



U N I V E R S I T Ä T
K O B L E N Z · L A N D A U

Fachbereich 4: Informatik

Konfiguration von Feature-Modellen durch Abbildung auf Goal-Modelle mithilfe von Beschreibungs-Logik

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von

Eduard Schleining

Betreuer: Diplom-Informatiker Gerd Gröner
Institute for Web Science and Technologies, Fachbereich 4:
Informatik

Dozent: Professor Doktor Steffen Staab
Institute for Web Science and Technologies, Fachbereich 4: In-
formatik

Koblenz, im Juli 2011

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Koblenz, den 28. Juli 2011

Eduard Schleining

Danksagung

An dieser Stelle bedanke ich mich bei allen Menschen, die mich bei der Erstellung der Bachelorarbeit unterstützt haben. Insbesondere möchte ich Gerd Gröner danken, der immer bereit war mir mit hilfreichen Kommentaren, Anregungen und Ideen zur Seite zu stehen. Zusätzlich bedanke ich mich bei Mark Schneider, Thomas Marx, Ramona Würstle und Christoph Ersfeld, die sich die Zeit genommen haben, die Bachelorarbeit Korrektur zu lesen. Des weiteren bedanke ich mich bei Mohsen Asadi, Bardia Mohabbati, Dragan Gašević und Fernando Silva Parreiras für die konstruktiven Telefon-Konferenzen, bei denen stets gute Ideen für die Arbeit entstanden sind.

Kurzdarstellung

Eine Software-Produkt-Linie ist eine Familie von Software-Produkten, die einige Funktionalitäten teilen. Diese Familien werden mithilfe von Feature-Modellen repräsentiert. Ein Feature-Modell kapselt die auftretenden Abhängigkeiten unter den einzelnen Funktionalitäten und repräsentiert alle Produkte, die aus dieser Familie hervorgehen können. Während der Entwicklung eines konkreten Produktes aus dieser Familie ist notwendig Anforderungen an das Produkt zu erheben, sodass entschieden werden kann, welche Funktionalitäten dieses beinhalten muss. Um die Anforderungen an ein Produkt zu modellieren werden Ziele und Aufgaben, die das zu entwickelnde Produkt erfüllen soll, in einem Goal-Modell repräsentiert, welches die Abhängigkeiten der einzelnen Ziele untereinander widerspiegelt. Die beiden Modelle abstrahieren das zu entwickelnde Produkt aus verschiedenen Sichtweisen. Das Feature-Modell bietet eine technische Sicht, während das Goal-Modell eher aus der Perspektive der Produkt-Nutzer entsteht. Wir werden in dieser Arbeit zeigen, wie die Konfiguration eines Feature-Modells mithilfe eines Goal-Modells vorgenommen werden kann, sodass beide Perspektiven abgedeckt werden. Beide Modelle werden zu diesem Zweck in einer Beschreibungs-Logik Wissensbasis beschrieben, auf deren Grundlage dann Abbildungen (Mappings) zwischen Elementen der beiden Modelle untersucht werden.

Inhaltsverzeichnis

1	Einleitung	15
2	Motivation	17
2.1	Einführung	17
2.2	Problem	19
3	Grundlagen	23
3.1	Feature-Modelle	23
3.2	Goal-Modelle	25
3.3	Beschreibungs-Logik	28
3.4	Reasoning	31
4	Transformation der Modelle	33
4.1	Transformation des Goal-Modells	33
4.2	Transformation des Feature-Modells	38
4.3	Auswertung der Mappings	41
5	Konfiguration des Feature-Modells	49
5.1	Beschriftung	49
5.2	Konfiguration	53
6	Abschluss	61
6.1	Evaluierung	61
6.2	Verwandte Arbeit und Diskussion	63
6.3	Zusammenfassung	64

Abbildungsverzeichnis

2.1	Ein Ausschnitt aus dem übersetzten eShop Feature-Modell.	18
2.2	Ein Ausschnitt aus dem übersetzten eShop Goal-Modell.	19
2.3	Benötigt-Einschränkung verletzt.	20
2.4	Obligatorische Features nicht übernommen.	20
3.1	Das Meta-Modell für Feature-Modelle dargestellt in UML.	23
3.2	Das Metamodell für Goal-Modelle dargestellt in UML.	26
4.1	Ein Beispiel für ein immer ungültiges Mapping.	43
4.2	Änderungen an den Modellen und Mappings.	48
5.1	Beschriftung des Beispiels aus Abbildung 2.2.	52
5.2	Entfernte Features aufgrund von Mappings.	57
5.3	Konfiguration des Feature-Modells aus Abbildung 2.1.	57

Tabellenverzeichnis

3.1	<i>ALC</i> Beschreibungs-Logik.	29
3.2	Regeln für <i>ALC</i> Modelle.	30
4.1	Mapping Validierung eines Features f zu einem Element g	43
4.2	Gültigkeit der Mappings aus Abbildung 2.3.	46
4.3	Gültigkeit der Mappings aus Abbildung 2.4.	46
5.1	Begründung für die Beschriftung.	53
5.2	Begründung für die Konfiguration.	58

Kapitel 1

Einleitung

Ein wichtiges Paradigma in der Software-Entwicklung ist das der Software-Produkt-Linie (engl. Software Product Line, Abk. SPL). Die einzelnen Produkte der Linie teilen sich dabei eine Menge von Funktionalitäten (Features) [PBL05], wodurch die Wiederverwendbarkeit in der Produktentwicklung steigt und die Entwicklungskosten für einzelne Produkte sinken [MMYJ10]. Feature-Modelle beschreiben die Menge der gemeinsamen Features in einer SPL und werden im Laufe der SPL Entwicklung erarbeitet. Sie enthalten technisches Wissen über die Struktur und Funktionsweise einer SPL.

In der frühen Software-Entwicklungsphase, in der die Ziele und Voraussetzungen des Wunsch-Systems erarbeitet werden, ist wenig technisches Wissen über das Wunsch-System bekannt, daher werden in dieser Phase nach dem GORE (Goal Oriented Requirements Engineering)[MCY99] Paradigma Goal-Modelle erarbeitet. Ein Goal-Modell beschreibt Benutzerwünsche und Gestaltungsalternativen des Ziel-Produktes auf einer hohen Abstraktionsebene, ohne auf technische Details einzugehen.

Ein Produkt der SPL muss die technischen Anforderungen, die aus dem Feature-Modell hervorgehen, einhalten und die Ziele und Aufgaben, die durch das Goal-Modell an das Produkt gestellt werden, erfüllen. Daher ist es notwendig beide Modelle während der Produkt-Entwicklung einzubeziehen.

Es gibt Literatur, welche sich mit dem GORE Paradigma auseinandersetzt, um eine SPL zu entwickeln [YPLLM08, YLL⁺08, LYM07], jedoch fehlt dort die Sicht auf die technische Realisierung des Produktes.

Wir stellen eine Beziehung zwischen den beiden Modellen her, indem wir

Abbildungen (im Folgenden Mappings genannt) zwischen den Modellen erzeugen. Eine solche Abbildung definiert, dass ein bestimmtes Element aus dem Goal-Modell durch ein Feature aus dem Feature-Modell realisiert wird. Mappings können ungültig sein, wenn z.B. ein Feature abhängig ist von einem anderen aber die Ziele, auf die diese Features abgebildet werden, sich gegenseitig ausschließen.

Diese Arbeit beschäftigt sich mit der Auswertung und Fehler-Erkennung von Abbildungen zwischen Feature-Modellen und Goal-Modellen mithilfe von Beschreibungs-Logik (engl. Description Logic im Folgenden DL)[BCM⁺03] und den gängigen Reasoning Verfahren. Dies ist nicht trivial möglich, da die beiden Modelle verschiedenartige Abhängigkeiten beschreiben. Auf der einen Seite werden die Feature-Modelle betrachtet, welche Details bezüglich der technischen Realisierung des Produktes enthalten, dem gegenüber stehen die Goal-Modelle, welche durch Ziele und Abhängigkeiten aus Benutzer-Sicht modelliert werden.

Die Validierung eines Mappings muss vorgenommen werden, um Fehler in der Beziehung zwischen Goal- und Feature-Modellen frühzeitig zu erkennen und zu korrigieren. Zusätzlich kann so bestimmt werden, ob es möglich ist, ein gegebenes Ziel-System (Goal-Modell) mithilfe eines Produktes aus der SPL (Feature-Modell) zu realisieren.

Diese Bachelorarbeit ist folgendermaßen gegliedert: Im nächsten Kapitel (Kapitel 2) stellen wir ein Feature- und ein Goal-Modell mit den zugehörigen Mappings vor und verdeutlichen die Problematik im Detail. Die Abschnitte 3.1 und 3.2 beschreiben, wie Feature-Modelle und Goal-Modelle aufgebaut sind. Eine Einführung in Beschreibungs-Logik und Reasoning Mechanismen wird im Abschnitt 3.3 gegeben. Darauf folgt Kapitel 4, in dem die Transformation der beiden Modelle und der Mappings in eine Darstellung der zuvor erwähnten Beschreibungs-Logik erläutert wird. Die Validierung der Mappings beschreibt der Abschnitt 4.3. Im Kapitel 5 wird darauf eingegangen, wie aus den Mappings und Modellen eine Konfiguration entsteht. Im Kapitel 6 evaluieren wir unseren Ansatz, geben eine Übersicht über ähnliche Arbeiten und fassen die Bachelorarbeit zusammen.

Kapitel 2

Motivation

2.1 Einführung

Der Fokus dieser Arbeit liegt darin, zwei verschiedene Modelle aus den zwei Phasen der Software-Produkt-Linien-Entwicklung zu vereinen. Die erste Phase ist die Domänen Entwicklung (engl. Domain Engineering).

Im Laufe des Domain-Engineering Prozesses werden gleichartige Bestandteile der Software-Produkt-Linie identifiziert. Es wird untersucht, welche Funktionalitäten jedes Produkt der Linie bieten muss. Zusätzlich suchen Domain-Engineers nach Punkten, in denen sich einzelne Produkte unterscheiden können (sog. Variation-Points). Das resultierende Modell der Software-Domain ist eine Beschreibung aller möglichen Produkte, die aus dieser Produkt-Linie hervorgehen können. Dieses Modell wird Feature-Modell genannt, da es alle Funktionen einer Produktlinie beschreibt.

In Abbildung 2.1 wird ein Ausschnitt aus dem eShop Feature-Modell dargestellt. Es werden die funktionalen Abhängigkeiten des Bestellprozesses beschrieben. Die Software muss Funktionalitäten implementieren, die sich mit der Bestellvorbereitung befassen. Bestellungen müssen angenommen und bestellte Waren erzeugt werden. Dürfen beliebige Kunden in dem eShop einkaufen, muss die Software zusätzlich Funktionen zur Kundenverifizierung bereitstellen.

Aus einem Feature-Modell, das aus n Features besteht, lassen sich maximal 2^n konkrete Produkte realisieren. Daher gilt es während der Entwicklungsphase des Produktes eine Verfeinerung des Feature-Modells vorzunehmen. Bei einer Verfeinerung werden nicht benötigte Features aus dem ursprünglichen

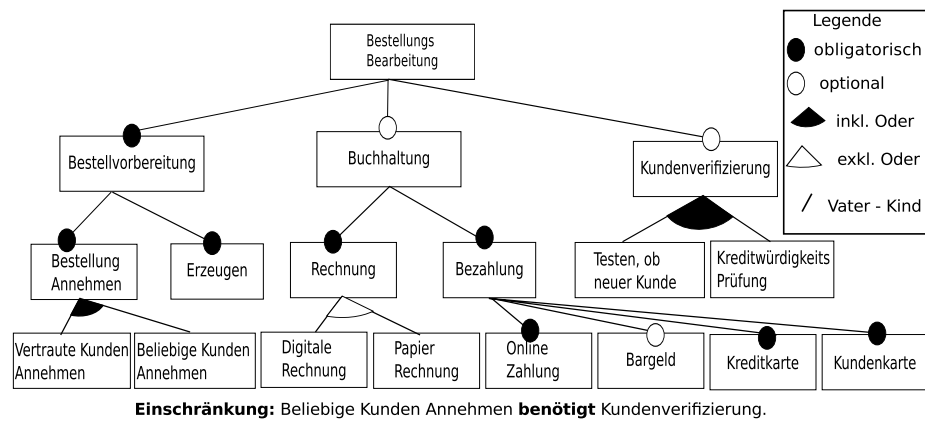


Abbildung 2.1: Ein Ausschnitt aus dem übersetzten eShop Feature-Modell.

Feature-Modell entfernt, sodass der Such-Raum für ein konkretes Produkt eingeschränkt werden kann.

Die zweite Phase, mit der wir uns auseinandersetzen, nennt sich Applikations-Entwicklung (engl. Application Engineering). In dieser Phase werden Anforderungen an ein Software Produkt erhoben. Dazu wird untersucht, was das zu entwickelnde Produkt leisten muss, also welche Ziele mit der Applikation erreicht werden sollen. Es wird in dieser Phase nach dem GORE (Goal Oriented Requirements Engineering) Paradigma vorgegangen. Das resultierende Modell beschreibt die Abhängigkeiten der einzelnen Ziele (engl. Goals) von einander und wird daher Goal-Modell genannt. Diese werden von uns genutzt, um ein konkretes Produkt aus einem Feature-Modell ableiten zu können. Der Prozess des Ableitens eines Produktes aus einem Feature-Modell kann schrittweise geschehen, indem in jedem Schritt bestimmte, für das Produkt nicht benötigte Features aus dem Feature-Modell entfernt werden. Daher wird dieser Vorgang auch Schrittweise-Konfiguration genannt. Ob ein Produkt ein Feature benötigt oder nicht, wird mithilfe des zugrunde liegenden Goal-Modells entschieden.

Abbildung 2.2 zeigt einen übersetzten Ausschnitt aus einem Goal-Modell für einen eShop. Dieser Ausschnitt beschreibt die Zerlegung des Ziels „Prozessabfolge“. Der Betreiber des eShops kann sich dazu entscheiden, zunächst die bestellten Waren zu erzeugen, um dann während der Lieferung die Abrechnung mit seinem Kunden vorzunehmen. Dies trägt positiv zur Zufriedenheit seiner Kunden bei, da sie die Ware erst bei Eintreffen bezahlen müssen. Eine Alternative hierzu ist, die Waren zu erzeugen, gleichzeitig die Abrechnung

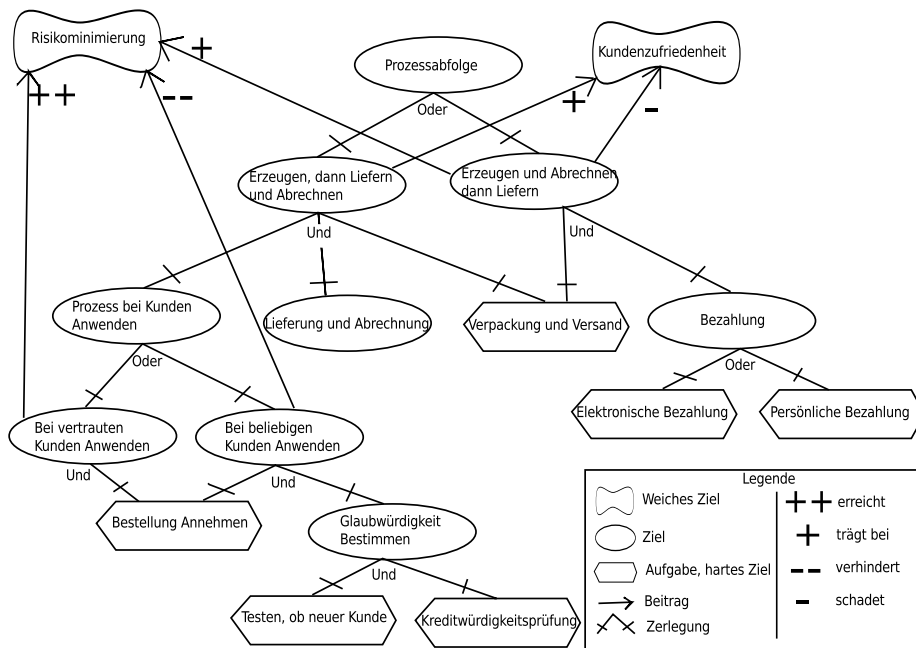


Abbildung 2.2: Ein Ausschnitt aus dem übersetzten eShop Goal-Modell.

vorzunehmen und erst dann auszuliefern. Dies hätte zwar negative Auswirkungen auf die Kunden-Zufriedenheit, würde zugleich aber auch das Risiko betrogen zu werden minimieren.

2.2 Problem

Goal-Modelle dienen in der Applikations-Entwicklung dazu, ein Produkt aus dem in der Domänen-Entwicklung erarbeiteten Feature-Modell abzuleiten. Zu diesem Zweck definieren wir Abbildungen zwischen Feature- und Goal-Modell. Die Auswahl der Features, die im konfigurierten Modell verbleiben, geschieht durch eine Auswahl von Goals (Features, welche auf „gewünschte“ Goals abgebildet wurden, verbleiben im konfigurierten Modell). Da das Goal-Modell andere Abhängigkeiten beschreibt als das Feature-Modell, können durch eine Goal-Basierte Feature-Auswahl Konfigurationen entstehen, die die Abhängigkeiten des ursprünglichen Feature-Modells nicht erfüllen (ungültige Konfigurationen). Mappings, die zu einer ungültigen Konfiguration führen, bezeichnen wir als inkonsistent. Zur Veranschaulichung geben wir zwei Beispiele für inkonsistente Mappings.

Abbildung 2.3 zeigt ein inkonsistentes Mapping, das durch die Verletzung

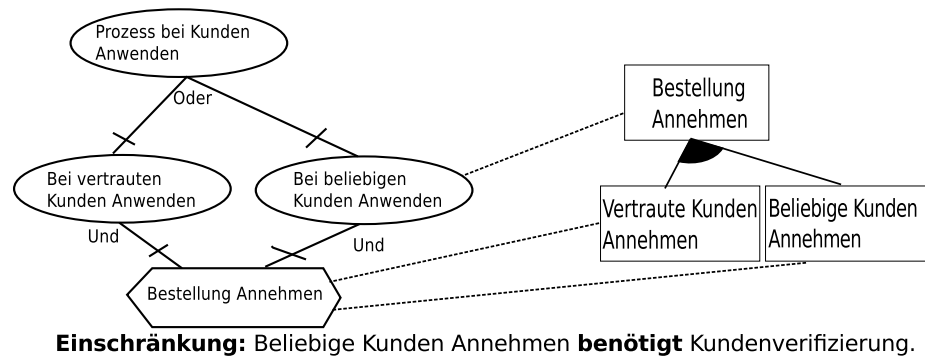


Abbildung 2.3: Benötigt-Einschränkung verletzt.

einer Benötigt-Einschränkung hervorgerufen wird. Die Aufgabe „Bestellung Annehmen“ des Goal-Modells wird durch die Features „Vertrauten Kunden Annehmen“ und „Beliebigen Kunden Annehmen“ realisiert. Das heißt, sobald die Aufgabe „Bestellung Annehmen“ erfüllt werden soll, müssen sowohl das Feature „Vertrauten Kunden Annehmen“ als auch das Feature „Beliebige Kunden Annehmen“ implementiert werden. Das Feature-Modell ist dadurch eingeschränkt, dass „Beliebige Kunden Annehmen“ das Feature „Kundenverifizierung“ benötigt. Das Goal-Modell stellt jedoch nicht sicher, dass durch „Bestellung Annehmen“ ein Ziel vorhanden ist, das durch „Kundenverifizierung“ erreicht wird. Bei der Konfiguration des Feature-Modells kann also aufgrund dieses inkonsistenten Mappings ein Feature-Modell entstehen, welches die Abhängigkeiten des ursprünglichen Feature-Modells nicht erfüllt.

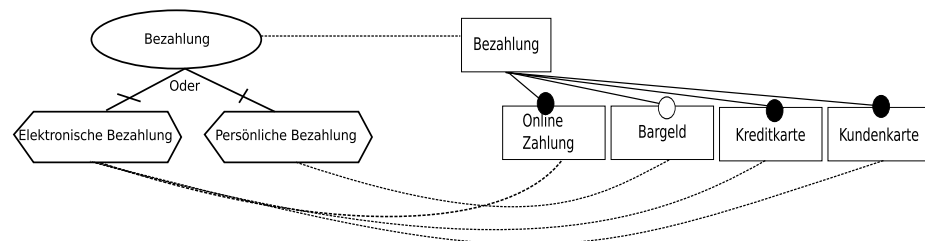


Abbildung 2.4: Obligatorische Features nicht übernommen.

Abbildung 2.4 gibt ein Beispiel für ein inkonsistentes Mapping aufgrund der Verletzung einer obligatorischen Vater-Kind-Beziehung im Feature-Modell. Es werden die Mappings zwischen dem Goal und dem Feature „Bezahlung“ und ihren Kindern dargestellt. Die Features „Kreditkarte“, „Kundenkarte“ und „Online Zahlung“ werden auf die Aufgabe „Elektronische Bezahlung“ abgebildet während eine Barzahlung die Aufgabe „Persönliche Bezahlung“ realisiert.

Die Anwendungs-Entwickler können sich dazu entschließen, nur die persönliche Bezahlung der Waren zuzulassen und somit auf sämtliche obligatorische Kinder des Bezahl-Features zu verzichten. Dies hat zur Folge, dass die Konfiguration die Abhängigkeiten des ursprünglichen Feature-Modells nicht erfüllt.

Wir stellen einen Mechanismus vor, mit dessen Hilfe Inkonsistenzen zwischen Feature-Modell und Goal-Modell gefunden werden. Zu diesem Zweck definieren wir für das Feature- und Goal-Modell eine Wissensbasis mit einer Beschreibungs-Logik. Dieser Basis fügen wir Mapping-Axiome hinzu, die beschreiben, wie die Abbildungen zwischen Feature- und Goal-Modell festgelegt sind. Sind diese beiden Schritte erfolgt, so können wir mithilfe von üblichen Reasoning Verfahren inkonsistente Mappings in unserer Wissensbasis identifizieren. Diese können von Hand beseitigt werden (durch Anpassen der Modelle oder der Mappings). Danach nehmen wir eine Beschriftung des Goal-Modells in der erzeugten Wissensbasis vor, anhand derer wir zu verwerfende von zu realisierenden Zielen unterscheiden. Die Beschriftung macht eine Konfiguration des Feature-Modells möglich, welche nur Features beinhaltet, die gewünschte Ziele realisieren.

Kapitel 3

Grundlagen

3.1 Feature-Modelle

Wir bezeichnen eine Funktionalität eines Produktes als Feature. Ein Feature-Modell beschreibt die gegenseitigen Abhängigkeiten der einzelnen Features einer SPL. Dieser Abschnitt stellt ein Meta-Modell für Feature-Modelle vor und erläutert dessen Semantik. Zum Schluss wird ein Feature-Modell formal definiert.

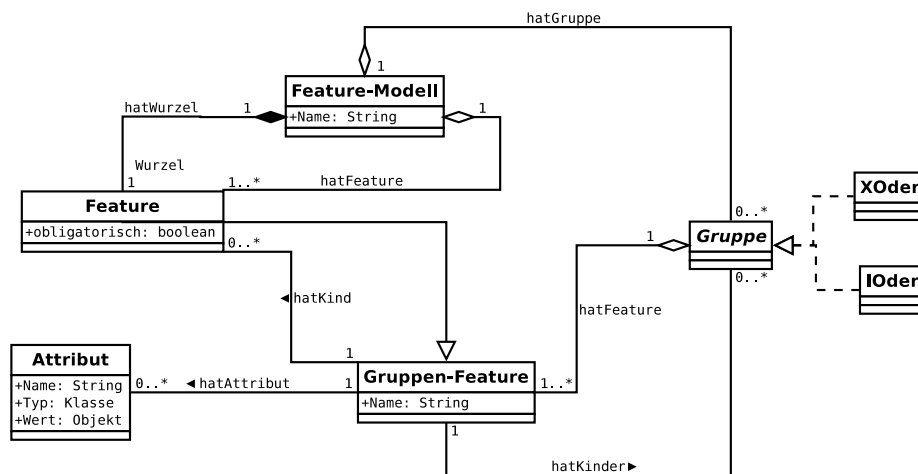


Abbildung 3.1: Das Meta-Modell für Feature-Modelle dargestellt in UML.

Abbildung 3.1 zeigt das UML Meta-Modell für Feature-Modelle. Ein Feature-Modell ist eine Baum-Struktur, bei der das Wurzel-Element ein Feature ist. In Abbildung 2.1 ist die Wurzel das Feature „Bestellungs Bearbeitung“.

Ein Feature kann in Kind-Features unterteilt werden, um eine feine Granularität von Funktionalitäten zu schaffen. Unser Beispiel in Abbildung 2.1 unterteilt die Wurzel „Bestellungs Bearbeitung“ in das obligatorische Feature „Bestellung Annehmen“ und die zwei optionalen Features „Buchhaltung“ und „Kundenverifizierung“.

Ist ein Feature obligatorisch, so muss es in allen Produkten enthalten sein, die sein Vater-Feature beinhalten. In unserem Fall setzt jedes Produkt, das eine Funktionalität zur Bestell-Vorbereitung liefert, auch eine Funktionalität zur Bestellungen-Annahme voraus. Optionale Features können implementiert werden, wenn das entsprechende Vater-Feature im Produkt enthalten ist, um die Funktionalität des Produktes zu erhöhen.

Zusätzlich können Kinder-Features in verschiedene Arten von Gruppen angeordnet werden, die definieren, wie viele Kinder die Produkte minimal und maximal umfassen müssen. In Abbildung 2.1 sind beispielsweise die Kinder des Features „Kunden-Verifizierung“ in einer *IOder*-Gruppe gegliedert. Eine solche Gruppe definiert, dass mindestens eines der Gruppen-Features in einer Konfiguration enthalten sein muss, wenn das Vater-Feature dort vorhanden ist.

Eine andere Gruppe ist in Abbildung 2.1 durch die Kinder des Features „Rechnung“ gegeben. Dort muss genau eines der Features „Digitale Rechnung“ und „Papier Rechnung“ in einem Produkt implementiert werden, wenn dieses Produkt eine Rechnungs-Funktionalität bietet. Eine solche Gruppe bezeichnen wir im Folgenden als *XOder*-Gruppe.

Zusätzlich zu den bereits aufgelisteten Beziehungen zwischen den einzelnen Features ist es notwendig, Einschränkungen der Produktlinie zu definieren, die von einer Baum-Struktur nicht ausgedrückt werden. Beispielsweise muss im Feature-Modell aus Abbildung 2.1 ein eShop, in dem beliebige Kunden Waren bestellen dürfen, eine Funktionalität zur Verifizierung der Kunden bieten. Solche Einschränkungen werden durch Attribute modelliert.

Ein Attribut bezieht sich immer auf genau ein Feature und hat einen Namen. In dieser Arbeit beschränken wir uns auf zwei Arten von Attributen, die durch ihre Namen unterschieden werden:

Das *benoetigt*-Attribut: Dieses Attribut legt fest, dass sein Bezugs-Feature ein anderes Feature benötigt. Demnach ist der Typ dieses Attributs die Klasse Feature und das Ziel ist ein Feature aus dem Feature-Modell. In Abbildung 2.1

benötigt das Bezugs-Feature „Beliebige Kunden Annehmen“ das Ziel-Feature „Kundenverifizierung“.

Das *schließtAus*-Attribut: Dieses legt den gegenseitigen Ausschluss von zwei Features fest. Es hat als Typ die Klasse Feature und als Ziel das Feature, welches von seinem Bezugs-Feature ausgeschlossen wird.

Ebenso erzeugen wir Mappings zwischen Elementen des Goal-Modells und Features durch die Angabe von Mapping-Attributen im Feature-Modell.

Definition 1 Ein Feature-Modell ist die Menge $\{\mathcal{F}, \mathcal{F}_{obl}, \mathcal{F}_{opt}, \mathcal{F}_{group}, \mathcal{R}_{VK}, \mathcal{R}_{VG}, \mathcal{A}\}$. \mathcal{F} ist die Menge aller Features eines Feature-Modells. $\mathcal{F}_{obl} \subseteq \mathcal{F}$ ist diejenige Teilmenge aus \mathcal{F} , die nur alle obligatorischen Features des Feature-Modells enthält. $\mathcal{F}_{opt} \subseteq \mathcal{F}$ ist diejenige Teilmenge aus \mathcal{F} , die nur alle optionalen Features des Feature-Modells enthält. $\mathcal{F}_{group} = \mathcal{F} \setminus (\mathcal{F}_{opt} \cup \mathcal{F}_{obl})$ ist die Menge der in Gruppen angeordneten Features. Die Menge \mathcal{R}_{VK} enthält Tupel der Form $(\mathcal{F} \times \mathcal{F}_{obl} \cup \mathcal{F}_{opt})$ und definiert die Vater-Kind Beziehung zwischen Vater-Features aus \mathcal{F} zu seinen Kind-Features aus $\mathcal{F}_{obl} \cup \mathcal{F}_{opt}$. Die Menge \mathcal{R}_{VG} enthält Tupel der Form $(\mathcal{F} \times \{IOder, XOder\} \times 2^{\mathcal{F}_{group}})$ und definiert eine Gruppen Beziehung zwischen dem Vater-Feature aus \mathcal{F} und einer Potenz-Menge von \mathcal{F}_{group} , die genau die gruppierten Kind-Features enthält. Der Typ der Gruppe wird entweder durch *IOder* oder *XOder* angegeben. Durch die Menge \mathcal{A} , welche aus einer Menge von Tupeln $(\mathcal{F} \times String \times Klasse \times Wert)$ besteht, werden Attribute eines Features angegeben.

3.2 Goal-Modelle

Ein Goal-Modell ist eine Repräsentation von zu realisierenden Zielen eines bestimmten Systems. Es stellt sowohl die Zerlegung von Intentionen oder Vorhaben dar als auch deren Verknüpfungen. In diesem Abschnitt stellen wir ein Meta-Modell vor, welches diejenigen Teile von Goal-Modellen repräsentiert, die für unsere Arbeit relevant sind. Wir verwenden die in [IT08] eingeführten Definitionen für die einzelnen Elemente eines Goal-Modells.

Abbildung 3.2 repräsentiert ein Meta-Modell mit den für diese Arbeit relevanten Elementen eines Goal-Modells.

Ein Goal-Modell beinhaltet eine Anzahl von intentionalen Elementen. Diese werden typischerweise in drei Kategorien eingeteilt, namentlich sind das erstens *qualitative Ziele*, zweitens *Ziele* und drittens *Aufgaben*. In den folgenden Abschnitten werden diese drei Typen erläutert.

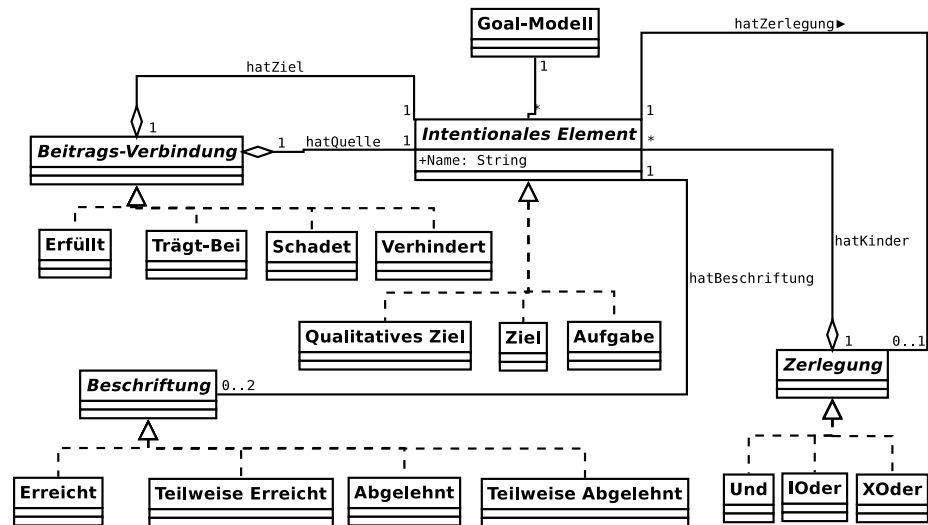


Abbildung 3.2: Das Metamodell für Goal-Modelle dargestellt in UML.

Qualitative Ziele beschreiben schwer zu messende und nicht-funktionale Qualitätsmerkmale des modellierten Systems. In unserem Beispiel-Modell (Abbildung 2.2) stellen „Kundenzufriedenheit“ und „Risikominimierung“ zwei solche Merkmale dar. Es ist plausibel, dass Kundenzufriedenheit nicht nur dadurch erreicht wird, dass der Kunde seine Ware erst bezahlen muss, wenn er sie in der Hand hält. Jedoch wird ein solcher Kunde leichter zufrieden zu stellen sein als jemand, der bereits die Waren bezahlt hat und noch einige Tage oder Wochen auf den Eingang dieser warten muss.

Ziele definieren Soll-Zustände der zu entwickelnden Anwendung. Unser Beispiel (Abbildung 2.2) gibt durch das Ziel „Bezahlung“ vor, dass die Anwendung irgendwann den Zustand „Bezahlung ist abgeschlossen“ erreichen muss, um das Ziel „Erzeugen und Abrechnen dann Liefern“ zu erfüllen (also den Zustand zu erreichen in dem die Waren abgeliefert und abgerechnet wurden).

Aus den Zielen leiten sich *Aufgaben* ab. Diese beschreiben den Weg zur Erfüllung eines Zieles. Wenn wir uns das Beispiel-Modell ansehen (Abbildung 2.2), erkennen wir, dass das Ziel „Bezahlung“ durch zwei Aufgaben erfüllt werden kann. Einerseits dadurch, dass der Kunde die Waren persönlich bezahlt („Persönliche Bezahlung“), andererseits durch eine elektronische Abwicklung innerhalb der Applikation („Elektronische Bezahlung“).

Einzelne intentionale Elemente können mit einer Beschriftung versehen werden, die Aussagen darüber trifft, wie gut das Element erfüllt wird. In [IT08] werden vier Beschriftungstypen vorgestellt (*erreicht*, *teilweise erreicht*, *teilweise abgelehnt* und *abgelehnt*). Die Bedeutung dieser Beschriftungen wird im Folgenden erläutert:

Erreicht (engl. satisfied): Die Beschriftung definiert, dass ein Ziel erreicht bzw. eine Aufgabe erfüllt wird.

Teilweise Erreicht (engl. partly satisfied): Diese Beschriftung sagt aus, dass ein Ziel teilweise erreicht oder eine Aufgabe teilweise erfüllt wird.

Teilweise Abgelehnt (engl. partly denied): Hierbei handelt es sich um die Aussage, dass das Ziel teilweise abgelehnt wird oder eine Aufgabe teilweise unerfüllt bleibt.

Abgelehnt (engl. denied): Ein abgelehntes Ziel wird nicht erreicht, eine abgelehnte Aufgabe wird nicht erfüllt.

Um Ziele und Aufgaben in Teile zu gliedern und so eine fein granulare Darstellung von Zielen zu erzeugen, gibt es die so genannten Zerlegungen (engl. decomposition). Diese haben einen von drei Typen: *Und*, *IOder* oder *XOder*. Ein Ziel, das durch eine *Und*-Zerlegung gegliedert wird, kann unter anderem erreicht werden, wenn alle seine Zerlegungs-Bestandteile erreicht wurden. Bei einer *IOder*-Zerlegung hingegen reicht es aus, wenn mindestens einer der Bestandteile erreicht wird, um das Ziel zu erfüllen. Entsprechend wird ein durch *XOder* zerlegtes Element erreicht, wenn genau ein Bestandteil der Zerlegung erreicht ist.

Es ist möglich, dass zwei intentionale Elemente in einer besonderen Beziehung zu einander stehen, bei der ein Element zur Erfüllung des anderen einen bestimmten (positiven oder negativen) Beitrag leistet. Dies wird durch die verschiedenen Beitrags-Verbindungen modelliert. Die *Erfüllt*-Verbindung beschreibt einen kompletten positiven Beitrag zur Erfüllung eines anderen Elementes, die *TraegtBei*-Verbindung sagt aus, dass ein Element teilweise positiv zu einem anderen Element korrespondiert. Negative Wechselwirkungen definieren wir durch die *Schadet*-(teilweise) und *Verhindert*-Verbindung (komplett). In unserem Modell (Abbildung 2.2) wird durch eine Einschränkung auf vertraute Kunden beispielsweise die „Risikominimierung“ erreicht, während das Zulassen von beliebigen Kunden eine Ablehnung dieses Qualitätsmerkmals zur Folge hat.

Definition 2 Ein Goal-Modell ist die Menge $\{\mathcal{G}, \mathcal{V}, \mathcal{Z}\}$. \mathcal{G} stellt die Menge aller intentionalen Elemente innerhalb des jeweiligen Modells dar. Die Menge \mathcal{V} enthält Relationen $(\mathcal{G} \times \{++, +, -, --\} \times \mathcal{G})$ und spiegelt die Verbindungen zwischen den Elementen des Goal-Modells wieder. Dabei steht $++$ für die Erfüllt-, $--$ für die Verhindert-Verbindung. Entsprechend beschreibt $+$ ($-$) eine Trägt-Bei (Schadet)-Verbindung. Die Menge \mathcal{Z} wird durch Relationen der Form $(\mathcal{G} \times \{IOder, XOder, Und\} \times 2^{\mathcal{G}})$ angegeben und definiert die Zerlegung eines intentionalen Elementes des Goal-Modells durch den Typ *IOder*, *XOder* oder *Und* in eine Potenzmenge von intentionalen Elementen $2^{\mathcal{G}}$.

3.3 Beschreibungs-Logik

Wir haben in den vorherigen Abschnitten zwei Arten von Modellen vorgestellt (Feature- und Goal-Modell), die im Fokus dieser Arbeit stehen. Um die Konfiguration des Feature-Modells vorzunehmen, haben wir uns entschlossen, diese Modelle zusammen mit den Mappings in eine Wissensbasis zu transformieren, die durch Beschreibungs-Logik formalisiert ist. In diesem Teil der Bachelorarbeit stellen wir die Beschreibungs-Logik vor und bauen in den folgenden Kapiteln darauf auf. Unsere Definitionen der Beschreibungs-Logik stammen aus [BCM⁺03].

Beschreibungs-Logiken werden dazu genutzt, eine Wissensbasis zu erzeugen. Diese definiert sogenannte Konzepte und teilt ihnen bestimmte, generelle Eigenschaften zu. Ein Konzept beschreibt also eine Menge von Objekten, welche sich durch gleiche Eigenschaften auszeichnen. Es verknüpft Eigenschaften mit einem Namen oder einem Oberbegriff für die Objekte, denen diese Eigenschaften zugeordnet werden können. Dieser Teil der Wissensbasis nennt sich TBox (engl. terminological box). Zusätzlich existieren konkrete Objekte, welche individuelle Merkmale besitzen und daher Individuen genannt werden. Diese stellen ein Objekt aus der Menge der einem Konzept zugeordneten Objekte dar, instantiiieren also ein Konzept. Es gibt einen Teil innerhalb der Wissensbasis, der ausschließlich Individuen beschreibt und ihnen Konzepte zuweist. Dieser Teil nennt sich ABox (engl. assertion box). Die Eigenschaften, welche die Konzepte und Individuen beschreiben, werden durch sogenannte Rollen realisiert.

Alle Beschreibungs-Logik-Sprachen liegen mindestens in der Kategorie \mathcal{AL} (engl. *Attributive Language*). Ist A ein atomares Konzept (also ein Konzept mit einem Namen), C und D komplexe Konzepte und R eine atomare Rolle, so

Tabelle 3.1: \mathcal{ALC} Beschreibungs-Logik.

Klasse	Name	Beschr. Logik	Interpretation
\mathcal{AL}	Atomares Konzept	A	A^I
	Top Konzept	\top	Δ^I
	Bottom Konzept	\perp	\emptyset
	Atomare Negation	$\neg A$	$\Delta^I \setminus A^I$
	Überschneidung	$C \sqcap D$	$C^I \cap D^I$
	Wert-Einschränkung	$\forall R.C$	$\{a \in \Delta^I \mid (a, b) \in R^I \rightarrow b \in C^I\}$
	Begrenzte Existenz Quantifizierung	$\exists R.\top$	$\{a \in \Delta^I \mid \exists b((a, b) \in R^I)\}$
\mathcal{ALE}	Unbegrenzte Existenz Quantifizierung	$\exists R.C$	$\{a \in \Delta^I \mid \exists b((a, b) \in R^I \wedge b \in C^I)\}$
\mathcal{ALU}	Vereinigung	$C \sqcup D$	$C^I \cup D^I$
\mathcal{ALC}	Komplexe Negation	$\neg C$	$\Delta^I \setminus C^I$

lässt sich ein weiteres komplexes Konzept einer \mathcal{AL} - Wissensbasis durch die \mathcal{AL} Konstrukte aus Tabelle 3.1 erzeugen.

Die Sprache, welche unbeschränkte existentielle Quantifizierung ($\exists R.C$) enthält, in der also das Ziel einer existentiellen Quantifizierung durch jedes beliebige Konzept eingeschränkt werden darf, wird \mathcal{ALE} (engl. *Attributive Language with unrestricted existential quantification*) genannt.

Zusätzlich zur Überschneidung von komplexen Konzepten (\sqcap) wird häufig die Vereinigung (\sqcup) komplexer Konzepte benötigt. Die am wenigsten mächtige Sprache, mit der die Vereinigung von Konzepten ausgedrückt werden kann, nennt sich \mathcal{ALU} (engl. *Attributive Language with concept union*).

Stellt C ein nicht-atomares Konzept dar, so lässt sich $\neg C$ nicht mehr mit der Sprache \mathcal{ALU} oder \mathcal{ALE} ausdrücken. Die Sprache, welche komplexe Negationen umfasst, nennt sich \mathcal{ALC} (engl. *Attributive Language with complex negation*). Wichtig ist, dass die Sprache \mathcal{ALC} die Sprache \mathcal{ALU} und \mathcal{ALE} enthält, denn jedes Konzept der Form $C \sqcup D$ lässt sich darstellen als $\neg(\neg C \sqcap \neg D)$ und jedes Konzept $\exists role.C$ als $\neg \forall role.\neg C$.

Atomare Konzepte werden durch sogenannte Axiome beschrieben. Ist A ein atomares Konzept und C ein beliebiges komplexes Konzept (vgl. Tabelle 3.1 Spalte 2), so ist $A \sqsubseteq C$ ein Axiom, welches A beschreibt. Wenn sowohl

Tabelle 3.2: Regeln für \mathcal{ALC} Modelle.

Name	Beschr. Logik	Modell, gdw.
Konzept Beschreibung	$A \sqsubseteq C$	$A^I \subseteq C^I$
Konzept Definition	$A \equiv C$	$A^I = C^I$

$A \sqsubseteq C$ als auch $C \sqsubseteq A$ gilt, so ist dies äquivalent zu dem Ausdruck $A \equiv C$. Dies wird als eine Definition des Konzeptes A bezeichnet.

Definition 3 Eine \mathcal{ALC} Wissensbasis Σ ist das Tupel $(\mathcal{T}, \mathcal{A})$. \mathcal{T} ist eine Menge von Axiomen $A \sqsubseteq C$ und $A \equiv C$, die Atomare Konzepte A durch komplexe Konzepte C beschreiben. \mathcal{T} wird TBox genannt. \mathcal{A} ist eine Menge von Zuweisungen $a : C$ von Individuen a zu komplexen Konzepten C und wird ABox genannt.

Die Interpretation einer Wissensbasis wird durch eine nicht leere Menge Δ^I , welche die Domäne der Interpretation genannt wird, und eine Funktion f_I angegeben. f_I muss eine Funktion sein, die jedem atomaren Konzept A eine Teilmenge A^I aus Δ^I ($A^I \subseteq \Delta^I$) und jeder atomaren Rolle R eine binäre Relation zwischen Elementen aus Δ^I zuweist ($R^I = \Delta^I \times \Delta^I$). Komplexe Konzepte lassen sich durch eine Interpretation I ($I = (\Delta^I, f_I)$) interpretieren, wie in der dritten Spalte der Tabelle 3.1 angegeben.

Eine Interpretation I ist genau dann ein Modell für ein Axiom $C \sqsubseteq F$, wenn $C^I \subseteq F^I$ gilt. Entsprechend ist die Interpretation ein Modell für eine Menge von Axiomen, wenn oberes für jede dieser Axiome gilt. Insbesondere ist eine Interpretation ein Modell für das Axiom $A \equiv C$, genau dann wenn sowohl $A^I \subseteq C^I$ als auch $C^I \subseteq A^I$ gilt, wenn also $A^I = C^I$ erfüllt ist.

Ein Konzept C , welches durch eine Menge von Axiomen \mathcal{T} beschrieben wird, bezeichnen wir als konsistent, wenn es ein Modell M für \mathcal{T} gibt, in dem $C^M \neq \emptyset$ gilt. Das Problem, zu entscheiden ob ein Konzept der Wissensbasis konsistent ist oder nicht, nennen wir Konzept-Erfüllbarkeits-Problem. Sind alle Konzepte einer Wissensbasis inkonsistent, so ist die Wissensbasis selbst inkonsistent (oder unerfüllbar).

Wenn wir im weiteren Verlauf von einer Wissensbasis Σ sprechen, so ist stets von einer Wissensbasis mit einer nicht-leeren TBox und einer leeren ABox die Rede ($\Sigma = (\mathcal{T}, \emptyset)$). Aus diesem Grund geben wir nur die TBox an und lassen die leere ABox weg.

3.4 Reasoning

Da wir im vorigen Abschnitt erklärt haben, was eine Wissensbasis ist, werden wir hier erörtern, welche Ziele ein Reasoner verfolgt. Wir sprechen von einem Reasoning-Prozess immer dann, wenn wir von gegebenen, expliziten Informationen einer Wissensbasis auf neue, implizite Informationen schließen. Die neuen Informationen lassen sich also aus der Wissensbasis ableiten, obwohl sie dort nicht explizit angegeben wurden. Ein Reasoner ist ein Programm, das diesen Prozess ausführt.

Eine der wichtigsten Aufgaben für einen Reasoner in dieser Arbeit ist zu überprüfen, ob ein Konzept konsistent ist oder nicht. Zur Wiederholung: Ein Konzept C einer TBox \mathcal{T} ist konsistent, wenn es ein Modell M für \mathcal{T} gibt, in dem C^M nicht leer ist. Dies wird Konzept-Erfüllbarkeits-Problem oder kurz Erfüllbarkeits-Problem genannt.

Daneben kann ein Reasoner überprüfen, ob ein Konzept ein anderes Konzept einschließt (engl. to subsume). Ein Konzept C einer TBox \mathcal{T} wird von einem Konzept D eingeschlossen, wenn für jedes Modell M von \mathcal{T} gilt $C^M \subseteq D^M$.

Entsprechend ist es möglich, mithilfe eines Reasoners zu entscheiden, ob ein Konzept C äquivalent zu einem Konzept D ist. Sie sind äquivalent, wenn in jedem Modell M gilt $C^M = D^M$.

Ein weiteres wichtiges Problem stellt die Entscheidung dar, ob zwei Konzepte C und D einer TBox \mathcal{T} disjunkt (engl. disjoint) sind. Sie sind disjunkt, wenn in jedem Modell M von \mathcal{T} gilt $C^M \cap D^M = \emptyset$.

Alle diese Probleme lassen sich auf das Erfüllbarkeits-Problem reduzieren. Ein Konzept C wird von dem Konzept D genau dann eingeschlossen, wenn das Konzept $C \sqcap \neg D$ inkonsistent ist.

Ebenso sind die zwei Konzepte C und D genau dann äquivalent, wenn sowohl $C \sqcap \neg D$ als auch $D \sqcap \neg C$ inkonsistent sind.

Zwei Konzepte C und D sind genau dann disjunkt (es existiert kein Modell, in dem ein Objekt sowohl eine Instanz von C als auch von D ist), wenn das Konzept $C \sqcap D$ inkonsistent ist.

Kapitel 4

Transformation der Modelle

In diesem Kapitel werden wir die Transformation eines Goal-Modells in eine \mathcal{ALC} -Wissensbasis, sowie die Transformation eines Feature-Modells in eine solche Wissensbasis vorstellen. Die Wissensbasis des Goal-Modells nennen wir im folgenden Σ_G , die des Feature-Modells Σ_F . Die Mappings zwischen Goal- und Feature-Modell werden in einer eigenen Wissensbasis Σ_M repräsentiert.

4.1 Transformation des Goal-Modells

Wir haben in Kapitel 3.2 ein Goal-Modell definiert. Das Ziel dieses Abschnitts ist eine Wissensbasis zu erzeugen, die alle Ziele, Aufgaben und qualitativen Ziele enthält und zusätzlich die Zerlegungen dieser Elemente in ihre Bestandteile beschreibt.

Algorithmus 1 transformiert ein Goal-Modell $\{\mathcal{G}, \mathcal{V}, \mathcal{Z}\}$ in eine Wissensbasis $\Sigma_{G_{naiv}}$. Der Algorithmus wird an dieser Stelle erläutert, um ein grundlegendes Verständnis für die Transformation zu schaffen.

Um die unterschiedlichen intentionalen Elemente in eine Wissensbasis zu transformieren, liegt es nahe gleichnamige Konzepte einzuführen. Daher nutzen wir in der Wissensbasis atomare Konzepte, die den gleichen Namen tragen, wie die intentionalen Elemente des Goal-Modells.

Zusätzlich benötigen wir Axiome, welche die zuvor hinzugefügten atomaren Konzepte passend zu den Zerlegungen im Goal-Modell beschreiben. Um zu gewährleisten, dass ein intentionales Element g , das durch eine *IOder*-Zerlegung in die Bestandteile b_1, \dots, b_n zerlegt wird, dann konsistent ist, wenn mindestens eines der Elemente b_1, \dots, b_n konsistent ist, beschreiben wir das Element g in diesem Ansatz durch ein Axiom $g \sqsubseteq b_1 \sqcup \dots \sqcup b_n$ (Zeilen 4-10). Ein

Algorithm 1 Naive Goal-Modell Transformation.

```

1: input=Goal-Modell  $\{\mathcal{G}, \mathcal{V}, \mathcal{Z}\}$ 
2:  $\Sigma_{G_{naiv}} = \emptyset$ .
3: for all  $g \in \mathcal{G}$  do
4:   if  $(g, IOder, Y) \in \mathcal{Z}$  then
5:      $vereinigung = \perp$ 
6:     for all  $y \in Y$  do
7:        $vereinigung = vereinigung \sqcup y$ 
8:     end for
9:      $\Sigma_{G_{naiv}} = \Sigma_{G_{naiv}} \cup g \sqsubseteq vereinigung$ 
10:  end if
11:  if  $(g, Und, Y) \in \mathcal{Z}$  then
12:     $schnitt = \top$ 
13:    for all  $y \in Y$  do
14:       $schnitt = schnitt \sqcap y$ 
15:    end for
16:     $\Sigma_{G_{naiv}} = \Sigma_{G_{naiv}} \cup g \sqsubseteq schnitt$ 
17:  end if
18:  if  $(g, XOder, Y) \in \mathcal{Z}$  then
19:     $vereinigung = \perp$ 
20:    for all  $y \in Y$  do
21:       $schnitt = y$ 
22:      for all  $z \in Y \setminus \{y\}$  do
23:         $schnitt = schnitt \sqcap \neg z$ 
24:      end for
25:       $vereinigung = vereinigung \sqcup schnitt$ 
26:    end for
27:     $\Sigma_{G_{naiv}} = \Sigma_{G_{naiv}} \cup g \sqsubseteq vereinigung$ 
28:  end if
29: end for
30: return  $\Sigma_{G_{naiv}}$ 

```

Konzept der Form $b_1 \sqcup \dots \sqcup b_n$ fassen wir im Folgenden zusammen zu $\bigsqcup_{i=1}^n b_i$.

Ein intentionales Element g , das durch eine *Und*-Zerlegung in die Bestandteile b_1, \dots, b_n zerlegt wird, beschreiben wir in diesem vereinfachten Ansatz entsprechend durch ein Axiom $g \sqsubseteq b_1 \sqcap \dots \sqcap b_n$ (Zeilen 11-17), um zu gewährleisten, dass das Konzept g dann konsistent ist, wenn alle *Und*-Bestandteile konsistent sind. Ein Konzept der Form $b_1 \sqcap \dots \sqcap b_n$ fassen wir im Folgenden zusammen zu $\prod_{i=1}^n b_i$.

Es gibt auch intentionale Elemente g , die durch *XOder*-Zerlegungen in die Bestandteile b_1, \dots, b_n zerlegt werden. In diesem Fall muss das Konzept g entsprechend der Semantik des Goal-Modells dann konsistent sein, wenn nur eines der Bestandteile g_1, \dots, g_n konsistent ist. Dies erreichen wir durch das Axiom $g \sqsubseteq (b_1 \sqcap \neg b_2 \sqcap \dots \sqcap \neg b_n) \sqcup \dots \sqcup (b_n \sqcap \neg b_1 \sqcap \dots \sqcap \neg b_{n-1})$ (Zeilen 18-28) oder etwas verkürzt durch $g \sqsubseteq \bigsqcup_{i=1}^n (b_i \sqcap (\prod_{j=1}^{i-1} \neg b_j \sqcap \prod_{j=1+i}^n \neg b_j))$.

Die so erzeugte Wissensbasis eignet sich nicht zur Konfiguration eines Feature-Modells. Es fehlt das Wissen um die Verbindungen der einzelnen intentionalen Elemente (\mathcal{V}). Außerdem wird durch die starke Hierarchisierung der einzelnen Elemente die Suche nach inkonsistenten Mappings zwischen Feature- und Goal-Modell erschwert.

In der verfeinerten Erzeugung der Wissensbasis, die durch den Algorithmus 2 angedeutet wird, führen wir eine Rolle ein (*istVorhanden*), um Bedingungen für die Erfüllung eines intentionalen Elementes zu modellieren.

Des Weiteren existiert zu einem Element g des Goal-Modells in der neuen Wissensbasis ein neues atomares Konzept. Es wird def_g (Definition von g) genannt. def_g repräsentiert die Voraussetzungen, die erfüllt sein müssen, damit das Element g erreicht ist. Wir werden diese Definitions-Konzepte verwenden, um fehlerhafte Mappings zu erkennen.

Die Definitions-Konzepte der einzelnen Elemente des Goal-Modells werden mithilfe der Rolle *istVorhanden* und den atomaren Konzepten beschrieben, die den gleichen Namen tragen wie ein intentionales Element. Das Vorgehen ist dem vereinfachten Ansatz ähnlich. In den Zeilen 5-11 wird für eine *IOder*-Zerlegungen eines Elementes g in die Bestandteile b_1, \dots, b_n das Konzept $\bigsqcup_{i=1}^n \exists istVorhanden.b_i$ erzeugt. Somit stellen wir sicher, dass def_g durch seine Zerlegung erfüllt wird, wenn mindestens eines der Zerlegungs-Bestandteile als

Algorithm 2 Goal-Modell Transformation mit Rollen.

```

1: input=Goal-Modell  $\{\mathcal{G}, \mathcal{V}, \mathcal{Z}\}$ .
2:  $\Sigma_G = \emptyset$ 
3: superkonzept = null
4: for all  $g \in \mathcal{G}$  do
5:   if  $(g, IOder, Y) \in \mathcal{Z}$  then
6:     vereinigung =  $\perp$ 
7:     for all  $y \in Y$  do
8:       vereinigung = vereinigung  $\sqcup$   $\exists istVorhanden.y$ 
9:     end for
10:    superkonzept = vereinigung
11:  end if
12:  if  $(g, Und, Y) \in \mathcal{Z}$  then
13:    schnitt =  $\top$ 
14:    for all  $y \in Y$  do
15:      schnitt = schnitt  $\sqcap$   $\exists istVorhanden.y$ 
16:    end for
17:    superkonzept = schnitt
18:  end if
19:  if  $(g, XOder, Y) \in \mathcal{Z}$  then
20:    vereinigung =  $\perp$ 
21:    for all  $y \in Y$  do
22:      schnitt =  $\exists istVorhanden.y$ 
23:      for all  $z \in Y \setminus \{y\}$  do
24:        schnitt = schnitt  $\sqcap$   $\neg \exists istVorhanden.z$ 
25:      end for
26:      vereinigung = vereinigung  $\sqcup$  schnitt
27:    end for
28:    superkonzept = vereinigung
29:  end if
30:  for all  $(f, ++, g) \in \mathcal{V}$  do
31:    if superkonzept  $\neq$  null then
32:      superkonzept = superkonzept  $\sqcup$   $\exists istVorhanden.f$ 
33:    else
34:      superkonzept =  $\exists istVorhanden.f$ 
35:    end if
36:  end for
37:  for all  $(f, --, g) \in \mathcal{V}$  do
38:    if superkonzept  $\neq$  null then
39:      superkonzept = superkonzept  $\sqcup$   $\neg \exists istVorhanden.f$ 
40:    else
41:      superkonzept =  $\neg \exists istVorhanden.f$ 
42:    end if
43:  end for
44:  if superkonzept  $\neq$  null then
45:     $\Sigma_G = \Sigma_G \cup def_g \equiv$  superkonzept
46:  else
47:     $\Sigma_G = \Sigma_G \cup def_g \equiv \top$ 
48:  end if
49: end for
50: return  $\Sigma_G$ 

```

erreicht markiert ist. Wir werden Konzepte der Form $\exists istVorhanden.g$ später nutzen, um das intentionale Element g als erreicht oder abgelehnt zu beschriften.

Die Transformation der *Und*-Zerlegungen in den Zeilen 12-18 erklärt sich analog. In den Zeilen 19-29 werden alle *XOder*-Zerlegungen repräsentiert. Wird ein intentionales Element durch eine *XOder*-Zerlegung zerlegt, so kann es erfüllt werden, wenn genau eines der Zerlegungs-Bestandteile erreicht ist. Es wird nicht durch die Zerlegung erfüllt, wenn keines oder mehrere der Bestandteile erreicht sind. Diesen gegenseitigen Ausschluss der Zerlegungs-Bestandteile b_1, \dots, b_n beschreiben wird durch das Konzept $\bigsqcup_{i=1}^n (\exists istVorhanden.b_i \sqcap \prod_{j=1}^{i-1} \neg \exists istVorhanden.b_j) \sqcap \prod_{j=1+i}^n \neg \exists istVorhanden.b_j$.

Eine *Erfüllt*-Verbindung ($++$ -Verbindung) zwischen zwei Elementen g und f des Goal-Modells $((g, ++, f) \in \mathcal{V})$ bedeutet, dass das Element g immer erreicht ist, wenn auch das Element f erreicht ist. Um dies in Σ_G zu repräsentieren, vereinigen (\sqcup) wir in den Zeilen 30-36 das bisher berechnete Konzept mit dem Konzept $\exists istVorhanden.f$. Die Vereinigung mit dem Konzept $\neg \exists istVorhanden.f$ in den Zeilen 37-43 beruht auf einer *Verhindert*-Verbindung zwischen g und f $((g, --, f) \in \mathcal{V})$.

Für die Konfiguration eines Feature-Modells spielen die *TraegtBei*- oder *Schadet*-Verbindungen keine Rolle, da sich mit ihnen nicht eindeutig festlegen lässt, ob ein Ziel erreicht wird oder nicht. Genau diese Unterscheidung müssen wir aber treffen, um eindeutig entscheiden zu können, ob ein Feature, das auf ein intentionales Element abgebildet ist, in eine Konfiguration aufgenommen wird oder nicht.

In unserem Beispiel (Abbildung 2.3) werden die Elemente „Prozess bei Kunden Anwenden“ (Abk. $PbKa$), „Bei beliebigen Kunden Anwenden“ (Abk. $BbKa$), „Bei vertrauten Kunden Anwenden“ (Abk. $BvKa$) und „Bestellung Annehmen“ (Abk. BA)

durch folgende Axiome beschrieben:

$$\begin{aligned} def_{PbKa} &\equiv \exists istVorhanden.BbKa \sqcup \exists istVorhanden.BvKa \\ def_{BbKa} &\equiv \exists istVorhanden.BA \\ def_{BvKa} &\equiv \exists istVorhanden.BA \\ def_{BA} &\equiv \top \end{aligned}$$

Das Goal-Modell aus der Abbildung 2.4, welches aus den Elementen „Bezahlung“ (Abk. *B*), „Persönliche Bezahlung“ (Abk. *PB*) und „Elektronische Bezahlung“ (Abk. *EB*) besteht, wird in die folgende TBox transformiert:

$$\begin{aligned} def_B &\equiv \exists istVorhanden.PB \sqcup \exists istVorhanden.EB \\ def_{PB} &\equiv \top \\ def_{EB} &\equiv \top \end{aligned}$$

4.2 Transformation des Feature-Modells

In Kapitel 3.1 wurden Feature-Modelle definiert. Basierend auf dieser Definition, werden wir in diesem Abschnitt die Transformation dieser Modelle in eine *ALC*-Wissensbasis vorstellen, die alle Features durch ihre Beziehung zueinander und ihre Einschränkungen beschreibt.

Die Wissensbasis, die wir erzeugen wollen, soll ähnlich zu der Wissensbasis des Goal-Modells sein, damit wir die einzelnen Konzepte der beiden Wissensbasen miteinander vergleichen können. Aus diesem Grund verwenden wir auch in der Feature-Modell Wissensbasis eine Rolle mit dem Namen *istVorhanden*, mit deren Hilfe wir die Beziehungen zwischen Features beschreiben. Der Algorithmus 3 zur Transformation eines Feature-Modells, den wir in diesem Abschnitt beschreiben, arbeitet ähnlich wie der Algorithmus zur Transformation des Goal-Modells (Algorithmus 2).

Durch die *istVorhanden*-Rolle und *def*-Konzepte werden, wie in der Goal-Modell-Wissensbasis, notwendige Bedingungen definiert, die erfüllt sein müssen, damit ein Feature in einer Konfiguration verbleiben darf. Sollte bei einer Konfiguration des Feature-Modells ein Feature entfernt werden, so muss auch sein Vater-Feature entfernt werden, wenn es obligatorisch von diesem abhängt.

Algorithm 3 Feature-Modell Transformation.

```

1: input=Feature-Modell  $\{\mathcal{F}, \mathcal{F}_{obl}, \mathcal{F}_{opt}, \mathcal{F}_{grup}, \mathcal{R}_{VK}, \mathcal{R}_{VG}, \mathcal{A}\}$ .
2:  $\Sigma_F = \emptyset$ .
3: for all  $f \in \mathcal{F}$  do
4:    $superkonzept = \top$ 
5:   for all  $(f, k) \in \mathcal{R}_{VK}$  do
6:     if  $k \in \mathcal{F}_{obl}$  then
7:        $superkonzept = superkonzept \sqcap \exists istVorhanden.k$ 
8:     end if
9:   end for
10:  for all  $(f, IOder, Y) \in \mathcal{R}_{VG}$  do
11:     $vereinigung = \perp$ 
12:    for all  $y \in Y$  do
13:       $vereinigung = vereinigung \sqcup \exists istVorhanden.y$ 
14:    end for
15:     $superkonzept = superkonzept \sqcap vereinigung$ 
16:  end for
17:  for all  $(f, XOder, Y) \in \mathcal{Z}$  do
18:     $vereinigung = \perp$ 
19:    for all  $y \in Y$  do
20:       $schnitt = \exists istVorhanden.y$ 
21:      for all  $z \in Y \setminus \{y\}$  do
22:         $schnitt = schnitt \sqcap \neg \exists istVorhanden.z$ 
23:      end for
24:       $vereinigung = vereinigung \sqcup schnitt$ 
25:    end for
26:     $superkonzept = superkonzept \sqcap vereinigung$ 
27:  end for
28:  for all  $(f, benoetigt, Feature, k) \in \mathcal{A}$  do
29:     $superkonzept = superkonzept \sqcap \exists istVorhanden.k$ 
30:  end for
31:  for all  $(f, schließtAus, Feature, k) \in \mathcal{A}$  do
32:     $superkonzept = superkonzept \sqcap \neg \exists istVorhanden.k$ 
33:  end for
34:   $\Sigma_F \cup def_f \equiv superkonzept$ 
35: end for
36: return  $\Sigma_F$ 

```

Deshalb erstellen wir zu jeder Beziehung (f, k) in \mathcal{R}_{VK} , in der k obligatorisch ist ($k \in \mathcal{F}_{obl}$) ein *istVorhanden*-Konzept ($\exists istVorhanden.k$, Zeilen 5-9).

Es ist notwendig, dass wir auch für *IOder*-Gruppen *istVorhanden*-Axiome in Σ_F aufnehmen. Wenn in einer Konfiguration einzelne Features k_i aus der Gruppe entfernt werden, so darf das Vater-Feature f noch in diese Konfiguration aufgenommen werden. Wird aber jedes Feature aus einer solchen Gruppe bei der Konfiguration entfernt, so muss auch das Vater-Feature entfernt werden. Dies lässt sich durch ein Vereinigungs-Konzept der *istVorhanden*-Rolle und allen Mitgliedern der *IOder*-Gruppe bewerkstelligen ($\bigsqcup_{i=1}^n \exists istVorhanden.k_i$, Zeilen 10-16).

Da es im Feature-Modell Gruppen des Typs *XOder* gibt, muss die Wissensbasis auch für diese Art von Gruppen $\exists istVorhanden$ -Konzepte enthalten. Der Algorithmus fügt daher zu jeder Beziehung $(f, XOder, Y)$ ein Konzept hinzu, das den gegenseitigen Ausschluss der Gruppen-Features in Y beschreibt ($\bigsqcup_{i=1}^n (\exists istVorhanden.y_i \sqcap \prod_{j=1}^{i-1} \neg \exists istVorhanden.y_j \sqcap \prod_{j=i+1}^n \neg \exists istVorhanden.y_j)$). Dies ist in den Zeilen 17-27 angedeutet.

Die Attribute eines Feature-Modells können viele Informationen beinhalten. In dieser Arbeit sind die Attribute relevant, welche Einschränkungen der Produktlinie kodieren. Existiert zu einem Feature f ein Attribut mit dem Namen *benoetigt*, das auf ein anderes Feature k des Modells zeigt, so kann f nur in der Konfiguration bleiben, wenn k dort vorhanden ist ($(f, benoetigt, Feature, k) \in \mathcal{A}$). Diese Bedingung wird durch das Konzept $\exists istVorhanden.k$ in den Zeilen 28-30 ausgedrückt.

Der gegenteilige Fall, in dem ein Feature f nur dann in die Konfiguration aufgenommen werden darf, wenn ein anderes Feature k dort nicht vorhanden ist, wird durch ein *schließtAus*-Attribut ($(f, schließtAus, Feature, k) \in \mathcal{A}$) im Feature-Modell kodiert. Wenn solch ein Attribut existiert, so wird es in der Wissensbasis durch das Konzept $\neg \exists istVorhanden.k$ repräsentiert (Zeilen 31-33).

Transformieren wir das Feature-Modell aus Abbildung 2.3 mit den Features „Bestellung Annehmen“ (Abk. *BA*), „Vertraute Kunden Annehmen“ (Abk. *VKA*), „Beliebige Kunden Annehmen“ (Abk. *BKA*) und „Kundenverifizierung“ (Abk. *KV*),

so erhalten wir eine Wissensbasis mit den folgenden Axiomen:

$$\begin{aligned} def_{BA} &\equiv \exists istVorhanden.VKA \sqcup \exists istVorhanden.BKA \\ def_{VKA} &\equiv \top \\ def_{BKA} &\equiv \exists istVorhanden.KV \end{aligned}$$

Das Feature-Modell aus Abbildung 2.4, bestehend aus den Features „Bezahlung“ (Abk. *Bz*), „Online Zahlung“ (Abk. *OZ*), „Bargeld“ (Abk. *BG*), „Kreditkarte“ (Abk. *KrK*) und „Kundenkarte“ (Abk. *KuK*). Es wird in die folgende Wissensbasis transformiert:

$$\begin{aligned} def_{Bz} &\equiv \exists istVorhanden.OZ \sqcap \exists istVorhanden.KrK \sqcap \exists istVorhanden.KuK \\ def_{KuK} &\equiv \top \\ def_{OZ} &\equiv \top \\ def_{BG} &\equiv \top \\ def_{KrK} &\equiv \top \end{aligned}$$

4.3 Auswertung der Mappings

Ein Mapping zwischen einem intentionalen Element g des Goal-Modells und einem Feature f des Feature-Modells wird durch ein Attribut im Feature-Modell modelliert $((f, hatMapping, IntElement, g) \in \mathcal{A})$. Wir nehmen in einem späteren Schritt eine Beschriftung des Goal-Modells vor, anhand derer wir die intentionalen Elemente in die Kategorien *erreicht* und *abgelehnt* einteilen. Die auf erreichte Elemente abgebildeten Features werden in das konfigurierte Feature-Modell aufgenommen. Die Unterteilung in *erreicht* und *abgelehnt* lässt sich für *qualitative Ziele* nicht vornehmen. Daher ist es notwendig, die Mappings auf eine Abbildung zwischen *Aufgaben* und *Zielen* des Goal-Modells und den *Features* aus dem Feature-Modell zu reduzieren.

Um alle Mappings des Feature-Modells auswerten zu können, müssen wir die Axiome der beiden Wissensbasen Σ_G und Σ_F in einer neuen Wissensbasis Σ_M vereinen. Damit wir erkennen, aus welchem Modell (Feature oder Goal) ein atomares Konzept ursprünglich stammt, indizieren wir alle atomaren Konzepte des Feature-Modells mit den Buchstaben *FM*, die des Goal-Modells mit

den Buchstaben GM .

Um ein Mapping überhaupt überprüfen zu können, müssen die aufeinander abgebildeten Elemente auch in der Wissensbasis aufeinander abgebildet werden. Wird ein Feature f auf ein Element g abgebildet, so soll es das Element g in der Wissensbasis repräsentieren und umgekehrt. Dies erreichen wir durch das Hinzufügen von Äquivalenz-Axiomen. Es wird für jedes Mapping $((f, \text{hatMapping}, \text{IntElement}, g) \in \mathcal{A})$ ein Axiom $g_{GM} \equiv f_{FM}$ zu Σ_M hinzugefügt. So erreichen wir, dass der Reasoner die aufeinander abgebildeten Konzepte in den def -Konzepten synonym verwendet. Es macht also keinen Unterschied mehr, ob die Abhängigkeiten eines Features (def -Konzepte) durch die atomaren Konzepte von Σ_F oder durch die entsprechenden Konzepte aus Σ_G definiert wurden. Sei beispielsweise ein def -Konzept $def_{f_{FM}} \equiv \exists \text{istVorhanden}.fk_{FM}$ aus Σ_F und $def_{g_{GM}} \equiv \exists \text{istVorhanden}.gk_{GM}$ aus Σ_G gegeben. Weiter sei in diesem Beispiel das Feature fk auf das Ziel gk abgebildet. Durch das Hinzufügen von $gk_{GM} \equiv fk_{FM}$ zu Σ_M erreichen wir, dass $def_{g_{GM}} \equiv def_{f_{FM}} \equiv \exists \text{istVorhanden}.fk_{FM} \equiv \exists \text{istVorhanden}.gk_{GM}$ in Σ_M gilt.

Für die Validierung eines Mappings zwischen dem Feature f und einem Element des Goal-Modells g spielen die Konzepte $def_{g_{GM}}$ und $def_{f_{FM}}$ eine Rolle. Wir wissen, dass ein Mapping nur dann konsistent sein kann, wenn in einer Konfiguration sowohl das Konzept $def_{g_{GM}}$, das die Voraussetzungen für das Vorhandensein des Elements g darstellt als auch das Konzept $def_{f_{FM}}$, welches die Voraussetzungen für das Feature f definiert, gültig sind. Also kann ein Mapping nur dann valide sein, wenn insgesamt das Konzept $def_{g_{GM}} \sqcap def_{f_{FM}}$ konsistent (also erfüllbar) ist.

Wenn $def_{g_{GM}} \sqcap def_{f_{FM}}$ konsistent ist, ist es dennoch möglich, dass durch eine ungünstige Auswahl von Elementen des Goal-Modells die Abhängigkeiten des Feature-Modells verletzt werden. Daher führen wir eine Überprüfung durch, ob die Abhängigkeiten des Goal-Modells immer (in jeder möglichen Konfiguration) die Abhängigkeiten des Feature-Modells enthalten. Wir überprüfen, ob aus den Abhängigkeiten von g die Abhängigkeiten von f folgen. In Beschreibungs-Logik kodiert testen wir, ob das Konzept $\neg def_{g_{GM}} \sqcup def_{f_{FM}}$ äquivalent ist zum Top Konzept ($\neg def_{g_{GM}} \sqcup def_{f_{FM}} \equiv? \top$).

Wir können somit drei Fälle unterscheiden, im ersten ist ein Mapping inkonsistent, weil keine Konfiguration möglich ist, in der die Abhängigkeiten des Goal-Modells und gleichzeitig die Abhängigkeiten des Feature-Modells eingehalten werden. Der zweite Fall bedeutet, dass es Konfigurationen gibt, in

Tabelle 4.1: Mapping Validierung eines Features f zu einem Element g .

Valide?	Bedingung
immer	$\neg def_g \sqcup def_f \equiv \top$
manchmal	$def_{FM} \sqcap def_g \sqsubseteq \top$
nie	$def_{FM} \sqcap def_g \sqsubseteq \perp$

denen beide Abhängigkeiten eingehalten werden. Im dritten Fall erhalten die Abhängigkeiten des Goal-Modells die Abhängigkeiten eines Features zu seinen Kindern in jeder Konfiguration. In der Tabelle 4.1 werden alle drei Möglichkeiten für die Validität eines Mappings (immer valide, manchmal valide, nie valide) aufgeführt.

Es ist zu erkennen, dass ein Mapping zwischen einem Feature f , zu dem keine Attribute oder Kind-Features existieren und einem Goal-Modell-Element g , das nicht zerlegt ist und keine Verbindung zu anderen Elementen des Goal-Modells hat, immer gültig ist. Dort ist das def -Konzept äquivalent zu \top und somit $\neg def_g \sqcup def_f$ äquivalent zu $\neg \top \sqcup \top$, was wiederum äquivalent zu \top ist.

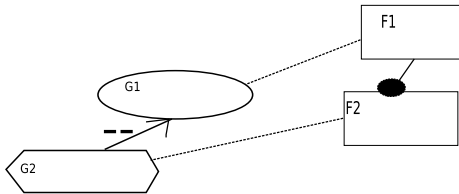


Abbildung 4.1: Ein Beispiel für ein immer ungültiges Mapping.

Existieren in einer Mapping-Wissensbasis Σ_M Mappings, die nie gültig sind, so ist es erforderlich, diese Mappings entweder zu entfernen oder die Modelle so anzupassen, dass die Mappings zumindest manchmal gültig sind. Sonst ist es unmöglich eine Konfiguration des Feature-Modells zu finden, sodass sowohl das Feature-Modell als auch das Goal-Modell valide sind. Ein Beispiel für diese Situation ist durch die Modelle in Abbildung 4.1 gegeben.

In dem Goal-Modell aus Abbildung 4.1 kann das Ziel $G1$ nur erreicht werden, wenn die Aufgabe $G2$ nicht erfüllt wird. Das Feature $F1$ hängt obligatorisch von $F2$ ab. $F1$ wird abgebildet auf $G1$ und $F2$ auf $G2$. Das Mapping zwischen $F1$ und $F2$ ist immer ungültig, denn das Ziel $G2$ schließt $G1$ aus und

$F1$ benötigt $F2$. In Σ_M finden wir folgende Axiome:

$$G1_{GM} \equiv F1_{FM}$$

$$G2_{GM} \equiv F2_{FM}$$

$$def_{G1_{GM}} \equiv \neg \exists istVorhanden.G2_{GM}$$

$$def_{F1_{FM}} \equiv \exists istVorhanden.F2_{FM}$$

Das Konzept $def_{F1_{FM}} \sqcap def_{G1_{GM}}$ ist inkonsistent, da es äquivalent zu $\neg \exists istVorhanden.G2_{GM} \sqcap \exists istVorhanden.G2_{GM}$ ist. In diesem Fall muss also eines der Mappings entfernt, die negative Verbindung zwischen $G1$ und $G2$ entfernt oder die obligatorische Beziehung zwischen $F1$ und $F2$ in eine optionale Beziehung umgewandelt werden, damit eine Konfiguration des Feature-Modells möglich wird.

Die Wissensbasis Σ_M für die Abbildung 2.3 enthält folgende Axiome:

$$def_{PbKa_{GM}} \equiv \exists istVorhanden.BbKa_{GM} \sqcup \exists istVorhanden.BvKa_{GM}$$

$$def_{BbKa_{GM}} \equiv \exists istVorhanden.BA_{GM}$$

$$def_{BvKa_{GM}} \equiv \exists istVorhanden.BA_{GM}$$

$$def_{BA_{GM}} \equiv \top$$

$$def_{BA_{FM}} \equiv \exists istVorhanden.VK_{AFM} \sqcup \exists istVorhanden.BK_{AFM}$$

$$def_{VK_{AFM}} \equiv \top$$

$$def_{BK_{AFM}} \equiv \exists istVorhanden.KV_{FM}$$

$$BA_{GM} \equiv VK_{AFM} \equiv BK_{AFM}$$

$$BbKa_{GM} \equiv BA_{FM}$$

Die Wissensbasis für die Mappings aus Abbildung 2.4 besteht aus den Axiomen:

$$def_{B_{GM}} \equiv \exists istVorhanden.PB_{GM} \sqcup \exists istVorhanden.EB_{GM}$$

$$def_{PB_{GM}} \equiv \top$$

$$def_{EB_{GM}} \equiv \top$$

$$def_{Bz_{FM}} \equiv \exists istVorhanden.OZ_{FM} \sqcap \exists istVorhanden.KrK_{FM}$$

$$\sqcap \exists istVorhanden.KuK_{FM}$$

$$def_{KuK_{FM}} \equiv \top$$

$$def_{OZ_{FM}} \equiv \top$$

$$def_{BG_{FM}} \equiv \top$$

$$def_{KrK_{FM}} \equiv \top$$

$$EB_{GM} \equiv OZ_{FM} \equiv KrK_{FM} \equiv KuK_{FM}$$

$$PB_{GM} \equiv BG_{FM}$$

$$B_{GM} \equiv Bz_{FM}$$

Die Tabelle 4.2 zeigt die Gültigkeit (Spalte 3) der einzelnen Mappings aus Abbildung 2.3 (Spalten 1 und 2) und die Begründung für die Gültigkeit (Spalte 4). Es sind alle Mappings immer gültig, außer das zwischen dem Feature „Beliebige Kunden Annehmen“ und dem Ziel „Bestellung Annehmen“. Dies ist

Tabelle 4.2: Gültigkeit der Mappings aus Abbildung 2.3.

Goal-Modell Element	Featue	Gültig?	Begründung
Bestellung Annehmen	Vertraute Kunden Annehmen	immer	$\neg def_{BAGM} \sqcup def_{VKAFM}$ $\equiv \neg \top \sqcup \top$ $\equiv \top$
	Beliebige Kunden Annehmen	manchmal	$def_{BAGM} \sqcap def_{VKAFM}$ $\equiv \top \sqcap \exists istVorhanden.KV_{FM}$ $\equiv \exists istVorhanden.KV_{FM}$ $\sqsubseteq \top$
Bei beliebigen Kunden Anwenden	Bestellung Annehmen	immer	$\neg def_{BbKaGM} \sqcup def_{BA_{FM}}$ $\equiv \neg \exists istVorhanden.BA_{GM} \sqcup \exists istVorhanden.VK_{AFM} \sqcup \exists istVorhanden.BK_{AFM}$ $\equiv \neg \exists istVorhanden.BA_{GM} \sqcup \exists istVorhanden.BA_{GM}$ $\equiv \top$

Tabelle 4.3: Gültigkeit der Mappings aus Abbildung 2.4.

Goal-Modell Element	Featue	Gültig?	Begründung
Elektronische Bezahlung	Online Zahlung	immer	$\neg def_{EBGM} \sqcup def_{BG_{FM}}$ $\equiv \neg \top \sqcup \top$ $\equiv \top$
	Kreditkarte	immer	$\neg def_{EBGM} \sqcup def_{KrK_{FM}}$ $\equiv \neg \top \sqcup \top$ $\equiv \top$
	Kundenkarte	immer	$\neg def_{EBGM} \sqcup def_{KuK_{FM}}$ $\equiv \neg \top \sqcup \top$ $\equiv \top$
Persönliche Bezahlung	Bargeld	immer	$\neg def_{PBGM} \sqcup def_{BG_{FM}}$ $\equiv \neg \top \sqcup \top$ $\equiv \top$
Bezahlung	Bezahlung	manchmal	$def_{BGM} \sqcap def_{Bz_{FM}}$ $\equiv (\exists istVorhanden.PB_{GM} \sqcup \exists istVorhanden.EB_{GM}) \sqcap (\exists istVorhanden.OZ_{FM} \sqcap \exists istVorhanden.KrK_{FM} \sqcap \exists istVorhanden.KuK_{FM})$ $\equiv \exists istVorhanden.EB_{GM}$ $\sqsubseteq \top$

der Fall, weil die Beschränkung aus dem Feature-Modell bei der Konfiguration eventuell verletzt wird, wie im Kapitel 2 angedeutet.

Die Tabelle 4.3 stellt die Auswertung der entsprechenden Mappings aus Abbildung 2.4 dar. Es ist zu erkennen, dass hier, wie im Kapitel 2 erläutert, eine ungültige Konfiguration entstehen kann, weil das Mapping zwischen dem Ziel und dem Feature „Bezahlung“ nicht immer gültig ist.

Wir beschränken uns in der restlichen Arbeit auf *immer* gültige Mappings. Solch ein Mapping hält in jeder Konfiguration die Abhängigkeiten eines Features zu seinen Kind-Features ein. Jedoch kann auch bei einem *immer* gültigen Mapping eine fehlerhafte Konfiguration entstehen, da ein Feature nicht nur abhängig ist von seinen Kind-Features. Wird ein Feature in eine Konfiguration aufgenommen, so muss sicher gestellt sein, dass auch sein Vater-Feature in dieser Konfiguration enthalten ist.

Mappings, die manchmal gültig sind, müssen bei einer konkreten Konfiguration erneut validiert werden, um ungültige Konfigurationen zu verhindern. Aus diesem Grund legen wir uns auf immer gültige Mappings fest und fordern folgende Anpassungen der Modelle und Mappings, wenn es dort nie oder nur manchmal gültige Mappings gibt: Wir ändern die Abhängigkeiten im Feature-Modell, die Abhängigkeiten im Goal-Modell oder es werden Mappings zwischen Goal-Modell Elementen und Features hinzugefügt bzw. entfernt.

In unserem Beispiel werden die Mappings dadurch immer gültig, dass wir folgende Anpassungen vornehmen: Die Aufgaben „Testen, ob neuer Kunde“ und „Kreditwürdigkeitsprüfung“ aus dem Goal-Modell werden auf die Features mit den gleichen Namen abgebildet. Das Mapping zwischen dem Ziel „Bei beliebigen Kunden Anwenden“ und dem Feature „Bestellung Annehmen“ wird entfernt und durch zwei Mappings ersetzt. Das erste ersetzende Mapping bildet das Ziel „Bei vertrauten Kunden Anwenden“ auf das Feature „Vertraute Kunden Annehmen“ ab, das zweite das Ziel „Bei beliebigen Kunden Anwenden“ auf das Feature „Beliebige Kunden Annehmen“. Die Mappings der Aufgabe „Bestellung Annehmen“ werden beide entfernt. Um zu garantieren, dass das Feature „Kundenverifizierung“ vorhanden ist, wenn das Ziel „Bei beliebigen Kunden Anwenden“ realisiert werden soll, wird das Ziel „Glaubwürdigkeit Bestimmen“ auf das Feature „Kundenverifizierung“ abgebildet. Zusätzlich wird das Ziel „Bezahlung“ im neuen Goal-Modell mit einer Und-Zerlegung zerlegt anstatt mit einer Oder-Zerlegung, damit das Ziel „Bezahlung“ die Abhängigkeiten des gleichnamigen Features erhält. Der Rest der

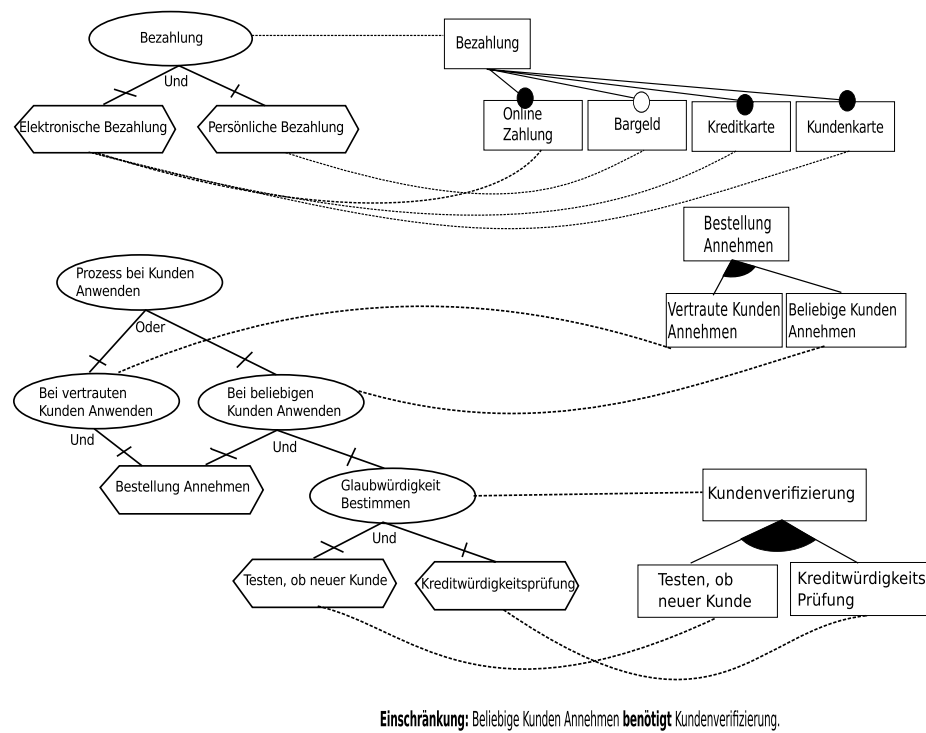


Abbildung 4.2: Änderungen an den Modellen und Mappings.

Modelle und Mappings bleibt unverändert. Die Änderungen werden durch die Abbildung 4.2 verdeutlicht.

Kapitel 5

Konfiguration des Feature-Modells

In diesem Kapitel werden wir die Beschriftung eines Goal-Modells vorstellen (Abschnitt 5.1). Bei einer Beschriftung entscheiden wir, ob ein Ziel oder eine Aufgabe erreicht oder abgelehnt wird. Mithilfe der Beschriftung wählen wir Features aus, die aus der Konfiguration des Feature-Modells ausgeschlossen werden. Da bei der Konfiguration Features entfernt werden, ist nicht sicher gestellt, dass die Konfiguration die Abhängigkeiten aus dem Feature-Modell erhält. Daher ist es notwendig zu überprüfen, ob eine Konfiguration gültig ist, also die Abhängigkeiten aus dem ursprünglichen Modell erhält. Diesen Vorgang erläutert der Abschnitt 5.2.

5.1 Beschriftung

Dieser Abschnitt erklärt, wie die Beschriftung eines Goal-Modells mithilfe von Beschreibungs-Logik vorgenommen wird.

In der Literatur wird bei der Beschriftung eines Goal-Modells zwischen zwei Verfahren unterschieden. Das erste nennt sich Rückwärts-Propagierung (engl. Backward Propagation). Dort wird für die obersten Ziele und Aufgaben entschieden, ob sie erreicht werden oder nicht. Die Rückwärts-Propagierung berechnet, welche „tiefen“ Aufgaben und Ziele erreicht/abgelehnt sein müssen, um die oberen Ziele entsprechend zu erfüllen bzw. abzulehnen. In [SGM04] wird dieses Verfahren vorgestellt.

Wir verwenden eine Vorwärts-Propagierung (engl. Forward Propagation).

Bei diesem Ansatz entscheidet der Entwickler für eine Anzahl möglichst “tiefer“ Elemente (d.h. für Elemente, die nicht weiter zerlegt werden), ob sie erreicht sind oder nicht. Auf dieser Grundlage wird eine Beschriftung für Elemente, die über den Ausgewählten stehen berechnet. Das bedeutet, eine initiale Beschriftung liegt für alle Aufgaben und Ziele des Goal-Modells ohne Zerlegungen und Verbindungen vor und eine Beschriftung für die Elemente der höheren Ebenen wird sukzessive berechnet. Diesen Ansatz erläutert [GMNS02].

Wir nutzen zur Beschriftung die Wissensbasis Σ_G aus Kapitel 4. Die Bedingungen zur Erfüllung eines Elementes aus dem Goal-Modell werden dort durch *def*-Konzepte beschrieben. Das Ziel dieses Abschnitts ist Beschriftungskonzepte hinzuzufügen, die zeigen, ob ein Element erreicht oder abgelehnt ist. Dazu führen wir zu jedem Element g ein neues Konzept $besch_g$ ein. Dieses *besch*-Konzept wird äquivalent zu \top sein, wenn g erreicht ist und äquivalent zu \perp , wenn g abgelehnt ist.

Um eine Beschriftung des Goal-Modells vornehmen zu können, ist eine initiale Beschriftung erforderlich. Der hier beschriebene Ansatz verlangt, dass eine initiale Beschriftung für alle Input-Elemente angegeben wird. Ein Input-Element ist ein Element des Goal-Modells, welches weder eingehende Verbindungen besitzt, noch zerlegt wird. Die Input-Elemente eines Goal-Modells $\{\mathcal{G}, \mathcal{V}, \mathcal{Z}\}$ (Definition 2) sind gegeben durch die Menge:

$$\mathcal{G}_{in} = \mathcal{G} \setminus \{g | d \in \{IOder, XOder, Und\}, Y \in 2^{\mathcal{G}}((g, d, Y) \in \mathcal{Z}) \\ \vee X \in \mathcal{G}, z \in \{+, ++, -, --\}((X, z, g) \in \mathcal{V})\}$$

Es ist die Menge aller Goal-Modell Elemente (\mathcal{G}) ohne diejenigen Elemente, die zerlegt werden ($\{g | d \in \{IOder, XOder, Und\}, Y \in 2^{\mathcal{G}}((g, d, Y) \in \mathcal{Z})$) und ohne Elemente, zu denen eine eingehende komplett positive oder negative Verbindung existiert ($\{g | X \in \mathcal{G}, z \in \{++, --\}((X, z, g) \in \mathcal{V})$).

Im Algorithmus 4 werden die initial erreichten Elemente durch die Menge *ERREICHT*, die initial abgelehnten Elemente durch die Menge *ABGELEHNT* in Zeile 1 angegeben. Es ist notwendig, dass alle Input-Elemente des Goal-Modells eine initiale Beschriftung erhalten, um das Goal-Modell vollständig beschriften zu können. Es darf kein Element existieren, welches initial sowohl mit erreicht als auch mit abgelehnt beschriftet wird, um eine eindeutige Beschriftung des gesamten Goal-Modells zu erzeugen. Diese Bedingungen werden in Zeile 2 überprüft.

Algorithm 4 Hinzufügen von Beschriftungs-Axiomen zu Σ_M .

```

1: input: Goal-Modell  $\{\mathcal{G}, \mathcal{V}, \mathcal{Z}\}$ , daraus transformierte Wissensbasis
    $\Sigma_G, ABGELEHNT \subseteq \mathcal{G}, ERREICHT \subseteq \mathcal{G}$ 
2: assert:  $ABGELEHNT \cup ERREICHT = \mathcal{G}_{in} \wedge ABGELEHNT \cap$ 
    $ERREICHT = \emptyset$ 
3:  $\Sigma_{GL} = \Sigma_G$ 
4: for all  $g \in \mathcal{G}$  do
5:   if  $g \in ABGELEHNT$  then
6:      $\Sigma_{GL} = \Sigma_{GL} \cup besch_g \sqsubseteq \perp$ 
7:   else
8:     if  $g \in ERREICHT$  then
9:        $\Sigma_{GL} = \Sigma_{GL} \cup besch_g \equiv \top$ 
10:    else
11:       $\Sigma_{GL} = \Sigma_{GL} \cup besch_g \equiv def_{g_g}$ 
12:    end if
13:  end if
14:   $\Sigma_{GL} = \Sigma_{GL} \cup besch_g \equiv \exists istVorhanden.g$ 
15: end for
16: return  $\Sigma_{GL}L$ 

```

Ist ein Goal-Modell Element initial mit abgelehnt zu beschriften, so wird sein *besch*-Konzept in den Zeilen 5-7 als ein Subkonzept von \perp ($besch_g \sqsubseteq \perp$) zu Σ_{GL} hinzugefügt. Soll es hingegen initial mit erreicht beschriftet werden, so wird die Wissensbasis Σ_{GL} um das Axiom $besch_g \equiv \top$ erweitert, sodass alle konsistenten Konzepte zu einem Subkonzept von diesem *besch*-Konzept werden (Zeilen 8-10). Ist das betrachtete Element kein Input-Element, so kann es nur mit erreicht beschriftet werden, wenn es durch eine Zerlegung oder eine eingehende Verbindung des Goal-Modells erfüllt wird. Diese Bedingungen haben wir in Kapitel 4 durch *def*-Konzepte modelliert. Aus diesem Grund fügen wir in diesem Fall ein Axiom hinzu, welches das *besch*-Konzept eines Nicht-Input-Elementes äquivalent zu seinem *def*-Konzept beschreibt (Zeile 11).

Wir haben nun erreicht, dass alle *besch*-Konzepte von Input-Elementen entweder inkonsistent sind ($besch_g \sqsubseteq \perp$) oder ein Superkonzept aller anderen Konzepte darstellen ($besch_g \equiv \top$). Damit auch die *besch*-Konzepte von Nicht-Input-Elementen entweder durch \top oder durch \perp beschrieben werden, ist es erforderlich, dass alle *besch*-Konzepte die Bedingungen der *def*-Konzepte eindeutig erfüllen oder nicht erfüllen. Das Konzept $besch_g$ eines erreichten Elementes g soll repräsentieren, dass das Element g für die restlichen Elemente des Goal-Modells vorhanden ist. Wird ein Element hingegen mit abgelehnt beschriftet, so soll es für die anderen Elemente des Modells als nicht vorhanden gekennzeichnet werden. Dies wird erreicht, indem ein *besch*-Konzept zu einem Element g immer äquivalent zu $\exists istVorhanden.g$ ist (Zeile 14).

Tabelle 5.1: Begründung für die Beschriftung.

Goal-Modell Element	Beschriftung	Begründung
Bestellung Annehmen	init. erreicht	$besch_{BA_g}$ $\equiv \top$
Testen, ob neuer Kunde	init. erreicht	$besch_{TonK}$ $\equiv \top$
Kreditwürdigkeitsprüfung	init. erreicht	$besch_{Kw}$ $\equiv \top$
Glaubwürdigkeit Bestimmen	erreicht	$besch_{GB}$ $\equiv def_{GB}$ $\equiv besch_{TonK} \sqcup besch_{Kw}$ $\equiv \top \sqcup \top$
Bei beliebigen Kunden Anwenden	erreicht	$besch_{BbKa_g}$ $\equiv def_{BbKa_g}$ $\equiv besch_{BA} \sqcap besch_{GB}$ $\equiv \top \sqcap \top$
Bei vertrauten Kunden Anwenden	erreicht	$besch_{BvKa_g}$ $\equiv def_{BvKa_g}$ $\equiv besch_{BA}$ $\equiv \top$
Elektronische Bezahlung	init. abgelehnt	$besch_{EB}$ $\sqsubseteq \perp$
Persönliche Bezahlung	init. abgelehnt	$besch_{PB}$ $\sqsubseteq \perp$
Bezahlung	abgelehnt	$besch_B$ $\equiv def_B$ $\equiv besch_{EB} \sqcap besch_{PB}$ $\equiv \perp \sqcap \perp$

Bezahlung“ und „Persönliche Bezahlung“ sind abgelehnt. Alle Elemente, welche durch diese initiale Beschriftung mit abgelehnt zu beschriften sind, wurden in der Abbildung durchgestrichen.

Die Tabelle 5.1 veranschaulicht die Begründungen für die Beschriftungen einiger Elemente aus Abbildung 5.1. Oben stehen die Input-Elemente, für die eine initiale Beschriftung gegeben ist. Aus diesen lassen sich die weiter unten stehenden Begründungen ableiten.

5.2 Konfiguration

In diesem Abschnitt werden wir Features aus dem ursprünglichen Modell entfernen, sodass wir ein neues Feature-Modell erhalten, das wir eine Konfiguration nennen. Entfernt werden diejenigen Features, die nur auf abgelehnte Ele-

mente des beschrifteten Goal-Modells abgebildet wurden. Wir verwenden die Wissensbasis Σ_F aus Kapitel 4.2, die das Feature-Modell in DL repräsentiert und die Wissensbasis Σ_{GL} , die das beschriftete Goal-Modell aus dem vorigen Abschnitt darstellt.

Im Abschnitt 4.3 haben wir die Mappings zwischen Goal- und Feature-Modell in drei Kategorien eingeteilt: *manchmal*-, *nie*- und *immer*-gültig. Dort haben wir uns entschlossen, nur *immer* gültige Mappings zur Konfiguration zuzulassen. Bei einer Konfiguration mithilfe von *immer* gültigen Mappings zu einem Goal-Modell müssen wir nur noch sicherstellen, dass die Vater-Features der übernommenen Features ebenfalls in die Konfiguration aufgenommen werden. Die Abhängigkeiten, die sich von einem Feature zu seinen Kind-Features ergeben, werden durch *immer* gültige Mappings bei der Konfiguration immer eingehalten.

Wir indizieren jedes Vorkommen eines atomaren Konzeptes in Σ_{GL} mit den Buchstaben GM , die so erzeugte neue Wissensbasis nennen wir Σ_{GLGM} . Auf die gleiche Weise indizieren wir jedes atomare Konzept in Σ_F mit FM und nennen die neue Wissensbasis Σ_{FFM} . Somit ist sichergestellt, dass in Σ_{FFM} kein atomares Konzept vorkommt, welches auch in Σ_{GLGM} vorkommt.

Der Algorithmus 5 erzeugt die Wissensbasis Σ_{ML} , welche die Konfiguration des Feature-Modells beinhaltet. Wir übergeben dem Algorithmus das Feature-Modell $\{\mathcal{F}, \mathcal{F}_{obl}, \mathcal{F}_{opt}, \mathcal{F}_{group}, \mathcal{R}_{VK}, \mathcal{R}_{VG}, \mathcal{A}\}$, die Menge der intentionalen Elemente aus dem Goal-Modell (\mathcal{G}) und die beiden oben beschriebenen Wissensbasen Σ_{GLGM} und Σ_{FFM} (Zeile 1).

In Zeile 2 werden die Axiome aus Σ_{GLGM} und Σ_{FFM} zu der Konfigurations-Wissensbasis hinzugefügt. Die Wissensbasis Σ_{FFM} beinhaltet für alle Features, zu denen es keine Abhängigkeiten gibt, ein *def*-Konzept, welches äquivalent zu \top ist. Diese Axiome wurden bei der Überprüfung der Mappings benötigt, müssen aber in der Konfigurations-Wissensbasis entfernt werden (Zeilen 4-6), da ein *def*-Konzept hier nur dann konsistent sein soll, wenn das entsprechende Feature in der Konfiguration übernommen wird.

Das *konfig*-Konzept eines Features repräsentiert, wie die *besch*-Konzepte im Goal-Modell, das Vorhandensein des Features in der Konfiguration. Daher wird der Wissensbasis in Zeile 7 zu jedem Feature f das Axiom $konfig_{f_{FM}} \equiv \exists istVorhanden.f_{FM}$ hinzugefügt. Ein Feature kann nur dann in die Konfiguration aufgenommen werden, wenn sein *def*-Konzept konsistent ist und das

Algorithm 5 Erzeugung der Konfigurations-Wissensbasis Σ_{ML} .

```

1: input:  $\Sigma_{FFM}, \Sigma_{GL}, \{\mathcal{F}, \mathcal{F}_{obl}, \mathcal{F}_{opt}, \mathcal{F}_{group}, \mathcal{R}_{VK}, \mathcal{R}_{VG}, \mathcal{A}\}, \mathcal{G}$ 
2:  $\Sigma_{ML} = \Sigma_{FFM} \cup \Sigma_{GLGM}$ 
3: for all  $f \in \mathcal{F}$  do
4:   if  $def_{FFM} \equiv \top \in \Sigma_{FFM}$  then
5:      $\Sigma_{ML} = \Sigma_{ML} \setminus \{def_{FFM} \equiv \top\}$ 
6:   end if
7:    $\Sigma_{ML} = \Sigma_{ML} \cup konfig_{f_{FM}} \equiv \exists istVorhanden.f_{FM}$ 
8:   if  $((v, \_, G) \in \mathcal{R}_{VG} \wedge f \in G) \vee (v, f) \in \mathcal{R}_{VK}$  then
9:      $\Sigma_{ML} = \Sigma_{ML} \cup konfig_{f_{FM}} \equiv \exists istVorhanden.v_{FM} \sqcap def_{f_{FM}}$ 
10:  else
11:     $\Sigma_{ML} = \Sigma_{ML} \cup konfig_{f_{FM}} \equiv def_{f_{FM}}$ 
12:  end if
13:   $i = 0$ 
14:  for all  $g \in \mathcal{G}$  do
15:    if  $(f, hatMapping, IntElement, g) \in \mathcal{A}$  then
16:       $i = i + 1$ 
17:       $union_i = besch_{g_{GM}}$ 
18:    end if
19:  end for
20:  if  $i > 0$  then
21:     $\Sigma_{ML} = \Sigma_{ML} \cup konfig_{f_{FM}} \equiv \bigsqcup_{j=1}^i union_j$ 
22:  end if
23: end for
24: return  $\Sigma_{ML}$ 

```

Vater-Feature ebenfalls in die Konfiguration aufgenommen wird. Diese Bedingungen werden durch die Axiome aus den Zeilen 8-10 modelliert. Hat das betrachtete Feature kein Vater-Feature (es ist das Wurzel-Feature), so muss nur das entsprechende *def*-Konzept konsistent sein, um es aufzunehmen (Zeilen 10-12).

Existieren zu einem Feature f Mappings zu den Goal-Modell Elementen g_1, \dots, g_n , so fügen wir der Wissensbasis das Axiom $config_{f_{FM}} \equiv \bigwedge_{i=1}^n besch_{g_{i_{GM}}}$ hinzu, wie in den Zeilen 14-22 angedeutet. Damit erreichen wir, dass Features, welche auf ein Element des Goal-Modells abgebildet werden, das erreicht ist, in der Konfiguration verbleiben ($config_{f_{FM}} \equiv \top$). Features, welche nur auf abgelehnte Goal-Modell Elemente abgebildet sind, werden aus der Konfiguration ausgeschlossen ($config_{f_{FM}} \equiv \perp$).

Da in den *config*-Konzepten auch modelliert wird, dass das Eltern-Feature aufgenommen sein muss, werden auch Features, deren Eltern nicht in der Konfiguration verbleiben ausgeschlossen. Dies kann in folgenden Situationen zu Wissensbasen führen, zu denen es kein Modell gibt: Ein Feature f mit dem Vater-Feature v wird auf ein Ziel abgebildet, das mit erreicht beschriftet wird ($config_f \equiv \top \equiv config_v \sqcap def_f$). Das Vater-Feature v von f wird nicht in die Konfiguration aufgenommen ($config_v \equiv \perp$). Nun erhalten wir $config_f \equiv \top \equiv \perp$. In solchen Fällen bezeichnen wir die Beschriftung des Goal-Modells als unpassend zum Feature-Modell. Durch die Einschränkung auf immer gültige Mappings, haben wir ausgeschlossen, dass eine Beschriftung aufgrund der Abhängigkeiten eines Features zu seinen Kindern unpassend ist.

Ist die erzeugte Wissensbasis erfüllbar, so können wir alle Features aus dem Feature-Modell entfernen, deren *config*-Konzept inkonsistent ist und erhalten eine Konfiguration, in der die Abhängigkeiten des ursprünglichen Feature-Modells erhalten bleiben.

Die Abbildung 5.2 zeigt, welche Features aufgrund der Mappings aus der Abbildung 4.2 und der Beschriftung des Goal-Modells (Abbildung 5.1) bei der Konfiguration entfernt werden müssen. In der Abbildung 5.3 wird veranschaulicht, welche Features zusätzlich entfernt werden müssen, weil ihre obligatorischen Kind-Features oder ihre Eltern-Features nicht in der Konfiguration vorhanden sind.

Die Tabelle 5.2 gibt eine Begründung für einen Teil der Konfiguration an.

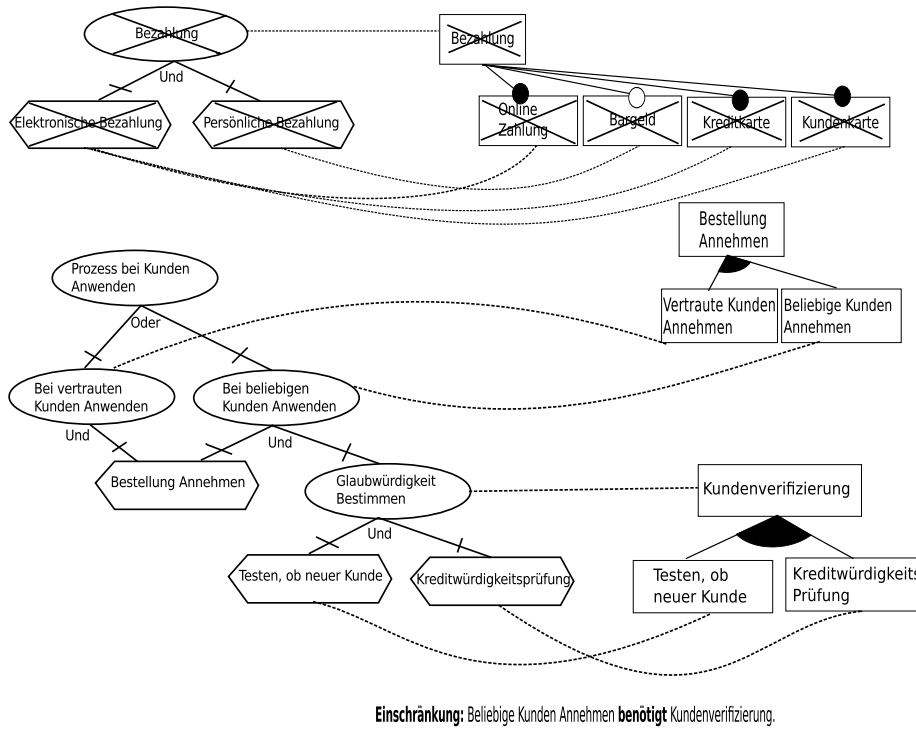


Abbildung 5.2: Entfernte Features aufgrund von Mappings.

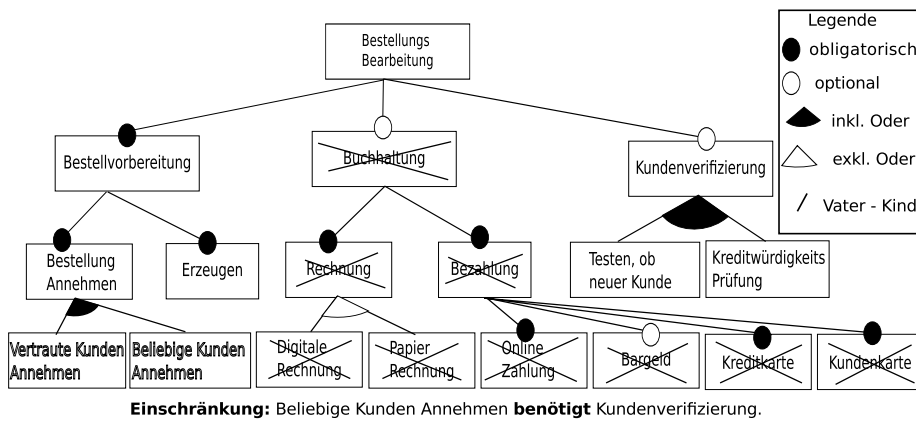


Abbildung 5.3: Konfiguration des Feature-Modells aus Abbildung 2.1.

Tabelle 5.2: Begründung für die Konfiguration.

Feature	übernommen?	Begründung
Testen, ob neuer Kunde	ja	$config_{ToNK_{FM}}$ $\equiv besch_{ToNK_{GM}}$ $\equiv \top$
Kreditwürdigkeitsprüfung	ja	$config_{Kw_{FM}}$ $\equiv besch_{Kw_{GM}}$ $\equiv \top$
Kundenverifizierung	ja	$config_{Kv_{FM}}$ $\equiv besch_{Gv_{GM}}$ $\equiv \top$
Online Zahlung, Kreditkarte, Kundenkarte	nein	$config_{*_{FM}}$ $\equiv besch_{EB_{GM}}$ $\equiv \perp$
Bargeld	nein	$config_{B_{FM}}$ $\equiv besch_{PB_{GM}}$ $\equiv \perp$
Bezahlung	nein	$config_{B_{FM}}$ $\equiv besch_{B_{GM}}$ $\equiv \perp$
Buchhaltung	nein	$config_{Bh_{FM}}$ $\equiv def_{Bh_{FM}} \sqcap config_{BB_{FM}}$ $\equiv config_{R_{FM}} \sqcap config_{B_{FM}} \sqcap$ $config_{BB_{FM}}$ $\equiv config_{R_{FM}} \sqcap \perp \sqcap config_{BB_{FM}}$ $\equiv \perp$
Rechnung	nein	$config_{R_{FM}}$ $\equiv def_{R_{FM}} \sqcap config_{Bh_{FM}}$ $\equiv def_{R_{FM}} \sqcap \perp$ $\equiv \perp$
Digitale Rechnung, Papier Rechnung	nein	$config_{*_{FM}}$ $\equiv def_{*_{FM}} \sqcap config_{R_{FM}}$ $\equiv def_{*_{FM}} \sqcap \perp$ $\equiv \perp$

Werden in einer Zeile mehrere Features angegeben (Digitale Rechnung, Papier Rechnung und Online Zahlung, Kreditkarte, Kundenkarte) so wird in dieser Zeile ein „*“ als Platzhalter in den *def*- und *konfig*-Konzepten benutzt.

Kapitel 6

Abschluss

Dieses Kapitel stellt eine abschließende Betrachtung der Bachelorarbeit dar. Die erarbeiteten Ergebnisse und eine Evaluierung werden im Abschnitt 6.1 zusammengetragen. Im Abschnitt 6.2 setzen wir uns mit verwandten Arbeiten auseinander. Der Abschnitt 6.3 fasst diese Arbeit zusammen.

6.1 Evaluierung

Um diese Arbeit zu evaluieren, haben wir ein Goal-Modell und ein Feature-Modell aus einer eshop Fallstudie [Lau06] benutzt. Die beiden Modelle wurden unabhängig voneinander entwickelt, um einen realitätsnahen Testfall zu betrachten. Um unseren Ansatz zu überprüfen, haben wir Mappings zwischen Feature- und Goal-Modell erzeugt. Zur Modellierung des Feature-Modells und der Mappings wurde das Feature Mapper Tool¹ eingesetzt. Das Goal-Modell wurde mithilfe des jUCMNav Tools² modelliert. Unser Programm ist Teil des TWOUSE Toolkits³ und wurde als Eclipse-Plugin⁴ implementiert.

Das verwendete Goal-Modell besteht insgesamt aus 24 Zielen, 45 Aufgaben und 6 qualitativen Zielen. Diese intentionalen Elemente wurden mit 49 Zerlegungen und 19 Verbindungen miteinander verknüpft. Im Feature-Modell existieren 287 Features von denen 70 obligatorisch sind. Die Features sind durch 50 Vater-Kind Relationen, 30 IOder-Gruppierungen und 3 XOder-Gruppierungen gegliedert. Wir haben 87 Mappings modelliert. Die Mappings wurden mithilfe eines Feature-Attributes namens *mapsTo* im Feature-Modell realisiert, welches

¹FeatureMapper update Seite: <http://featuremapper.org/update/>

²jUCMNav tool Seite: <http://www.ohloh.net/p/11712>

³TWOUSE Google code Seite: <http://code.google.com/p/twouse/>

⁴Eclipse IDE Download Seite: <http://www.eclipse.org/downloads/>

als Wert den Namen des Goal-Modell Elementes hat, auf das das Feature abgebildet werden soll.

Sind diese Modelle gegeben, so erzeugt unser Programm drei Wissensbasen: Σ_F für das Feature-Modell (Abschnitt 4.2), Σ_G für das Goal-Modell (Abschnitt 4.1) und Σ_M , um die Mappings auszuwerten (Abschnitt 4.3). Die Beschreibungs-Logik Ausdrucksstärke dieser Wissensbasen ist \mathcal{ALC} . Nach dem Reasoning auf der Mapping-Wissensbasis gibt das Programm eine Liste von immer gültigen und eine Liste von manchmal oder nie gültigen Mappings aus (vgl. Abschnitt 4.3).

Anhand dieser Listen haben wir eine Anpassung des Feature-Modells und der Mappings vorgenommen, sodass jedes Mapping immer gültig wurde. Das Programm erzeugt zur Konfiguration des Feature-Modells die Wissensbasis Σ_{ML} , wie in Kapitel 5 beschrieben. Ob die Konfiguration valide ist oder nicht, entscheidet das Programm durch einen Erfüllbarkeits-Test über diese Wissensbasis.

Die Wissensbasen wurden mithilfe der OWL-API⁵ erzeugt. Unser Test-System war ein Notebook, mit einem Intel Core 2 Duo T7300 Prozessor (2.0 GHz, 800 MHz FSB, 4 MB L2 Cache und 2GB DDR2 RAM). Wir haben in der Java-VM 256MB Speicher genutzt. In diesem System nahm die Auswertung der Mapping-Wissensbasis (Σ_M) 2863ms Zeit in Anspruch. Die Auswertung der Konfigurations-Wissensbasis dauerte 3462ms.

Obwohl die Zeit-Komplexität für das Reasoning theoretisch *exponentiell* ist, war das Reasoning auf den Wissensbasen in diesem Test nach einigen Sekunden abgeschlossen (3,4s). Es hat sich herausgestellt, dass das Anpassen des Feature-Modells und der Mappings zur Vorbereitung auf die Konfiguration den langwierigsten Arbeitsschritt darstellt, wir haben etwa eine Stunde damit verbracht. Unser Programm kann anhand von gegebenen Beschriftungen des Goal-Modells und der Mappings aus dem Feature-Modell entscheiden, ob eine gültige Konfiguration zustande kommt oder nicht. Das Programm liefert keine Anhaltspunkte für Anpassungen der Goal-Modell-Beschriftung, um eine gültige Konfiguration des Feature-Modells zu erhalten.

⁵OWL-API Seite: <http://owlapi.sourceforge.net/>

6.2 Verwandte Arbeit und Diskussion

In [IT08] wird das Meta-Modell für Goal-Modelle angegeben, auf die wir uns beziehen. Die Modellierung der Feature-Modelle stammt aus [PBL05].

Das Papier [YPLLM08] stellt eine Methode vor, mit der aus Goal-Modellen Feature-Modelle erzeugt werden. Dazu wird das Goal-Modell um eine Reihe von Annotationen erweitert. Darauf aufbauend wird in [YLL⁺08] ein systematischer Prozess vorgestellt, der verschiedene Design Sichten aus einem Goal-Modell erzeugt, wobei die Variabilität des Goal-Modells (die verschiedenen Möglichkeiten, wie ein Ziel erfüllt werden kann) erhalten bleibt. Wie ein Geschäfts-Prozess-Modell (engl. Business Process Model BPM) aus einem Goal-Modell erzeugt und konfiguriert werden kann, wird in [LYM07] erläutert. Die Verwandtschaft zwischen Feature- und Goal-Modellen wird in [AAS09] und [BS09] untersucht. Diese Papiere erörtern einen Prozess, um Feature-Modelle mithilfe von heuristischen Regeln aus einem Goal-Modell zu erzeugen.

In allen diesen Arbeiten werden Feature-Modelle aus Goal-Modellen abgeleitet, während wir einen Ansatz vorstellen, bei dem das Feature-Modell unabhängig von einem Goal-Modell erzeugt wurde. Wir versuchen dann eine Konfiguration des Feature-Modells zu finden, welche sowohl zum Goal-Modell passt als auch die Bedingungen des Feature-Modells erfüllt.

Um Feature-Modell Konfigurationen zu verifizieren, nutzen die Autoren aus [CP06] OCL (Object Constraint Language) Bedingungen und SAT-Solver. In [TBKC07] wird dieses Verfahren erweitert und der SAT-Solver zusätzlich dazu genutzt, Referenzen auf undefinierte Methoden, Klassen und Variablen zu identifizieren. Die Korrektheit von Mappings zwischen Feature- und Komponenten-Modellen wird in [JB08] und [VDS07] mithilfe von Aussagenlogik überprüft. Ein DL-Basierter Ansatz zur Überprüfung von Feature-Modellen mithilfe von Workflow-Patterns wird in [GWB⁺11] vorgestellt.

Diese Ansätze beschäftigen sich alle mit der Identifizierung von Inkonsistenzen zwischen Design-Modellen und Feature-Modellen, um eines der Modelle zu ändern. Wir spezialisieren uns auf die Validierung von Abbildungen zwischen Feature- und Goal-Modellen, um in einem weiteren Schritt eine konkrete Konfiguration des Feature-Modells vorzunehmen.

Wir nutzen zur Validierung der Mappings zwischen Goal- und Feature-Modellen die Ideen aus [ABGH11], wo ein Verfahren vorgestellt wird, das ent-

scheidet, ob ein Goal-Modell zur Konfiguration des Feature-Modells genutzt werden kann.

Zur Konfiguration des Feature-Modells beschriften wir das Goal-Modell. Das Papier [GMNS02] stellt ein Rückwärts-Propagierungs Verfahren vor, das mithilfe von SAT-Solvern die Beschriftung vornimmt. In [GMNS03] wird der Vorwärts-Propagierungs Ansatz erläutert, den wir mithilfe von Beschreibungs-Logik implementiert haben.

6.3 Zusammenfassung

Wir haben in dieser Arbeit ein Verfahren vorgestellt, dass ein Feature-Modell mithilfe von Abbildungen auf ein Goal-Modell konfiguriert. Dazu werden beide Modelle in eine Beschreibungs-Logik Wissensbasis transformiert (Σ_G und Σ_F). Diese werden dann zusammen mit den Abbildungen zu einer neuen Wissensbasis Σ_M vereinigt. Nach dem Reasoning Prozess auf Σ_M können wir jede Abbildung in eine Kategorie einteilen, *immer-*, *nie-*, oder *manchmal-*gültig. Sollte es Abbildungen in Σ_M geben, die nie oder nur manchmal gültig sind, so ist eine Anpassung der Modelle oder Mappings erforderlich, um die Anzahl unpassender Feature-Modell Konfigurationen einzuschränken.

Existieren nur immer gültige Mappings zwischen Feature- und Goal-Modell, können wir im nächsten Schritt eine Wissensbasis (Σ_{GL}) aus dem Goal-Modell und einer initialen Beschriftung ableiten, die der Beschriftung des Goal-Modells dient. Diese Wissensbasis wird zusammen mit Σ_F dazu genutzt, das Feature-Modell zu konfigurieren. Dazu werden Σ_{GL} und Σ_F in einer neuen Wissensbasis Σ_{ML} vereinigt und um einige Axiome erweitert. Ob sich eine gültige Konfiguration des Feature-Modells erzeugen lässt oder nicht, wird mithilfe eines Erfüllbarkeitstests auf Σ_{ML} entschieden. Ist dieser Test erfolgreich (Σ_{ML} ist erfüllbar), lässt sich die Konfiguration des Feature-Modells nach einem Reasoning über Σ_{ML} ablesen.

Literaturverzeichnis

- [AAS09] ANTÓNIO, Sandra ; ARAÚJO, João ; SILVA, Carla: Adapting the i* Framework for Software Product Lines. In: *Proc. of the ER 2009 Workshops on Advances in Conceptual Modeling - Challenging Perspectives*, Springer, 2009. – ISBN 978-3-642-04946-0, S. 286–295
- [ABGH11] ASADI, Mohsen ; BAGHERI, Ebrahim ; GASEVIC, Dragan ; HATALA, Marek: Goal-Driven Software Product Line Engineering. In: *The Proceedings of the 26th Annual ACM Symposium on Applied Computing (SAC 2011)*, 2011, S. 691–698
- [BCM⁺03] BAADER, Franz (Hrsg.) ; CALVANESE, Diego (Hrsg.) ; MCGUINNESS, Deborah L. (Hrsg.) ; NARDI, Daniele (Hrsg.) ; PATELSCHNEIDER, Peter F. (Hrsg.): *The Description Logic Handbook: Theory, Implementation, and Applications*. New York, NY, USA : Cambridge University Press, 2003
- [BS09] BORBA, Clarissa ; SILVA, Carla: A Comparison of Goal-Oriented Approaches to Model Software Product Lines Variability. In: *Proc. of the ER 2009 Workshops on Advances in Conceptual Modeling - Challenging Perspectives*, Springer, 2009, S. 244–253
- [CP06] CZARNECKI, Krzysztof ; PIETROSZEK, Krzysztof: Verifying feature-based model templates against well-formedness OCL constraints. In: *Proc. of GPCE '06*. New York, NY, USA : ACM, 2006, S. 211–220
- [GMNS02] GIORGINI, Paolo ; MYLOPOULOS, John ; NICCHIARELLI, Eleonora ; SEBASTIANI, Roberto: Reasoning with Goal Models. In: *Proc. of the 21st International Conference on Conceptual Modeling (ER) Bd. 2503*, Springer, 2002 (LNCS), S. 167–181
- [GMNS03] GIORGINI, Paolo ; MYLOPOULOS, John ; NICCHIARELLI, Eleonora ; SEBASTIANI, Roberto: Formal Reasoning Techniques for Goal Models. In: *J. Data Semantics* 1 (2003), S. 1–20

- [GWB⁺11] GRÖNER, Gerd ; WENDE, Christian ; BOŠKOVIĆ, Marko ; PARRERAS, Fernando S. ; WALTER, Tobias ; HEIDENREICH, Florian ; GAŠEVIĆ, Dragan ; STAAB, Steffen: Validation of Families of Business Processes. In: *The Proceedings of the 23rd CAiSE Conference (CAiSE2011)* Bd. 6741, Springer, 2011 (LNCS), S. 551–565
- [IT08] ITU-T: *Recommendation Z.151 (09/08): User Requirements Notation (URN) – Language Definition*. Geneva, Switzerland, 2008
- [JB08] JANOTA, Mikolás ; BOTTERWECK, Goetz: Formal Approach to Integrating Feature and Architecture Models. In: *FASE* Bd. 4961, 2008 (LNCS), S. 31–45
- [Lau06] LAU, Sean Q.: *Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates*. Waterloo, University of Waterloo, Diplomarbeit, 2006. – 168 S.
- [LYM07] LAPOUCHNIAN, Alexei ; YU, Yijun ; MYLOPOULOS, John: Requirements-Driven Design and Configuration Management of Business Processes. In: *5th Int. Conf. on Business Process Management, BPM*, Springer, 2007 (LNCS), S. 246–261
- [MCY99] MYLOPOULOS, John ; CHUNG, Lawrence ; YU, Eric: From Object-oriented to Goal-oriented Requirements Analysis. In: *Commun. ACM* 42 (1999), January, S. 31–37
- [MMYJ10] MCGREGOR, J.D. ; MUTHIG, D. ; YOSHIMURA, K. ; JENSEN, P.: Successful Software Product Line Practices. In: *Software, IEEE* 27 (2010), Nr. 3, S. 16–21
- [PBL05] POHL, Klaus ; BÖCKLE, Günter ; LINDEN, Frank J. d.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005
- [SGM04] SEBASTIANI, Roberto ; GIORGINI, Paolo ; MYLOPOULOS, John: Simple and Minimum-Cost Satisfiability for Goal Models. In: *CAiSE* Bd. 3084, Springer, 2004 (LNCS), S. 20–35
- [TBKC07] THAKER, Sahil ; BATORY, Don S. ; KITCHIN, David ; COOK, William R.: Safe Composition of Product Lines. In: *Proceedings of the GPCE '06*, 2007, S. 95–104
- [VDS07] VAN DER STORM, T.: Generic Feature-based Software Composition. In: *6th Int. Conference on Software Composition* Bd. 4829, 2007 (LNCS), S. 66–80

- [YLL⁺08] YU, Yijun ; LAPOUCHNIAN, Alexei ; LIASKOS, Sotirios ; MYLOPOULOS, John ; LEITE, Julio: From Goals to High-Variability Software Design. In: *Foundations of Intelligent Systems* Bd. 4994. Springer, 2008, S. 1–16
- [YPLLM08] YU, Yijun ; PRADO LEITE, Julio Cesar S. ; LAPOUCHNIAN, Alexei ; MYLOPOULOS, John: Configuring Features with Stakeholder Goals. In: *Proceedings of the 2008 ACM Symposium on Applied Computing*. New York, NY, USA : ACM, 2008 (SAC '08), S. 645–649