann die oben angedeutete Modellierung mittels Constraints sinnvoll sein, wenn dadurch die Verwaltung von Kennzahlen vereinfacht wird. Das entwickelte Modell muss vielmehr den spezifischen unternehmensinternen Anforderungen gerecht werden.

4.2 Graphische Darstellung

Nach der formalen Darstellung des Portfolios soll dieses zur Verdeutlichung noch graphisch illustriert werden. Der entstehende Konfigurationsgraph besteht aus den beiden verschiedenen Knotentypen Komponente (dargestellt als Rechtecke) sowie Konnektor (dargestellt als Ellipsen). Hierarchische Abhängigkeiten zwischen Knoten werden mittels gerichteter, durchgehend gezeichneter Pfeile dargestellt. Nichthierarchische Abhängigkeiten werden als gerichtete, gepunktete Pfeile zwischen den involvierten Komponenten dargestellt. Der aus der Spezifikation des Beispielportfolios entstandene Konfigurationsgraph ist in Abbildung 4 dargestellt.

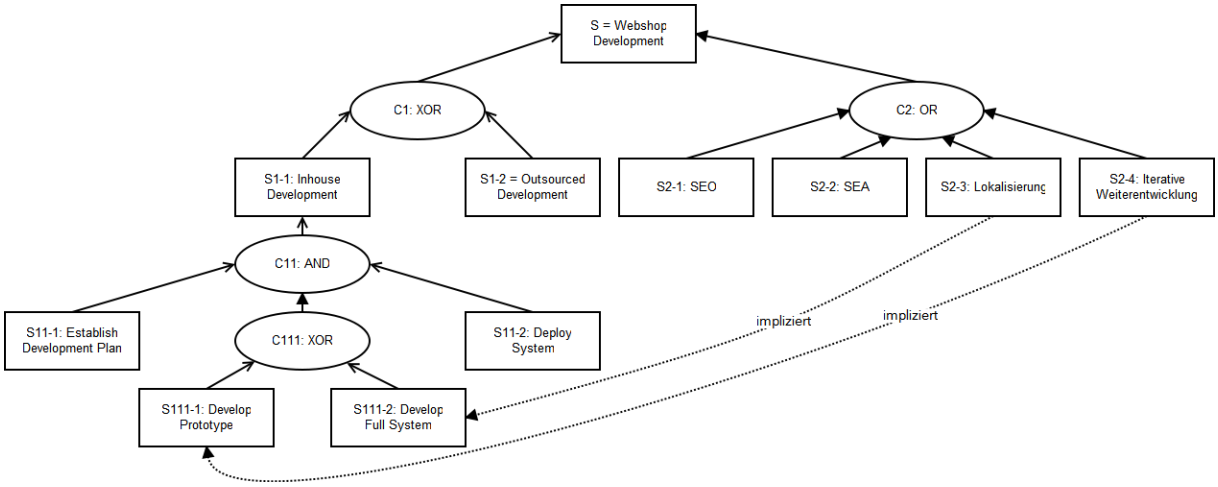

Abbildung 4 Graphische Darstellung des Webshop-Portfolios

4.3 Integration temporaler Abhängigkeiten

Neben den logischen Beziehungen zwischen Komponenten ist es wie in Abschnitt 2.6 beschrieben auch möglich, Bedingungen an den zeitlichen Verlauf der Dienstleistungen zu

formulieren. Dazu ist eine genauere Spezifikation der einzelnen Dienstleistungskomponenten (bzw. der dahinter liegenden Prozesse) notwendig, da nur damit Aktivitäten, die zur Erbringung einer Komponente notwendig sind, identifiziert werden können. Diese sind in Abbildung 5(für den Bereich der Entwicklung) sowie Abbildung 6 (für den Bereich der optionalen Module *Lokalisierung* und *Iterative Weiterentwicklung*) in Form ereignisgesteuerter Prozessketten dargestellt.

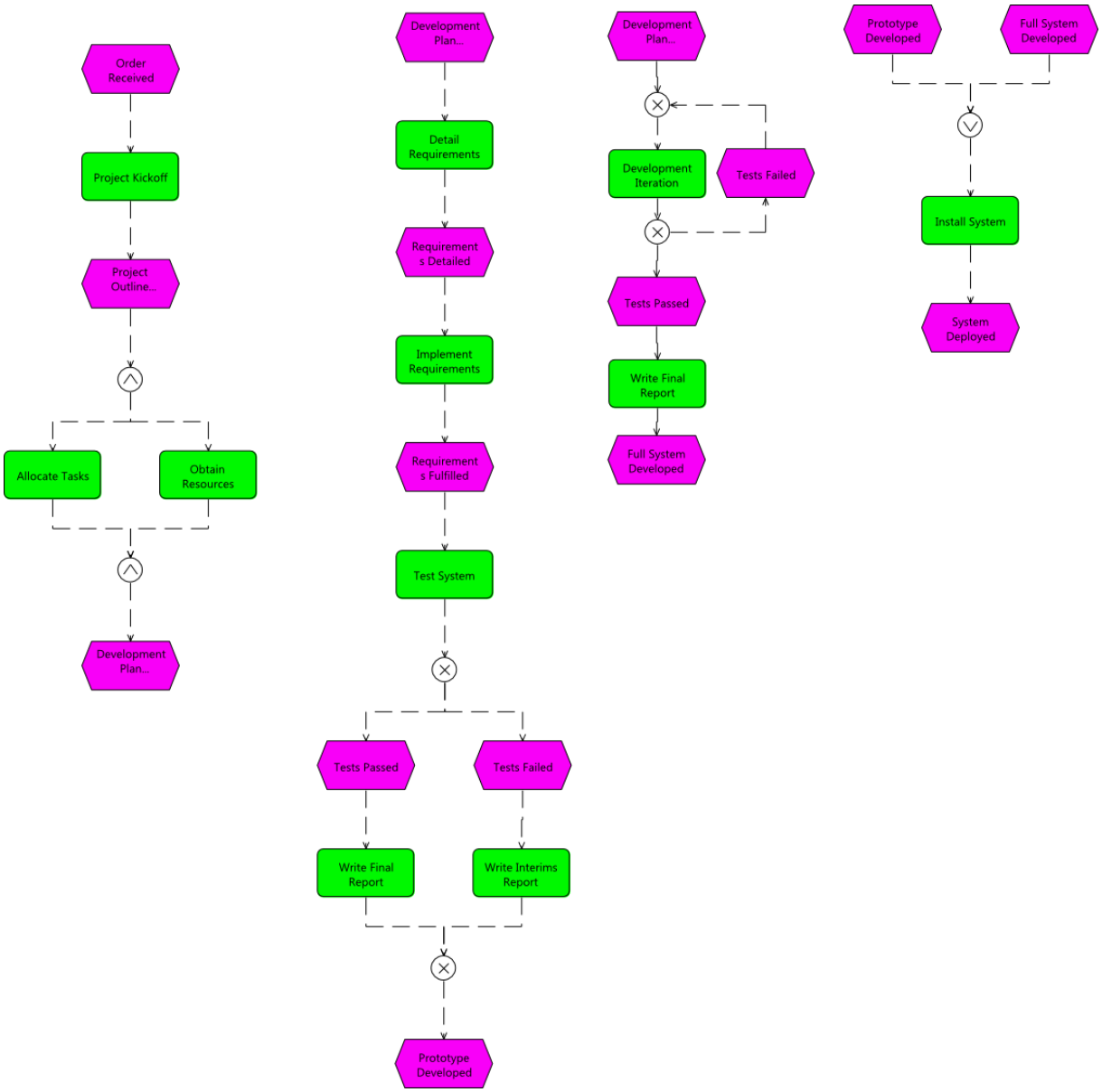

Abbildung 5 Prozessmodelle für die Entwicklung des Webshops

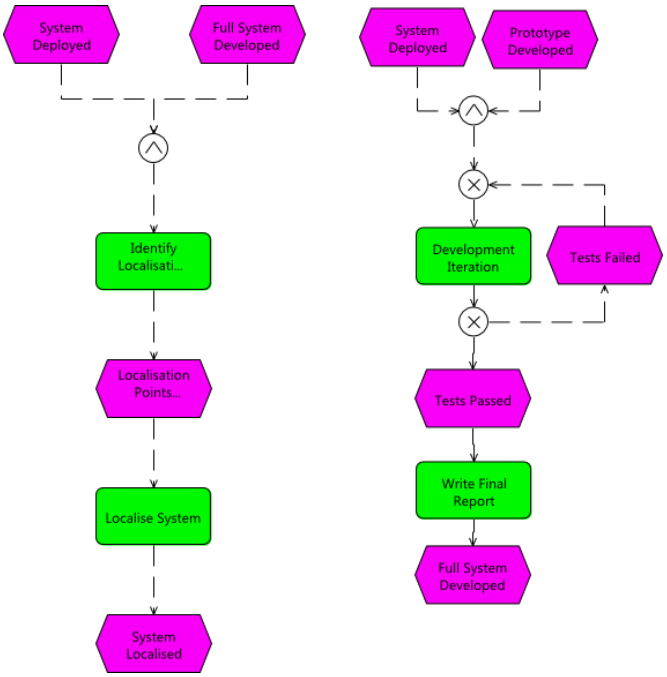

Abbildung 6 Prozessmodelle für die optionalen Module des Webshops

Anhand dieser Prozesse lassen sich neben den bereits identifizierten logischen auch temporale Abhängigkeiten ablesen. So lässt sich feststellen, dass die Komponenten *Develop Prototype* und *Develop Full System* die Komponente *Establish Development Plan* voraussetzen, da das auslösende Ereignis beider Komponenten *Development Plan Established* ist. Die gleiche Logik ergibt sich für die Komponente *Deploy System*. Diese setzt eine der beiden Entwicklungsmethoden voraus, da sie auf die Ereignisse *Prototype Developed* und *Full System Developed* wartet. Analog dazu ergibt sich neben der logischen Verknüpfung zwischen den optionalen Komponenten *Lokalisierung* und *Iterative Weiterentwicklung* zu den jeweiligen Entwicklungskomponenten auch eine temporale Abhängigkeit, da die beiden optionalen Komponenten ein ausgeliefertes System (entweder als Prototyp oder vollständig) benötigen. Die temporalen Abhängigkeiten werden wie folgt spezifiziert.

- Solange kein Entwicklungsplan aufgestellt wurde, darf weder der Prototyp noch das Gesamtsystem entwickelt werden
$$\neg(\neg S_{11-1} US_{111-1}), \neg(\neg S_{11-1} US_{111-2})$$
- Erst wenn das System oder ein Prototyp entwickelt wurden, darf die Anwendung ausgeliefert werden
$$\neg(\neg(S_{111-1} \vee S_{111-2}) US_{11-2})$$
- Die Lokalisierung bzw. Iterative Weiterentwicklung darf erst dann beginnen, wenn das Komplettsystem bzw. der Prototyp entwickelt wurde
$$\neg(\neg S_{111-1} US_{2-4}), \neg(\neg(S_{111-2} US_{2-3})$$

5 Implementierung als UML-Modell

Aufbauend auf den in den vorigen Abschnitten vorgestellten formalen Konzepten lässt sich ein UML-Modell zur Beschreibung komponentenbasierter Dienstleistungen entwickelt. Dadurch wird die Implementierung des Metamodells in einem Softwarewerkzeug erleichtert.

Das Modell soll weiterhin den Vergleich des vorgestellten Metamodells mit existierenden Modellen für Dienstleistungen ermöglichen. Die grundlegenden Bestandteile eines UML-Modells sind Klassen zur Klassifikation von Elementen, Assoziationen, mit denen verschiedene Klassen miteinander verknüpft werden sowie Bedingungen an die Struktur von Modellen (**Fehler! Verweisquelle konnte nicht gefunden werden.**).

5.1 Klassen des Modells

Im Folgenden werden die Bestandteile des in Abbildung 7 dargestellten UML-Modells zur Modellierung von Dienstleistungskomponenten vorgestellt. Die Klasse *Portfolio* bildet dabei den Ausgangspunkt. Jedes Portfolio besteht aus einer Reihe verschiedener, miteinander verknüpfbarer Elemente, dargestellt durch die Klasse *ConnectableElement*. Die Klasse dient der Kapselung der beiden spezifischen Notationselemente *Component* und *Connector*.

Zur Verbindung verschiedener Elemente miteinander und damit zur Etablierung der hierarchischen Abhängigkeiten besitzt jedes ConnectableElement die Attribute *prenode* (zur Repräsentation der Vorgängerknoten) und *postnodes* (zur Repräsentation der Nachfolgerknoten). Jedes Element besitzt dabei maximal einen Vorgänger und beliebig viele Nachfolger. Aus technischen Gründen erhalten alle Elemente weiterhin einen modellweit eindeutigen Identifikator.

Komponenten sind durch eine beliebige Anzahl von Attributen und Produktivitätskennzahlen gekennzeichnet. Während die Attribute (dargestellt durch die Klasse *Attribute*) einen Namen und Wert besitzen, sind Kennzahlen (Klasse *KPI*) durch ihren Namen und eine Formel gekennzeichnet. Bei der Definition der Formel kann außerdem auf externe Variablen zugegriffen werden, die sich mit der Klasse *Variable* modellieren lassen.

Konnektoren besitzen mindestens eine Kardinalität, um die Verbindung der Elemente zu beschreiben. Kardinalitäten selber werden durch die Klasse *Cardinality* dargestellt und besitzen einen minimalen und einen maximalen Wert. Um die Arbeit mit Konnektoren zu vereinfachen, sind im UML-Modell die spezifischen Konnektoren *AND*, *OR*, *XOR*, *OR+*, *XOR** bereits vordefiniert und lassen sich verwenden.

Die Modellierung von Dienstleistungsportfolios wird abgeschlossen mit der Definition nichthierarchischer Abhängigkeiten. Diese sind durch einen Namen sowie ihre Formel gekennzeichnet und werden durch die Klasse *Dependency* modelliert. Um zwischen logischen und temporalen Abhängigkeiten zu unterscheiden, stellt das UML-Modell die Klassen *LogicalDependency* und *TemporalDependency* bereit.

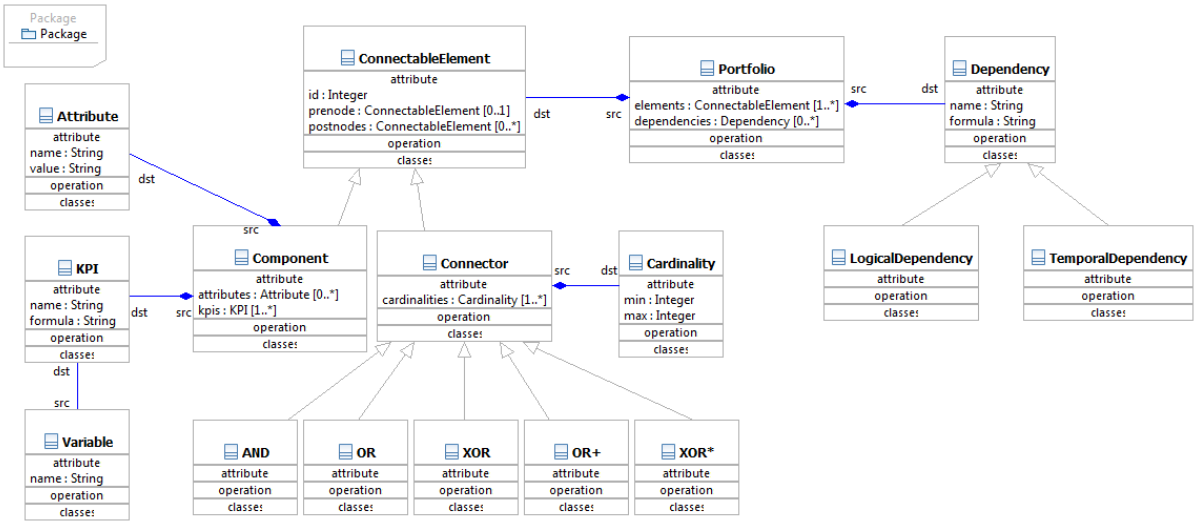

Abbildung 7 UML-Modell für Dienstleistungskomponenten

5.2 Strukturelle Bedingungen

Einige der Bedingungen an das Komponentenmodell ergeben sich implizit aus der Definition des Metamodells. So findet sich die Anforderung, dass Elemente maximal ein Elternelement besitzen dürfen, durch die Definition des Attributs *prenode* im Stereotypen *ConnectableElement* wieder. Dem stehen Bedingungen gegenüber, die sich nicht einfach aus dem Modell ablesen lassen. Durch die Verwendung des Stereotypen *ConnectableElement* ist es bspw. nicht ohne weitere möglich, Komponenten direkt ohne notwendige Konnektoren miteinander zu verbinden.

Um das UML-Modell genauer zu definieren sowie die Erstellung invalider Portfolios zu verhindern, müssen die oben angegebenen Bedingungen an den Komponentenbaum in eine UML-Form gebracht werden. Dazu eignet sich die Object Constraint Language (OCL). Diese ermöglicht die Spezifikation von Invarianten sowie Pre- und Postbedingungen für Elemente eines UML-Modells (**Fehler! Verweisquelle konnte nicht gefunden werden.**). Die Abbildung der oben definierten Bedingungen als OCL Constraints ist in Tabelle 4 dargestellt. Zu beachten ist, dass die Constraints dort aus Lesbarkeitsgründen getrennt aufgeführt sind. Im realen Modell sind die beiden am Element *ConnectableElement* sowie die drei am Element *Connector* definierten Constraints an einem Stück spezifiziert.

Tabelle 4 OCL-Constraints für das UML-Modell der Dienstleistungskomponenten

Bedingung	Constraint in OCL
Zyklenfreiheit	context ConnectableElement inv CycleConstraint: (not (self.prenode = self)) and (not self.postnodes->closure(postnodes)->includes(self))
Komponenten dürfen nicht direkt miteinander verbunden werden	context ConnectableElement inv ComponentConnectionConstraint: if self.oclIsTypeOf(Component) then self.postnodes->forAll(

	node: ConnectableElement |
	node.oclIsTypeOf(Connector)
	)
	else
	true
	endif

Maximum der Kardinalitäten darf nicht größer als die Anzahl nachfolgender Knoten sein	context Connector inv MaximumCardinalitiyConstraint: self.cardinalities->forAll(c: Cardinality | c.upperBound <= self.postnodes->size())

Kardinalitäten dürfen keine Überlappungen enthalten	context Connector
	inv CardinalityOverlapConstraint:
	self.cardinalities->forAll(
	c1: Dependencies::Cardinality |
	self.cardinalities->forAll(
	c2: Dependencies::Cardinality |
	if c1 <> c2 then
	not ((c1.lowerBound <= c2.lowerBound and
	c2.lowerBound <= c1.upperBound) or
	(c1.lowerBound <= c2.upperBound and
	c2.upperBound <= c1.upperBound))
	else
	true
	endif
	)
	)

Standardwerte müssen gültig für die definierten Kardinalitäten sein, Minimalwert einer Kardinalität darf nicht größer als deren Maximalwert sein	context Connector inv DefaultCardinalityConstraint self.cardinalities->forAll(c: Cardinality | c.lowerBound <= self.defaultCardinality and c.upperBound >= self.defaultCardinality)

6 Fazit und Ausblick

In dieser Arbeit wurde ein formalisiertes Modell für die Modellierung von Dienstleistungen aus Komponenten vorgestellt. Komponenten werden dabei als einzelne Teilprozesse einer Gesamtleistung betrachtet und lassen sich unter Einhaltung bestimmter Regeln beliebig miteinander kombinieren. Dies ermöglicht den Aufbau komplexer Portfolios, in denen einzelne Teilleistungen sowie Abhängigkeiten zwischen den Leistungen enthalten sind. Trotz der bisherigen Möglichkeiten sind noch einige wichtige Analysen durchzuführen, um das Potential von Komponenten in der Dienstleistungsdomäne vollständig zu entfalten.

Um die Arbeit mit komplexen Portfolios weiter zu vereinfachen ist neben deren Modellierung auch die werkzeuggestützte Konfiguration zu betrachten. Es ist zwar möglich, die Konfiguration anhand einer gegebenen Menge von Komponenten sowie Constraints zwischen

diesen manuell durchzuführen; allerdings ist dies sehr arbeitsaufwändig und vor allem fehleranfällig. Daher ist es notwendig, Methoden und Visualisierungen zu entwickeln, mit denen die Konfiguration unterstützt werden kann. Dabei sind mehrere Ansätze denkbar. So ist es einerseits möglichen, einen Top-Down-Ansatz zu verfolgen. Dieser basiert darauf, dass zuerst die groben Oberkomponenten und dann davon ausgehend die spezifischeren Komponenten gewählt werden. Das direkte Gegenstück dazu stellen Bottom-Up-Ansätze dar, bei denen eine komplexe Leistung aus kleinen, feingranularen Teilleistungen konstruiert wird. Eine gänzlich andere Herangehensweise wird verwendet die interaktive, dialogbasierte Konfiguration (Tiihonen, et al., 2003). Vor allem im Product Engineering (und dort bei der Konfiguration von Autos) wird diese Art der Konfiguration bereits heutzutage eingesetzt; eine Übertragung auf die Dienstleistungsdomäne scheint daher realistisch.

Zur weiteren Formalisierung der Konfiguration gehört auch die Analyse der Auswirkungen der Kombination verschiedener Komponenten auf die Gesamtproduktivität des Systems. So können zwei Komponenten bspw. individuell betrachtet eine sehr hohe Produktivität besitzen; in Kombination kann sich allerdings die Produktivität der Gesamtleistung verringern. Dies kann u.a. dadurch ausgelöst werden, dass die Aktivitäten beider Komponenten von denselben Personen ausgeführt werden müssen und sich daher nur sequentiell abarbeiten lassen. Eine andere Komponente kann hingegen von einer anderen Person und damit parallel ausgeführt werden, wodurch sich die Produktivität des Gesamtsystems erhöht (jedenfalls aus einer zeitlichen Perspektive betrachtet). Diese Abhängigkeiten lassen sich momentan noch nicht modellieren. Dies setzt u.a. die formale Definition der für die Erbringung einer Komponente notwendigen Ressourcen voraus. In (Böttcher, 2009) wurden erste Ansätze zur Definition der Ressourcen entwickelt. Neben den ressourcenbedingten Auswirkungen auf die Produktivität sind auch andere Einflussfaktoren zu beachten, die gegebenenfalls explizit im Modell gekennzeichnet werden müssen. Dazu sind Kennzahlen zu entwickeln, die die Auswirkungen der Kombination verschiedener Komponenten repräsentieren. Diese betriebswirtschaftliche Herangehensweise muss zudem durch das Metamodell unterstützt werden, indem die Konzepte der oben angesprochenen *Cross-Tree Calculation* genauer untersucht werden.

Durch den zunehmenden Druck zur Internationalisierung von Dienstleistungen, festgestellt z.B. in (Husen, et al., 2008), müssen Unternehmen ihr Portfolio auch auf fremden Märkten anbieten. Dabei ist zu beachten, dass die gesetzlichen Regelungen teilweise sehr unterschiedlich sind, wodurch die Erbringung von Dienstleistungen i. A. angepasst werden muss. Dadurch sind Anbieter gezwungen, nicht nur dienstleistungsinterne Abhängigkeiten zu formulieren sondern auch je nach Region z.B. zusätzliche Aktivitäten auszuführen bzw. sich auf spezielle Gegebenheiten einzustellen. Ein Beispiel dafür bietet das Cloud Computing, welches in Deutschland anderen gesetzlichen Regelungen im Bereich der Datenhaltung unterliegt als in den USA. Um ein Dienstleistungsportfolio am globalen Markt anzubieten ist es daher notwendig, diese Anforderungen in die Spezifikation mit einfließen zu lassen.

Für die reale Anwendung von Komponenten in der Dienstleistungsdomäne ist es schließlich unerlässlich, geeignete Methoden zu entwickeln, mit denen sich existierende oder neu zu entwickelnde Dienstleistungen in Komponenten schneiden lassen. Dabei ist einerseits zu beachten, dass die Komponenten feingranular genug sind, anderseits führt eine zu feine Strukturierung wiederum zum Anstieg der Komplexität. Ähnlich wie auch im Bereich des

Software Engineering sind daher Best Practices und Vorgehensmodelle für die komponentenbasierte Entwicklung von Dienstleistungen notwendig.

7 Literaturverzeichnis

Akkermans, H. et al., 2004. Value Webs: Using Ontologies to Bundle Real-World Services. *IEEE Intelligent Systems,* Band 19, pp. 57-66.

Böttcher, M., 2009. *Architektur integrierter Dienstleistungssysteme,* s.l.: s.n.

Böttcher, M. & Fähnrich, K.-P., 2009. *Service Systems Modeling.* s.l., Logos.

Böttcher, M. & Klingner, S., 2010. *Komponenten in der Dienstleistungsdomäne,* s.l.: s.n.

Böttcher, M. & Klingner, S., 2011. Komponentisierung zur Steigerung der Dienstleistungsproduktivität. In: M. Bruhn & K. Hadwich, Hrsg. *Dienstleistungsproduktivität.* s.l.:Gabler, pp. 59-80.

Böttcher, M. & Klingner, S., 2011. Providing a Method for Composing Modular B2B-Services. *Journal of Business \& Industrial Marketing.*

Grönroos, C. & Ojasalo, K., 2004. Service productivity: Towards a conceptualization of the transformation of inputs into economic results in services. *Journal of Business Research,* 57(4), pp. 414-423.

Husen, C., Fuchs, B., Böttcher, M. & Meyer, K., 2008. Ermittlung von Problemfeldern bei der Entwicklung IT-basierter Dienstleistungen. In: K. Fähnrich & C. Husen, Hrsg. *Entwicklung IT-basierter Dienstleistungen.* s.l.:Physica-Verlag HD, pp. 29-39.

Ioannidis, Y., Ramakrishnan, R. & Winger, L., 1993. Transitive closure algorithms based on graph traversal. *ACM Trans. Database Syst.,* Band 18, pp. 512-576.

Kang, K. C. et al., 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study,* s.l.: s.n.

Mendonca, M., Wasowski, A. & Czarnecki, K., 2009. *SAT-based analysis of feature models is easy.* s.l., Carnegie Mellon University, pp. 231-240.

OMG, 2010. *Object Constraint Language,* s.l.: s.n.

OMG, 2010. *OMG Unified Modeling Language Superstructure,* s.l.: s.n.

O'Sullivan, J. J., 2008. *Towards a precise understanding of service properties,* s.l.: s.n.

Pnueli, A., 1977. The temporal logic of programs. *Foundations of Computer Science, Annual IEEE Symposium on,* Band 0, pp. 46-57.

Thum, T., Batory, D. & Kastner, C., 2009. *Reasoning about edits to feature models.* s.l., IEEE Computer Society, pp. 254-264.

Tiihonen, J., Soininen, T., Niemelä. & Sulonen, R., 2003. *A practical tool for mass-customising configurable products.* s.l., s.n., pp. 1290-1299.

Vazsonyi, A., 1954. The Use of Mathematics in Production and Inventory Control. I. *Management Science,* 1(1), pp. 70-85.