Fraunhofer Institut
Software- und
Systemtechnik

Formale Spezifikation des strukturierten Anforderungsmanagements für das Domain Engineering anhand eines Beispiels aus der Automobilindustrie

Alexander Borusan
Augustin Kebemou
Marek Feldo

Berlin, 24. September 2003

Inhalt

1 Abstrakt

Der Erfolg eines Softwareprojekts hängt maßgeblich von der Qualität der Ergebnisse des Requirements Engineering ab, also von der Vollständigkeit, der Konsistenz und der Adäquatheit der erhobenen Anforderungen. Obwohl man sich in Wissenschaft und Praxis schon seit vielen Jahren mit Methoden und Werkzeugen für das Requirements Engineering auseinandersetzt, scheitern auch heute noch viele Projekte aufgrund von unzulänglich oder fehlerhaft analysierten Anforderungen. Das Erfassen von Anforderungen erfolgt naturgemäß in der sehr frühen Phase von Projekten. Hierbei wird in der Regel verbal beschrieben, was von einem Produkt gefordert wird oder auch was ein Produkt leisten soll. Auf der Basis solch einer Erfassung werden nachfolgende Modelle erstellt, welche das Produkt genauer spezifizieren. Es existieren viele Ansätze, die über die methodische Erfassung und Bearbeitung der Anforderungen hinausgehen.

Dieser Papier stellt eine komparative Spezifikation einer Methodik des Fraunhofer ISST für den Bereich der Automobilindustrie dar, welche als Grundphilosophie die Wiederverwendbarkeit von Artefakten der Anforderungsanalyse inne hat. Dieser Ansatz unterscheidet zwischen zwei grundlegenden Entwicklungsbereichen: dem Domain Engineering und dem Application Engineering.

2 Hintergrund

2.1 Einführung in das Requirements Engineering

Jede Anwendungsentwicklung erfordert die Ermittlung und Verwaltung von Anforderungen. Anforderungen (oder englisch: Requirements) beschreiben die Erwartungen des Nutzers an ein Produkt oder einen Dienst. Nur bei einer genauen Vorstellung davon, was die Kunden benötigen und wollen, kann die gewünschte Anwendung entwickelt werden. Je früher es dabei gelingt, die Anforderungen der Kunden korrekt festzuhalten, desto kostengünstiger und schneller kann eine passende Lösung entwickelt werden. Solche Anforderungen können dabei z.B. als (natürlichsprachliche) Texte formuliert sein oder auch in Form von grafischen Darstellungen (z.B. USE-CASES aus der UML) beschrieben sein. Ein effizientes Requirements Engineering gibt u.a. Aufschluss darüber, welche Auswirkungen Änderungen in einer Anforderung auf andere Anforderungen und damit auf das Projekt haben.

In größeren Projekten, wie z.B. in der Fahrzeugindustrie, ist mit Tausenden von Anforderungen aller Art zu rechnen. Um den Überblick nicht zu verlieren werden die Anforderungen strukturiert . Eine mögliche Strukturierung ist im Kapitel 3.1.3 zu sehen. Weiterhin besteht die Möglichkeit die Anforderungen zu klassifizieren. Anforderungen können dabei z.B. in funktionale oder nichtfunktionale Anforderungen kategorisiert werden, je nachdem ob der Funktionalitätsumfang der Software betroffen ist oder die "Art und Weise" wie diese Funktionalität erbracht werden soll. Auch nach der betrachteten Granularität können die Anforderungen kategorisiert werden bspw. in Kundenanforderungen (diese beschreiben eher grobgranular, die Anforderungen eines Systemanwenders) und Systemanforderungen (diese beschreiben detailliert, wie die Kundenanforderungen zu realisieren sind).

2.2 Domain und Application Engineering

Die Wiederverwendung von Software-Artefakten oder -Strukturen ist unabdingbar. Ohne einen systematischen Ansatz von Wiederverwendbarkeit ist es nicht möglich, Software-Systeme für die wachsende Wirtschaft mit den sich schnell ändernden Anforderungen zu produzieren. Ansätze der Wiederverwendbarkeit sind Domainspezifisch und basieren auf zwei Schlüsselelementen: Domain Engineering und Application Engineering. Im Domain Engineering werden Erfahrungen mit Systemen oder Teilen von Systemen einer spezifischen Domäne als Artefakte gesammelt, organisiert und archiviert. Weiterhin versteht man unter Domain Engineering das zur Verfügung stellen ausreichender Mittel für die Wiederverwendung dieser Artefakte beim Aufbau neuer Systeme. Der Aufbau dieser neuen Systeme wird als Application Engineering verstanden.

3 Ansatz eines Applikations- und Domänenmodells für das Anforderungsmanagement: ein Beispiel für die Automobilindustrie.

Die Automobilindustrie hat sich während der letzten Jahre rasant entwickelt. Die Fortschritte in Elektronik und Informationstechnologie haben dieses Wachstum vorangetrieben, und dies für alle Hersteller. Das technologische Niveau der Hersteller hat sich dadurch fast ausgeglichen. Deshalb hat ein Fahrzeugprojekt nur dann Erfolg, wenn das Produkt schnell und mit einer guten Qualität auf der Markt kommt. Im Kampf um den Marktanteil unter den Automobilherstellern haben nur die Hersteller Erfolgschancen, die auf kontinuierliche Innovation setzen. Für sie hat ein Projekt nur dann gute Erfolgschancen, wenn die Anforderungen an das Fahrzeug richtig erfasst und verwaltet sind. Dafür muss es einen effizienten Ansatz geben. Nachfolgend wird ein Ansatz für ein Applikationsmodell und ein Domänenmodell vorgestellt, welcher von der Abteilung Verlässliche Technische Systeme des Fraunhofer ISST Berlin in einem Industrieprojekt entwickelt wurde.

3.1 Das Applikationsmodell

In dem nun vorgestellten Ansatz eines Applikations- und Domänenmodells für das Anforderungsmanagement in der Automobilindustrie werden die Anforderungen in Kunden- und Systemanforderungen klassifiziert. Der nächste Abschnitt beschreibt die Organisation der beiden Klassen.

3.1.1 Die Kundenanforderungen

Die Kundenanforderungen beschreiben die erlebbare Funktionalität, d.h. diejenigen Merkmale, die ein Kunde von dem Fahrzeug erwartet. Allerdings umfasst der Begriff „Kunde“ nicht nur die Benutzer des Fahrzeugs, sondern auch andere Interessenten, wie z.B. den Gesetzgeber, der beispielsweise die Sicherheitsmaßnahmen festlegt oder auch die Vertriebsabteilung, die im Vergleich zur Konkurrenz ihre Forderungen stellt. Die Kundenanforderungen abstrahieren von den technischen Details, wie man in den folgenden Beispielen erkennt:

- Beim Fahrzeugzugang ist eine Benutzeridentifikation vorzunehmen.
- Das Fahrzeug kann aus der Ferne entriegelt werden.

3.1.2 Die Systemanforderungen

Die Systemanforderungen beschreiben, was das Fahrzeug bzw. System können soll, d.h., was von den Entwicklern verlangt wird. Im

Unterschied zu Kundenanforderungen können hier technische Gegebenheiten wichtig sein, wie die folgenden Beispiele zeigen.

- Der Fahrzeugzugang ist bei einer *Fingerabdruck-Erkennung* zu gewähren.
- Das Fahrzeug ist durch einen *Funkschlüssel* zu entriegeln.

Die Systemanforderungen beziehen sich auf technische Umsetzungen der Kundenanforderungen, gehen dabei aber nicht auf die technische Realisierung ein [BCGH02]. In den Beispielen wird weder gesagt, welche Technologie eingesetzt wird, noch welche Geräte eingebaut werden, um einen Fingerabdruck zu erkennen. Das folgende Beispiel verdeutlicht den Unterschied zwischen einer Kundenanforderung, Systemanforderung und der technischen Realisierung.

- Kundenanforderung: Die Luft im Fahrzeug ist zu verbessern.
- Systemanforderung: Luftaustrittsöffnungen sind bereitzustellen.
- Technische Realisierung: Ein Feuchtigkeitssensor PRX-123- 4567 ist einzubauen.

3.1.3 Strukturierung der Anforderungen im Applikationsmodell

Um eine übersichtliche Darstellung der Anforderungen zu erhalten und um Anforderungen schneller lokalisieren und wiederfinden zu können, werden sowohl Kunden- als auch Systemanforderungen hierarchisch strukturiert. Hierzu werden Oberbegriffe eingeführt, anhand derer die Anforderungen klassifiziert werden können. Dadurch befinden sich kontextbezogen eng verwandte Anforderungen innerhalb der Struktur nah beieinander.

Dies ergibt eine Baumstruktur, an deren Blattknoten die Anforderungen stehen, welche als atomare Informationen behandelt werden. Alle anderen Knoten sind sogenannte Strukturierungsknoten, welche die Anforderungen entsprechend ihren Zuordnungen gliedern. Die Strukturierungsknoten dienen zur Lokalisierung der Anforderungen.

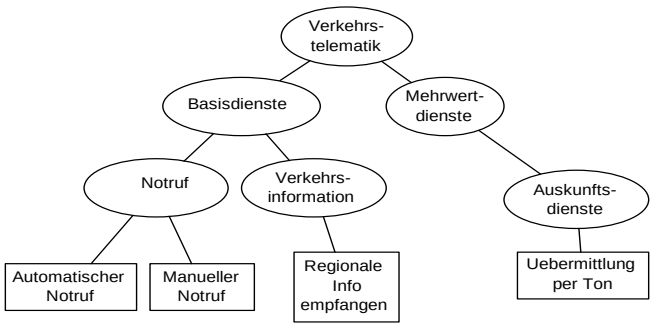

Abbildung 1: Ein Anforderungsbaum.

3.1.4 Beziehungen zwischen Anforderungen: „Needs“ und „Excludes“ Beziehungen

Für ein Fahrzeugprojekt werden die Kundenanforderungen in einem Anforderungsbaum, wie er in Abbildung 1 zu sehen ist, erfasst. Dabei kann es vorkommen, dass sich Anforderungen gegenseitig ausschließen oder einander erfordern.

So schließen sich die beiden folgenden Anforderungen gegenseitig aus, da ein Fahrzeug nicht gleichzeitig einen Dieselmotor und einen Benzinmotor besitzen kann.

- „Das Fahrzeug soll einen Diesel Motor haben“
- „Das Fahrzeug soll einen Benzin Motor haben“

Die beiden nachfolgenden Anforderungen hingegen erfordern sich gegenseitig.

- „Belüften mit Aussenluft“
- „Zuführen von Frischluft“.

3.1.5 Beziehung zwischen den Kunden- und Systemanforderungen: die Konkretisierungsbeziehung

Die Anforderungen für die Systementwickler sind in den Systemanforderungen festgelegt, während die für den Kunden bzw. den Verkauf des Systems relevanten Merkmale in den Kundenanforderungen beschrieben sind. Durch den Entwicklungsprozess muss sichergestellt werden, dass die Kundenanforderungen durch die Systemanforderungen auch vollständig abgedeckt werden. Deshalb wird im Modell eine n:m Assoziations- Beziehung zwischen Kunden- und Systemanforderungen, nämlich die „Konkretisierungsbeziehung“ eingeführt. Ein Beispiel ist in der nachfolgenden Abbildung zu sehen.

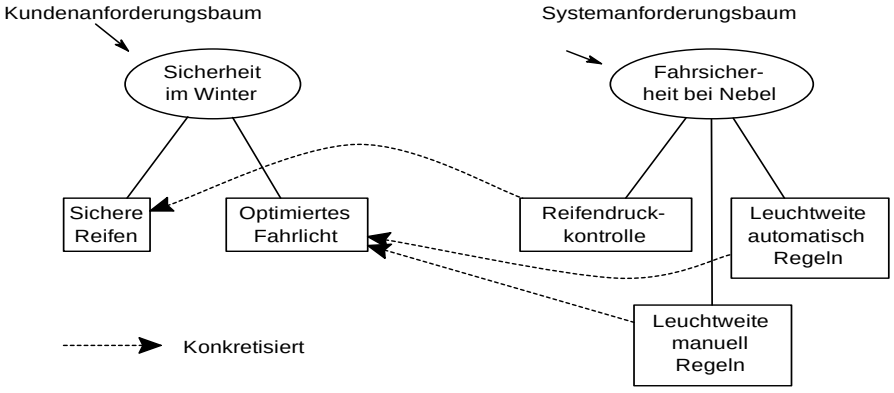

Abbildung 2: Konkretisierungsbeziehung.

3.1.6 Konfiguration von Anforderungen (Fahrzeugvarianten)

Um die Anforderungen und deren Beziehungen in Bezug zu Fahrzeugprojekten zu bringen, werden Fahrzeugvarianten beschrieben und den Anforderungen zugeordnet. Fahrzeugvarianten sind Merkmale von Fahrzeugen, welche diese eindeutig identifizieren. Diese Identifizierung kann z.B. der Motor in Kombination mit der Karosserie sein. Jede Fahrzeugvariante kann also durch ein n-Tupel von Werten dargestellt werden, wobei jeder der Werte einen bestimmten Typ (Wertebereich) hat.

Jedoch werden die Fahrzeugvarianten nur den Systemanforderungen zugeordnet, da nur diese in Fahrzeugprojekten direkt realisiert werden. Will man die Information über die Realisierung einer Kundenanforderung in einer bestimmten Fahrzeugvariante haben, lässt sich diese mit Hilfe der Konkretisierungs- Assoziation ermitteln.

Für eine gegebene Anforderung in einer Fahrzeugvariante wird zusätzlich ein Realisierungs- Attribut vergeben. Dieses Attribut sagt aus, ob die Anforderung als Serienausstattung (Serie) in dem Fahrzeug realisiert wurde oder nur zur Sonderausstattung(SA) gehört.

3.1.7 Kontextbereiche

Ein wichtiges Merkmal des hier beschriebenen Modells besteht darin, dass die Erfahrungen aus vorhandenen Fahrzeugprojekten in anderen Fahrzeugprojekten wiederverwendet werden können. Bestimmte Aspekte von Anforderungen manifestieren sich nicht nur ausschließlich in den Anforderungen selbst, sondern auch in ihrer Gliederung. Aus diesem Grund bekommen gewisse Strukturierungsknoten des Applikationsmodells den Charakter von Vorlagen. Solche Strukturierungen sind Teilbäume des Anforderungsbaums. Sie werden Kontextbereiche genannt. Sie können für einen späteren Gebrauch ausgezeichnet und in einem Repository abgelegt werden. Dieser Vorgang wird sowohl im Kundenanforderungsbaum als auch im Systemanforderungsbaum ausgeführt. Um solche Kontextbereiche zu markieren, werden bestimmte Strukturierungsknoten mit sogenannten *Kontexten* assoziiert. Ein Kontext kann ein beliebiger Begriff sein wie z.B. ein Themenbereich (Infotainment, Karosserie). Ein Kontextbereich erstreckt sich von der jeweiligen Wurzel des Teilbaumes bis zu den Blättern bzw. dem nächsten Kontextbereich. Die Blätter, d.h. die Anforderungen selbst, gehören nicht zum Kontextbereich. Ein Kontextbereich kann somit Strukturierungsknoten, die Verbindungen zwischen den Strukturierungsknoten sowie die Verbindungen zu den Anforderungen beinhalten.

In Abbildung 3 wird ein Beispiel dargestellt, in welchem ein Kontextbereich mit dem Strukturierungsknoten „Medien abspielen“ beginnt und zwei Strukturierungsknoten „analoge Medien“ und „digitale Medien“ beinhaltet. Zwei weitere Kontextbereiche beginnen jeweils mit dem Strukturierungsknoten „Video abspielen“ und reichen bis zu den Anforderungen. Beide Teilstücke haben den gleichen Kontext

(angedeutet durch die gleiche Umrandung), unterscheiden sich jedoch in der Struktur, da sie eine unterschiedliche Anzahl der Verbindungen zu den Anforderungen besitzen. Im linken Teilbaum beinhaltet der Strukturierungsknoten „Ton abspielen“ zwei Verbindungen zu den Anforderungen, im rechten Teilbaum hingegen drei. Aufgrund dieser Eigenschaft, dass ein Kontext unterschiedliche Strukturen besitzen kann, können Varianten innerhalb eines Anforderungsbaumes erstellt werden.

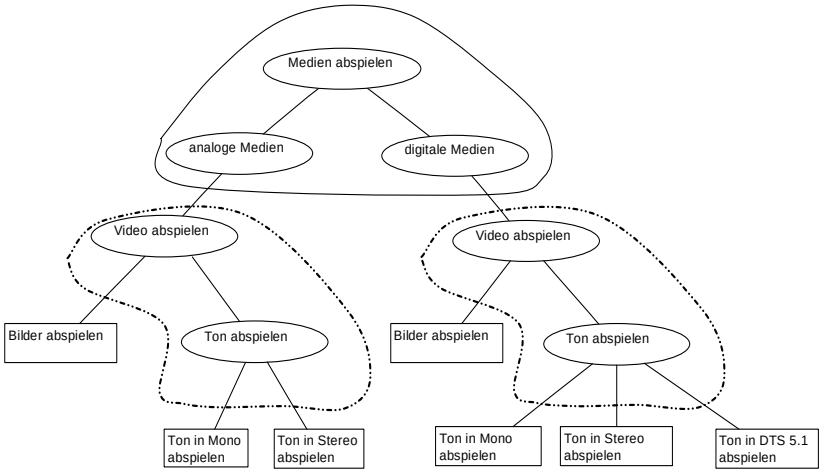

Abbildung 3: Kontextbereiche.

3.1.8 Aktivitäten auf den Anforderungsbäumen

Im folgendem werden Aktivitäten angegeben, die im Applikationsmodell durchzuführen sind. Diese Aktivitäten werden im späteren Verlauf dieses Berichts formal beschrieben.
Der Kundenanforderungsbaum wird gepflegt durch Aktivitäten wie:

- Strukturierungsknoten hinzufügen, löschen, ändern
- Anforderungen hinzufügen, löschen, ändern
- Excludes- und needs- Beziehungen hinzufügen, löschen, ändern
- Kontextknoten auszeichnen.

Die Aktivitäten auf der Systemanforderungsbaum sind z. B.:

- Alle Aktivitäten auf dem Kundenanforderungsbaum
- Fahrzeugvarianten hinzufügen, löschen, ändern.

3.2 Das Domänenmodell

Das Domänenmodell enthält im Gegensatz zum Applikationsmodell, in welchem die Anforderungen zu konkreten Fahrzeugprojekten modelliert werden, Informationen, die sich bereits in verschiedenen Fahrzeugprojekten bewährt haben. Hierzu werden Modelle benötigt, welche die Informationen aus konkreten Produkten abstrahieren. Diese

Informationen bestehen aus Kontextbereichen und Kunden- und Systemanforderungen.

3.2.1 Kontexte und Anforderungen

Die Kontexte spielen neben den Anforderungen im Domänenmodell die zentrale Rolle. So werden nicht nur die Anforderungen für eine spätere Wiederverwendung bereitgestellt, sondern eben auch die Kontexte. Die Struktur eines Kontextes selbst ist, ebenso wie im Applikationsmodell, eine Baumstruktur. Es können verschiedene Varianten von Kontexten existieren. Kontexte können im Domänenmodell miteinander assoziiert werden. Die Assoziation zweier Kontexte drückt aus, dass diese beiden Strukturen in einem Projekt irgendwann einmal miteinander verbunden waren. Hierzu existieren an den Blättern der Kontextstruktur entsprechende Verknüpfungspunkte. Ebenso wie andere Kontexte können diese auch mit Anforderungen assoziiert werden. Es wird sogar gefordert, dass jede Anforderung im Domänenmodell mit mindestens einer Kontextstruktur assoziiert wird. Kontexte selbst brauchen jedoch nicht unbedingt mit Anforderungen assoziiert zu werden. Zwei Kontextvarianten unterscheiden sich durch die Menge dieser Verknüpfungspunkte.

Diese Assoziationen sind allgemeine *n:m*-Beziehungen, so dass beispielsweise eine Anforderung mit mehreren Kontextstrukturen assoziiert werden kann. Dies sagt dann aus, dass diese Anforderung in unterschiedlichen Projekten in verschiedenen Kontexten verwendet wurde.

In Abbildung 4 ist die im vorherigen Abschnitt vorgestellte Struktur als Domänenmodell dargestellt. Die kleinen Rechtecke stellen die Verknüpfungspunkte dar.

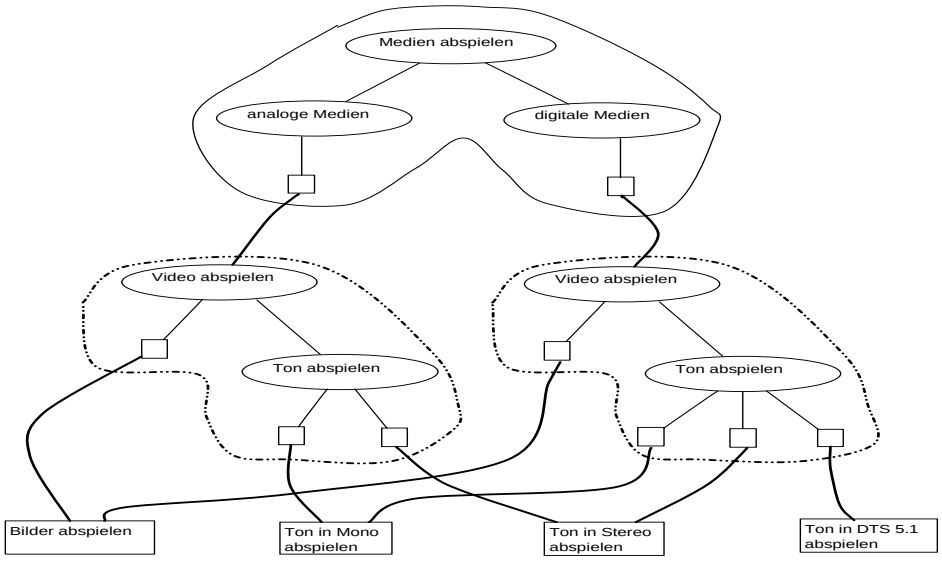

Abbildung 4: Das Domänenmodell.

3.2.2 Aktivitäten auf dem Domänenmodell

Da das Domain Repository nur die Informationen, die im Application Repository vorhanden sind, enthält, findet keine Informationsänderung im Domänenmodell statt. Das Domain Repository wird ausschließlich durch das Hinzufügen und die Löschung von Elementen aus dem Applikationsmodell gepflegt.

4 Formale Spezifikation des Fallbeispiels aus der Automobilindustrie.

Viele Menschen mit unterschiedlichen Kenntnissen und Kompetenzen sind an der Entwicklung eines komplexen Softwaresystems beteiligt. Die Kommunikation zwischen Ihnen spielt dabei eine entscheidende Rolle. Die Spezifikation soll bei der Kommunikation helfen.

Eine Spezifikation soll die Anforderungen des Systems, seine Prozesse und ihren Ablauf beschreiben. Die Beschreibung kann in natürlichsprachlicher Form geschehen. In einer Aufgabe, in der die Präzision eine wichtige Rolle spielt, kommt dieses Mittel nur schlecht zur Anwendung, da alle unsere natürlichen Sprachen sehr unpräzise sind.

Eine Graphische Spezifikation hat hingegen den Vorteil, kompakt und leicht verständlich zu sein, wenn die benutzen graphischen Symbole einer Standardisierung unterliegen. Sie ist aber ebenfalls nicht eindeutig genug und man benötigt großen Aufwand, um ein System in allen Sichten (statische Zustände und kontinuierliches Verhalten) zu beschreiben. Sehr bekannt in dieser Kategorie ist die UML (Unified Modelling Language).

Die Mängel der beiden angesprochenen Spezifikationssprachmittel möchte man mit einer formalen Spezifikation ausgleichen. Eine formale Spezifikation wird in einer Sprache ausgedrückt, deren Vokabular, Syntax und Semantik formal definiert sind und die auf mathematischen Konzepten beruht.

In den nächsten Abschnitten wird das Fallbeispiel mittels der Graphentheorie und mit der Spezifikationssprache Z beschrieben.

4.1 Formale Spezifikation des Applikations- und Domänenmodells mittels Graphentheorie

4.1.1 Definition eines Graphen

Ein Graph ist ein Paar $G = (V,E)$ von endlichen Mengen, mit $V \cap E = \phi$, wobei V die Knotenmenge (englisch: Vertex, Nodes) und E die Kantenmenge (englisch: Edge) ist. Die Kanten sind Paare von Knoten, d. h. $E \subseteq V^2 (\forall xy \in E; x \in V \wedge y \in V)$. Zur Notation: $(x,y) = xy$. Wenn die Kanten und die Knoten von Graphen mit weiteren Informationen annotiert sind, spricht man von bewerteten Graphen [NST96].

4.1.2 Bäume

Bäume sind spezielle Graphen. Ein Baum ist ein zusammenhängender, ungerichteter, azyklischer Graph, wo jedes Knotenpaar genau durch einen Weg verbunden ist. Bäume sind i. a. dynamische Datenstrukturen.

Sie besitzen eine Wurzel, Äste und Blätter. Jeder nicht leere Baum besitzt mindestens einen Knoten, die Wurzel. Weiter besitzt jeder Knoten eine (eventuell leere) Menge von Nachfolgern. Hat ein Knoten keinen Nachfolger, so spricht man von einem Blatt. Hat er Nachfolger, ist er ein Ast. Die Anzahl der Äste ist dabei beliebig [CGM79]. Die Bäume des Fallbeispiels speichern die Informationen (Anforderungen) nur in den Blättern. Die Äste enthalten ausschließlich Schlüssel (Strukturierungsknoten), mit denen man sich zu den richtigen Blättern navigieren kann . Man bezeichnet solche Bäume als blattorientiert.

In diesem Bericht bezeichnet der Begriff Datenelemente die Anforderungen selbst.

4.1.3 Beschreibung des Modells mittels Graphen

4.1.3.1 Entwurf der Struktur

Ein Baum kann durch seine Wurzel und seine Unterbäume eindeutig identifiziert werden. Die Unterbäume sind die Bäume, deren Wurzeln die Kinder der oberen Wurzel sind. Um die Aktivitäten auf dem Anforderungsmanagementbaum spezifizieren zu können, wollen wir aber einen Baum wie folgt beschreiben.
$B = (V,r,E)$ mit $E \subseteq V \times V$; $r \in V$.

- *V* ist die Menge aller Knoten und Blätter im Baum. *V* ist eine endliche Menge.
- *r* ist die Wurzel des Baumes. Die Wurzel ermöglicht jederzeit einen Zugriff auf den Baum. *r* ist ein Knoten des Baumes.

- *E* ist die Menge aller Kanten im Baum.

Definition: Ein Baum ist eine endliche Knoten- und Blättermenge *V* und eine binäre Relation über *V*.
Die binäre Relation ist entweder eine Menge von geordneten Paaren von Elementen von *V* oder eine Funktion von *V* zu der Potenzmenge (P_0) von *V* .

Definieren wir die Abbildung „*kind*" wie folgt:
kind: $E \rightarrow V$ sodass
für eine Kante e eines Baums ist kind(e) der Endknoten dieser Kante. Man sagt auch, dass ein Knoten y „Kind von" x ist, wenn x ein Knoten ist und wenn eine Kante von x nach y besteht.

Wir führen die Funktion „Nachfolger" ein und notieren sie:

nachfolger: $V \rightarrow P_0$ (V)
Wir definieren *nachfolger(v)* als die Menge aller direkten Nachfolger von v, d. h. die Menge aller Kinder von v, wenn v ein Knoten in V ist.

$\forall x,y \in V$, ist xy die Kante zwischen x und y, wenn y ein Kind von x ist, d. h. wenn $y \in$ nachfolger(x).

Die surjektive Hülle von *nachfolger* ist eine bijektive Funktion. Ihre invertierte Funktion ist die Funktion *vorgänger*.

Es existieren im Kundenanforderungsbaum zwei Sorten von Knoten: die Strukturierungsknoten und die Informationsknoten.

Es wird in diesem Bericht festgelegt, das es in einem konsistenten Baum keine Informationsknoten ohne Strukturierungsknoten geben darf. Werden Anforderungen gesammelt, so kann das durchaus auch ohne eine Strukturierung geschehen. Die Festlegung wird getroffen, da die Strukturierung der Anforderungen für das Domänenmodell im hier vorgestellten Ansatz eine entscheidende Rolle spielt.

Ein nicht leerer konsistenter Kundenanforderungsbaum hat mindestens eine Anforderung, d. h. ein Blatt und einen Strukturierungsknoten. Der Baum *B* ist leer, wenn keine Anforderung existiert.

Zur Notation:

$V = S \cup I$, wobei S die Menge der Strukturierungsknoten ist. Im Fallbeispiel sind diese ausschließlich Äste des Kundenanforderungsbaums, d. h. mögliche Wurzeln eines Unterbaums.

I = Menge der Informationsknoten oder Blätter.

Dabei gilt : $S \cap I = \varnothing$, d. h. eine Anforderung (Informationsknoten) kann nicht als Strukturierungsknoten benutzt werden. Sie kann im Kundenanforderungsbaum nur ein Blatt sein, und genauso dürfen die Strukturierungsknoten nur Äste sein.

Für jede Menge X bezeichnet $|X|$=Card(X) die Mächtigkeit von X.

Wir führen die „parent" Beziehung ein und notieren sie p:

$p: V \rightarrow V$

p(v) = Parent von v ist die Menge der Vorgänger von v

Eigenschaften von p:

p ist eine Funktion, da jeder Knoten in einem Baum maximal einen Vater hat.

p ist nicht injektiv.

Beweis:

p injektiv, wenn $\forall x,y \in V$; $x \neq y \Rightarrow p(x) \neq p(y)$ oder äquivalent $p(x)=p(y) \Rightarrow x=y$.

Manche Strukturierungsknoten sind Vorgänger von mehreren Anforderungen. D.h. $\exists x,y \in V / x \neq y \wedge p(x)=p(y)$ ist erlaubt;

p ist nicht surjektiv.

Beweis:

p ist surjektiv, wenn $(\forall y \in V)(\exists x \in V)$ sodass $y=p(x)$.

Die Wurzel hat keinen Parent. Trotzdem kann man intuitiv sehen, dass kind die Rolle der invertierten Funktion von parent spielt. In der Tat, für beliebige x und y, zwei Knoten eines Baums, gilt:

„x ist Kind von y" ist äquivalent zu „y ist parent von x".

Anmerkung: Für alle v in S, nachfolger(v) $\neq \emptyset$. Jeder Strukturierungsknoten dient nur dazu, die tatsächlichen Anforderungen zu gliedern.

Sei *r* die Wurzel des gesamten Baums, für alle *v* in $V-\{r\}$, $\exists!\ w \in V$ | $p(v)=w$.

p(r) existiert nicht. In einem Baum ist der Eingangsgrad aller Knoten mit Ausnahme der Wurzel 1. Die Restriktion von *parent* auf $V-\{r\}$ ist surjektiv für einen konsistenten Baum.

Die definierte Datenstruktur für die Anforderungsbäume im Applikationsmodell lautet somit: *(V,r,E,nachfolger,parent).*

Diese Datenstruktur soll uns ermöglichen vom mathematischen Denken zu profitieren, da aus mathematischer Sicht Datenstrukturen Algebren sind. Wir wollen die Datenstruktur „Baum" weiter abgekürzt mit *(V,r,E)* beschreiben.

Wir nennen :

B_0 die Menge aller Bäume, die die oben beschriebenen Eigenschaften besitzen.

S_0 die Menge aller möglichen Strukturierungsknoten und I_0 die Menge aller möglichen Informationsknoten.

4.1.3.2 Aktivitäten in B_0

Für alle Aktivitäten haben die Bäume $B \in B_0$ die Form $B=(V,r,E)$ mit $V=S \cup I$.

Im folgendem werden die Aktivitäten auf den Bäumen formal beschrieben.

- Hinzufügen eines Anforderungsknotens

Das Hinzufügen eines Anforderungsknotens ist sowohl auf einem nicht leeren Baum sowie auf einem leeren (erster Knoten) möglich.

Der Baum ist nicht sortiert und besitzt keine besonderen Eigenschaften, die die Suche nach der Einfügestelle erleichtern. Die Einfügestelle befindet sich unter einem ausgewählten Strukturierungsknoten. Existiert der gewünschte Strukturierungsknoten im Baum nicht, muss er zuerst eingefügt werden.

Ist der Baum leer, muss ebenfalls zuerst ein Navigationsknoten eingefügt werden, unter dem die Anforderung angefügt wird.

Seien $wo \in S$ der ausgewählte Ast, $v \in I$ der einzufügende Informationsknoten.

v wird unmittelbar unter *wo* eingefügt. Der Vorgang kann folgendermaßen formal spezifiziert werden:

Sei $B \in B_0$, $B=(V,r,E)$ mit $V=S \cup I$

Sei $wo \in S$, $\forall\ v \in I_0$ - *nachfolger(wo)* ,

$$Hinzufügen(v,wo) \Leftrightarrow nachfolger(wo) = nachfolger(wo) + \{v\} \wedge p(v) = wo$$

Diese Schreibweise hat einige semantische Probleme.

- $\forall v \in I_0$ - *nachfolger(wo)* soll beschreiben, dass sich die Anforderung *v* vor Ausführung der Operation nicht bereits unter dem gewünschten Strukturierungsknoten befindet.
- *nachfolger(wo)* = *nachfolger(wo)* + *{v}* beschreibt für Informatiker eine Zuweisung. In der Mathematik hat diese Beschreibung jedoch eine andere Bedeutung, denn es wird damit ausgesagt, dass die Menge *nachfolger(wo)* gleich bleibt, auch wenn neue Elemente hinzukommen. Was natürlich an dieser Stelle falsch ist.
- Die Beschreibung *p(v)* = *wo* reicht alleine nicht aus, um die Aktivität *Hinzufügen(v,wo)* korrekt zu spezifizieren. Sie sagt nur aus, dass *wo* der parent von *v* ist. Das bedeutet noch nicht, dass *v* gerade unter dem Knoten wo hinzugefügt worden ist.

An dieser Stelle gibt es keine Möglichkeit, die Zustandsänderung formal zu beschreiben, ohne zu sagen „Vorher....“ und „Nachher....“. Aber wir dürfen nicht vergessen, dass die Syntax aus der Mathematik nur deshalb eingesetzt wurde, um bestimmte Sachverhalte eines Informationssystems zu beschreiben. Hier liegt offensichtlich ein Interpretationsproblem vor. Es fehlt offenbar an einer geeigneten Semantik für die hier benutze Syntax.

Sehen wir uns eine weitere Aktivität an.

- Löschen eines Anforderungsknotens

Beim Löschen einer Anforderung, die erst einmal ausgewählt werden soll, können zwei Fälle auftreten:

- die Anforderung hängt an einem Strukturierungsknoten, der andere Kinder hat. Hier kann ohne weiteres der Informationsknoten gelöscht werden.
- die Anforderung ist das einzige Kind seines *parents*. Hier muss der Strukturierungsknoten auch gelöscht werden. Ein Strukturierungsknoten darf nämlich, wie oben bereits beschrieben, nicht ohne darrunterhängende Anforderungen existieren.

So könnte die Aktivität beschrieben werden:

Sei $v \in I$ sodass $p(v) = wo$

1. Fall:

Wenn $|nachfolger(wo)|>1$;

$Löschen(v) \Leftrightarrow nachfolger(wo) = nachfolger(wo)-\{v\} \wedge p(v) = \varnothing$

2. Fall:

Wenn $nachfolger(wo) = \{v\}$; $Löschen(v) \Leftrightarrow Löschen(v) \wedge Löschen(wo)$

Im 1. Fall stoßen wir wieder auf die gleichen Probleme wie bei der Aktivität „Hinzufügen eines Anforderungsknotens".

Im 2. Fall kommen noch weitere Probleme hinzu. Zum einen müsste die Operation ein rekursives Löschen von Knoten aus dem Baum durchführen. Schließlich muss auch der Vorgänger des Vaters (*wo*) der Anforderung v gelöscht werden, falls er keine weiteren Nachfolger hat. Dies ist in der oberen Beschreibung nicht spezifiziert. Die Spezifikation ist somit nicht ausreichend. Weiterhin ist der Operator „$\wedge$" symmetrisch. D. h., man kann damit die zeitliche Reihenfolge der Aktivitäten nicht festlegen. Es geht hier um zwei Aktivitäten, die nacheinander durchgeführt werden müssen, um die gewünschte Änderung zu erreichen. *Löschen(wo)* könnte vor *Löschen(v)* erfolgen, was natürlich nicht wünschenswert ist. Die Reihenfolge muss in diesem Fall einfach stimmen, und *v* muss vor *wo* gelöscht werden. Man muss also die Reihenfolge der Aktionen festlegen.

4.1.4 Zusammenfassung

Bäume sind spezielle Graphen, d. h. Datenstrukturen im Sinne der Informatik und Algebren im Sinne der Mathematik. Sie lassen sich mit der Graphentheorie gut beschreiben, genauso auch die Beziehungen in den Bäumen, (*needs*- und *excludes*-Beziehungen). Aber die Operationen auf den Bäumen, die die Struktur des Baums ändern, lassen sich in dieser Form nicht beschreiben. Hierfür muss eine geeignete Datenstruktur mit einem Vokabular, einer Syntax und einer passenden Semantik eingesetzt werden. Die Semantik der Graphentheorie ist statisch. Sie kann kontinuierliche Veränderungen nur sehr schwer beschreiben.

4.2 Formale Spezifikation des Modells mit der Z-Notation

Im folgendem Abschnitt wird die Z-Notation verwendet um das Fallbeispiel aus der Automobilindustrie zu beschreiben. Die Z-Notation ist eine zustandsbasierte Spezifikationssprache und basiert auf der Mengentheorie und der Prädikatenlogik [Spi92]. Eine Z-Spezifikation besteht üblicherweise aus einer Anzahl von Zustand- und Operationsschemadefinitionen. Eine Schemadefintion kapselt Variablendeklarationen und Einschränkungen auf die möglichen Variablenbelegungen. Ein Operationsschema ist ein Schema, das die Beziehung zwischen Vor- und Nachzustand definiert. In den folgenden Abschnitten wird das Fallbeispiel mit Hilfe der Z Spezifikation formal beschrieben.

Um die formale Spezifikation der Anforderungsbäume zu vereinfachen, wird die Struktur der Bäume und die Anforderungen, die in den Blättern der Bäume stehen, in zwei unterschiedlichen Datenstrukturen behandelt. Diese Datenstrukturen werden mit zwei Z Schemata beschrieben. Das eine Schema „Kundenanforderungsbaum“ enthält die Beschreibung der Strukturierungsknoten und ihrer Beziehungen untereinander. In dem anderen Schema „Anforderungsliste“ werden die Anforderungen und die needs- und excludes Beziehungen verwaltet. Die Beziehung zwischen den Strukturknoten und den Anforderungen stellen sogenannte *Links* her. Die nachfolgende Abbildung stellt die verwendete Struktur grafisch dar.

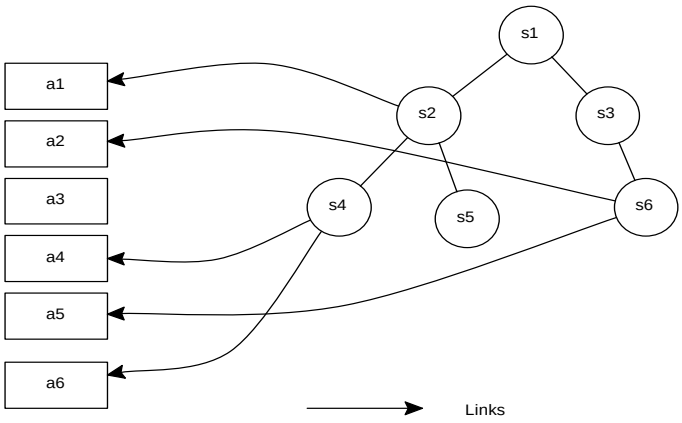

Abbildung 5: Links zwischen den Strukturknoten und Anforderungen

Die getrennte Spezifikation der Anforderungen und Strukturierungsbäume hat aber einen weiteren Vorteil. In das Domänenmodell können Anforderungen, auch ohne die Struktur hineingestellt werden. Aber auch Strukturen können für die Wiederverwendung geeignet sein, und ohne Anforderungen im Domänenrepository gespeichert werden. Die hier verwendete Spezifikation unterstützt diesen Ansatz. Im folgenden werden die beiden Z Schemata vorgestellt. Auch werden einige Aktivitäten auf den Datenstrukturen formal beschrieben.

4.2.1 Der Strukturierungsbaum

∪_*Kundenanforderungsbaum*_____
→*AnforderungsListe*
→*StruktKnoten:* Π *STRUKT_K*
→*Wurzel: STRUKT_K*
→*Kanten: STRUKT_K* φ *STRUKT_K*
→*parent: STRUKT_K* ♣ *STRUKT_K*
→*kinder: STRUKT_K* ♣ Π *STRUKT_K*
→*Links: INFO_K* φ *STRUKT_K*
→*srcLinks: INFO_K* ♣ *STRUKT_K*
→*tgLinks: STRUKT_K* ♣ Π *INFO_K*
∩_____

→*StruktKnoten* ⌊ 0 ⇔ *Wurzel* ε *StruktKnoten*	*[1]*
→dom *parent* = *StruktKnoten* \ {*Wurzel*}	*[2]*
→ran *parent* ζ *StruktKnoten*	*[3]*
→dom *Kanten* ζ *StruktKnoten*	*[4]*
→ran *Kanten* ζ *StruktKnoten*	*[5]*
→A*x: StruktKnoten* ∞ (*x*, *x*) ™ *Kanten*	*[6]*
→dom *kinder* = ran *parent*	*[7]*
→ran *kinder* ζ Π dom *parent*	*[8]*
→A*x*, *y: StruktKnoten* | *x* ⌊ *y* ∞ *x* = *parent y* ⇔ *y* ε *kinder x* ⇔ (*x*, *y*) ε *Kanten*	*[9]*
→*Kanten* ⌊ 0 ⇒ *StruktKnoten* \ {*Wurzel*} ⌊ 0	*[10]*
→A*x*, *y: StruktKnoten* ∞ *x* ⌊ *y* ⇔ (disjoint ©*kinder x*, *kinder y*⟩)	*[11]*
→dom *srcLinks* = dom *Links* = *InfoKnoten*	*[12]*
→dom *tgLinks* = ran *Links* = ran *srcLinks* ζ *StruktKnoten*	*[13]*
→A*x: InfoKnoten; y: StruktKnoten*	*[14]*
→ ∞ *y* = *srcLinks x* ⇔ *x* ε *tgLinks y* ⇔ (*x*, *y*) ε *Links*	*[15]*
→A*x*, *y: StruktKnoten* ∞ *x* ⌊ *y* ⇔ (disjoint ©*tgLinks x*, *tgLinks y*⟩)	*[16]*

∠_____

Der Strukturierungsbaum für die Kundenanforderungen unterscheidet sich nicht von dem Strukturierungsbaum für die Systemanfoderungen. Sämtliche Knoten des Baums sind Strukturierungsknoten (StruktKnoten).

Alle Strukturierungsknoten, bis auf die Wurzel, besitzen genau einen Vorgänger (Parent) [2] [11].

Weiterhin besitzt jeder Strukturknoten eine Menge von Nachfolgern. (kinder)

Die binäre Relation „Kanten“ ermöglicht die Wahl zwischen einem gerichteten und einem ungerichteten Graph. Die Verwendung der kinder- und parent- Relation zwischen zwei Knoten drückt auch aus, dass eine Kante zwischen den beiden existiert [9]. Folglich ist zu verstehen, dass ein Strukturierungsbaum ohne Kanten leer ist oder nur aus der Wurzel besteht.

Die Relation „Links“ stellt die Verbindungen zwischen den Strukturknoten und den Anforderungen her.

In einer „Link"-Beziehung ist der Strukturierungsknoten der *srcLink* (source link) und die Anforderung der *tgLink* (target link). Einem Strukturierungsknoten können mehrere *tgLinks,* also Anforderungen, zugewiesen werden. Jedoch darf jede Anforderung maximal unter einem Strukturierungsknoten platziert werden [16].

Die binäre Relation *Links* spielt in Bezug von *srcLinks* und *tgLinks* die gleiche Rolle wie *Kanten* in Bezug von *parent* und *kinder*.

An einem Beispiel soll das Schema noch einmal verdeutlicht werden:

Im Kundenanforungsbaum in Abbildung 5 ist zu sehen, dass:

StruktKnoten={s1,s2,..s6};Wurzel=s1;
Kanten={(s1,s2),(s1,s3),(s2,s4),(s2,s5),(s3,s6)};

Das gleiche kann auch folgendermaßen ausgedrückt werden:

Kanten(s2)={s4,s5}; Kanten(s3)={s6};

parent(s4)=s2; parent(s5)=s2; parent(s2)= s1;

kinder(s2)={s4,s5}; kinder(s3)={s6};

s4, s5, s6 haben keine Kinder.

Für die Kanten der Wurzel s1 gilt:

Kanten(s1) = StruktKnoten \ {s1} .

Die Verbindung zu den Anforderungen sieht folgendermaßen aus:

Links={(a1,s2);(a2,s6);(a4,s4);(a5,s6);(a6,s4)} oder

Links(s4)={a4,a6}; Links(s2)= {a1}; Links(s3)=Links(s1)=∅;

srcLinks(a1)=s2; srcLinks(a4)=s4; tgLinks(s4)={a4,a6}; tgLinks(s3)= tgLinks(s5)=∅;

Die folgenden Schemata beschreiben die Aktivitäten auf dem Anforderungsbaum.

- **Hinzufügen eines Strukturknotens**

∪_*AddStruktKnoten* _____
→Δ*Kundenanforderungsbaum*
→*strukt?, vater?: STRUKT_K*
∩_____
→*vater?* ε *StruktKnoten ƒ strukt?* ™ *StruktKnoten* *[1]*
→*kinder' vater? = kinder vater?* Y {*strukt?*} *[2]*
→*kinder' strukt?* = 0 *[3]*
∠_____

AddStruktKnoten beschreibt das Hinzufügen eines Strukturierungsknotens in einen Baum. Der einzufügende Knoten „*strukt?*“ befindet sich vor der Aktivität nicht im Baum [1]. Durch Angabe des Vorgänger-Knotens (*vater?*) wird die Einfügestelle festgelegt. Anschließend wird der neue Knoten in den Baum eingefügt, indem der neue Knoten zur Menge der Kinderknoten des Vaterknotens hinzugefügt wird [2]. Die nachfolgende Abbildung verdeutlicht den Vorgang.

AddStrukturierungsknoten (*strukt? = ns ; vater? = s3*).

Abbildung 6: Einfügen eines Strukturierungsknotens in einen Baum.

- **Löschen eines Strukturierungsknotens**

∪_*DeleteStruktKnoten*_____
→Δ*Kundenanforderungsbaum*
→*strukt?: STRUKT_K*
∩_____
→*strukt?* ε *StruktKnoten ƒ strukt?*⎣ *Wurzel* *[1]*
→*tgLinks' strukt?* = 0 *[2]*
→*strukt?* ™ *StruktKnoten'* *[3]*
→*kinder'* (*parent strukt?*) *[4]*
→ = *kinder* (*parent strukt?*) Y *kinder strukt?* \ {*strukt?*} *[5]*
∠_____

Beim Löschen eines Strukturierungsknotens müssen alle Nachfolger des zu löschenden Strukturierungsknotens erhalten bleiben. Die direkten Nachfolgeknoten erhalten hierzu einen neuen Vorgänger, nämlich den Vater des gelöschten Strukturierungsknotens [5].

Die Wurzel darf nicht durch diese Operation gelöscht werden, denn sie hat keinen Vorgänger [1]. Die Verbindungen zu den Anforderungen, die der zu löschende Knoten enthält, werden ebenfalls mitgelöscht [2]. Der Vorgang lässt sich im folgenden Bild visualisieren.

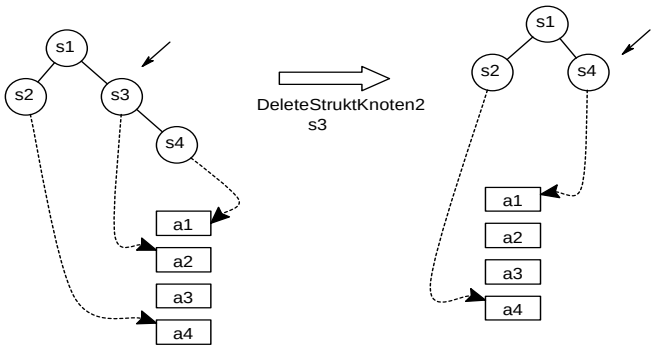

Abbildung 7: Löschen eines Strukturierungsknotens.

- **Umstrukturierung des Anforderungsbaums**

∪_MoveStruktKnoten_____
→ΔKundenanforderungsbaum
→strukt?, destination?: STRUKT_K
∩_____
→strukt? ⌊ destination? [1]
→*f* strukt? ™ kinder destination? [2]
→*f* strukt? ε StruktKnoten [3]
→*f* destination? ε StruktKnoten [4]
→kinder' destination? = kinder destination? Y {strukt?} [5]
∠_____

MoveStruktKnoden beschreibt die Umstrukturierung des Strukturierungsbaums.

Als Eingabe wird zum einen der Strukturierungsknoten angegeben, der umbewegt werden soll (*strukt?*). Weiterhin wird das Ziel durch einen Knoten angegeben, unter den der Knoten eingefügt werden soll (*destination?*).

Die Vorbedingung besagt, dass *strukt?*, der umzugliedernde Strukturierungsknoten, kein Kind vom *destination?* sein darf [2]. Denn dann würde der Knoten an der gleichen Stelle verbleiben. Weiterhin ist es nicht erlaubt, dass ein Knoten durch eine Umstrukturierung sein eigenes Kind wird [1]. Falls die Vorbedingungen erfüllt sind, wird der Knoten unter den neuen Wunschknoten eingefügt. Durch die Definition im Schema *Kundenanforderungsbaum*, sagt [5] auch aus, dass der Knoten *strukt?* einen neuen *parent* bekommt. Dadurch, dass jeder Knoten nur einen Vorgänger haben darf, wird die Verbindung zum alten Vorgängerknoten automatisch mitgelöscht.

Diese Operation könnte auch *MoveTeilBaum* genannt werden, da der ganze Teilbaum unter *strukt?*, einem neuen Vorgänger untergliedert wird. Die nachfolgende Abbildung zeigt ein Beispiel für das Umstrukturieren eines Baums.

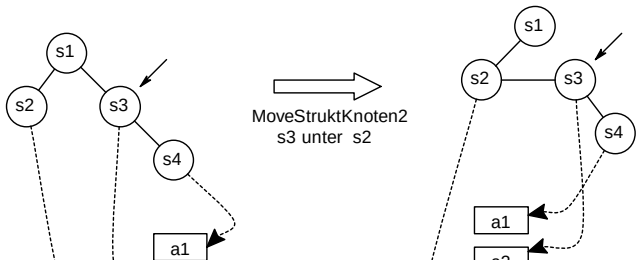

Abbildung 8: Umstrukturierung des Kundenanforderungsbaums (*strukt?=s3; destination?=s2*).

- **Konfiguration von Anforderungen**

```
∪_Fahrzeugvarianten _____________________________
→AttrNamen: seq ATTR                                    [1]
→attrWerte: ATTR ϕ ΠATTR                                [2]
→Realisierung: ATTR                                     [3]
→RealisierungsArten: ΠATTR                              [4]
→ FzvListe: Π (seq ATTR)                                [5]

_________________
→dom attrWerte = { x: ATTR | Ei: N ∞ AttrNamen i = x }   [6]
→RealisierungsArten = attrWerte Realisierung            [7]
→AFzv: FzvListe ∞ Ai: N | i ⟩ # Fzv ∞ Fzv i ε attrWerte (AttrNamen i)   [8]
→AFzv: FzvListe; i: N ∞ let i == # Fzv ∞ Fzv (i + 1) ε RealisierungsArten   [9]
∠_____________________________________________
```

Um einen Bezug zwischen Anforderungen und Fahrzeugprojekten zu schaffen, werden Fahrzeugvarianten beschrieben und den Systemanforderungen zugewiesen. Das obere Schema „Fahrzeugvarianten" zeigt, wie die Beschreibung einer Fahrzeugvariante aussehen kann. Es existieren vordefinierte Dimensionswerte (*AttrNamen*) in einer festen Reihenfolge [1]. Jeder Dimension kann eine Reihe von Attributwerten (*attrWerte*) zugewiesen werden [2]. Dazu wird jeder Fahrzeugvariante ein Realisierungsattribut zugeordnet [3]. Die Attributwerte werden in einer durch die Attributnamen festgelegten Sequenz angeordnet [8]. Am Ende der Sequenz wird das Realisierungsattribut hinzugefügt [9].

Die nachfolgende Abbildung zeigt ein Beispiel für zwei Fahrzeugvarianten, mit den Dimensionen „Motorvariante", „Karosserievariante" und „Länderausstattung".

AttrNamen	Motorvariante	Karosserievariante	Länderausstattung	Realisierung
FzvListe	3 Zylinder	Cabrio	Europa	Serie
	5 Zylinder	Limousine	USA	SA

Abbildung 9: Fahrzeugvarianten

4.2.2 Die Anforderungsliste

∪ *AnforderungsListe*_____
→*InfoKnoten:* Π *INFO_K*
→*needs: INFO_K* ♣ Π *INFO_K*
→*excludes: INFO_K* ♣ Π *INFO_K*
∩_____
→dom *needs* = dom *excludes* ζ *InfoKnoten* *[1]*
→ran *needs* = ran *excludes* ζ Π *InfoKnoten* *[2]*
→A*x: InfoKnoten* ∞ *x* ™ *excludes x f x* ™ *needs x* *[3]*
→A*x, y: InfoKnoten* ∞ *y* ε *needs x* ⇒ *y* ™ *excludes x* *[4]*
→A*x, y, z: InfoKnoten* ∞ *y* ε *needs x f z* ε *needs y* ⇒ *z* ε *needs x* *[5]*
→A*x, y, z: InfoKnoten* ∞ *y* ε *needs x f z* ε *excludes y* ⇒ *z* ε *excludes x* *[6]*
∠_____

Das Schema *AnforderungsListe* beschreibt die Menge aller Anforderungen (*InfoKnoten*), sowie die Eigenschaften der needs- und excludes-Beziehungen zwischen den Anforderungen. Die in dem Schema festgelegten Einschränkungen lauten:

- Eine Anforderung kann sich nicht selbst ausschließen. [3]
- Es darf nicht zwischen zwei Anforderungen gleichzeitig eine needs- und eine excludes-Beziehung bestehen. D. h. eine Anforderung kann nicht gleichzeitig eine andere erfordern und ausschließen. [4]
- Die needs-Beziehung ist transitiv. [5]
- Die Komposition von needs- und excludes-Beziehungen führt zu einer excludes-Beziehung.
  Beispiel:
  Wenn gilt: Anf1 needs Anf2 und Anf2 excludes Anf3, dann muss gelten: Anf1 excludes Anf3. [6]

In solch eine Anforderungsliste können die Anforderungen ohne große Mühe verwaltet werden. Sie können mit wenig Aufwand gelöscht und eingefügt werden. Auch die Beziehungen zwischen den Anforderungen können sehr leicht gepflegt werden. Die Aktivitäten auf der Anforderungsliste werden nun mit Hilfe der Z-Spezifikation formal beschrieben.

- **Hinzufügen von Anforderungen**

∪ *AnforderungHinzufügen* _____
→Δ*AnforderungsListe*
→*neueAnforderung?: INFO_K*
∩_____

→*neueAnforderung?* ™ *InfoKnoten* *[1]*
→*neueAnforderung?* ε *InfoKnoten'* *[2]*
∠_____

Bei der Initialisierung der Anforderungsliste wird eine leere Liste angelegt, in der die Anforderungen nacheinander eingefügt werden können. Vor der Aktivität befindet sich die Anforderung nicht in der Menge der Anforderungen [1]. Nach durchführen der Aktivität ist die Anforderung Teil der Anforderungsmenge [2]. Man kann die Anforderungen ohne Rücksichtnahme auf ihre Beziehungen zu anderen Anforderungen in die Liste einfügen. Auch die Strukturierung der Anforderungen kann erst zum späteren Zeitpunkt passieren. Um eine Anforderung direkt unter einem Strukturierungsknoten einzufügen, muss ein Link zum gewünschten Strukturierungsknoten gezogen werden, und die folgende Operation (*AnforderungHinzufügen* ∧ *Setze_Link*) ausgeführt werden.

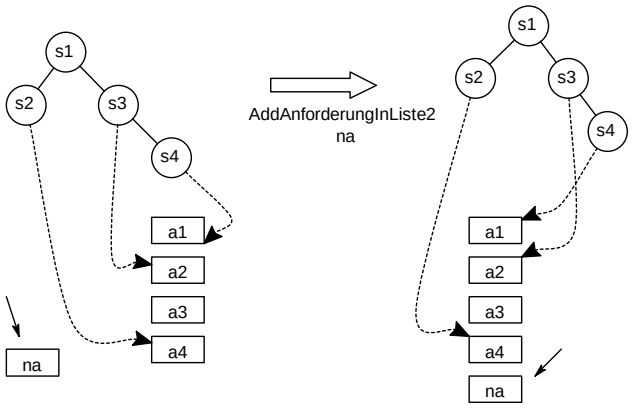

Abbildung 10 : Hinzufügen einer Anforderung (*neueAnforderung?=na*).

- **Strukturierung der Anforderungen**

∪_*Setze_Link* _____
→Δ*Kundenanforderungsbaum*
→Ξ*AnforderungsListe*
→*src?: STRUKT_K*
→*tg?: INFO_K*
∩_____
→A*x: StruktKnoten* ∞ *tg?* ™ *tgLinks x f src?* ε *StruktKnoten f tg?* ε *InfoKnoten* *[1]*
→*tgLinks' src? = tgLinks src?* Y {*tg?*} *[2]*
∠_____

Das Schema *Setze_Link* beschreibt das Anlegen einer Verbindung (Link) zwischen einem Strukturierungsknoten und einer Anforderung. Die Operation verursacht Veränderungen im Strukturierungsbaum und möglicherweise auch in der Anforderungsliste (wenn needs- und excludes-Beziehungen betroffen sind). Ein Link wird von einem Strukturierungsknoten (*scr?*) zu einem Informationsknoten (*tg?*) kreiert. Dabei darf zwischen einem Strukturknoten und einer Anforderung nur ein Link existieren [1]. Weiterhin darf jede Anforderung nur einem Strukturknoten zugewiesen sein.

- **Setzen von excludes-Beziehungen**

∪_*AddExcludesRelation* _____
→Δ*AnforderungsListe*
→*src?, tg?: INFO_K*
∩_____
→*src?* ε *InfoKnoten*
→*f tg?* ε *InfoKnoten*
→*f tg?* ™ *excludes src?*
→*f src?* ™ *excludes tg?*
→*f tg?* ™ *needs src?*
→*f src?* ™ *needs tg?*
→*excludes' src? = excludes src?* Y {*tg?*}
∠_____

Das obere Schema besagt, dass das Einfügen einer excludes-Beziehung zwischen zwei Anforderungen nur dann erfolgt, wenn:

- keine excludes-Beziehung zwischen den Anforderungen bereits existiert, die in die gleiche Richtung deutet.
- keine needs-Beziehung zwischen den Anforderungen existiert.

- **Setzen von needs- Beziehungen**

∪_*AddNeedsRelation*_____
→Δ*AnforderungsListe*
→*src?, tg?: INFO_K*
∩_____
→*src?* ε *InfoKnoten* *[1]*
→*ƒ tg?* ε *InfoKnoten* *[2]*
→*ƒ tg?* ™ *excludes src?* *[3]*
→*ƒ tg?* ™ *needs src?* *[4]*
→*ƒ src?* ™ *excludes tg?* *[5]*
→*needs' src? = needs src?* Y {*tg?*} *[6]*
∠_____

Die Vorbedingungen für das Hinzufügen einer needs-Beziehung zwischen zwei Anforderungen lauten:

- Die Anforderungen schließen sich bislang gegenseitig noch nicht aus [3], [5]. Es würde ein Widerspruch existieren, wenn eine Anforderung eine andere Anforderung gleichzeitig benötigt (needs) und ausschließt (excludes).
- Zwischen zwei Anforderungen darf in jede Richtung nur eine needs-Beziehung existieren [4], [6].

5 Zusammenfassung

Dieser Bericht stellt im ersten Abschnitt eine am Fraunhofer Institut für Software- und Systemtechnik entwickelte Methodik des Anforderungsmanagements vor. Der Ansatz ermöglicht eine methodische Erfassung sowie eine effiziente Bearbeitung und die sichere Kontrolle der Anforderungen während der Entwicklung eines Projektes. Dafür wird für das Requirements Engineering jeweils ein Strukturierungs- und ein Vorgehensmodell im Bereich des Application- und des Domain Engineering beschrieben. Der Bericht begründet die Notwendigkeit einer formalen Spezifikation des Ansatzes und stellt verschiedene mögliche Mittel dazu vor.

Auch wenn die Graphentheorie genug Ausdrücke zur Darstellung mathematischer Strukturen aufweist, ist sie nicht wirklich für die Beschreibung der Aktivitäten, die zum Aufbau und zur Pflege der Modelle in diesem Ansatz durchgeführt werden müssen, geeignet. Es ist z.B. nicht möglich mittels der Graphentheorie die Aktivitäten, die eine Zustandsänderung einer Struktur hervorrufen, präzise zu beschreiben, da die Graphentheorie keine Mittel besitzt, mit denen eine kontinuierliche Zustandsänderung beschrieben werden kann. So kann z.B. bei einer Komposition mehrerer Aktivitäten die zeitliche Abfolge der verschiedenen Aktivitäten nicht festgelegt werden, was für unser Vorgehen nicht vorteilhaft ist.

Eine weitere Möglichkeit der formalen Spezifikation stellt die Aussagenlogik dar. Hier werden die Datenstrukturen wie Bäume durch Signaturen beschrieben [EMCG+98]. Die Signaturen beinhalten nur die Benennung von Sorten und Operationssymbolen auf einer syntaktischen Ebene, während Datenstrukturen die zugehörigen semantischen Modelle sind. Nach einem erfolglosen Versuch ein semantisches Modell aufzubauen, müssen wir feststellen, dass dieser Weg das Problem ebenfalls nicht löst. Es reicht nicht, eine Datenstruktur „Baum“ zu definieren, denn sie legt nur die Syntax fest, sodass das Risiko, dass die Aussagen bei jeder möglichen Interpretation verschiedene Wahrheitswerte erhalten, zunimmt. Um diese Probleme zu lösen, muss eine Sprache verwendet werden, die die Eindeutigkeit der Wahrheitswerte jeder Aussage garantiert.

Die Z-Notation erfüllt diese Bedingungen. Damit werden die Zustände der Datenstrukturen und die Operationen auf den Datenstrukturen mit einem sehr befriedigenden Ergebnis formal beschrieben. Das Verständnis einer solchen Spezifikationen setzt allerdings eine rigorose Interpretation der Quantifikatoren und der logischen Verknüpfungen voraus.

Literatur

[CGM79] Chachra, V.; Ghare, PM.; Moore, JM.; Application of Graph Theorie Algorithms; North Holland; New York 1979

[NST96] Nägler, G.; Stopp, F.; Teubner, BG; Graphen und Anwendungen; Verlaggesellschaft Stuttgart Leipzig 1996

[EMCG+98] Ehrig, H.; Mahr, B.; Cornelius, F.; Große-Rhode, M.; Zeitz, B.; Mathematisch-strukturelle Grundlagen der Informatik; Springer Verlag, Berlin, Heidelberg, New-York, 1998

[BFGH02] Berthomieu, C.; Feldo, M.; Große-Rhode, M.; Hardt, M. ; Lindow, S.; Mackenthun, R.; Webers, W.: Anforderungsmodellierung und Anforderungsmanagement im Domain-Engineering; Technischer Bericht des Fhg-ISST Berlin, 2002, erscheint demnächst

[Spi92] Spivey, J.M.; „The Z Notation: A Reference Manual, Second Edition“ ; Oriel College, Oxford, OX1 4EW, England; 1992