Interner Bericht

Bessere Änderbarkeit durch Verfolgbarkeit: Auswahl eines Ansatzes

Kerstin Brockhage
Christiane Differding
Antje von Knethen

301 / 99

Fachbereich Informatik

Universität Kaiserslautern • Postfach 3049 • D-67653 Kaiserslautern

Kurzfassung

Im Bereich des Software Engineering werden komplexe Software-Entwicklungsprojekte betrachtet. Im Rahmen dieser Projekte werden große Mengen von Informationen bearbeitet. Diese Informationen werden in Software-Artefakten (z.B. in Projektplänen oder Entwicklungsdokumenten, wie Anforderungsbeschreibungen) festgehalten. Die Artefakte werden während der Entwicklung und der Wartung eines Softwaresystems häufig geändert. Änderungen einer Information in einem Artefakt haben häufig Änderungen im selben und in anderen Artefakten zur Folge, da Beziehungen innerhalb und zwischen den in den Artefakten festgehaltenen Informationen bestehen. Die Beziehungen liegen meist nicht explizit vor, so daß die Konsequenzen einer Änderung schwer zu überblicken sind. In dieser Arbeit wurde ein Verfolgbarkeitsansatz ausgewählt, der den Benutzer bei der Durchführung von Änderungen an Artefakten unterstützt. Unterstützung bedeutet hierbei, daß der Aufwand zur Durchführung einer Änderung reduziert wird und weniger Fehler bei der Durchführung gemacht werden.

In der Arbeit wurden Anforderungen an einen auszuwählenden Verfolgbarkeitsansatz gestellt. Eine Anforderung war, daß er auf verschiedene Bereiche des Software Engineering, wie z.B. Systementwurf oder Meßplanung, mit jeweils sehr unterschiedlichen Artefakten, anwendbar sein sollte. Die durchgeführte Literaturrecherche und die anschließende Bewertung anhand der gestellten Anforderungen ergaben, daß das Prinzip der Metamodellierung in Verbindung mit Wissensbankverwaltungssystemen ein geeigneter Verfolgbarkeitsansatz ist. Eine Evaluation, die sich auf Fallstudien aus den Bereichen "Objektorientierter Entwurf mit UML" und "Meßplanung mit GQM" bezog, ergab, daß das Wissensbankverwaltungssystem ConceptBase, das auf der Wissensrepräsentationssprache O-Telos basiert, ein geeignetes Werkzeug zur Unterstützung des Verfolgbarkeitsansatzes ist.

Inhaltsverzeichnis

Kapitel 1

Einführung

Software Engineering betrachtet komplexe Software-Entwicklungsprojekte. Im Rahmen dieser Projekte wird mit große Mengen von Informationen gearbeitet. Die Informationen werden in Software-Artefakten dokumentiert (z.B. in einem "umfassenden Projektplan" oder in Entwicklungsdokumenten, wie Anforderungsbeschreibungen). Dabei sind die Notationen sehr unterschiedlich, die die Informationen in den verschiedenen Software-Artefakten festhalten (z.B. Notationen für Zustandsdiagramme, Klassendiagramme, Meßpläne oder Projektpläne).

Die Artefakte werden während der Entwicklung und Wartung von Softwaresystemen häufig geändert. Da Beziehungen innerhalb (z.B. zwischen verschiedenen Sichten in der Anforderungsbeschreibung) und zwischen den Software-Artefakten (z.B. zwischen Anforderungs- und Entwurfsbeschreibung) bestehen, haben Änderungen einer Information in einem Software-Artefakt häufig Änderungen im selben und in anderen Artefakten zur Folge.

Die Beziehungen innerhalb und zwischen Software-Artefakten liegen meist nicht explizit vor, so daß die Konsequenzen einer Änderung schwer zu überblicken sind. Wird eine Information geändert, so ist es schwierig, die Informationen, die von dieser Änderung ebenfalls betroffen sind, zu identifizieren. In [BA96] wird die Fähigkeit, Beziehungen zwischen Software-Artefakten, die während des Lebenszyklus eines Software-Produkts erzeugt und modifiziert werden, explizit definieren und verfolgen zu können, mit "Traceability" (Verfolgbarkeit) bezeichnet.

Im folgenden unterscheiden wir zwischen drei Arten von Verfolgbarkeit: 1. Verfolgbarkeit innerhalb einer Sicht in einem Software-Artefakt, 2. Verfolgbarkeit zwischen verschiedenen Sichten innerhalb eines Software-Artefakts und 3. Verfolgbarkeit zwischen verschiedenen Software-Artefakten.

Das Ziel dieser Arbeit ist die Auswahl eines Verfolgbarkeitsansatzes zum Zwecke der Verbesserung von Änderbarkeit innerhalb eines Software-Artefakts (Verfolgbarkeit innerhalb einer Sicht und zwischen Sichten). Verbesserung von Änderbarkeit bedeutet hierbei, daß der Aufwand zur Durchführung einer Änderung reduziert wird und weniger Fehler bei der Durchführung gemacht werden. Der Verfolgbarkeitsansatz soll allgemein verwendbar sein, da er in verschiedenen Bereichen des Software Engineering, wie z.B. Meßplanung und Softwareentwurf, mit sehr unterschiedlichen Arten von Software-Artefakten angewendet werden soll.

Kapitel 2 dieser Arbeit erläutert das oben angerissene Problem ausführlich. Kapitel 3 legt die Ziele der Arbeit fest und leitet aus diesen Zielen Anforderungen an geeignete Lösungsansätze ab. In Kapitel 4 wird der aktuellen Forschungsstand betreffend des Problems und der Ziele charakterisiert. Die bei der Recherche gefundenen Lösungsansätze werden anhand der Anforderungen aus Kapitel 3 bewertet, und ein geeigneter Ansatz ausgewählt. In Kapitel 5 wird der gewählte Lösungsansatz ausführlich erläutert. Kapitel 6 beschreibt verschiedene Werkzeuge zur Unterstützung des Lösungsansatzes. Anhand der Anforderungen aus Kapitel 3 und zusätzlicher Anforderungen, eher die technische Seite des Werkzeugs betreffend, wird ein geeignetes Werkzeug ausgewählt.

Kapitel 7 beschreibt die Evaluation des gewählten Ansatzes und Werkzeugs anhand zweier Fallstudien aus den Software Engineering Bereichen "Meßplanung mit GQM" und "Objektorientierter Entwurf mit UML" und identifiziert erforderliche Erweiterungen des gewählten Ansatzes und Werkzeugs. Kapitel 8 enthält abschließend eine Zusammenfassung der Arbeit und einen Ausblick auf zukünftige Tätigkeiten in diesem Bereich.

Kapitel 2

Problem

In komplexen Software-Entwicklungsprojekten werden große Mengen von Informationen verwaltet und verarbeitet. Dies bedeutet, vorliegende Informationen werden repräsentiert, und Änderungen an diesen Informationsrepräsentationen durchgeführt. Information ist Wissen über die reale Welt, d.h. Tatsachen und Zusammenhänge zwischen Tatsachen. Nachfolgend führen wir zwei Beispiele ein, die an dieser Stelle zur Erläuterung des Problems dienen und sich durch die gesamte Arbeit fortsetzen werden. Die Beispiele sind klein gehalten und daher unvollständig. Sie stellen nur die Aspekte dar, die wichtig im Hinblick auf das Problem sind.

Beispiel Entwicklungsaufwand

Das erste Beispiel stammt aus dem Bereich "Meßplanung mit GQM". Dieses Beispiel basiert auf dem "Goal-Question-Metric"-Ansatz ([Rom91], [Bas95]). Die gegebene Situation in der realen Welt sei ein Software-Entwicklungsprojekt. Es soll der Aufwand für dieses Entwicklungsprojekt gemessen werden (Entwicklungsaufwand). Dazu wird ein geeigneter Meßplan (GQM-Plan) benötigt.

Beispiel Lichtregelungssystem

Das zweite Beispiel stammt aus dem Bereich "Objektorientierter Entwurf eines Softwaresystems mit UML" (Unified Modeling Language) [Rat97]. Die Informationen der realen Welt sind die Anforderungen, die an das System gestellt werden. Es sei ein Raum mit einer Lichtquelle gegeben. Wenn eine Person den Raum betritt (d.h. der Raum ist belegt), und es draußen dunkel ist, soll das System das Licht anschalten (Anf1). Wenn die letzte Person den Raum verläßt (d.h. der Raum ist leer), soll das System das Licht ausschalten (Anf2). Wird es hell draußen, soll das System das Licht ausschalten (Anf3). Ist der Raum belegt, und es wird dunkel draußen, soll das System das Licht anschalten (Anf4).

Die Informationen, die in der realen Welt vorliegen, werden zunächst in einer geeigneten Form dargestellt und abgelegt. Dieser Prozeß heißt "Informationsrepräsentation" [Ric92]. Außerdem sind die abgelegten Informationen "Änderungen" unterworfen. Dafür werden Informationen teilweise gelöscht oder neue Information eingefügt. Nachfolgend werden wir die beiden Punkte "Informationsrepräsentation" und "Informationsänderung" detaillierter betrachten.

2.1 Repräsentation von Information

Im folgenden wird der Begriff "Informationsrepräsentation" definiert und anschließend das Problem der Konsistenz von Informationsrepräsentationen erläutert.

2.1.1 Definition

Zur Darstellung und Verwaltung von Informationen (I) sind geeignete Informationsrepräsentationen (I') notwendig (siehe Bild 1). Mittels Informationsrepräsentationen kann Wissen mitgeteilt werden. Eine Repräsentationssprache (S), z.B. Text, Diagramme, usw. erlaubt die Beschreibung von Informationsrepräsentationen. Es gibt verschiedene Repräsentationssprachen, deren Formalitätsgrad von informell bis formal variiert.

Die Ebene der Informationen wird als kognitive Ebene bezeichnet, während die Ebene der Informationsrepräsentationen die Repräsentationsebene darstellt [Ric92]. Informationen der kognitiven Ebene bezeichnen wir im folgenden mit Großbuchstaben (I, A, B, usw.), deren Informationsrepräsentationen auf der Repräsentationsebene mit apostrophierten Großbuchstaben (I', A', B', usw.).

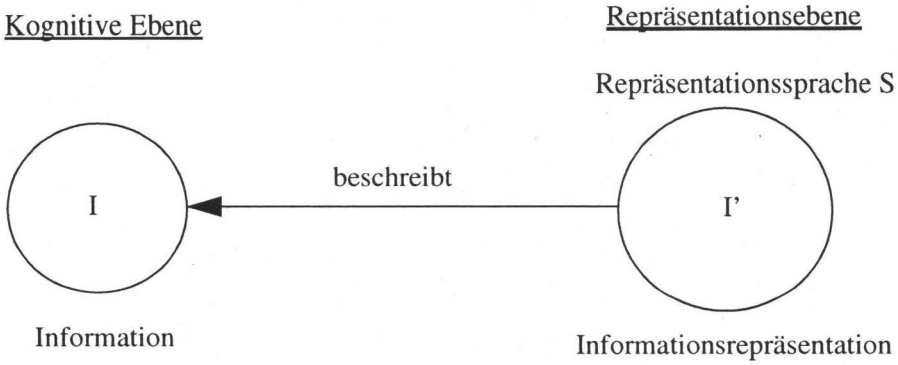

Bild 1: Informationsrepräsentation

Beispiel Entwicklungsaufwand

Der Entwicklungsprozeß eines betrachteten Softwaresystems (in Bild 1 Information I) kann durch ein Prozeßmodell "Entwicklungsprozeß" (in Bild 1 Informationsrepräsentation I') beschrieben werden (siehe Bild 3 auf Seite 7, Sicht "Projektpläne"). Dies geschieht in diesem Beispiel in Form einer Aggregationsbeziehung, die angibt, aus welchen Subprozessen sich *Entwicklungsprozeß* zusammensetzt. Diese Subprozesse sind *Anforderungsanalyse*, *System-Entwurf* und *Kodieren*. Die Repräsentationssprache S, die wir hierbei verwenden, ist eine Prozeßmodellierungssprache.

Beispiel Lichtregelungssystem

Die Anforderung Anf2 an das Lichtregelungssystem (in Bild 1 Information I) wird durch ein Sequenzdiagramm "Licht aus" (in Bild 1 Informationsrepräsentation I') beschrieben (siehe Bild 4 auf Seite 9, Sicht "Sequenzdiagramme").

Die Instanz *Anwesenheitssensor* wird alle 5 Sekunden abgefragt. Liefert der Sensor *leer* zurück, so meldet die Instanz *Anwesenheit* *anwesenheit() = leer* an die Instanz *Raum*. Falls es draußen dunkel ist, ruft *Raum* die Methode *ausschalten()* der Instanz *Licht* auf. Falls das Licht an ist (*zustand = an*), wird bei der Instanz *Lichtaktor* die Methode *aus()* aktiviert. Die Repräsentationssprache S, die wir hier verwenden, ist die Notation von UML für Sequenzdiagramme.

2.1.2 Konsistenz einer Informationsrepräsentation

Eine Repräsentation der Informationen der realen Welt ist konsistent, wenn keine Fehler in der Repräsentation bestehen. Fehler können verschiedener Art sein.

Zunächst kann die syntaktische Konsistenz verletzt werden, d.h. die Syntaxregeln der Repräsentationssprache werden nicht eingehalten. Weiterhin kann die semantische Konsistenz verletzt werden, d.h. es treten Widersprüche innerhalb der Repräsentation auf. Diese können durch fehlerhafte Beschreibung einzelner Aspekte der Realität entstehen. Die Semantik der Repräsentationssprache kann mithilfe semantischer Beziehungen formuliert werden (mit Hilfe einer informellen, semi-formalen oder formalen Sprache).

Beispiel Entwicklungsaufwand

Die Repräsentation des Entwicklungsprozesses (siehe Bild 3 auf Seite 7, Sicht "Projektpläne") ist konsistent, sie enthält keine Fehler.

Eine syntaktische Konsistenzverletzung würde auftreten, falls z.B. die Aggregationsbeziehung nicht nur Prozesse, sondern auch ein Produkt enthalten würde. Dies widerspricht den Syntaxregeln der verwendeten Prozeßmodellierungssprache.

Eine semantische Konsistenzverletzung würde auftreten, falls z.B. *Anforderungsanalyse* wieder verfeinert würde zu *Entwicklungsprozeß*, da dies einen Zyklus ergeben würde, der nicht sinnvoll wäre.

Beispiel Lichtregelungssystem

Das Sequenzdiagramm (siehe Bild 4 auf Seite 9, Sicht “Sequenzdiagramme”) ist konsistent.

Eine syntaktische Konsistenzverletzung würde auftreten, falls z.B. der Pfeil, der mit der Methode *aus()* beschriftet ist, von der Instanz *Licht* ins Leere zeigen würde und nicht auf die Instanz *Lichtaktor*, weil jeder Pfeil auf eine Instanz zeigen muß. Dies widerspricht den Syntaxregeln von UML.

Eine semantische Konsistenzverletzung würde auftreten, falls z.B. die Instanz *Anwesenheit* an die Instanz *Raum* anwesenheit(belegt) liefern würde, obwohl die Instanz *Anwesenheitssensor* leer zurückliefert.

2.2 Sichten

Liegt eine komplexe Informationsmenge vor, so ist es schwierig, die gesamte Informationsrepräsentation auf einmal zu erstellen oder zu verstehen. Eine Möglichkeit der Bewältigung der Komplexität ist die Unterteilung der Informationsmenge in logisch zusammenhängende Einheiten. Um die Informati-

onsmenge zu verstehen, können dann Ausschnitte aus der Gesamtinformation betrachtet werden. Im folgenden definieren wir das Konzept der Sichten, erläutern Überschneidungen von Sichten und das sich daraus ergebende Problem der Konsistenz zwischen Sichten.

2.2.1 Definition

Sei eine Gesamtinformationsmenge G gegeben, und seien A, B, usw. Ausschnitte aus dieser Gesamtinformationsmenge G (siehe Bild 2). Sichten auf G sind Informationsrepräsentationen A', B', usw., die die Ausschnitte A, B, usw. aus der Gesamtinformationsmenge G repräsentieren. Sichten sind also Repräsentationen eines Teils der zugrunde liegenden Gesamtinformation und können in verschiedenen Repräsentationssprachen (S_1, S_2, usw.) formuliert sein.

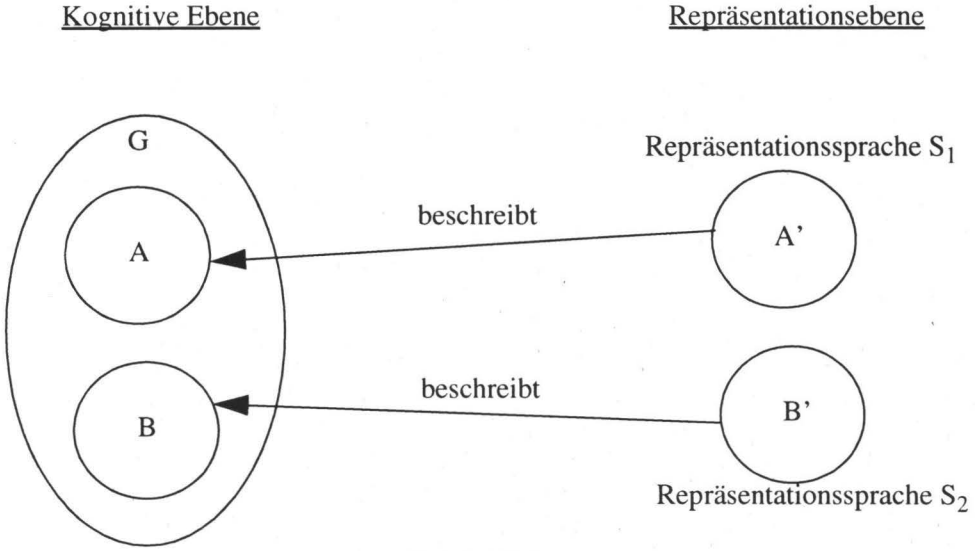

Bild 2: Sichten

Beispiel Entwicklungsaufwand

Die unter 2.1.1 eingeführte Beschreibung des Entwicklungsprozesses (in Bild 2 Sicht A') repräsentiert nicht die gesamte in der realen Welt vorliegende Information (in Bild 2 G), sondern nur einen Ausschnitt daraus. Das Prozeßmodell "Entwicklungsprozeß" gehört zur Sicht "Projektpläne" auf die vorliegenden Informationen (siehe Bild 3 auf Seite 7). In diesem Beispiel existieren noch zwei weitere Sichten, die Sichten "Qualitätsmodelle" (in Bild 2 Sicht B') und "GQM-Pläne" (im Bild wäre Sicht C' hinzuzufügen). Die Information ist in diese Ausschnitte unterteilt, da jede Sicht Informationen für bestimmte Rollen zur Verfügung stellt. Der GQM-Plan unterstützt z.B. die Qualitätssicherer, das Prozeßmodell z.B. den Projekt- und das Qualitätsmodell den Qualitätsmanager.

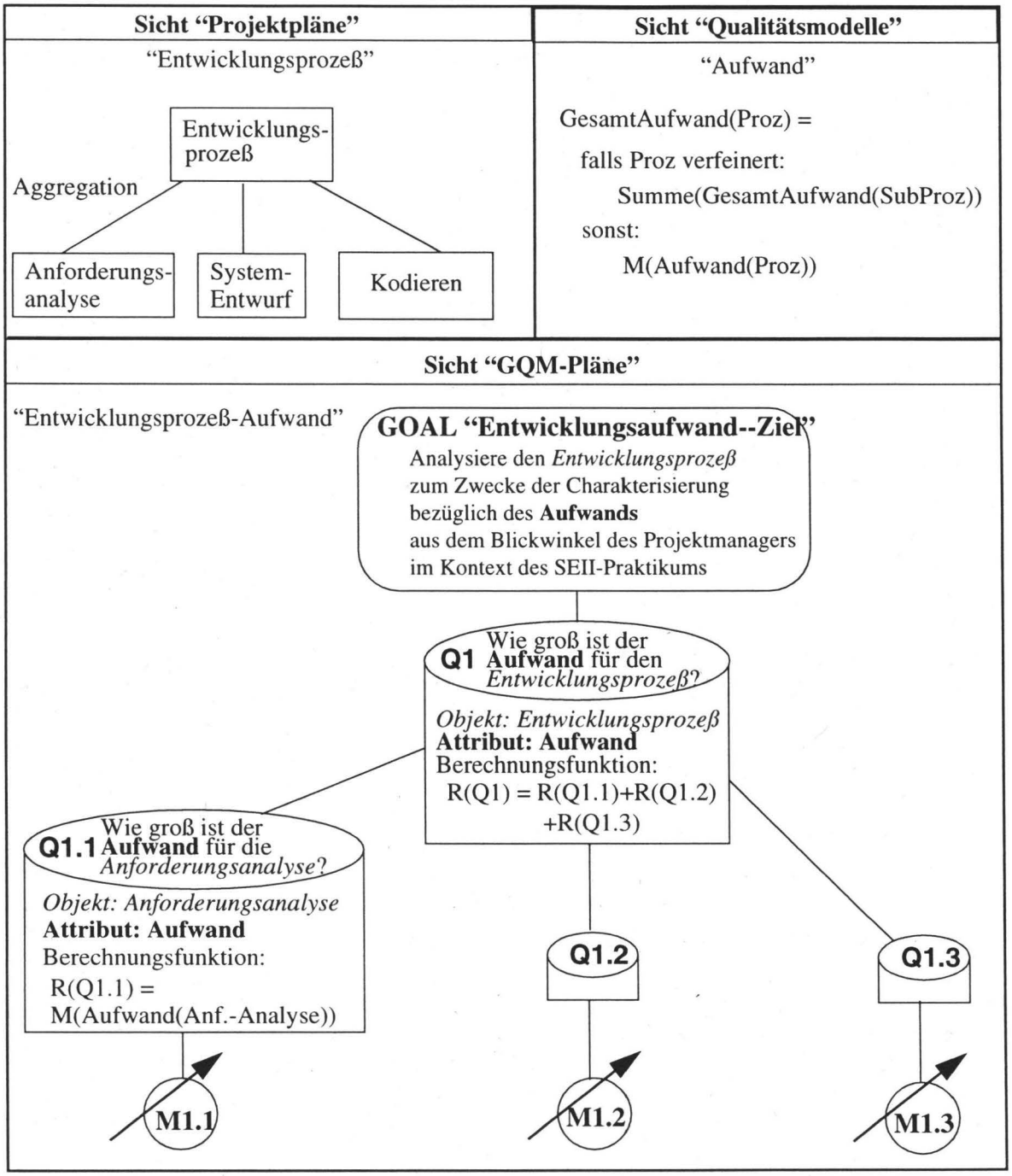

Bild 3: Sichten - Beispiel Entwicklungsaufwand

In der Sicht "Qualitätsmodelle" existiert das Qualitätsmodell "Aufwand" (in Bild 2 B'). Es beschreibt, wie der Gesamtaufwand eines Prozesses (Proz) ermittelt wird (siehe Bild 3, Sicht "Qualitätsmodelle"). Falls der Prozeß in Unterprozesse verfeinert wird, wird rekursiv der Gesamtaufwand dieser Unterprozesse ermittelt und aufsummiert. Falls der Prozeß nicht weiter verfeinert wird, ist der Gesamtaufwand dieses Prozesses gleich dem gemessenen Aufwand für diesen Prozeß. Die benutzte Repräsentationssprache lehnt sich an Programmiersprachen an.

Aus dem oben beschriebenen Prozeß- und dem Qualitätsmodell läßt sich ein GQM-Plan "Entwicklungsprozeß-Aufwand" (in Bild2 wäre C' hinzuzufügen) ableiten. Das *Objekt* der Frage *Q1* des GQM-Plans bezieht sich auf das Prozeßmodell "Entwicklungsprozeß", das *Attribut* bezieht sich auf das Qualitätsmodell "Aufwand" (siehe Bild 3 auf Seite 7, Sicht "GQM-Pläne"). Daraus ergibt sich also der Text der Frage *Q1* "Wie groß ist der Aufwand für den Entwicklungsprozeß?". Die Berechnungsfunktion für das Attribut "Aufwand" wird aus dem Qualitätsmodell abgeleitet. Außerdem wird *Q1* gemäß Prozeßmodell in Unterfragen (*Q1.1* bis *Q1.3*) verfeinert. Die Repräsentationssprache, die hier für GQM-Pläne verwendet wird, besteht aus Schablonen für die einzelnen GQM-Planelemente (Ziele, Fragen, Maße). Der komplette GQM-Plan "Entwicklungsprozeß-Aufwand" befindet sich in Anhang B.

Beispiel Lichtregelungssystem

Die Sichten, die beim objektorientierten Entwurf mit UML außer der Sicht Sequenzdiagramme noch existieren, sind u.a. Klassendiagramme zur Beschreibung der statischen Aspekte und Zustandsdiagramme zur Beschreibung der dynamischen Aspekte der Instanzen einer Klasse. Es gibt verschiedene Möglichkeiten, das System zu strukturieren. Die hier gegebene Struktur führt zu einer Reduktion der Kommunikation im System.

In der statischen Sicht "Klassendiagramme" existiert ein Klassendiagramm "Lichtregelungssystem" (siehe Bild 4 auf Seite 9). Dieses umfaßt eine Klasse *Raum*, die die Attribute *zustand_anwesenheit* und *zustand_helligkeit* und die Methoden *anwesenheit(zustand)* und *helligkeit(zustand)* besitzt. Außerdem existieren die Klassen *Sensor*, *Helligkeit* und *Anwesenheit*, die zur Überprüfung der Lichtverhältnisse draußen und der Anwesenheit von Personen im Raum benötigt werden.

Die Klasse *Raum* ist über eine Aggregation mit der Klasse *Licht* verbunden, d.h. eine Instanz von *Raum* enthält eine Instanz von *Licht*. Weiterhin existiert noch die Klasse *Lichtaktor*, die mit ihren Methoden *an()* und *aus()* für die Lichtschaltung zuständig ist. Die benutzte Repräsentationssprache ist die Notation von UML für Klassendiagramme.

Das dynamische Verhalten der Instanzen einer Klasse wird durch die Sicht "Zustandsdiagramme" beschrieben (siehe Bild 4 auf Seite 9). Die Sicht umfaßt die Zustandsdiagramme "Raum" und "Licht".

Eine Instanz der Klasse *Raum* kann vier verschiedene Zustände annehmen, *leer/hell*, *leer/dunkel*, *belegt/hell* und *belegt/dunkel* (jeweils Anwesenheit/Lichtverhältnisse draußen). Der Zustand wird je nach Sensormeldungen gewechselt, z.B. geht die Instanz *Raum* von *belegt/dunkel* nach *leer/dunkel* über, falls die Methode *anwesenheit(leer)* aufgerufen wird. Außerdem ruft die Instanz *Raum* dann noch die Methode *ausschalten()* von *Licht* auf.

Eine Instanz der Klasse *Licht* kann zwei verschiedene Zustände annehmen, *an* und *aus*. Der Übergang zwischen diesen Zuständen geschieht jeweils durch die Methoden *ausschalten()* und *anschalten()*. Die benutzte Repräsentationssprache ist die Notation von UML für Zustandsdiagramme.

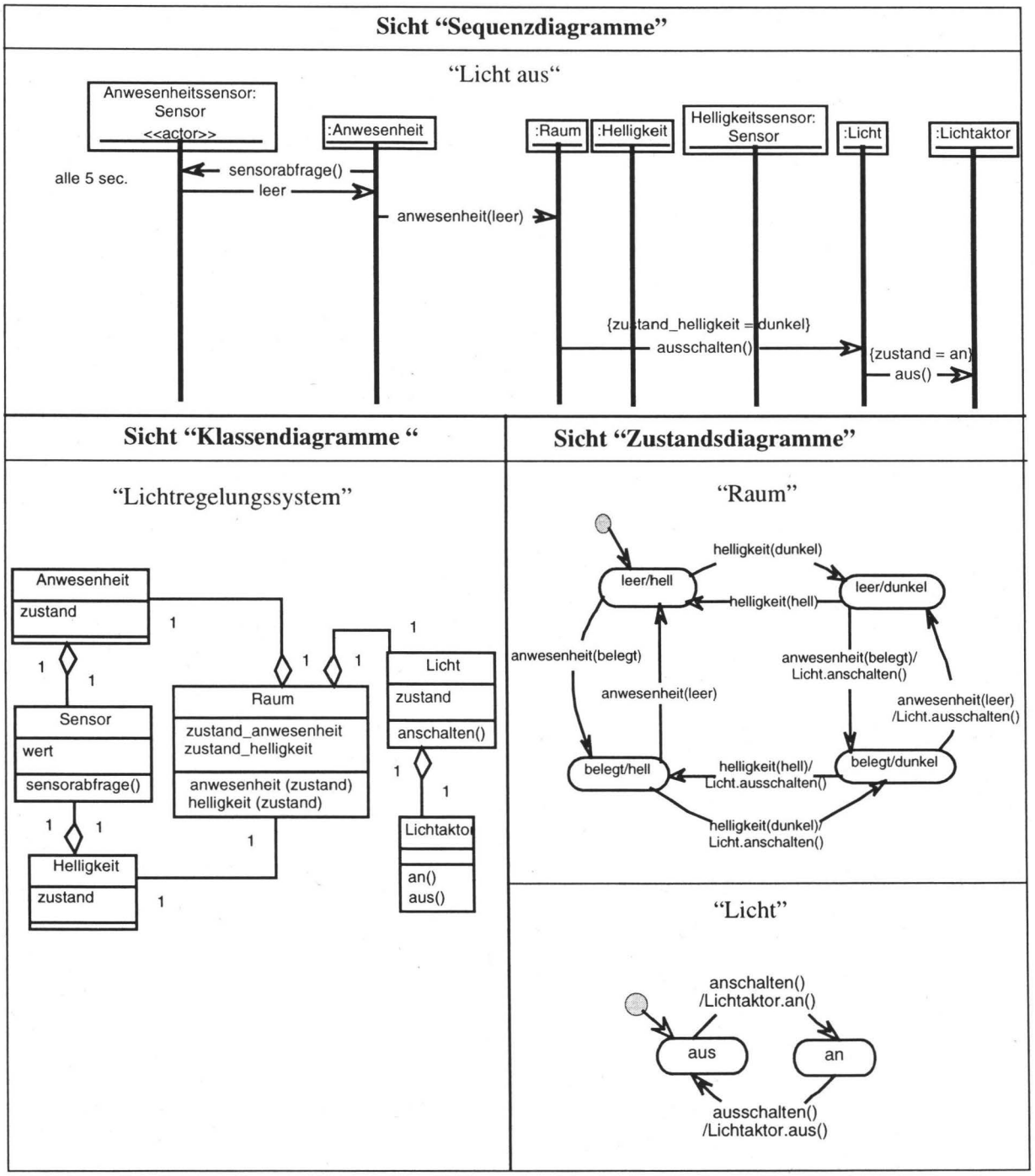

Bild 4: Sichten - Beispiel Lichtregelungssystem

2.2.2 Überschneidung von Sichten

Sichten können sich überschneiden, d.h. sie beschreiben dieselbe zugrunde liegende Information. Seien eine Informationsrepräsentation A' in Repräsentationssprache S_1 und eine Informationsrepräsentation B' in Repräsentationsprache S_2 gegeben. A' beschreibt eine Informationsmenge A, und B' beschreibt eine Informationsmenge B (siehe Bild 5 auf Seite 10). Die Informationsmengen A und B überschneiden sich. Die Schnittmenge zwischen A und B bezeichnen wir mit C.

Die Gesamtinformationsmenge ist die Vereinigung von A und B. Diese bezeichnen wir mit G. Somit sind A' und B' Sichten auf G. C_A' ist der Teil von A', der C beschreibt, und C_B' ist der Teil von B', der C beschreibt. A' und B' überschneiden sich, da sie sich überschneidende Ausschnitte A und B aus der Gesamtinformation G repräsentieren.

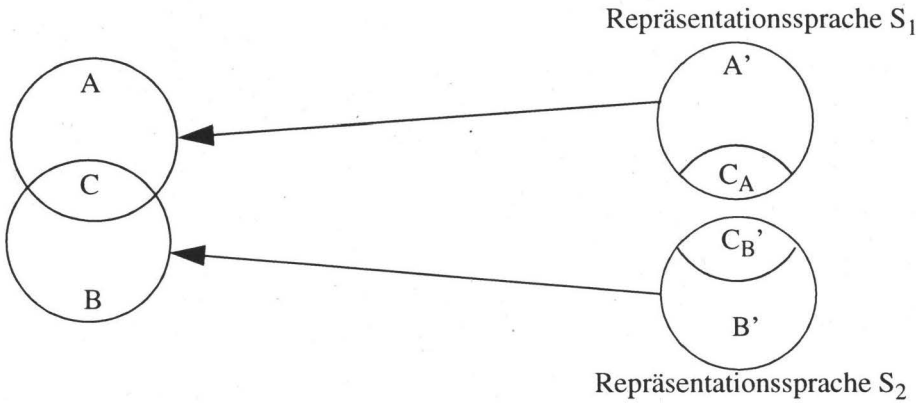

Bild 5: Sichten-Überschneidung

Sichten-Überschneidungen erläutern wir im folgenden an den Beispielen. Dabei betrachten wir zur Vereinfachung nur ausschnittsweise einige Überschneidungen jeweils zweier Sichten, obwohl sich auch mehrere Sichten gemeinsam überschneiden können.

Beispiel Entwicklungsaufwand

Sicht "Projektpläne" - Sicht "GQM-Pläne": Das *Objekt* der Frage *Q1* des GQM-Plans bezieht sich auf das Prozeßmodell "Entwicklungsprozeß" (siehe Bild 3 auf Seite 7). Der GQM-Plan wird gemäß dieses Prozeßmodells verfeinert.

Sicht "Qualitätsmodelle" - Sicht "GQM-Pläne": Das *Attribut* der Frage *Q1* des GQM-Plans bezieht sich auf das Qualitätsmodell "Aufwand", woraus die Berechnungsfunktion abgeleitet wird.

Beispiel Lichtregelungssystem

Sicht "Sequenzdiagramme" - Sicht "Klassendiagramme": In der Sicht "Sequenzdiagramme" werden Instanzen der Klassen aus der Sicht "Klassendiagramme" betrachtet, z.B. ist *Anwesenheitssensor* eine Instanz der Klasse *Sensor* (siehe Bild 4 auf Seite 9). Die Methoden in der Sicht "Sequenzdiagramme" sind dieselben wie in der Sicht "Klassendiagramme".

Sicht "Klassendiagramme" - Sicht "Zustandsdiagramme": Übergänge in der Sicht "Zustandsdiagramme" haben als Beschriftung Bedingungen, die Methoden-Aufrufe enthalten können. Dieselben Methoden sind in der Sicht "Klassendiagramme" definiert, z.B. *ausschalten()* bei der Klasse *Licht* (siehe Bild 4 auf Seite 9). Klassen aus der Sicht "Klassendiagramme" haben zugehörige Zustandsdiagramme in der Sicht "Zustandsdiagramme", falls sie ein dynamisches Verhalten aufweisen, hier für die Klassen *Raum* und *Licht* dargestellt.

2.2.3 Konsistenz zwischen Sichten

Durch die Überschneidung von Sichten entsteht das Problem der Konsistenz zwischen Sichten. Die beiden Sichten A' und B' heißen konsistent, wenn C_A' und C_B' dieselbe Information C der realen Welt beschreiben.

Es können semantische Beziehungen zwischen den Sichten formuliert werden, die aus der realen Welt abgeleitet werden. Um die semantische Beziehungen zwischen den Sichten zu definieren, müssen die Überschneidungen der Sichten identifiziert werden. Zwischen Sichten treten nur semantische Konsistenzverletzungen auf, da keine Syntaxregeln zwischen den verschiedenen Repräsentationssprachen der einzelnen Sichten existieren.

Beispiel Entwicklungsaufwand

Die Überschneidungen der Sicht "GQM-Pläne" mit den Sichten "Projektpläne" und "Qualitätsmodelle" sind konsistent, sie beschreiben dieselben Informationen. Die Verfeinerung der Fragen stimmt mit der Sicht "Projektpläne", die Berechnungsfunktion mit der Sicht "Qualitätsmodelle" überein.

Der GQM-Plan wäre (im Bezug zur Überschneidung mit der Sicht "Projektpläne") semantisch inkonsistent, falls z.B. Frage *Q1.1* einen im Prozeßmodell nicht existierenden Unterprozeß *Testen* als *Objekt* hätte, also "Wie groß ist der Aufwand für Testen?" lauten würde.

Beispiel Lichtregelungssystem

Konsistenz besteht zwischen allen drei Sichten, z.B. haben alle Instanzen in der Sicht "Sequenzdiagramme" zugehörige Klassen in der Sicht "Klassendiagramme", alle Methoden in den Sichten "Sequenzdiagramme" und "Zustandsdiagramme" sind auch in der Sicht "Klassendiagramme" vertreten.

Das Sequenzdiagramm wäre (im Bezug zur Überschneidung mit der Sicht "Klassendiagramme") semantisch inkonsistent, falls z.B. an einem Pfeil der Name einer Klasse stehen würde anstelle des Namens einer Methode aus dem Klassendiagramm.

Die Konsistenz zwischen Sichten heißt Inter-Sicht-Konsistenz. Die Konsistenz innerhalb einer Sicht (siehe 2.1.2 auf Seite 5, Konsistenz einer Informationsrepräsentation) heißt Intra-Sicht-Konsistenz.

Es gibt zwei Möglichkeiten der Datenhaltung bei Existenz mehrerer Sichten. Die erste Möglichkeit besteht darin, aus einer zugrunde liegenden gemeinsamen Datenmenge Sichten abzuleiten. Falls Änderungen gemacht werden, werden sie auf dieser gemeinsamen Datenmenge gemacht. In diesem Fall entstehen keine Konsistenzprobleme zwischen verschiedenen Sichten. Die zweite Möglichkeit der Datenhaltung ist, daß die Daten der verschiedenen Sichten getrennt ermittelt und dann integriert werden. Änderungen werden auf den getrennten Datenmengen der einzelnen Sichten gemacht. Dies ist insbesondere dann sinnvoll, wenn verschiedene Rollen die Daten erfassen. Nachteil dieser Möglichkeit ist, daß Konsistenzprobleme zwischen Sichten auftreten können. Da die untersuchten Beispiele Datenhaltung entsprechend der zweiten Möglichkeit vornehmen, werden wir im weiteren auch nur die zweite Möglichkeit betrachten.

2.3 Informationsänderungen

Im Laufe eines Software-Projekts werden die verschiedenen Software-Artefakte geändert, d.h. Änderungen jeder Art werden auf den Informationsrepräsentationen durchgeführt. Dadurch kann sowohl die Konsistenz innerhalb der Sichten als auch die Konsistenz zwischen den Sichten verletzt werden. Zunächst erläutern wir die grundsätzliche Vorgehensweise beim Ändern einer Informationsrepräsentation. Anschließend betrachten wir verschiedene Arten von Konsistenzen, die im Laufe der Änderung überprüft werden müssen.

2.3.1 Grundsätzliche Vorgehensweise bei einer Änderung

In [BA96] wird die grundsätzliche Vorgehensweise bei einer Änderung beschrieben. Der angegebene Änderungsprozeß bezieht sich auf die Änderung verschiedener Software-Artefakte. Da wir nur die Änderung innerhalb eines Software-Artefakts betrachten, haben wir die beschriebene Vorgehensweise angepaßt.

Ausgehend von einem Vorschlag zur Änderung eines Elements in einem Artefakt, wird eine Auswirkungsanalyse durchgeführt, d.h. die möglichen Konsequenzen der Änderung werden identifiziert. Es wird abgeschätzt, welche anderen Elemente modifiziert werden müssen, um die Änderung zu vollenden.

Ausgangsdokumente für die Durchführung der Änderung sind: die vorgeschlagene Änderung und die aktuell vorliegenden Sichten eines Artefakts (siehe Bild 6).

Beispiel: Lichtregelungssystem

Eine vorgeschlagene Änderung könnte das Einfügen einer neuen Funktionalität in den objektorientierten Entwurf sein. Beispielsweise könnte gefordert werden, das Licht erst nach einer bestimmten Zeit t1 auszuschalten.

Die vorliegenden Sichten im objektorientierten Entwurf könnten sein: Klassendiagramm, Sequenzdiagramme und Zustandsdiagramme.

Aus dem Änderungsvorschlag und den Sichten des Artefakts wird initial eine Startmenge von offensichtlichen Auswirkungen erstellt (direkte Auswirkungen, Auswirkungen erster Ordnung). In einer "Change-Map" wird die Änderung mit den offensichtlichen Auswirkungen in Beziehung gesetzt.

Beispiel: Lichtregelungssystem

Die gewünschte Änderung erfordert eine Änderung des Sequenzdiagramms "Licht aus". Wenn vom Licht die Methode ausschalten() aufgerufen wird, muß das Licht Methoden eines neuen Objekts Zeit aufrufen, um die Zeitspanne t1 zu warten. Erst danach wird das Licht ausgeschaltet.

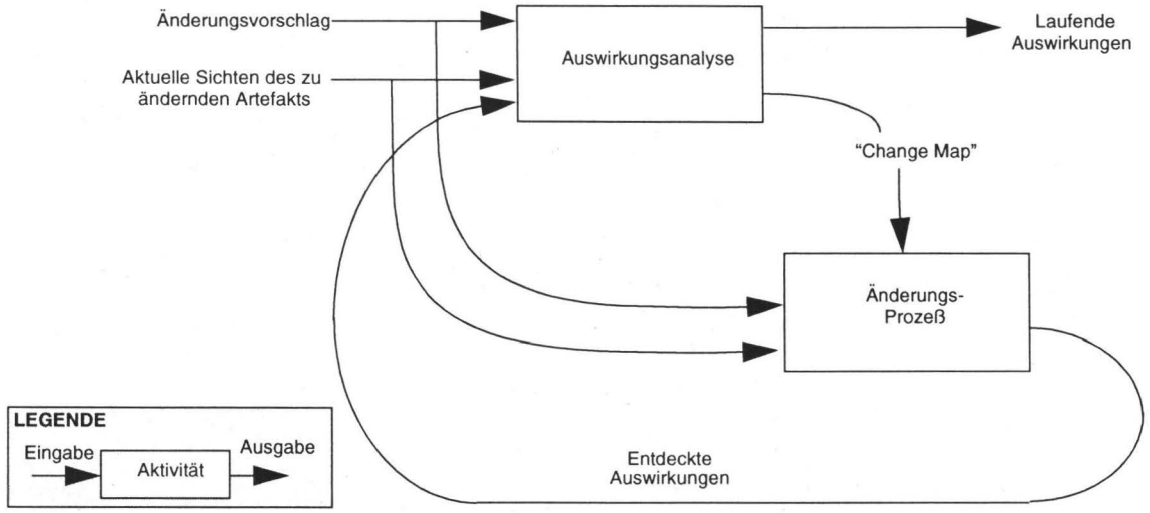

Bild 6: Auswirkungsanalyse bei einer Änderung, angelehnt an [BA96]

Danach erfolgt eine Abschätzung der möglichen "Ripple Effects" der Änderung (indirekte Auswirkungen, Auswirkungen n-ter Ordnung). Dies sind mögliche Auswirkungen von Änderungen der Elemente der Startmenge.

Beispiel: Lichtregelungssystem

Das Einführen von weiteren Objekten und Methodenaufrufen in das Sequenzdiagramm macht eine Überarbeitung anderer Sichten erforderlich. In das Klassendiagramm müssen ebenfalls neue Klassen und Methoden aufgenommen werden. Außerdem muß das Zustandsdiagramm der Klasse Licht erweitert werden.

Die Menge der geschätzten "Ripple Effects" wird zu der Startmenge hinzugefügt. Die beiden Mengen zusammen bilden die Menge der "laufenden Auswirkungen" (siehe Bild 6), d.h. der Auswirkungen, die gerade bearbeitet werden. Diese Menge dient dem Management zum Abschätzen der Ressourcen und der Zeit, die benötigt wird, um die Änderung mit ihren Auswirkungen durchzuführen.

Die möglichen "Ripple Effects" werden auch zur Erweiterung der "Change-Map" herangezogen, welche an den Änderungsprozeß weitergereicht wird. Die "Change-Map" dient innerhalb des Änderungsprozesses als Checkliste der Elemente, die gegebenenfalls geändert werden müssen, und als eine Art Navigationskarte zum Lokalisieren dieser Elemente.

Während des Änderungsprozesses werden die wirklich auftretenden "Ripple Effects" entdeckt. Diese werden als "entdeckte Auswirkungen" (siehe Bild 6 auf Seite 13) wieder an die Auswirkungsanalyse weitergereicht. Die Elemente, die während des Änderungsprozesses wirklich geändert werden, werden in der Menge der aktuellen Auswirkungen gesammelt.

Traditionell geschieht das Auffinden der Abhängigkeiten, die zur Aufdeckung der Auswirkungen führen, während des Änderungsprozesses im Kopf von Software-Experten mithilfe einer iterativen Folge von Untersuchungen und Auswirkungsentdeckungen.

Beispiel: Lichtregelungssystem

Eine Abhängigkeit ist z.B.: Methoden, die eine Klasse im Sequenzdiagramm geschickt bekommt, müssen ebenfalls im Klassendiagramm als Methoden der Klasse auftreten.

Abhängigkeiten können meist nur manuell aufgedeckt werden. Dies liegt daran, daß diese Abhängigkeiten meist nur implizit sind. Das Auffinden der Abhängigkeiten kann z.B. durch Inspektionen oder "Walkthroughs" geschehen. Dies genügt für kleine Programme, ist jedoch nicht geeignet für Änderungen an großen Softwaresystemen, da der Arbeitsaufwand bei dieser Vorgehensweise zu groß wird.

Die Kernpunkte der Auswirkungsanalyse sind die Repräsentation der Abhängigkeiten und Techniken zum Auffinden der Auswirkungen. Detaillierte Untersuchungen der Abhängigkeiten und Techniken zum Auffinden der Auswirkungen tragen dazu bei, daß implizite Abhängigkeiten expliziter gemacht werden. Existieren die Abhängigkeiten explizit und formal, so kann die Auswirkungsanalyse automatisch (durch ein Werkzeug) stattfinden, was voraussichtlich eine enorme Zeit- und Arbeitsersparnis bedeutet [BA96] und auch die Anzahl der gemachten Fehler reduziert. Die automatische Unterstützung ist also essentiell für die Effektivität und die Effizienz der Auswirkungsanalyse.

Die Abhängigkeiten innerhalb und zwischen Software-Artefakten sind semantische Beziehungen. Die Auswirkungsanalyse besteht in einer Überprüfung dieser semantischen Beziehungen, um inkonsistente Stellen in den Software-Artefakten aufzufinden.

2.3.2 Konsistenzprüfungen bei der Durchführung von Änderungen

Liegen mehrere Sichten in unterschiedlichen Repräsentationssprachen vor, so unterteilt sich die Überprüfung der Konsistenz in zwei Komponenten, die Überprüfung der Intra-Sicht-Konsistenz und die Überprüfung der Inter-Sicht-Konsistenz.

Wird mit der Repräsentation der Information begonnen, so existieren noch keine Informationen in den Sichten. Bei der Erstellung der ersten Sicht muß dann zunächst nur die Intra-Sicht-Konsistenz überprüft werden. Bei der Ersterstellung der anderen Sichten muß, wie im Falle einer Änderung, die Intra-Sicht-Konsistenz und die Konsistenz zwischen der neuen und den schon existierenden Sichten überprüft werden.

Seien zwei Sichten A' und B' in den Repräsentationssprachen S_1 und S_2 gegeben, die sich überschneiden (siehe Bild 5 auf Seite 10). Sei ein Element E' der Sicht A' zu E'_{update} geändert worden. Nun wird die Intra-Sicht-Konsistenz innerhalb der geänderten Sicht A'_{update} und die Inter-Sicht-Konsistenz zwischen A'_{update} und B' überprüft, um die Auswirkungen der Änderung festzustellen.

2.3.3 Intra-Sicht-Konsistenzprüfung

Die Intra-Sicht-Konsistenzprüfung überprüft die Beziehungen innerhalb einer Sicht. Innerhalb von A'_{update} werden die syntaktische und die semantische Konsistenz überprüft (siehe Bild 7 auf Seite 15).

2.3.3.1 Syntaktische Konsistenzprüfung

Bei der syntaktischen Konsistenzprüfung werden die Syntaxregeln der Repräsentationssprache S_1 überprüft. Ist die syntaktische Konsistenz verletzt, so wird E'_{update} solange geändert und auf syntaktische Konsistenz überprüft, bis E'_{update} den Syntaxregeln von S_1 genügt. Hier muß nur E'_{update} betrachtet werden und keine anderen Elemente aus A'_{update}. Dies genügt, da alle anderen Fehler, die auftreten können, semantischer Art sind, wie z.B. zwei gleiche Namen. Diese Fehler werden in der semantischen Konsistenzprüfung erkannt. Die semantische Konsistenzprüfung wird erst dann durchgeführt, wenn eine syntaktisch konsistente Sicht vorliegt.

2.3.3.2 Semantische Konsistenzprüfung

Die semantische Konsistenzprüfung stellt fest, ob die semantischen Beziehungen der Repräsentationssprache zwischen den Elementen von A'_{update} eingehalten werden. Hierbei werden die semantischen Beziehungen innerhalb von A'_{update} überprüft. Werden die semantische Beziehungen verletzt, so werden die inkonsistenten Stellen in A'_{update} aufgelistet (initiale Liste von Auswirkungen).

Werden die inkonsistenten Stellen betrachtet, so können mögliche "Ripple Effects" entdeckt werden, d.h. es würden weitere semantische Beziehungen verletzt, falls die Elemente der initialen Liste geändert würden. Die inkonsistenten Stellen werden zur initialen Liste von Auswirkungen hinzugefügt.

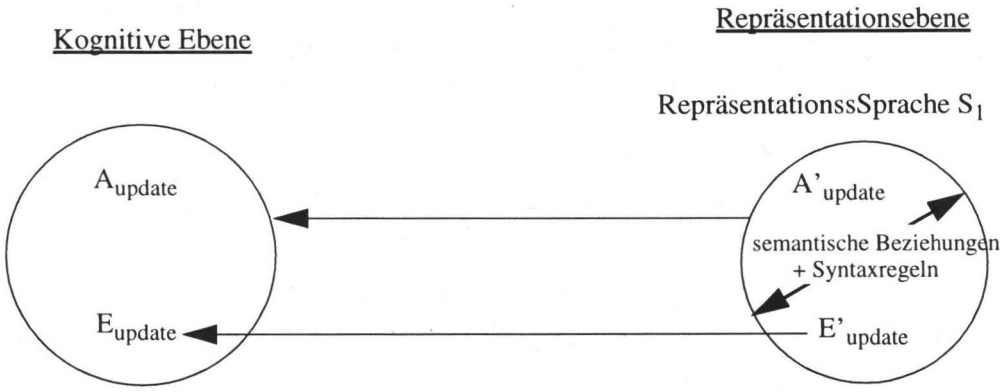

Bild 7: Intra-Sicht-Konsistenzprüfung bei Änderung der Sicht A'

2.3.4 Inter-Sicht-Konsistenzprüfung

Die Inter-Sicht-Konsistenzprüfung überprüft die Einhaltung der Beziehungen zwischen verschiedenen Sichten. Hier wird die Konsistenz zwischen A'_{update} und B', die in den Repräsentationssprachen S_1 und S_2 vorliegen, überprüft (siehe Bild 8 auf Seite 16).

2.3.4.1 Syntaktische Konsistenzprüfung

Zwischen Sichten, die in unterschiedlichen Repräsentationssprachen vorliegen, existieren keine Syntaxregeln. Deshalb ist keine syntaktische Inter-Sicht-Konsistenzprüfung notwendig.

2.3.4.2 Semantische Konsistenzprüfung

Die semantische Konsistenzprüfung stellt fest, ob die semantischen Beziehungen zwischen den Repräsentationssprachen von den Elementen von A'_{update} und B' eingehalten werden. Sind die semantische Beziehungen verletzt, so wird die initiale Liste von Auswirkungen (siehe Semantische Konsistenzprüfung der Intra-Sicht-Konsistenz) um die inkonsistenten Stellen erweitert (erweiterte Liste von Auswirkungen).

Befindet sich das geänderte Element E' in $A'_{update} \backslash C_A'$, so wird die semantische Konsistenz nicht verletzt, da B' dieses Element nicht beschreibt. Befindet sich E' jedoch in C_A', so wird C_A' zu $C_A'_{update}$ und beschreibt nun auch eine geänderte Information C_{update}. C_B' beschreibt jedoch weiterhin die alte Information C, d.h. die Inter-Sicht-Konsistenz zwischen A'_{update} und B' ist verletzt.

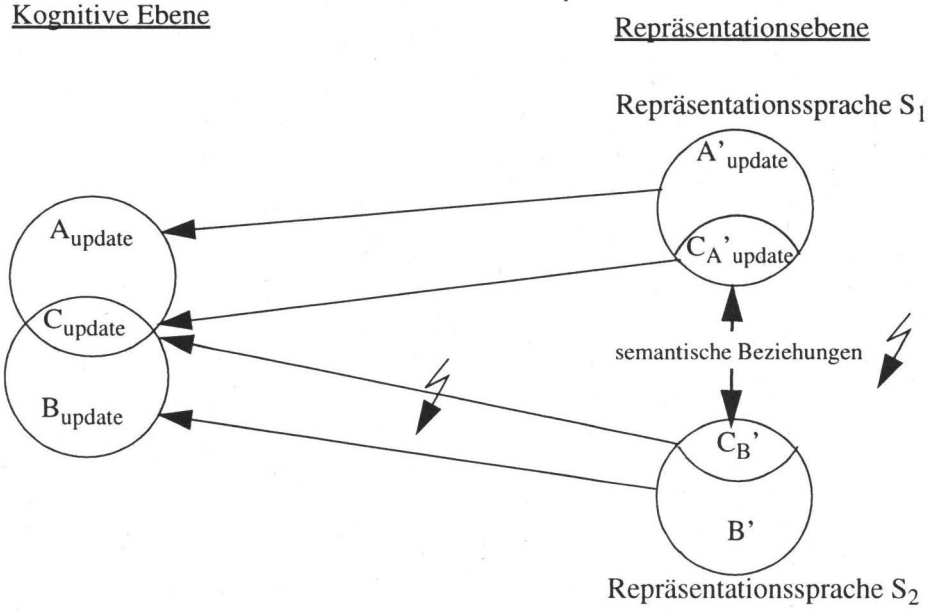

Bild 8: Inter-Sicht-Konsistenzprüfung bei Änderung der Sicht A'

Werden die inkonsistenten Stellen betrachtet, so können mögliche "Ripple Effects" identifiziert werden, d.h. es würden weitere semantische Beziehungen verletzt. Diese Konsistenzverletzungen können sowohl innerhalb der betrachteten, als auch zwischen der betrachteten und anderen Sichten auftreten. Die inkonsistenten Stellen werden zur erweiterten Liste von Auswirkungen hinzugefügt.

Beispiel Entwicklungsaufwand

Angenommen, das Prozeßmodell "Entwicklungsprozeß" wird geändert, indem noch ein vierter Subprozeß, nämlich *Benutzbares-System-erstellen* hinzugefügt wird (siehe Bild 9, Sicht "Projektpläne").

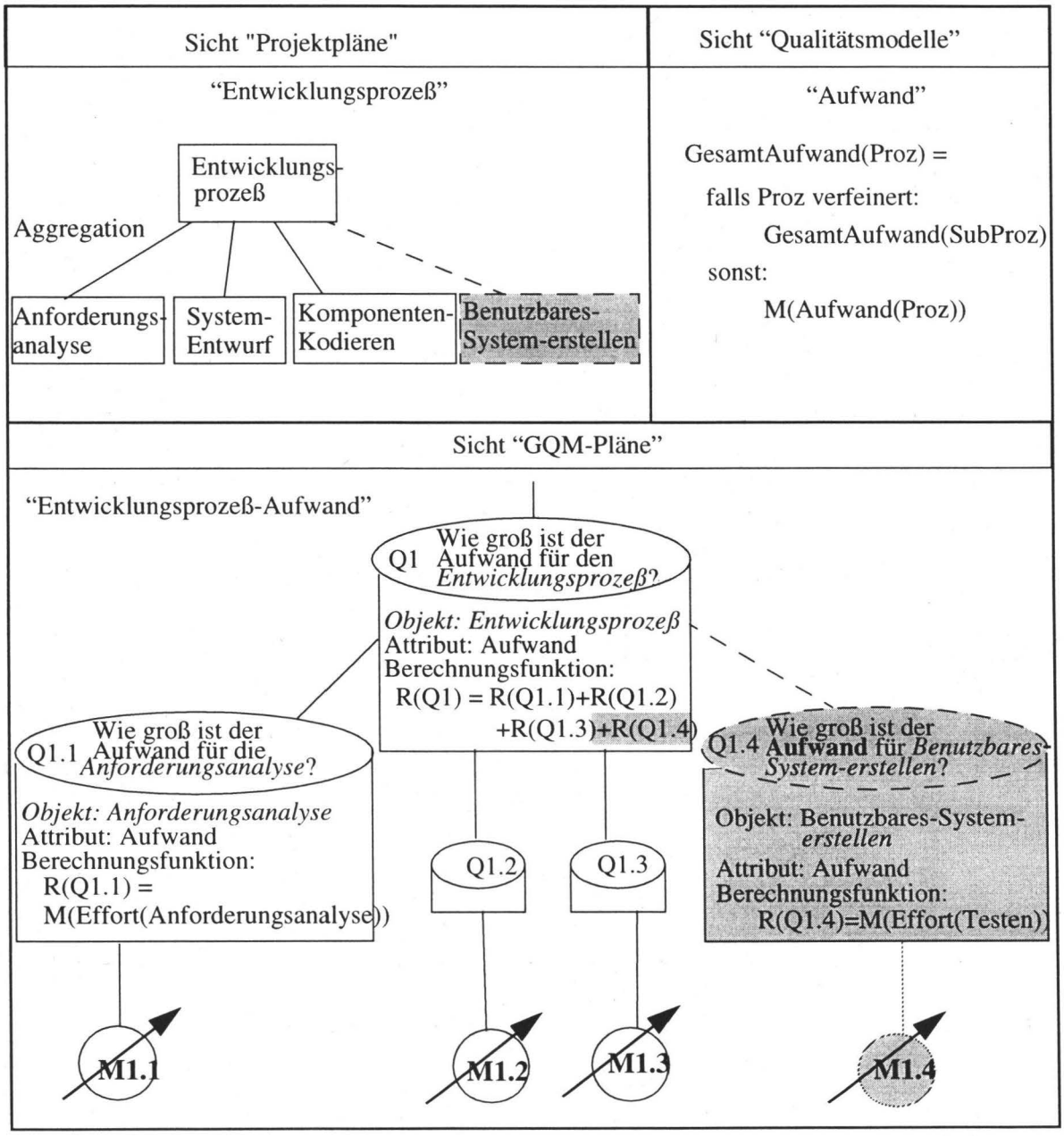

Bild 9: Geänderte, wieder konsistente Sichten - Beispiel Entwicklungsaufwand

Die syntaktische Intra-Sicht-Konsistenz von der Sicht "Projektpläne" bleibt erhalten, *Benutzbares-System-erstellen* ist syntaktisch korrekt dargestellt. Die semantische Intra-Sicht-Konsistenz wird auch nicht verletzt, da z.B. kein Zyklus entsteht.

Bei der Inter-Sicht-Konsistenzprüfung muß nur die Sicht "GQM-Pläne" (das GOAL wurde aus Platzgründen weggelassen, es bleibt unverändert) betrachtet werden, da keine Überschneidung von der Sicht "Projektpläne" mit der Sicht "Qualitätsmodelle" existiert.

Die Inter-Sicht-Konsistenz ist verletzt, da *Q1* nun nicht mehr gemäß Prozeßmodell verfeinert ist. Um die Konsistenz wiederherzustellen, würde in der Sicht "GQM-Pläne" zunächst eine vierte Unterfrage *Q1.4* mit *Objekt Benutzbares-System-erstellen* hinzukommen, das dazugehörige Maß *M1.4*, und außerdem würde die Berechnungsfunktion von *Q1* um *R(Q1.4)* ergänzt werden.

Da eine Überschneidung von der Sicht "GQM-Pläne" mit der Sicht "Qualitätsmodelle" existiert, muß auch hier die Inter-Sicht-Konsistenz geprüft werden. Diese wird durch die Änderungen nicht verletzt, d.h. es würden keine Änderungen in der Sicht "Qualitätsmodelle" ausgeführt werden.

Beispiel Lichtregelungssystem

Falls die Sensoren ständig wechselnde Werte liefern, also z.B. der Helligkeitssensor bei Morgengrauen ständig zwischen den Meldungen "hell" und "dunkel" schwankt, so wird das Licht ständig an- und ausgeschaltet, es flackert. Um dieses Flackern zu vermeiden, werden Anf2 und Anf3 dahingehend geändert, daß, wenn die Person den Raum verläßt (bzw. es draußen hell wird), das Licht erst nach t_1 Minuten ausgeschaltet wird.

Wir betrachten hier nur das Sequenzdiagramm "Licht aus" für Anf2. Dort wird eine weitere Instanz *Zeit* benötigt, die jeweils die aktuelle Zeit t_{akt} mitteilt (siehe Bild 10 auf Seite 19). Wird bei *Licht ausschalten()* aufgerufen, fragt *Licht* die aktuelle Eintrittszeit t_{entry} mithilfe der Methode *abfrage()* ab (siehe Bild 10 auf Seite 19, Sicht "Sequenzdiagramme"). Danach fragt *Licht* alle 30 Sekunden die aktuelle Zeit t_{akt} mithilfe der Methode *abfrage()* ab. Sobald $t_{akt} - t_{entry} > t_1$ Minuten ist, und das Licht vorher an war (*zustand* = *an*), und nicht *anschalten()* in der Zwischenzeit aufgerufen wurde, wird die Methode *aus()* von *Lichtaktor* aufgerufen.

Die syntaktische Konsistenz der Sicht "Sequenzdiagramme" bleibt erhalten, da alle Elemente im Sequenzdiagramm "Licht aus" syntaktisch korrekt dargestellt sind. Die semantische Intra-Sicht-Konsistenz bleibt auch erhalten, da z.B. keine Widersprüche bei den Bedingungen im Sequenzdiagramm auftreten.

Die Inter-Sicht-Konsistenz mit der Sicht "Zustandsdiagramme" wird verletzt, da im Zustandsdiagramm "Licht" immer noch direkt das Licht ausgeschaltet wird. Dies könnte behoben werden, indem ein neuer Zustand *Wartend* eingefügt wird, in dem regelmäßig die Zeit t_{akt} abgefragt wird (Do-Anweisung). Dieser wird erst verlassen (nach *Aus*), falls $t_{akt} - t_{entry} > t_1$ Minuten ist (siehe Bild 10 auf Seite 19, Sicht Zustandsdiagramme), oder falls die Methode *anschalten()* aufgerufen wird (nach *An*).

Die Intra-Sicht-Konsistenz von der Sicht "Zustandsdiagramme" würde durch diese Änderung nicht verletzt. Das Zustandsdiagramm "Raum" bliebe unverändert (deshalb in Bild 10 auf Seite 18 weggelassen).

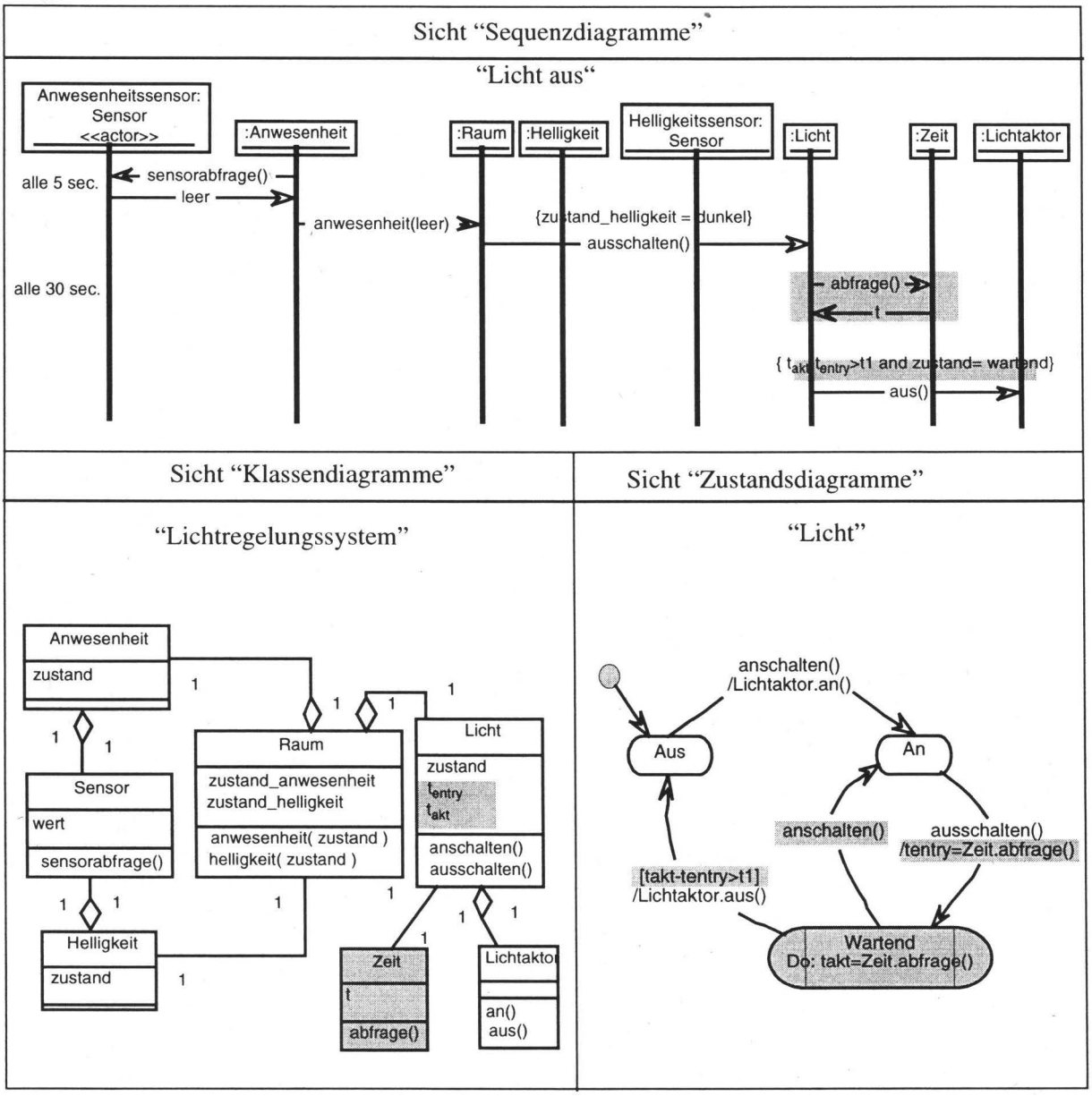

Bild 10: Geänderte, wieder konsistente Sichten - Beispiel Lichtregelungssystem

Da sich aber die Sicht "Zustandsdiagramme" mit der Sicht "Klassendiagramme" überschneidet, müßte zunächst dort die Inter-Sicht-Konsistenz geprüft werden. Sie wäre verletzt, da im Klassendiagramm "Lichtregelungssystem" keine Klasse *Zeit* mit der Methode *abfrage()* und einem Attribut *t* existiert. Diese Klasse würde eingefügt werden (siehe Bild 10 auf Seite 19, Sicht "Klassendiagramme"). Außerdem fehlen der Klasse *Licht* die Attribute t_{entry} und t_{akt}. Auch sie würden eingefügt werden. Die Intra-Sicht-Konsistenz von der Sicht "Klassendiagramme" würde dadurch nicht verletzt, und die Inter-Sicht-Konsistenz sowohl mit der Sicht "Zustandsdiagramme" als auch mit der Sicht "Sequenzdiagamme" wäre damit wiederhergestellt.

Kapitel 3

Ziel der Arbeit und Anforderungen an Lösungsansätze

Dieses Kapitel erläutert das Ziel dieser Arbeit und leitet Anforderungen an Lösungsansätze von diesem Ziel ab.

3.1 Ziel der Arbeit

Ausgehend von dem gegebenen Problem aus Kapitel 2 wird hier das Ziel der Arbeit erläutert. Seien mehrere Sichten in unterschiedlichen Repräsentationssprachen gegeben. Ändert der Benutzer ein Element einer Sichten, so ist die Konsistenz sowohl innerhalb als auch zwischen den Sichten nicht mehr gewährleistet. Konsistenzverletzungen können durch Syntaxregeln und semantische Beziehungen innerhalb und zwischen den Sichten lokalisiert werden. Auf diese Weise kann auch die Auswirkungsanalyse von Änderungen unterstützt werden. Um die Konsistenz innerhalb des Artefakts wiederherzustellen, muß der Benutzer das geänderte Element korrigieren, bis es syntaktisch korrekt ist, die Elemente, die in semantischer Beziehung zu dem geänderten Element stehen, entsprechend ändern oder die Änderung zurücknehmen. Da die semantischen Beziehungen jedoch meist nur implizit sind, muß der Benutzer von Hand die inkonsistenten Stellen suchen und gegebenenfalls ändern (Änderungspropagierung). Bei komplexen Informationsmengen ist dies zeit- und arbeitsintensiv, sowie fehleranfällig.

Das Ziel der Arbeit ist die Auswahl eines Verfolgbarkeitsansatzes zum Zwecke der Änderungsunterstützung. Verfolgbarkeit heißt hier, daß semantische Beziehungen, die dem Zweck der Änderungsunterstützung innerhalb und zwischen Sichten dienen, explizit dargestellt werden. Es soll eine Verbesserung der Änderungsunterstützung erreicht werden. Dies bedeutet, daß der Aufwand von Änderungen reduziert wird und weniger Fehler bei der Durchführung von Änderungen gemacht werden. Dies umfaßt eine Reduktion des Aufwands für die Lokalisierung von der Änderung betroffener Komponenten. Wie schon in 2.3.1 auf Seite 13 erläutert, besteht die Annahme, daß diese Verbesserung erreicht wird, falls ein die Änderungen unterstützendes Werkzeug zur Verfügung steht. Mit dessen Hilfe sollen die Sichten, die Syntaxregeln und semantischen Beziehungen repräsentiert und überprüft werden, und gegebenenfalls automatisch Änderungen propagiert werden.

Aus der Zielsetzung lassen sich Anforderungen an einen Ansatz zur Lösung des Problems ableiten. Der nächste Abschnitt erläutert die abgeleiteten Anforderungen, aufgrund derer wir eine umfassende Recherche nach Lösungsansätzen durchgeführt haben.

3.2 Anforderungen an Lösungsansätze

Die aus dem Ziel abgeleiteten Anforderungen (siehe Tabelle 12 auf Seite 24) unterteilen sich in Anforderungen an die Repräsentationssprache (ARS = Anforderungen an die Repräsentationssprache) und Anforderungen an ein unterstützendes Werkzeug (AW = Anforderungen an das Werkzeug).

3.2.1 Anforderungen an die Repräsentationssprache

Mithilfe der gesuchten Repräsentationssprache S soll es möglich sein, die gegebenen Informationen der realen Welt (kognitive Ebene) darstellen zu können (ARS-Darstellung). Die Verfolgbarkeit innerhalb von Informationsrepräsentationen (eine Sicht) muß hergestellt werden können, d.h. explizite Intra-Sicht-Beziehungen und Syntaxregeln müssen in der Repräsentationssprache S formulierbar sein (ARS-Intra-Regeln). Die Repräsentationssprache S muß die Fähigkeit besitzen, verschiedene Sichten, die in verschiedenen Repräsentationssprachen S_i vorliegen, beschreiben zu können (ARS-Sichten) und Verfolgbarkeit zwischen diesen Sichten herstellen zu können (ARS-Inter-Regeln). Dies geschieht mithilfe von expliziten semantischen Inter-Sicht-Beziehungen.

Zur Änderungsunterstützung durch ein Werkzeug wird die Ausführbarkeit der Repräsentationssprache S auf einem Rechner benötigt. Dies führt neben den schon eingeführten Ebenen (kognitive und Repräsentationsebene) zur Implementationsebene (siehe Bild 11).

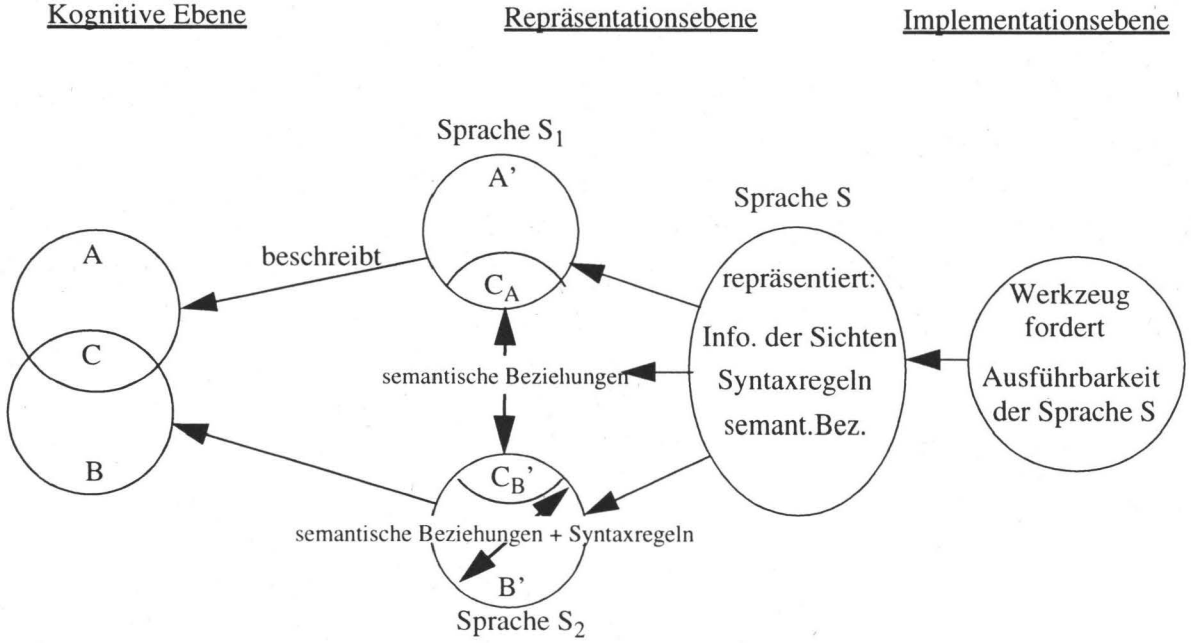

Bild 11: Integration einer Implementationsebene

Auf der Implementationsebene werden Werkzeuge betrachtet. Das Werkzeug implementiert sowohl die Informationsrepräsentationen der Sichten mithilfe der Repräsentationssprache S, als auch Syntaxregeln und semantische Beziehungen innerhalb oder zwischen den Sichten, die durch die Repräsentationssprache S explizit ausgedrückt werden können. Die Implementationsebene fordert eine formale Basis von der Repräsentationssprache S (ARS-Formal). Ist die Repräsentationssprache S informell, so kann sie nicht auf einem Rechner ausgeführt werden, d.h. automatische Konsistenzprüfungen sind z.B. unmöglich.

3.2.2 Anforderungen an ein unterstützendes Werkzeug

Das Werkzeug, das basierend auf der Repräsentationssprache S implementiert ist, muß zur Unterstützung von Änderungen verschiedene Mechanismen zur Verfügung stellen (siehe Tabelle 12).

Zunächst muß eine Prüfung der Konsistenz (AW-Konsistenz) möglich sein, um festzustellen, ob eine Informationsrepräsentation (eine Sicht) oder auch mehrere Sichten konsistent sind. Ein Werkzeug soll gegebenenfalls die Konsistenzprüfungen automatisch durchführen, falls sich Beziehungen formulieren lassen, die automatisch überprüfbar sind. Der Benutzer erhält dadurch zunächst die Information, ob Konsistenz vorliegt oder nicht. Weiterhin soll das Werkzeug die Auswirkungsanalyse (AW-Auswirkungen) soweit wie möglich unterstützen, d.h. dem Benutzer die Suche nach inkonsistenten Stellen, die von einer Änderung betroffen sind und gegebenenfalls geändert werden müssen (Änderungspropagierung), erleichtern. Damit wird dem Benutzer die Möglichkeit geboten, die inkonsistenten Stellen direkt zu ändern, ohne sie suchen zu müssen. Er kann die Änderung andererseits zurückzunehmen, falls ihm der Aufwand zur Bearbeitung der Auswirkungen zu groß erscheint.

Je nach Art der Inkonsistenzen und der damit notwendigen zusätzlichen Änderungen zur Wiederherstellung der Inter-und Intra-Sicht-Konsistenz soll das Werkzeug automatische Änderungspropagierung unterstützen (AW-Änderungen). Es wäre denkbar, daß in einfachen und häufig vorkommenden Fällen von Inkonsistenz (welche das sind, muß der Benutzer entscheiden) Änderungen vom Werkzeug automatisch durchgeführt werden. Im Falle des Lichtregelungssystems könnte eine solche Änderung beispielsweise die konsistente Umbenennung einer Klasse sein. Bei schwierigeren Fällen sollten dem Benutzer jedoch nur die betroffenen Stellen angegeben werden. Dieser kann die Änderungen dann von Hand durchführen.

Tabelle 12: Anforderungen an Lösungsansätze

Aspekt	Bezeichnung	Erläuterung
Repräsentationssprache	ARS-Darstellung	Fähigkeit, die gegebenen Informationen der realen Welt (kognitive Ebene) darzustellen.
	ARS-Sichten	Fähigkeit, verschiedene Repräsentationssprachen verschiedener Sichten zu beschreiben.
	ARS-Intra-Regeln	Fähigkeit, Verfolgbarkeit innerhalb von Informationsrepräsentationen, d. h. in einer Sicht, herzustellen durch explizite Intra-Sicht-Beziehungen + Syntaxregeln.
	ARS-Inter-Regeln	Fähigkeit, Verfolgbarkeit zwischen Sichten herzustellen durch explizite Inter-Sicht-Beziehungen.
	ARS-Formal	Formale Basis der Repräsentationssprache (wird zur Ausführbarkeit benötigt).
Werkzeug	Änderungsunterstützung durch:	
	AW-Konsistenz	Konsistenzprüfung
	AW-Auswirkungen	Auswirkungsanalyse
	AW-Änderungen	Automatische Änderungspropagierung

Ausgehend von diesen Anforderungen haben wir eine Literaturrecherche durchgeführt. Das Ziel der Recherche war, den heutigen Stand der Änderungsunterstützung mit Werkzeugen bei Existenz mehrerer Sichten in unterschiedlichen Repräsentationssprachen zu erfassen. Es kristallisierte sich heraus, daß mehrere Ansätze und auch Werkzeuge existieren, über die im folgenden Kapitel ein kurzer Überblick gegeben wird.

Kapitel 4

Stand der Forschung

Um den derzeitigen Stand der Forschung zu charakterisieren, haben wir eine Literaturrecherche durchgeführt, welche auf der in Kapitel 3 vorgestellten Zielsetzung basiert. Die gefundenen Ansätze haben wir beurteilt anhand der Anforderungen aus Kapitel 3, die sich aus der Zielsetzung ableiten (siehe Bild 13).

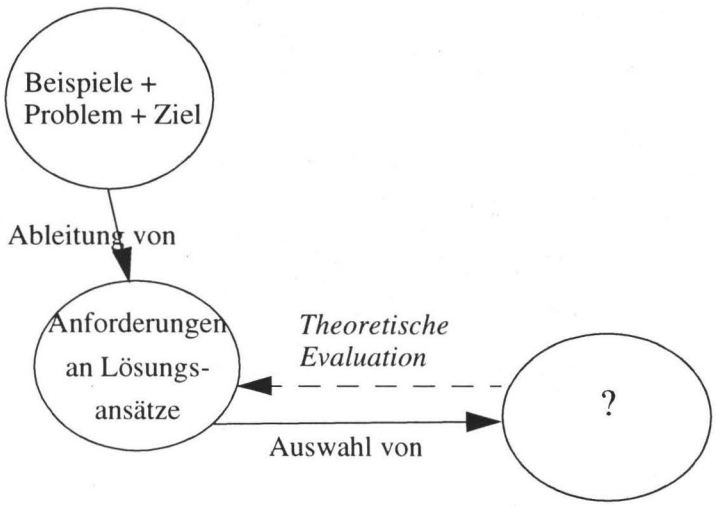

Bild 13: Evaluation von Lösungsansätzen

Verschiedene Gebiete der Informatik beschäftigen sich mit dem Problem der Änderungsunterstützung und somit auch Konsistenzprüfung bei Existenz mehrerer Sichten. Beispielsweise bei der Integritätssicherung in klassischen Datenbanksystemen, bei den Techniken zur Modellierung von Anforderungen und Entwürfen, sowie bei der Integration verschiedener Werkzeuge zur Unterstützung der Softwareentwicklung (CASE-Umgebungen) treten die Probleme auf. Im folgenden werden wir die Problemlösungsansätze in den einzelnen Gebiete beschreiben und hinsichtlich der aus der Zielsetzung abgeleiteten Anforderungen (ARS-Darstellung, -Sichten, -Intra-Regeln, -Inter-Regeln, -Formal und AW-Konsistenz, -Auswirkungen, -Änderungen) bewerten. Am Ende des Kapitels befindet sich eine Tabelle, die die vorgenommenen Bewertungen zusammenfaßt. Die Tabelle ermöglicht es, die Auswahl des für das Problem und das Ziel geeignetsten Ansatzes nachzuvollziehen.

4.1 Sicherung der Integrität in klassischen Datenbanksystemen

Datenbanksysteme (DBS) bestehen aus einer Datenbank, in der Daten gespeichert werden, und einem Datenbankverwaltungssystem (DBVS), das Mechanismen zur Verfügung stellt, um Daten in der Datenbank abzulegen, zu verwalten, zu verarbeiten und auszuwerten. In Datenbanksystemen werden Informationen der realen Welt modelliert. Dies geschieht mithilfe sogenannter Informationsmodelle [Har87]. Zunächst gehen wir auf die Informationsmodelle in klassischen Datenbanksystemen ein, danach erläutern wir die verschiedenen Arten der Integritätssicherung in klassischen Datenbanksystemen und abschließend die Mechanismen der DBVS.

4.1.1 ARS-Darstellung: Repräsentation von Information

Informationen der realen Welt werden in Datenbanksystemen mithilfe von Informationsmodellen abgebildet. Diese sind zunächst konzeptioneller Art (conceptual modeling) und werden in klassischen (relationalen) DBS mit Entity-Relationship-Diagrammen (relationale Informationsmodelle) dargestellt. Danach folgt der logische Schema-Entwurf, der das Informationsmodell in ein Datenmodell abbildet. Es gibt verschiedene Arten von Datenmodellen, z.B. Relationen-, Netzwerk- oder hierarchische Datenmodelle. Die Datenmodelle (Schemata) werden mittels formalen Datenbeschreibungssprachen (DDL - Data Description Languages) dem Rechner vermittelt (ARS-Formal). Die Datenbeschreibungssprachen sind z.B. mengentheoretisch-orientiert (z.B. Relationenalgebra), prädikatenlogisch-orientiert (Relationenkalkül) oder abbildungsorientiert (z.B. SQL).

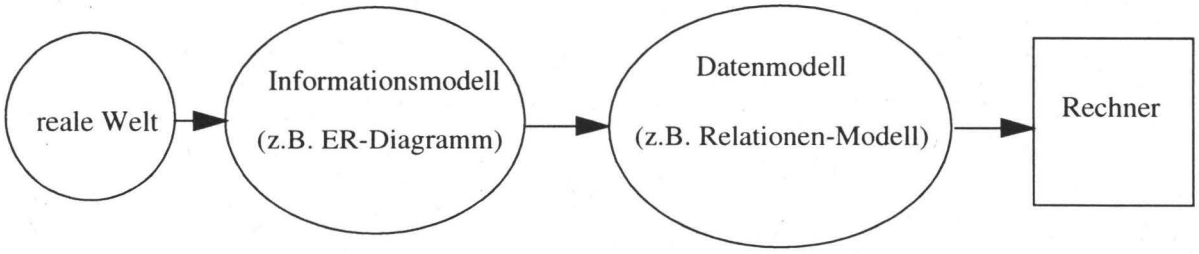

Bild 14: Informations-/ Datenmodelle in DBS

4.1.2 ARS-Sichten: Repräsentation von Repräsentationssprachen

Mithilfe von erweiterten ER-Diagrammen (Metaentities, -relationships, -attribute) und somit auch erweiterten Datenmodellen könnten verschiedene Sichten in verschiedenen Repräsentationssprachen beschrieben werden. In klassischen (relationalen) Datenbanksystemen ist dies nicht möglich. Allerdings werden zur Zeit Anstrengungen unternommen, um die Datenmodelle von relationalen DB zur Repräsentation von Repräsentationssprachen zu erweitern [Fla98].

4.1.3 ARS-Intra-Regeln: Konsistenz

Auf dem Gebiet der Datenbanken wird Konsistenz oft mit "Integrität" bezeichnet [Rei92]. Integrität bedeutet jedoch im allgemeinen, daß die Abbildung der realen Welt in die Datenbank-Schemata ohne Fehler ist (siehe Bild 15 auf Seite 27). Konsistenz heißt dagegen, daß sich die in der Datenbank existierenden Daten nicht widersprechen. Die Integrität ist innerhalb des Datenbanksystems nicht prüfbar, da das Datenbanksystem keinen Zugriff auf die Daten der realen Welt hat. Das Datenbankverwaltungssystem kann nur die Konsistenz der Daten sichern [Här97]. Mit "Integritätsprüfung" ist deshalb "Konsistenzprüfung" in unserem Sinne gemeint. Aus diesem Grunde werden wir hier den Begriff "Konsistenz" weiterverwenden.

Der interessierende Teil der realen Welt (Miniwelt) wird im DBS durch einen Datenbankzustand beschrieben. Ein Datenbankzustand wird definiert durch die existierenden Datenelemente und ihre Werte. Ändert sich die reale Welt, so wird dies im Datenbanksystem durch einen Zustandsübergang simuliert, so daß der neue Datenbankzustand die veränderte reale Welt beschreibt (siehe Bild 15 auf Seite 27). Die Erhaltung der logischen Datenkonsistenz geschieht durch Spezifizierung "akzeptabler" Datenbank-Zustände. Dies erfolgt durch Beschreibung der korrekten Daten durch Konsistenzbedingungen (= Integritätsbedingungen). Dies sind sogenannte Invarianten, d.h. Prädikate über den Daten, die immer erfüllt sein müssen.

Dabei existieren modellinhärente (oder strukturelle) Konsistenzbedingungen, die auf der Struktur des zugrundeliegenden Informationsmodells beruhen, beim relationalen Informationsmodell z.B. referentielle Konsistenz (= relationale Invariante). Diese werden "Constraints" (Einschränkungen) genannt. Diese muß der Benutzer nicht explizit definieren, sie sind systemkontrolliert. Weiterhin existieren benutzerdefinierte semantische Konsistenzbedingungen, welche "Assertions" (Zusicherungen) genannt werden. Im folgenden betrachten wir die referentielle und die semantische Konsistenzsicherung (im Datenbanken-Sprachgebrauch Integritätssicherung), sowie die Konsistenzsicherung bei Schema-Integration.

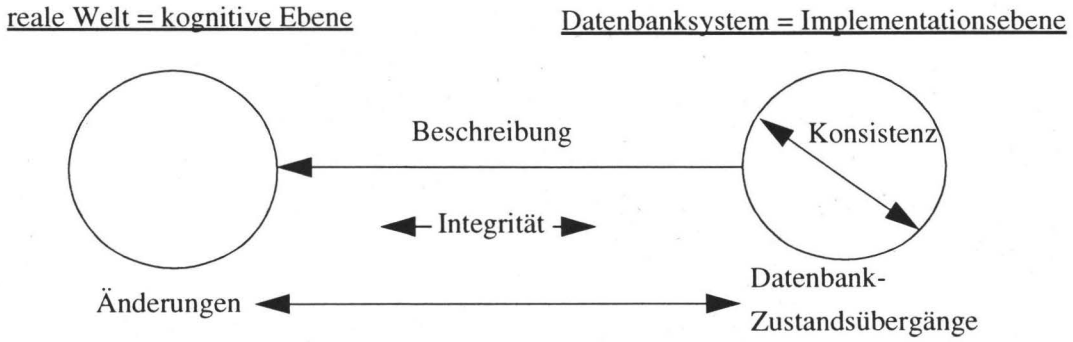

Bild 15: Integrität/Konsistenz in Datenbanken

Referentielle Konsistenzsicherung (= ref. Integritätssicherung)

Die referentielle Konsistenz sichert ein Modellierungsprinzip des relationalen Informationsmodells. Referentielle Konsistenzbedingungen dienen dazu, erlaubte, d.h. in sich konsistente, Datenbankzustände zu beschreiben. In [Rei92] wird als Datenmodell das Relationen-Modell zugrundegelegt. Einträge in Tabellen (Relationen) sind jeweils durch ihren Primärschlüssel pk eindeutig bestimmt. Ein

Fremdschlüssel fk eines Eintrages verweist auf einen Primärschlüssel pk eines Eintrages in einer anderen Tabelle (Relation). Was passiert nun, wenn pk geändert oder gelöscht wird? Zur Lösung dieses Problems werden Trigger mit Constraints auf den Tabellen angegeben, die festlegen, ob der Fremdschlüssel fk auf NULL gesetzt wird, die Änderung auf fk kaskadiert werden, oder die Änderung zurückgewiesen wird.

Diese verschiedenen Strategien werden durch ein Triggerkonzept realisiert. Hier findet also im wesentlichen eine automatische Änderungspropagierung (AW-Änderungen) zur Wiederherstellung der referentiellen Konsistenz statt [HR92]. Bei automatischen Änderungspropagierungen stellt sich das Problem der Effizienz zur Laufzeit. Weiterhin stellt sich noch das Problem von Konflikten der Auswirkungen, dessen Behandlung zur Zeit noch sehr großen Aufwand erfordert.

Semantische Konsistenzsicherung (= sem. Integritätssicherung)

Um die reale Welt im Datenbanksystem möglichst genau abzubilden, ist es wünschenswert, inhaltliche Zusicherungen über den Daten zu definieren und aufrechtzuerhalten. Diese Zusicherungen heißen Konsistenzbedingungen zur Aufrechterhaltung der semantischen Konsistenz in einer Datenbank (ARS-Intra-Regeln). Sie enthalten ein Prädikat, das wahr sein muß, die Menge der Datenelemente (Relationen), auf die sich die Bedingung bezieht und eine Reaktionsregel. Die Reaktionsregel definiert die Reaktion des Datenbanksystems, wenn die Bedingung verletzt ist. Strenge Konsistenzbedingungen erfordern ein Zurücksetzen der Operation, die zur Verletzung der Bedingung geführt hat. Schwache Konsistenzbedingungen melden nur eine Warnung an den Benutzer und selbstkorrigierende Konsistenzbedingungen führen zur automatischen Änderung der Werte, die zur Wiederherstellung der Konsistenz notwendig sind [Web81]. Die Überprüfung der Konsistenzbedingungen erfolgt durch sogenannte Prüfroutinen (AW-Konsistenz), die den Zustand der Datenbank auf Konsistenz überprüfen. Bei allen Ansätzen zur Sicherung der Konsistenz (= Integrität) in Datenbanksystemen stellt sich das Problem der Effizienz zur Laufzeit. Die Forschungen auf diesem Gebiet befassen sich unter anderem damit, Strukturen zu entwickeln, die die Konsistenzprüfung schneller machen.

View-Integration

Liegen verschiedene Sichten auf die Daten einer Datenbank vor, so beschäftigt sich das Thema "View"-Integration mit der Vereinigung dieser Sichten. Dabei sollen Inkonsistenzen beim Integrationsprozeß vermieden werden. Hier existieren verschiedene Ansätze, die sich unter anderem darin unterscheiden, welches Datenmodell zugrunde gelegt wird.

Nach [BC85] wird ein Datenschema D durch Relationen-Tabellen R und Konsistenz-Constraints C beschrieben (D = < R | C >), falls das relationale Datenmodell vorliegt. Liegen mehrere Views vor, so wird zur View-Integration eine formalen Spezifikationssprache benötigt, um Beziehungen (Integration-Constraints I) zwischen Views zu beschreiben. Ausgegangen wird von D = < R | C | I >. D wird transformiert zu D' = < R' | C' | { } >, falls möglich automatisch. Falls Konflikte auftreten, findet die Transformation mithilfe des Benutzers statt [FP92].

4.1.4 AW-Konsistenz, -Auswirkungen, -Änderungen

In klassischen (relationalen) Datenbanksystemen (DBS) stellen die Datenbankverwaltungssysteme (DBVS) Mechanismen zur Verfügung, die sowohl Überprüfungen von modellinhärenten wie auch benutzerdefinierten Konsistenzbedingungen erlauben (AW-Konsistenz). Dazu müssen die Konsistenzbedingungen explizit festgelegt sein. Bei Änderungsoperationen auf den Daten werden "Qualitätskontrollen" durchgeführt (durch Prüfroutinen), d.h. es werden nur Änderungen zugelassen, die allen spezifizierten Konsistenzbedingungen entsprechen. Die meisten DBS haben noch Probleme mit solchen modellinhärenten Konsistenzbedingungen, wie z.B. referentielle Konsistenz, so daß noch nicht einmal die Modellierungsprinzipien des relativ kargen ER-Informationsmodells unterstützt werden. Durch Anfragen an das DBS in Verbindung mit Konsistenzbedingungen kann eine Auswirkungsanalyse (AW-Auswirkungen) von Änderungen realisiert werden. Automatische Änderungspropagierungen werden hier mithilfe des Triggerkonzeptes unterstützt (AW-Änderungen). Trigger werden durch sogenannte ECA-Regeln (Event-Condition-Action) realisiert. Ein Ereignis (Event = Änderungsoperation) startet automatisch Folgeänderungen (Aktionen) auf durch eine Bedingung (Condition) ausgewählten Datenelementen zur Wahrung der Datenbank-Konsistenz [Har87]. Dies wirft jedoch noch große Probleme auf und ist erst teilweise in einigen DBS verwirklicht (z.B. erst in SQL3).

4.1.5 Zusammenfassende Bewertung

Zusammenfassend ist zu sagen, daß alle Anforderungen an eine geeignete Repräsentationssprache erfüllt werden, außer ARS-Sichten und ARS-Inter-Regeln. Informationen der realen Welt (ARS-Darstellung) werden in Datenbanksystemen mithilfe von Informations- und Datenmodellen abgebildet. Die Datenmodelle werden mittels formalen Datenbeschreibungssprachen (ARS-Formal) modelliert, die von einem Rechner ausgeführt werden können. Es können explizite semantische Beziehungen innerhalb von Informationsrepräsentationen (ARS-Intra-Regeln) mithilfe von Konsistenzbedingungen formuliert werden (im Datenbanken-Sprachgebrauch Integritätsbedingungen). Liegen jedoch Informationsrepräsentationen in verschiedenen Repräsentationssprachen vor, so erlauben die gebräuchlichen Datenbeschreibungssprachen keine Beschreibung dieser Repräsentationssprachen (ARS-Sichten) und somit auch keine Integration der verschiedenen Repräsentationssprachen (ARS-Inter-Sichten).

Die Anforderungen an Werkzeuge werden durch klassische Datenbanksysteme erfüllt. DBVS Mechanismen erlauben sowohl Überprüfungen von modellinhärenten wie auch benutzerdefinierten Konsistenzbedingungen (AW-Konsistenz). Durch Anfragen an das DBS in Verbindung mit Konsistenzbedingungen kann eine Auswirkungsanalyse (AW-Auswirkungen) von Änderungen realisiert werden. Automatische Änderungspropagierungen werden mithilfe des Triggerkonzeptes unterstützt (AW-Änderungen).

4.2 Modellierungstechniken

Zur Modellierung eines zu entwickelnden Systems existieren eine Reihe von Modellierungstechniken, wie z.B. OMT. Modellierungstechniken umfassen Beschreibungssprachen, die je nach Fokus der Technik sehr unterschiedlich sein können. Beispielsweise könnte der Fokus einer Technik zur Modellierung

von Anforderungen auf einer möglichst guten Interaktion mit dem Kunden zur externen Verifikation der Anforderungen liegen. Solche Techniken umfassen meist visuelle Sprachen, da Diagramme leicht verständlich sind. Ein Fokus anderer Anforderungstechniken liegt beispielsweise auf einer automatischen internen Konsistenzprüfung der Anforderungen. Diese Techniken umfassen Sprachen, die eher formal sind, um eine automatische Überprüfung zu ermöglichen.

Im folgenden werden in jedem Abschnitt solche Techniken herausgegriffen, die die jeweilige betrachtete Anforderung besonders gut (relativ gesehen) unterstützen.

4.2.1 ARS-Darstellung: Repräsentation von Informationen

Informationen, die hier repräsentiert werden, sind Beschreibungen eines Softwaresystem (z.B. Anforderungen oder Entwürfe). Zur Darstellung existieren sogenannte Beschreibungssprachen, wie z.B. OMT [RBP+91], OOSE [Jac94], UML [Rat97], Statecharts [Har87] oder Z [Spi89].

Mithilfe derartiger Beschreibungssprachen kann die Informationsmenge aus der realen Welt (beispielsweise die Anforderungen an ein System) modelliert werden (ARS-Darstellung). Diese Sprachen sind jedoch meistens semi-formal (ARS-Formal), d.h. die Semantik der Sprache liegt größtenteils implizit vor (z.B. in Form von natürlicher Sprache als Erklärung der Symbole der Sprache).

Das Problem von formalen Beschreibungssprachen, wie z.B. Z, liegt darin, daß diese Sprachen meist schwer zu verstehen und zu erlernen sind, weshalb sie nicht oft verwendet werden (Akzeptanz-Probleme). Somit liegen noch wenige praktische Erfahrungen mit diesen Sprachen vor [Rom97]. Außerdem können bei einigen formalen Sprachen nur sehr kleine Informationsmengen repräsentiert werden, da der Beschreibungsaufwand für komplexere Informationsmengen zu groß ist.

4.2.2 ARS-Sichten: Repräsentation von Repräsentationssprachen

Zur Reduktion der Komplexität erlauben verschiedene Modellierungstechniken die Beschreibung von Sichten eines Systems (z.B. bei UML die Modellierung der statischen, dynamischen und funktionalen Aspekte eines Systems). Für die Beschreibung werden verschiedene Beschreibungssprachen verwendet (z.B. Klassendiagramme oder Zustandsdiagramme). Die Beschreibung von verschiedenen Repräsentationssprachen mithilfe einer Repräsentationssprache ist möglich, falls Sprachkonzepte zur Metamodellierung vorliegen, wie z.B. bei UML [Rat97] (vergleiche hierzu 4.4). Bei den meisten Beschreibungssprachen liegen diese jedoch nicht vor.

4.2.3 ARS-Intra- und Inter-Regeln: Konsistenz

Zur Unterstützung der Konsistenz werden an dieser Stelle verschiedene Konzepte unterschieden: Constraints und Verfolgbarkeitsbeziehungen [BA96]. Es wird Konsistenz innerhalb einer Sicht, zwischen Sichten in einem Software-Artefakt und zwischen Artefakten betrachtet.

ARS-Intra-Regeln: Verfolgbarkeit in einer Sicht

In [HL96] und [HJL96] wird die zustandsbasierte Konsistenzprüfung bei der SCR (Software Cost Reduction)-Anforderungstechnik betrachtet. Die Anforderungen liegen in Tabellen-Notation (Mode-Transition-Tabellen) vor und werden in endliche Automaten übersetzt. Konsistenz wird dann anhand dieser endlichen Automaten automatisch überprüft. Hierbei geht es um das Entdecken von Fehlern wie Nichtdeterminismen, fehlende Fälle (Unvollständigkeit) oder zirkuläre Definitionen. Der Prototyp des Konsistenz-Checkers benötigt die Interaktion mit dem Benutzer und ist nicht auf allgemeine Repräsentationssprachen anzuwenden.

ARS-Inter-Regeln: Verfolgbarkeit zwischen Sichten

Existieren verschiedene Sichten auf einer Ebene (z.B. auf Anforderungsebene), stellt sich das Problem der Konsistenzerhaltung der Sichten auf die gemeinsame Informationsmenge. Es ist wünschenswert, daß sich die verschiedenen Sichten (z.B. statische und dynamische Sicht) nicht widersprechen (d.h. konsistent sind). Wird eine Sicht geändert, so sollten die Änderungen in die anderen Sichten verfolgbar sein.

Werkzeuge, die Verfolgbarkeitstabellen, Crossreferenzen, usw. erstellen (siehe Verfolgbarkeit zwischen Software-Artefakten) können zur Visualisierung von horizontalen Verfolgbarkeitsbeziehungen eingesetzt werden. Dazu müssen die Verfolgbarkeitsbeziehungen zwischen Elementen der verschiedenen Sichten von Hand gesetzt werden [Wie95].

Verschiedene Ansätze zur Gewährleistung von Konsistenz zwischen Sichten sind mit dem Ziel entstanden, ein System aus verschiedenen Anwenderperspektiven beschreiben und diese Perspektiven anschließend integrieren zu können (Viewpoint-Ansatz) ([BC85], [EFK+94], [SF97]). Wichtig ist die Gewährleistung von Konsistenz zwischen den Perspektiven. Einen Überblick über verschiedene Ansätze gibt [Ver97].

Bei der Beschreibungssprache UML existiert eine Object Constraint Language (OCL). Mit dieser Sprache können sowohl die UML-Diagramme, als auch das UML-Metamodell angereichert werden. Dadurch sind explizite semantische Beziehungen innerhalb und zwischen Sichten formulierbar.

ARS-Inter-Regeln: Verfolgbarkeit zwischen Software-Artefakten

Werden die Anforderungsanalyse, der Entwurf und die Implementierung als Ebenen betrachtet, so liegen in der oberen Ebene jeweils abstraktere Informationen vor, als in der Ebene darunter. Die untere Ebene enthält also sowohl die Informationen der darüberliegenden Ebene, gegebenenfalls in anderer Form (Repräsentationssprache), als auch zusätzliche Informationen. Es sollte gewährleistet sein, daß beim Übergang von einer oberen zu einer darunterliegenden Ebene die gemeinsamen Informationen konsistent vorliegen, jeweils nur in einer anderen oder der gleichen Repräsentationssprache. Dazu müssen die Beziehungen zwischen den Informationen der verschiedenen Ebenen bekannt sein (vertikale Verfolgbarkeit). Wenn z.B. eine Anforderung auf der obersten Ebene geändert wird, sollte direkt erkennbar sein, wo im Entwurf und im Kode die Stelle ist, die geändert werden muß, um der geänderten Anforderung zu entsprechen.

Dafür existiert eine Reihe von Werkzeugen, in denen die Verfolgbarkeitsbeziehungen von Hand gesetzt werden müssen, und die eher informelle Artefakte grob verbinden. Die Visualisierung der Verfolgbar-

keitsbeziehungen geschieht dann z.B. durch Verfolgbarkeitstabellen, Crossreferenzen, Diagramme, usw. [Wie95]. Verfolgbarkeitstabellen z.B. listen jeweils für zwei Ebenen auf, welche Elemente der unteren Ebene von welchen Elementen der oberen Ebene abhängen.

Die in [CG95] betrachtete zustandsbasierte Konsistenzprüfung behandelt den Übergang von Anforderungen zum Entwurf. Es werden endliche Automaten zur formalen Beschreibung der Anforderungen und des Entwurfs eingesetzt. Basierend auf diesen endlichen Automaten kann eine automatische Überprüfung der Konsistenz zwischen Anforderungen und Entwurf stattfinden [CG96]. Diese Methode ist auf eine bestimmte Repräsentationssprache (SCR-Tabellen) beschränkt und kann nicht als allgemeiner Ansatz genutzt werden.

In [SHA96] wird die Verfeinerung von Datenflußdiagrammen (DFD) zu Prozeßdiagrammen (PAD-ProcessActionDiagram) betrachtet, deren Konsistenz dann anhand von Z (prädikatenlogische Sprache erster Stufe) überprüft wird. Ein DFD wird durch sogenannte MiniSpec's beschrieben. Eine MiniSpec stellt einen primitiven Prozeß innerhalb des DFD dar. Sie besteht aus einem Datendiagramm (ER-Diagramm) und aus einer Beschreibung der Operationen. Ein PAD hingegen beschreibt eine Prozeßstruktur mit Konstrukten wie Schleifen, usw. Die MiniSpec und das PAD werden in sogenannte Z-Schemata übersetzt. Ein Z-Schema besteht aus einer Schema-Signatur (Variablen und ihre Typen) und einem Schema-Prädikat (definiert Beziehungen zwischen Variablen). Z basiert auf Prädikatenlogik erster Stufe. Zur Konsistenzprüfung wird das Beweisverfahren der Reduktion angewandt. Ein PAD ist eine korrekte Verfeinerung einer MiniSpec, falls unter den Vorbedingungen, die in der MiniSpec definiert sind, jedes Verhalten des PAD, das möglich ist, in der MiniSpec erlaubt ist. Diese Konsistenzprüfung ist jedoch nur eine Methode und wird nicht durch ein Tool unterstützt, da die Übersetzung der Operationen der MiniSpec in Z nicht mechanisierbar sind. Außerdem ist dieses Verfahren nur für sehr kleine Bereiche anwendbar, da sonst der Aufwand zu groß ist.

4.2.4 AW-Konsistenz, -Auswirkungen, -Änderungen

Die zur Verfügung stehenden Modellierungswerkzeuge, die die einzelnen Modellierungstechniken unterstützen (z.B. Statemate, Rational Rose), führen nur syntaktische und fest implementierte semantische Konsistenzprüfungen durch. Sie unterstützen keine Überprüfung von benutzerdefinierten semantische Beziehungen (AW-Konsistenz). Die Modellierungswerkzeuge arbeiten auf festgelegten Repräsentationssprachen, deren Semantik im Werkzeug fest implementiert ist. Die Auswirkungsanalyse erfaßt somit nur syntaktische und fest implementierte semantische Auswirkungen einer Änderung (AW-Auswirkungen). Automatische Änderungspropagierung (AW-Änderungen) wird teilweise unterstützt (z.B. werden Aggregationen von einer Klasse automatisch gelöscht, wenn die Klasse gelöscht wird).

Neben den Modellierungswerkzeugen gibt es verschiedene Requirements Management- (oder Verfolgbarkeits-) Werkzeuge. Diese übernehmen das Darstellen und Verwalten von Beziehungen innerhalb und zwischen Software-Artefakten. Sie unterstützen eine Überprüfung der Konsistenz (AW-Konsistenz), eine Analyse der Auswirkungen von Änderungen (AW-Auswirkungen) und das Durchführen von Änderungen (AW-Änderungen), indem sie ein Navigieren über die von Hand angelegten Bezie-

hungen erlauben. Automatische Konsistenzüberprüfungen oder automatische Änderungspropagierungen sind nicht möglich.

4.2.5 Zusammenfassende Bewertung

Einige Beschreibungssprachen (z.B. UML) erfüllen größtenteils die Anforderungen an eine geeignete Repräsentationssprache durch das Konzept der Metamodellierung (vergleiche 4.4). Viele andere Beschreibungssprachen unterstützen nur einzelne Aspekte und kommen daher zur Verwendung in unserem Kontext nicht in Frage. Modellierungstechniken verwenden das Konzept der Metamodellierung, um Konsistenz zwischen Sichten zu gewährleisten z.B beim Viewpoint-Ansatz.

Die zugehörigen Modellierungs- und Requirements Management-Werkzeuge unterstützen nur ungenügend die geforderten Funktionalitäten (AW-Konsistenz, -Auswirkungen und -Änderungen).

4.3 Integration von Werkzeuge in CASE-Umgebungen

Im Rahmen des Software Engineering existieren verschiedene Werkzeuge, die den Benutzer beim Entwickeln von Softwaresystemen unterstützen. Für eine geeignete Unterstützung des Benutzers muß ein Austausch von Dokumenten zwischen den Werkzeugen möglich sein. Beispielsweise erlaubt ein Modellierungswerkzeug die Modellierung von Informationen der realen Welt mithilfe der Repräsentationssprache UML und ein anschließendes Generierung von Kode aus den UML-Modellen. Ein Programmierwerkzeug (Programmierumgebung) erlaubt die Bearbeitung des generierten Kodes. Sinnvollerweise sollten sich aus dem bearbeitetem Kode wieder UML-Modelle erstellen lassen. Im Rahmen einer CASE (Computer Aided Software Engineering) -Umgebung liegen verschiedene Werkzeuge integriert vor. In dieser CASE-Umgebung können Daten zwischen den Werkzeugen ausgetauscht werden. Dafür müssen jedoch die Beziehungen zwischen den Repräsentationssprachen der einzelnen Werkzeuge modelliert werden, damit sich die verschiedenen Werkzeuge "verstehen". Dies führt auf das Gebiet der Wissensmodellierung. Das Gebiet basiert auf Informationsmodellen, die mithilfe von Metamodellierungs- und Regelkonzepten erweitert sind und von Werkzeugen (Wissensbankverwaltungssysteme) unterstützt werden.

Um Wissen zu modellieren, das die Beziehungen zwischen zwei verschiedenen Werkzeugen mit verschiedenen unterliegenden Repräsentationssprachen beschreibt, eignet sich die Metamodellierung. Die Semantik von Repräsentationssprachen ist meist implizit definiert, d.h. die Definition liegt z.B. in Form von informellem Text vor, und es existiert keine formale, d.h. mathematisch präzise Beschreibung der Semantik der Sprache. Die Vorteile der Metamodellierung liegen in der Möglichkeit, Sprachkonzepte explizit zu definieren. Eine explizite Beschreibung eliminiert Zweideutigkeiten der Sprachbeschreibung.

4.3.1 ARS-Darstellung: Repräsentation von Information

Metamodelle sind erweiterte klassische Informationsmodelle. Diese werden durch das Prinzip der Metamodellierung ergänzt. Informationen der realen Welt können beschrieben werden (ARS-Darstellung), da die klassischen Informations- und Datenmodelle nur erweitert werden, und schon mit den klassischen Datenmodellen Informationsrepräsentationen möglich sind (siehe 4.1.1). Metamodelle können z.B. anhand von erweiterten Klassendiagrammen, konzeptuellen Graphen, mit semantischen Netzen, Frames usw. dargestellt werden (Informationsmodelle). Diese werden wiederum mit entsprechenden Datenmodellen realisiert und mithilfe von formalen Datenbeschreibungssprachen, z.B. objektorientierte Sprachen (C++), Netzwerksprachen (IXL [Ric92] orientiert an Graphgrammatiken), Graphbeschreibungssprachen (PROGRES orientiert an Graphgrammatiken) oder Ontologie-Sprachen (PROLOG orientiert an Prädikatenlogik, Hornlogik), dem Rechner vermittelt (ARS-Formal).

4.3.2 ARS-Sichten: Repräsentation von Repräsentationssprachen

Mittels Metamodellierung lassen sich auch die Konzepte von verschiedenen Repräsentationssprachen beschreiben. Die Repräsentationssprachen werden untersucht, und ihre Syntaxregeln werden festgelegt. Daraus wird ein Metamodell dieser Repräsentationssprachen erstellt. Informationen, die nun in diesen Repräsentationssprachen dargestellt werden, können anhand des Metamodells auf syntaktische Korrektheit überprüft werden.

4.3.3 ARS-Intra- und Inter-Regeln: Konsistenz

Die Syntaxregeln der Sprache werden durch modellinhärente Konsistenzbedingungen (diese sind nun auch gegenüber dem klassischen Informationsmodell erweitert, z.B. durch die Invariante für das Prinzip der Metamodellierung) repräsentiert. Um semantische Beziehungen (ARS-Intra-Regeln) in einer Sicht formulieren zu können, werden benutzerdefinierte Konsistenzbedingungen definiert. Diese legen einschränkende Bedingungen auf einer Sicht fest, sogenannte Invarianten, die zu jeder Zeit erfüllt sein müssen. Diese Konsistenzbedingungen werden Constraints genannt.

Liegen zwei verschiedene Repräsentationssprachen vor, so definiert man beide Sprachen mithilfe einer Metamodellierungssprache. Dann kann man Beziehungen zwischen den einzelnen Konzepten dieser beiden Repräsentationssprachen erkennen und in Konsistenzbedingungen festlegen. Kommt noch eine weitere Repräsentationssprache hinzu, so muß diese in derselben Metamodellierungssprache formuliert werden und kann dann in das Metamodell der beiden anderen Repräsentationssprachen eingefügt werden, d.h. Metamodelle können leicht nachträglich erweitert werden. Metamodelle ermöglichen es also, verschiedene Repräsentationssprachen zu beschreiben und mithilfe von Constraints zwischen den verschiedenen Metamodellen der Repräsentationssprachen miteinander in Beziehung zu setzen (ARS-Inter-Regeln).

4.3.4 AW-Konsistenz, -Auswirkungen, -Änderungen

Ist das Wissen, das in den verschiedenen Werkzeugen existiert, mithilfe von Metamodellen modelliert, so können jedoch noch nicht die semantischen Beziehungen zwischen den Werkzeugen automatisch (d.h. werkzeug-unterstützt) überprüft werden. Man benötigt dafür sogenannte Wissensbankverwaltungssysteme (WBVS). Dies führt uns auf das Gebiet der Künstlichen Intelligenz [Ric92]. Wissensbankverwaltungssysteme bestehen aus einer Wissensbank und einer Inferenzmaschine zur Herleitung von Wissen. Die Wissensbank enthält eine Datenbasis mit einer Menge von Informationen, die in einer Wissensrepräsentationssprache formuliert sind (Fakten und deduktive Regeln). Dazu muß das Datenmodell wiederum erweitert werden, diesmal um das Prinzip der Regeln. Zur Überprüfung von benutzerdefinierten Regeln und Konsistenzbedingungen (Constraints) dient die Inferenzmaschine (AW-Konsistenz). Damit ist ein Überprüfen der festgelegten semantischen Beziehungen zwischen Werkzeugen möglich (semantische Konsistenz). Anfragen werden mithilfe von deduktiven Regeln realisiert. Die Inferenzmaschine dient der Auswertung der Anfragen (Herleitung von Wissen: deshalb muß die Sprache eine unterliegende Theorie haben, Korrektheit + Vollständigkeit wird zur Herleitung benötigt). Mithilfe der Möglichkeiten von Anfragen und der Constraint-Sprache können auch hier inkonsistente Stellen aufgelistet werden (AW-Konsistenz und AW-Auswirkungen). Das Trigger-Konzept der Datenbanksysteme bietet die Möglichkeit, automatische Änderungen durchzuführen (AW-Änderungen). Dies wird durch die Inferenzmaschine im Vergleich zu klassischen DBVS stark verbessert.

4.3.5 Zusammenfassende Bewertung

Informationen der realen Welt können beschrieben werden (ARS-Darstellung) und existieren verschiedene formale Datenbeschreibungssprachen (ARS-Formal). Die Formulierung von Konsistenzbedingungen innerhalb und zwischen Sichten sowie die Beschreibung verschiedener Repräsentationssprachen erlaubt das Konzept der Metamodellierung (ARS-Intra-Regeln, ARS-Inter-Regeln, ARS-Sichten). Eine Werkzeugunterstützung bieten die sogenannten Wissensbankverwaltungssysteme. Zur Überprüfung von benutzerdefinierten Regeln und Konsistenzbedingungen (Constraints) dient die Inferenzmaschine (AW-Konsistenz). Mithilfe der Möglichkeiten von Anfragen und der Constraint-Sprache können auch hier inkonsistente Stellen aufgelistet werden (AW-Konsistenz und AW-Auswirkungen). Das Trigger-Konzept der Datenbanksysteme bietet die Möglichkeit, automatische Änderungen durchzuführen (AW-Änderungen).

4.4 Auswahl eines geeigneten Lösungsansatzes

Zur Auswahl eines geeigneten Lösungsansatzes fassen wir die Bewertung der einzelnen betrachteten Ansätze in einer Tabelle zusammen (siehe Tabelle 16 auf Seite 36). Wird eine Anforderung erfüllt, so ist dies mit einem "+" gekennzeichnet, ist sie nicht erfüllt, so ist dies mit einem "-" gekennzeichnet. Wird eine Anforderung nut teilweise erfüllt, so wird dies mit einem "(+)" dargestellt. Erfüllen bei den

Modellierungstechniken/Werkzeuge einige die Anforderungen und einige nicht, so steht in dieser Anforderungszeile "+/-".

Tabelle 16: Bewertung der Ansätze

Anforderungen / Ansätze/Werkzeuge		Klass. Informations-modelle/DBS	Modellierungstech-niken/Werkzeuge	Metamodelle/ WBVS
Repräsentationssprache	ARS-Darstellung	+	+	+
	ARS-Sichten	-	+/-	+
	ARS-Intra-Regeln	+	+/-	+
	ARS-Inter-Regeln	-	+/-	+
	ARS-Formal	+	+/-	+
Werkzeug	AW-Konsistenz	(+)	+/-	+
	AW-Auswirkungen	+	+/-	+
	AW-Änderungen	(+)	-	+

Bei den Datenbanksystemen erfüllen die klassischen Schema-Sprachen nicht ARS-Sichten und ARS-Inter-Regeln, d.h. verschiedene Repräsentationssprachen können nicht beschrieben und miteinander in Beziehung gesetzt werden. Deshalb eignet sich am besten die Metamodellierung im Rahmen eines Wissensbankverwaltungssystems. Diese Systeme haben als Grundlage ein Datenbanksystem und besitzen zusätzlich die Fähigkeit, verschiedene Repräsentationssprachen zu beschreiben und mithilfe von Constraints Beziehungen zwischen den Repräsentationssprachen zu definieren, die dann auch vom WBVS auf den Daten überprüft werden können. Außerdem ist das Triggerkonzept verwirklicht (AW-Änderungen).

Die meisten Beschreibungssprachen der Modellierungstechniken sind nicht geeignet, da sie keine Metamodellierungsfähigkeiten besitzen. Außerdem sind die meisten Modellierungswerkzeuge auf eine Repräsentationssprache beschränkt, und können keine Konsistenzüberprüfungen zwischen Informationen, die in verschiedenen beliebigen Repräsentationssprachen beschrieben sind, durchführen.

Kapitel 5

Lösungsansatz Metamodellierung/WBVS

Das vorangegangene Kapitel charakterisierte den Stand der Forschung in Bezug auf unsere Ziele. Während der Recherche stellte sich heraus, daß weitgehende Ansätze zur Lösung unseres Problems und auch dazugehörende Werkzeuge existieren. Der Ansatz, der alle unsere aus dem Problem und den Zielen abgeleiteten Anforderungen erfüllt und somit als Lösungskonzept ausgewählt wird, ist der Ansatz der Metamodellierung. Zur Realisierung der Metamodellierung existieren wiederum mehrere in Frage kommende mögliche Konzepte, nämlich konzeptuelle Graphen, ER- oder objektorientierte Metamodelle, welche wiederum durch verschiedene konkrete Werkzeuge unterstützt werden. Dieses Kapitel erläutert ausführlich das Konzept der Metamodellierung. Kapitel 6 beschreibt die verschiedenen Realisierungsarten der Metamodellierung und konkrete unterstützende Werkzeuge.

Das Konzept der Metamodellierung wird erläutert, indem wir zuerst eine allgemeine Einführung in die Metamodellierung geben und dann speziell auf die Metamodellierung mehrerer Sichten und die Modellierung von semantischen Beziehungen zwischen diesen Sichten eingehen.

5.1 Metamodellierung von Information

Um das Konzept der Metamodellierung zu erklären, gehen wir zunächst auf die zugrunde liegende Informationsmodellierung ein.

5.1.1 Informationsmodellierung

Seien Informationen der realen Welt gegeben (kognitive Ebene). Modellierung bedeutet hier, allgemeine Konzepte in diesen Informationen und Beziehungen zwischen diesen Konzepten zu identifizieren [HB98] (siehe Bild 17). Dies wird oft auch mit konzeptueller Modellierung bezeichnet.

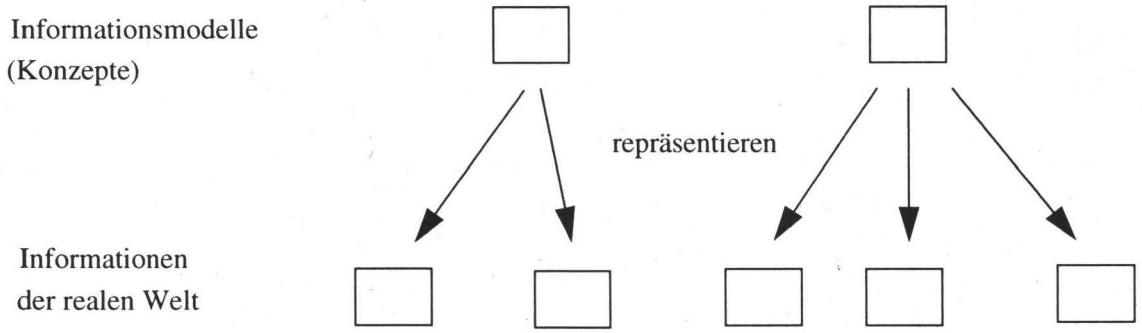

Bild 17: Informationsmodellierung

Die Konzepte können in verschiedenen Repräsentationssprachen dargestellt werden [Met98], z.B. in Form von Entity-Relationship-Diagrammen für relationale Modellierung, oder in Form von Klassen-Hierarchien (Klassendiagrammen) für objektorientierte Modellierung, oder in Form von konzeptuellen Graphen oder semantischen Netzen für graphorientierte Modellierung.

5.1.2 Metamodellierung - Definition

Metamodellierung bedeutet, ein Modell auf einer höheren Abstraktionsebene zu erzeugen als die Modelle, die beschrieben werden sollen [HB98]. Falls also Informationsmodelle vorliegen, werden Modelle von diesen Modellen gemacht (siehe Bild 18) [Atk97]. Metamodellierung wird dazu benutzt, eine Sprache zu definieren, die die Fähigkeit hat, alle Aspekte der Dinge, die modelliert werden sollen, beschreiben (repräsentieren) zu können [NJZ+95]. Dies wird auch oft mit konzeptueller Modellierung bezeichnet. Zur Metamodellierung wird eine Sprache (Repräsentationssprache) benötigt. Eine solche Meta-Sprache ist eine Repräsentationssprache, die genügend ausdrucksstark ist, um Aussagen über die zu modellierenden Elemente zu machen.

Im Falle von ER-Diagrammen muß die Repräsentationssprache Meta-Entities, Meta-Relationships und Meta-Attribute zur Verfügung stellen [Met98]. Im Falle von objektorientierter Modellierung werden objektorientierte Modellierungskonzepte zur Verfügung gestellt ([Atk97], [Atk98]), um die objektorientierte Modellierung selbst zu beschreiben (Meta-Klassen, Meta-Assoziationen, Meta-Attribute [Met98]]). Im Falle von konzeptuellen Graphen, stellt die Repräsentationssprache konzeptuelle "Meta-Graphen" zur Verfügung, d.h. Graphen, mit denen konzeptuelle Graphen beschrieben werden können. Im Falle von semantischen Netzen, müssen semantische Netze existieren, die wiederum semantische Netze beschreiben können.

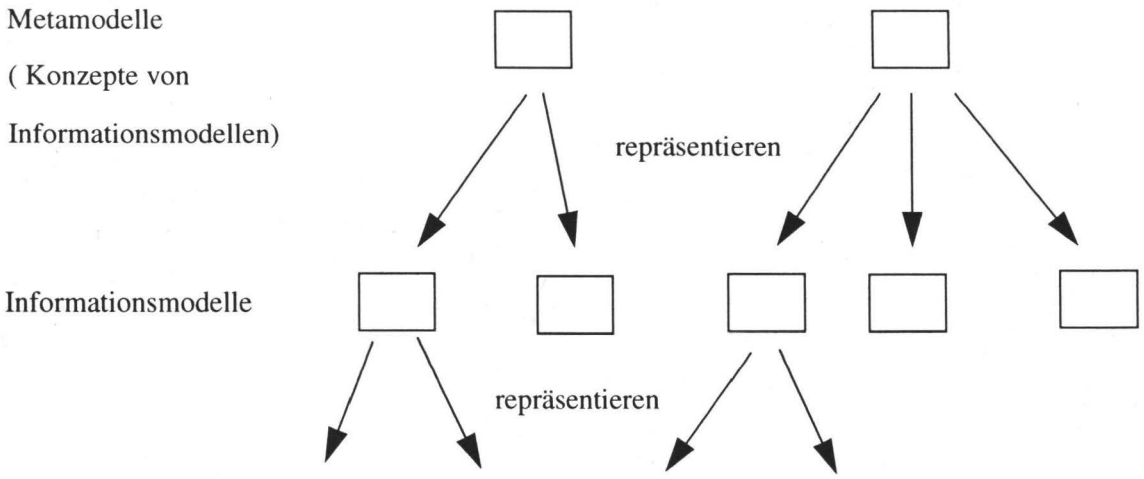

Bild 18: Metamodellierung

Der Begriff "Meta-" sollte nicht absolut, sondern relativ gesehen werden [Atk97]. "Meta-" ist keine Eigenschaft des Modells an sich, sondern eine Eigenschaft der Rolle, die das Modell in Beziehung zu einem anderen Modell spielt [Met98]. Deshalb erscheint es sinnvoll, die Ebenen der Modell-Hierarchie und die Elemente auf den Ebenen zu numerieren (z.B L_1 für Ebene 1), anstatt mit "Meta-", oder "Meta-meta-" zu bezeichnen [Atk97]. Wenn ein $Modell_2$ also das Metamodell eines anderen $Modells_1$ ist (siehe Bild 19 auf Seite 39), so bedeutet dies nur, daß das $Modell_2$ das $Modell_1$ beschreibt, oder anders ausgedrückt, daß das $Modell_2$ eine Repräsentationssprache definiert, in der das $Modell_1$ ausgedrückt werden kann. "Meta-" sagt hier nichts über die absolute Position des $Modells_1$ in der Modell-Hierarchie.

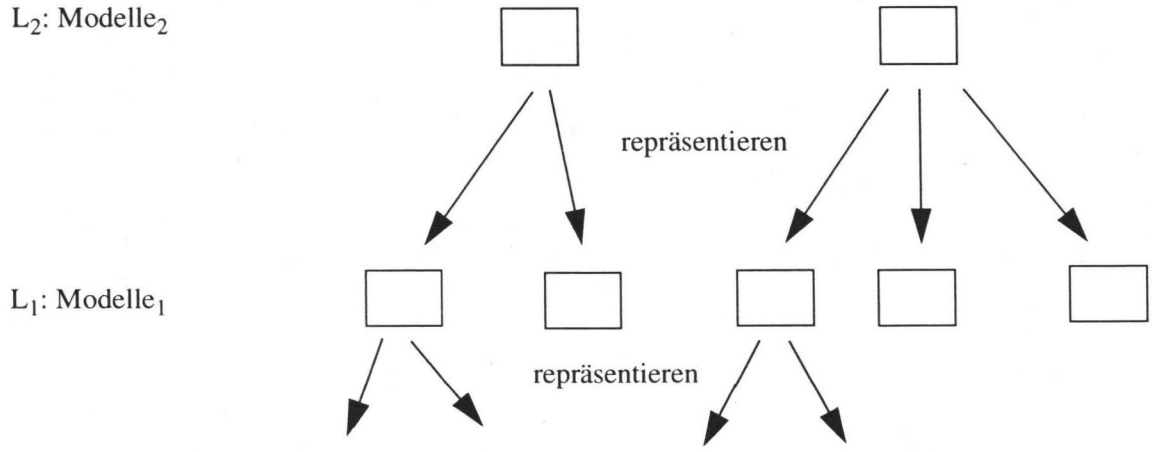

Bild 19: Metamodellierung -Numerierung der Ebenen

Ein allgemein akzeptiertes Organisationsprinzip für Metamodellierung ist die Vier-Ebenen-Architektur, siehe z.B. [Met98] und [Jar97].

5.1.3 Vier-Ebenen-Architektur

Es existiert ein IRDS-Standard (Information Resource Dictionary Systems Standard) für Metamodellierung [ISO90]. Dieser Standard enthält vier Ebenen. L_0 (Application Data) enthält die Informationen der realen Welt, welche mit "(Benutzer-)Daten" bezeichnet werden. L_1 (Level) enthält Modelle, die (Benutzer-)Daten beschreiben. L_2 (Definition Level) enthält (Meta-)Modelle dieser Modelle auf L_1. L_3 (Definition Schema Level) schließt die Modell-Hierarchie ab, indem auf dieser Ebene Elemente zur Verfügung gestellt werden, um das (Meta-)Modell auf L_2 zu beschreiben. Auf L_3 befindet sich also das (Metameta-)Modell der Modelle auf L_1. Es können auf den verschiedenen Ebenen unterschiedliche Repräsentationssprachen verwendet werden, z.B. ER-Diagramme, objekt- oder graphorientierte Repräsentationssprachen. Im folgenden wählen wir eine objektorientierte Repräsentationssprache (z.B. UML) zur Erläuterung des Konzeptes der Metamodellierung.

Im Falle von objektorientierten Repräsentationssprachen befinden sich auf der untersten Ebene L_0 wie bei relationaler Modellierung (Benutzer-)Daten. Auf der Ebene L_1 befinden sich Klassendiagramme, auf L_2 eine Beschreibung von Klassendiagrammen in der Repräsentationssprache von Klassendiagrammen, und auf L_3 Sprachelemente, um die Elemente auf L_2 zu beschreiben (siehe Bild 20 auf Seite 41).

Die unterste Ebene L_0 enthält also Instanzen der Klassen in Ebene L_1. Geht man von L_0 zu L_1 über erhält man Klassen der Instanzen auf L_0. Geht man umgekehrt von L_1 zu L_0 über, erhält man Instanzen der Klassen auf L_1. Dieser Mechanismus wiederholt sich auch für die übrigen Ebenen. Der Übergang von einer oberen Ebene auf eine untere entspricht einer Instanziierung, der Übergang von einer unteren auf eine darüberliegende Ebene entspricht einer Klassifizierung.

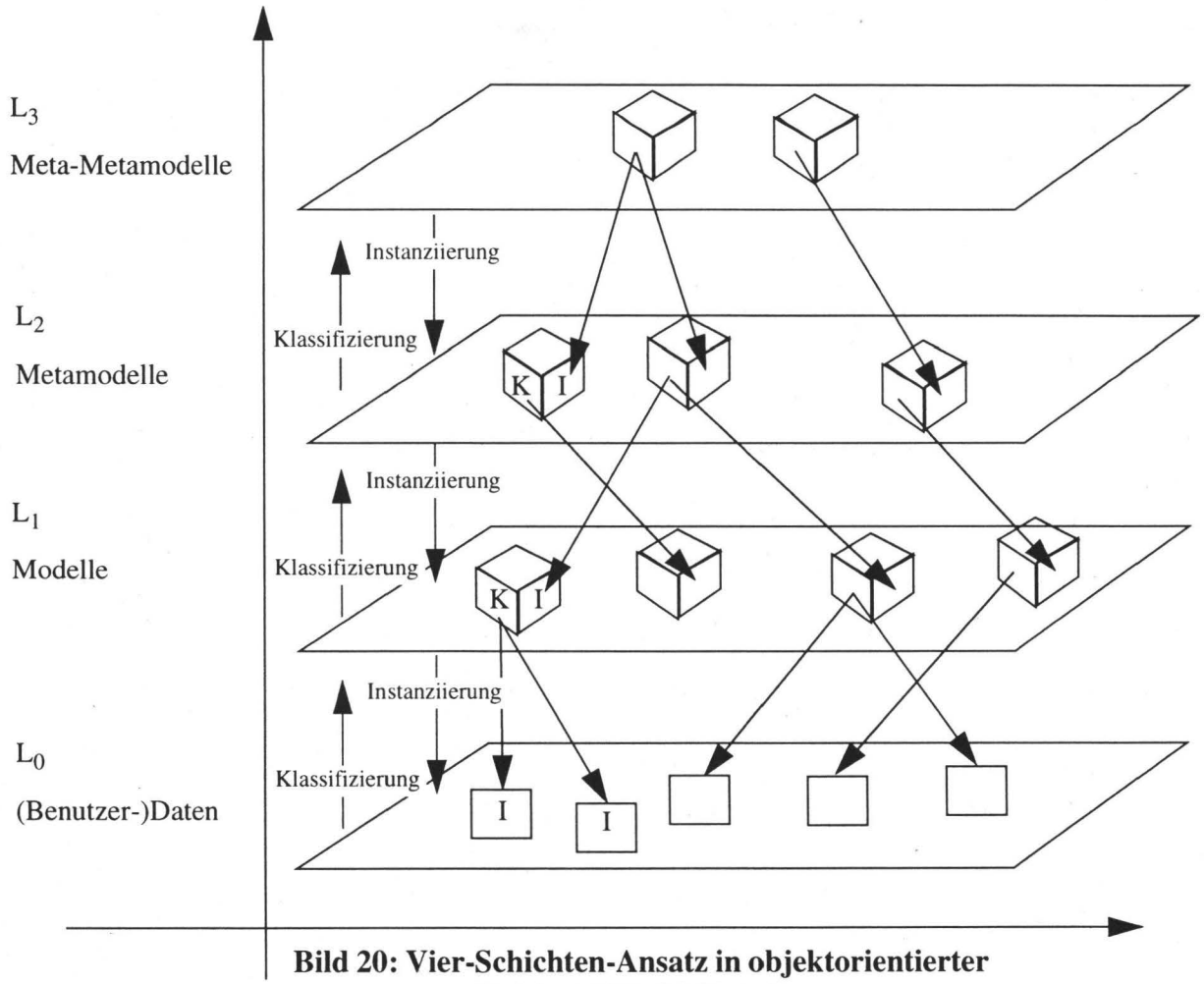

Bild 20: Vier-Schichten-Ansatz in objektorientierter Clabject-Würfel-Notation

Somit haben die Elemente der mittleren Ebenen zwei Facetten, einmal können sie als Klassen (K) für die Elemente der darunterliegenden Ebene, jedoch auch als Instanzen (I) der Elemente der darüberliegenden Ebene betrachtet werden. Deshalb können diese Elemente Clabjects genannt und als Würfel dargestellt werden [Atk98]. Die Elemente der untersten Ebene sind nicht weiter instanziierbar, d.h. sie haben nur die Instanzen-Facette (I), nicht jedoch die Klassen-Facette.

Auch Attribute und Assoziationen werden als Klassen gesehen und sind somit instanziierbar. Nur auf unterster Ebene haben die Attribute Werte und die Assoziationen sind nicht weiter instanziierbare Links.

Beispiel Lichtregelungssystem

Betrachtet man die Sicht Klassendiagramm, so enthält die Ebene L_1 die modellierten Klassen, z.B. die Klasse $Sensor_1$ (siehe Bild 21).

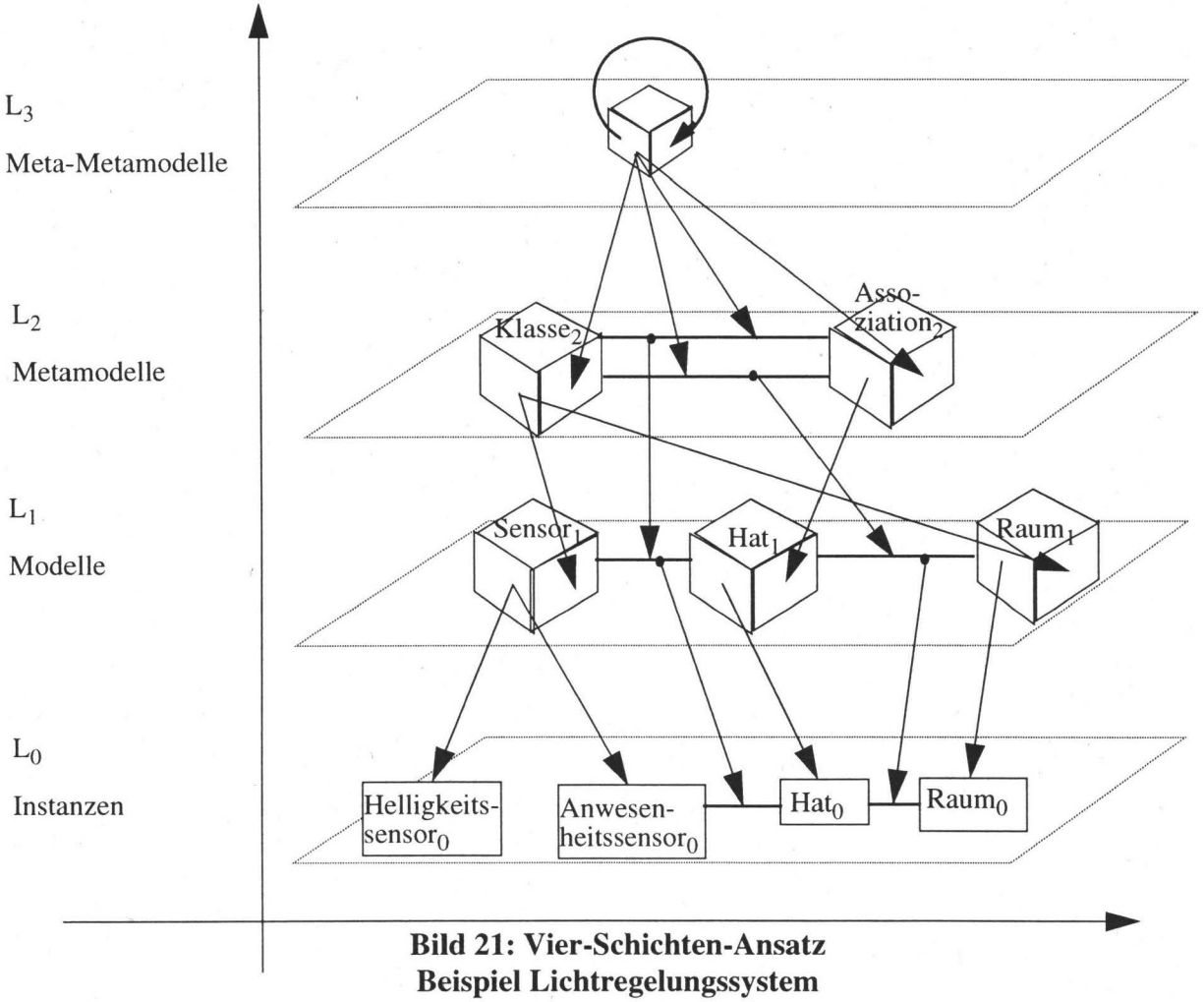

Bild 21: Vier-Schichten-Ansatz
Beispiel Lichtregelungssystem

L_0 enthält Instanzen dieser Klassen, also z.B. $Helligkeitssensor_0$ und $Anwesenheitssensor_0$ als Instanzen von Sensor. Die dafür geeignete Notation ist $Helligkeitssensor_0$: $Sensor_1$, um auszudrükken, daß $Helligkeitssensor_0$ eine Instanz von $Sensor_1$ ist. L_2 enthält Clabjects, die wiederum die Clabjects auf Ebene L_1 beschreiben. Um das Klassendiagramm modellieren zu können, braucht man also das Konzept der Klasse und das Konzept der Assoziation zwischen Klassen. L_2 enthält also ein Clabject $Klasse_2$ und ein Clabject $Assoziation_2$. $Sensor_1$ ist Instanz von $Klasse_2$ ($Sensor_1$:$Klasse_2$) und Hat_1 ist Element von $Assoziation_2$ (Hat_1: $Assoziation_2$). L_3 enthält Elemente, die wiederum die Clabjects $Klasse_2$ und $Assoziation_2$ von Ebene L_2 beschreiben können.

Metamodelle sind dazu geeignet, Konsistenz von Informationsrepräsentationen zu unterstützen. Syntaktische Konsistenz wird durch Typüberprüfung gesichert. Metamodelle können implizit vorhandene semantische Informationen explizit machen, wodurch sich semantische Konsistenz überprüfen läßt.

5.1.4 Explizite semantische Beziehungen innerhalb einer Sicht

Metamodelle sind dazu geeignet, implizit vorhandene semantische Informationen explizit zu machen.

Im Falle von objektorientierten Repräsentationssprachen transportiert die Klassen-Hierarchie semantische Informationen (Vererbung oder Generalisierung), sowie die Konzepte der Aggregation oder der Kardinalitäten bei Assoziationen (z.B. 1:n-Assoziation). Diese Repräsentationssprache stellt also schon in der Graphik viele semantische Informationen zur Verfügung. Natürlich können auch ER-Diagramme, semantische Netze oder konzeptuelle Graphen als Repräsentationssprachen benutzt werden.

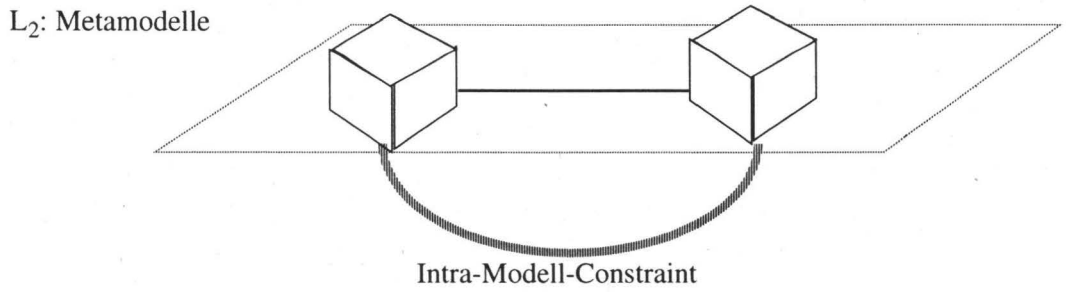

Bild 22: Intra-Modell-Constraint

Die semantische Information wird im Vier-Ebenen-Ansatz auf L_2 explizit dargestellt, wobei es hilfreich ist, das Konzept der Metamodellierung mit dem Konzept von Constraints (siehe Bild 22) zu verbinden, um eine größere Ausdruckskraft zu erreichen. Größere Ausdruckskraft kann auch mithilfe von informellem Text erreicht werden, der die graphischen Darstellungen erklärt. Dies ist jedoch ein semiformaler Ansatz, der nicht geeignet ist, um ein Tool zu erstellen, das automatische Unterstützung bei der Konsistenzprüfung liefert. Constraints sind eine formale Repräsentation der textuellen Erklärungen. Diese Constraints werden im Metamodell definiert (L_2), aber eine Ebene darunter auf der Ebene der Modelle überprüft (L_1).

Beispiel Lichtregelungssystem

Das Metamodell der Sicht Klassendiagramm (siehe Bild 21 auf Seite 42) enthält die Clabjects $Klasse_2$ und $Assoziation_2$. Um die Semantik zwischen diesen beiden Clabjects explizit zu definieren, werden Constraints zwischen ihnen eingeführt. Ein Klasse-Assoziation-$Constraint_2$ könnte z.B. festlegen, daß eine Assoziation immer mindestens mit zwei Klassen verbunden sein muß (siehe Bild 23 auf Seite 43). Dieses Constraint wird also im Metamodell definiert (L_2), aber eine Ebene darunter auf der Ebene der Modelle überprüft (L_1).

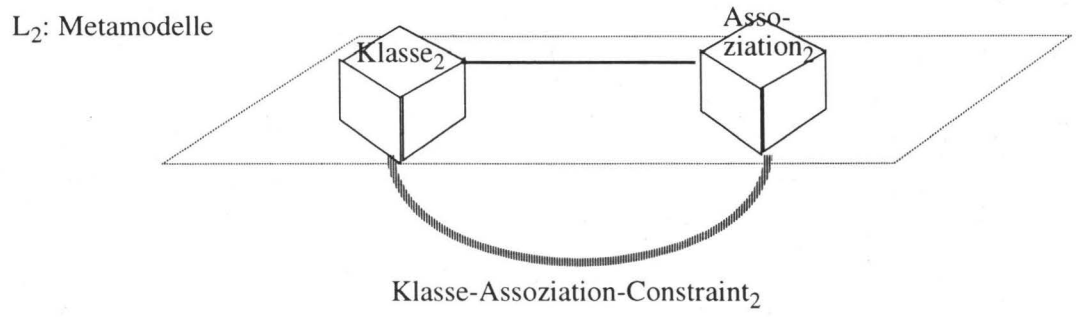

Bild 23: Klasse-Assoziation-Constraint

5.2 Metamodellierung zur Integration mehrerer Sichten

Liegen mehrere Sichten in verschiedenen Repräsentationssprachen (Sprachen) vor, so kann die Integration dieser Sichten mithilfe einer gemeinsamen Repräsentationssprache realisiert werden.

5.2.1 Metamodellierung mehrerer Sichten

Auf der Ebene der Instanzen werden zur Darstellung einer jeden Sicht Objekte, Links und Attribute auf Objekten und Links benötigt. Eine Ebene über der Ebene der Instanzen befindet sich die Ebene der Modelle. Hier werden die Instanzen konzeptuell beschrieben, wozu Klassen und Assoziationen benötigt werden. Für jede Sicht existiert ein Modell, also in Modell A werden die Objekte und Links der ersten Sicht, und in Modell B die Objekte und Links der zweiten Sicht beschrieben (siehe Bild 24 auf Seite 44).

Anhand von Metamodellierung werden zunächst die verschiedenen Repräsentationssprachen der Sichten beschrieben (Metamodell A und Metamodell B). Liegen die verschiedenen Sichten jeweils in der Metamodellierungssprache vor, so können vom Benutzer analoge Konzepte erkannt werden. An Stellen mit analogen Konzepten können die Metamodelle der verschiedenen Sichten integriert werden (siehe Bild 24 auf Seite 44). So bekommt der Benutzer Hinweise zur Änderungsunterstützung, da er die Beziehungen zwischen den Sichten erkennen kann.

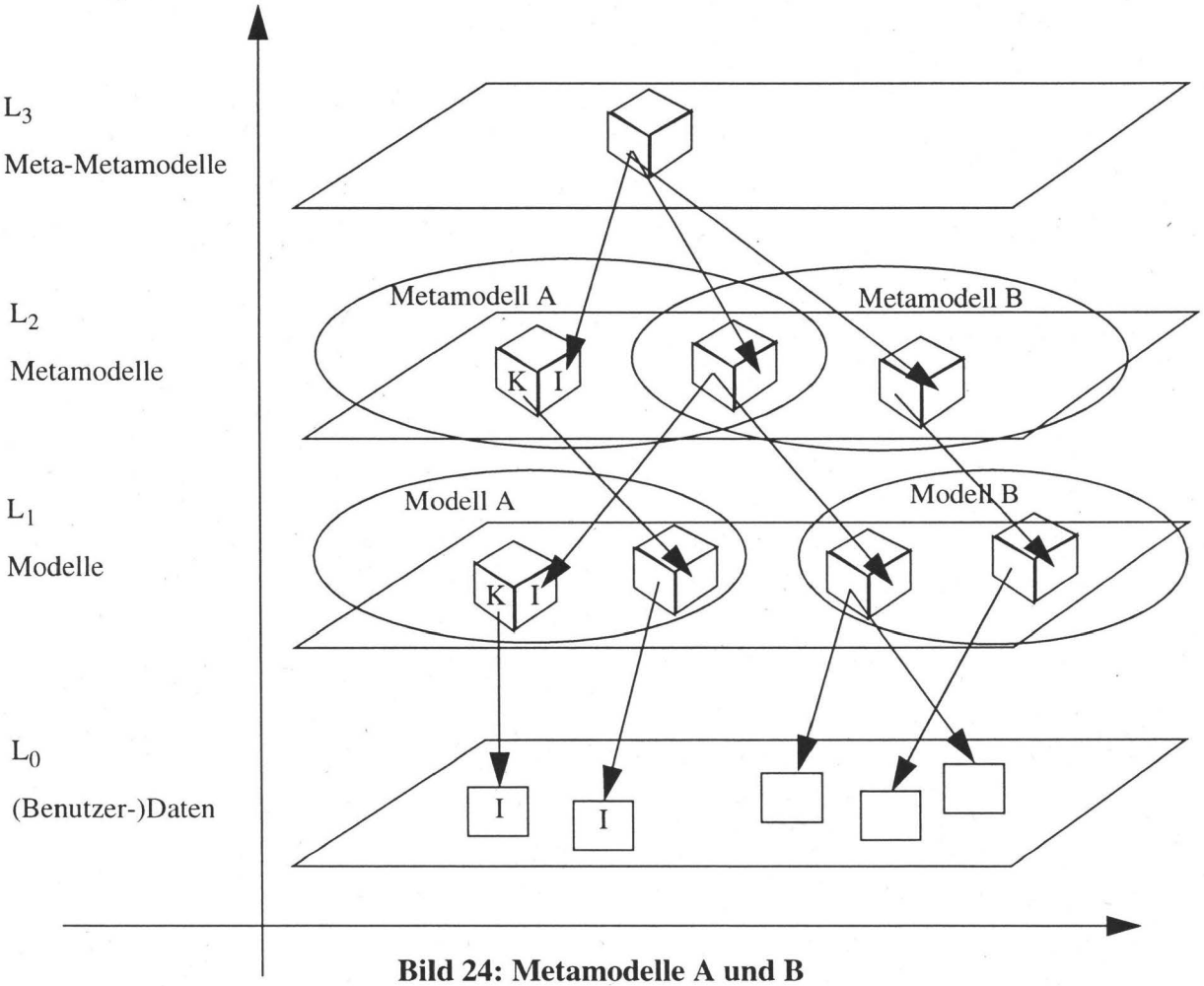

Bild 24: Metamodelle A und B

5.2.2 Explizite semantische Beziehungen zwischen Sichten

Um die Verbindung zwischen Metamodell A und Metamodell B herzustellen, können analoge Konzepte der Metamodelle geteilt werden.

Beispiel Lichtregelungssystem

Metamodell A könnte beispielsweise die Sicht Klassendiagramme modellieren. In dieser Sicht gibt es das Konzept Klasse und das Konzept Methode. Eine Klasse kann mehrere Methoden umfassen (z.B. die Klasse Licht die Methoden anschalten() und ausschalten()). Metamodell B könnte die Sicht Sequenzdiagramme modellieren. In dieser Sicht gibt es ebenfalls die Konzepte Klasse und Methode. Darüber hinaus existiert das Konzept Instanz. Methoden, die eine Instanz einer Klasse geschickt bekommt, müssen identisch sein mit den Methoden der Klasse im Klassendiagramm. Dies kann mit objektorientierten Konzepten ausgedrückt werden.

Eine andere Möglichkeit besteht darin, Constraints zwischen den analogen Konzepten einzuführen, sogenannte Inter-Modell-Constraints (siehe Bild 25 auf Seite 45).

Diese Constraints definieren explizit die semantischen Beziehungen zwischen den einzelnen Metamodellen. Mit Hilfe dieser Constraints entsteht dann auch ein integriertes Gesamt-Metamodell. Diese beiden Methoden können auch gemischt angewandt werden.

L_2: Metamodelle

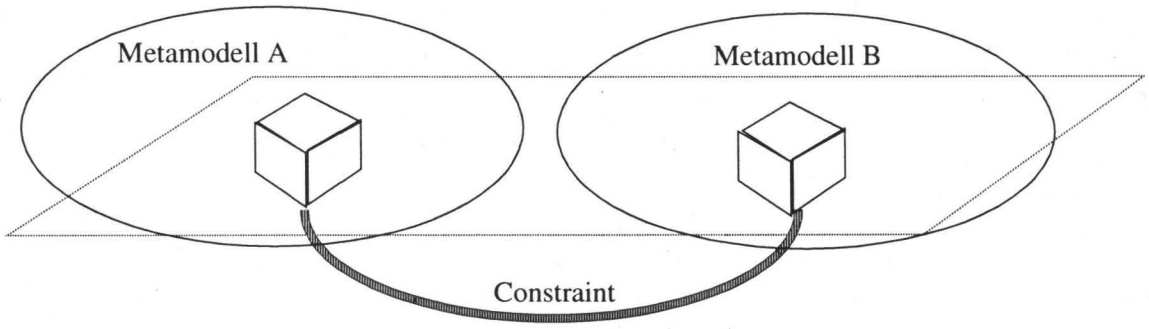

Bild 25: Inter-Modell-Constraint

Beispiel Lichtregelungssystem

Außer dem Metamodell der Sicht Klassendiagramm, werden auch Metamodelle der Sichten Zustandsdiagramm und Sequenzdiagramm erstellt. L_3 enthält dabei immer dieselben Elemente, d.h. die drei Metamodelle liegen in derselben Repräsentationssprache vor, und analoge Konzepte können erkannt werden. Wir demonstrieren das hier für die Sichten Klassendiagramm und Zustandsdiagramm.

Wird die Sicht Zustandsdiagramm betrachtet (siehe Bild 4 auf Seite 9), so werden die Konzepte Zustand und Übergang benötigt. Wir gehen nun von der Würfel-Clabject-Notation wieder zu einer flachen Notation über, um die folgenden Diagramme übersichtlicher zu gestalten.

Im Zustandsdiagramm von Licht wird vom Zustand "Aus" in den Zustand "An" gewechselt, wenn die Methode anschalten() aufgerufen wird (das Event geschickt wird). Dies wird im Metamodell so

ausgedrückt, daß $Übergang_2$ als Attribut "Event" erhält. $Anschalten_1$ hat als Wert von "Event" dann "anschalten()".

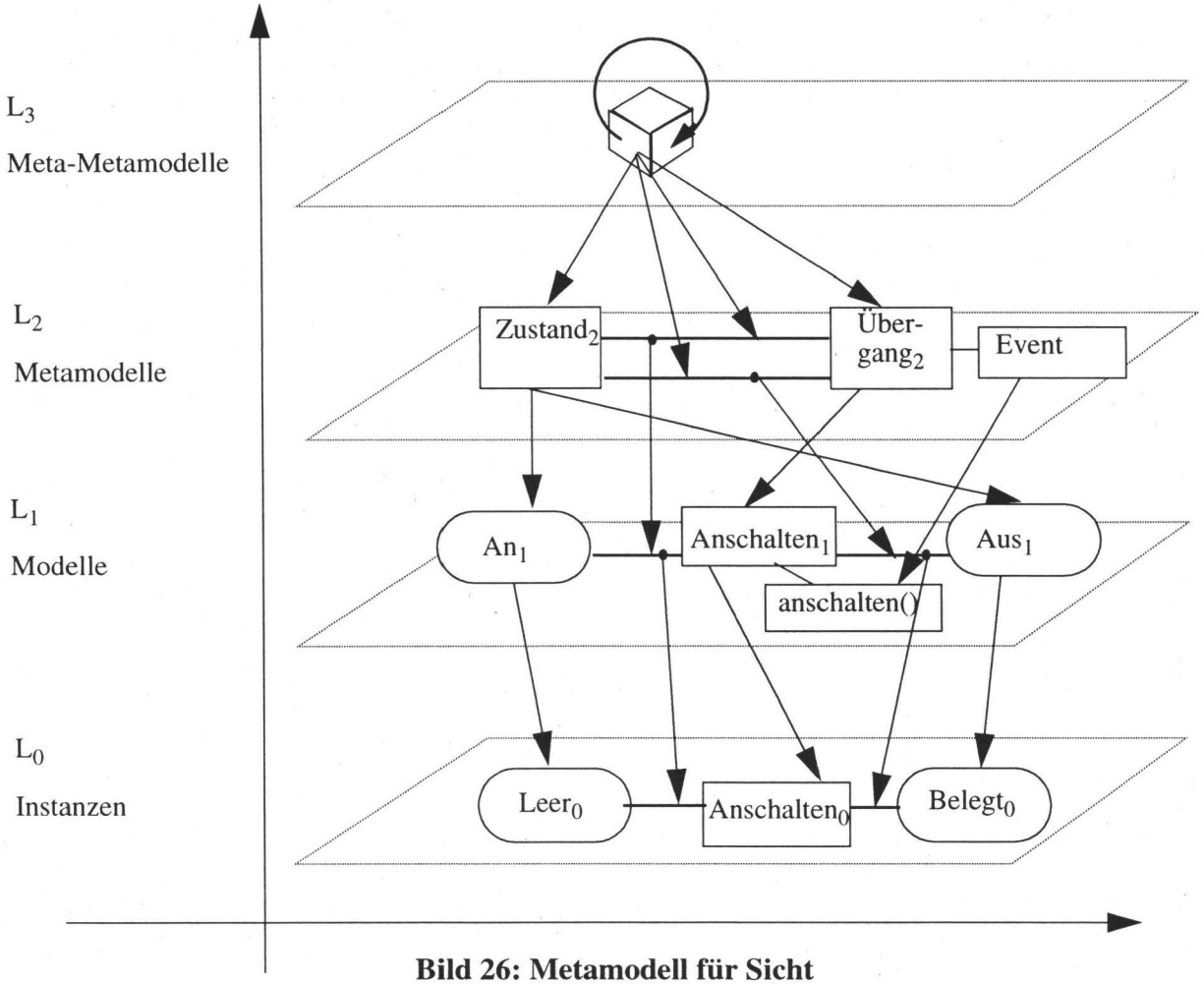

Bild 26: Metamodell für Sicht Zustandsdiagramme

Liegen die beiden Metamodelle vor, d.h Metamodell A für Klassendiagramme und Metamodell B für Zustandsdiagramme, so können sie nun auf Metamodell-Ebene L_2 integriert werden. Da beide Metamodelle in derselben Repräsentationssprache vorliegen, können analoge Konzepte identifiziert werden, z.B. enthalten die Bedingungen im Zustandsdiagramm Methoden aus dem Klassendiagramm.

Die semantische Inter-Sicht-Konsistenz wäre verletzt, falls eine Bedingung eines Zustandsdiagramms auf Ebene L_1 eine Methode enthielte, die in dem zugehörigen Klassendiagramm nicht definiert wäre. Diese semantische Einschränkung kann durch ein Inter-Modell-Constraint "Methode-Übergang-Constraint" explizit ausgedrückt werden (siehe Bild 27 auf Seite 47).

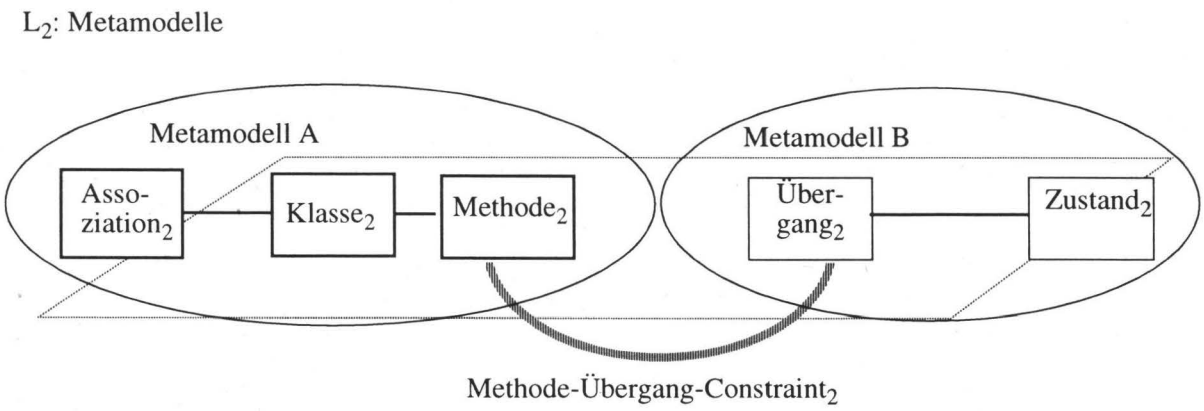

Bild 27: Inter-Modell-Constraint
Beispiel Lichtregelungssystem

5.3 Änderungsunterstützung durch Wissensbankverwaltungssysteme

Anhand dieses integrierten Gesamtmodells ist nun die Änderungsunterstützung möglich. Konsistenzprüfung (AW-Konsistenz) geschieht mithilfe der Constraints und einer Inferenzmaschine, die die Constraints überprüft. Auswirkungsanalyse (AW-Auswirkungen) geschieht mithilfe der Constraint-Überprüfung und Anfragen an die Datenbank. So können die Stellen ausfindig gemacht werden, an denen Änderungen Auswirkungen haben. Automatische Änderungspropagierung (AW-Änderungen) geschieht mithilfe der Auswirkungsanalyse und Triggern, die bei bestimmten Konsistenzverletzungen Änderungsaktionen starten. In einfachen Fällen kann diese Änderungspropagierung automatisch geschehen, in komplizierteren ist die Interaktion mit dem Benutzer notwendig.

Kapitel 6

Werkzeuge zur Metamodellierung

Wie im vorangegangenen Kapitel erwähnt, wird Metamodellierung in verschiedenen Werkzeugen auf unterschiedliche Arten realisiert. In den vorangegangenen Kapiteln haben wir die möglichen Werkzeuge nur allgemein betrachtet. Ab diesem Kapitel wenden wir uns konkreten Werkzeugen zu. Das Ziel dieses Kapitels ist die Auswahl eines Werkzeugs, das Metamodellierung unterstützt, bezogen auf unser Problem und Ziel. Dazu müssen die Werkzeuge anhand der Anforderungen aus Kapitel 4 bewertet werden.

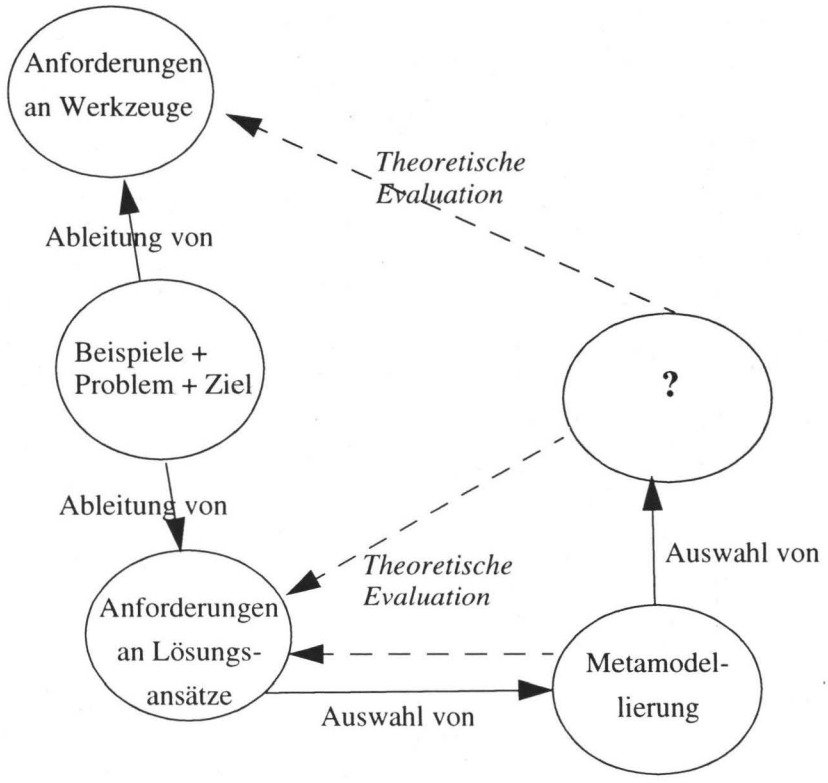

Bild 28: Theoretische Evaluation von Werkzeugen gegen die Anforderungen

Die Anforderungen aus Kapitel 4 werden durch zusätzliche Anforderungen, die auf das konkrete Werkzeug abzielen und eher technischer Art sind (z.B. Anforderungen an die Schnittstelle), erweitert (siehe Bild 28). Im folgenden werden wir zunächst die erweiterten Anforderungen erläutern. Danach werden wir die verschiedenen Realisierungsarten der Metamodellierung in den Werkzeugen beschreiben und

hinsichtlich der Anforderungen bewerten. Am Ende des Kapitels befindet sich eine Tabelle, die die vorgenommenen Bewertungen zusammenfaßt. Die Tabelle ermöglicht, die Auswahl eines für das Problem und das Ziel geeigneten Werkzeugs nachzuvollziehen.

6.1 Anforderungen

Zu den in Kapitel 4 eingeführten Anforderungen (ARS-Darstellung, -Sichten, -Intra- und -Inter-Regeln und AW-Konsistenz, -Auswirkungen, -Änderungen, siehe Tabelle 12 auf Seite 24), kommen nun bei der Betrachtung von konkreten Werkzeugen noch einige Anforderungen hinzu, die technischer Art sind (siehe Tabelle 29 auf Seite 50). Die gegebenen Anforderungen werden zunächst erweitert durch Anforderungen an die Schnittstelle. Es sollte eine graphische Schnittstelle (AW-Graphisch) existieren, da Graphiken leichter verständlich sind als Text, und einige der Software-Artefakte, die betrachtet werden sollen, graphische Notationen besitzen. Die Schnittstelle sollte außerdem sichtenbasiert sein (AW-Sichtenbasiert), damit Ausschnitte der Modelle betrachtet werden können und nicht nur die gesamte Datenmenge. Das Werkzeug sollte leicht erweiterbar sein (AW-Erweiterbar). Falls nicht alle Anforderungen erfüllt werden, kann das Werkzeug dann zukünftig gemäß der Anforderungen angepaßt werden. Abschließend ist es wichtig, daß das Werkzeug gut anwendbar ist (AW-Anwendbar). Mit dem Werkzeug sollte effektiv gearbeitet werden können, insbesondere sollten keine Datenverluste vorkommen. Außerdem sollte das Arbeiten mit dem Werkzeug effizient sein, insbesondere sollten keine Systemabstürze vorkommen, die Interaktion mit dem Werkzeug sollte nicht zu aufwendig, und das Werkzeug sollte nicht zu langsam sein.

Tabelle 29: Erweiterte Anforderungen

Aspekt	Bezeichnung	Erläuterung
Repräsentationssprache	(ARS-Darstellung)	Fähigkeit, die gegebenen Informationen der realen Welt (kognitive Ebene) darzustellen.
	(ARS-Sichten)	Fähigkeit, verschiedene Repräsentationssprachen verschiedener Sichten zu beschreiben.
	(ARS-Intra-Regeln)	Fähigkeit, Verfolgbarkeit innerhalb von Informationsrepräsentationen, d. h. in einer Sicht, herzustellen durch explizite Intra-Sicht-Beziehungen + Syntaxregeln.
	(ARS-Inter-Regeln)	Fähigkeit, Verfolgbarkeit zwischen Sichten herzustellen durch explizite Inter-Sicht-Beziehungen.
	(ARS-Formal)	Formale Basis der Repräsentationssprache (wird zur Ausführbarkeit benötigt).

Tabelle 29: Erweiterte Anforderungen

Aspekt	Bezeichnung	Erläuterung
Werkzeug	Änderungsunterstützung durch:	
	(AW-Konsistenz)	Konsistenzprüfung
	(AW-Auswirkungen)	Auswirkungsanalyse
	(AW-Änderungen)	Automatische Änderungspropagierung
	(AW-Graphisch)	Graphische Schnittstelle
	(AW-Sichtenbasiert)	Sichtenbasierte Schnittstelle
	(AW-Erweiterbar)	Erweiterbarkeit
	(AW-Anwendbar)	Anwendbarkeit

6.2 Realisierungen durch Werkzeuge

Metamodellierung kann auf verschiedene Arten stattfinden, durch erweiterte Informationsmodelle wie objektorientierte Metamodelle oder konzeptuelle Graphen. Diese Arten der Wissensmodellierung benutzen unterschiedliche Konzepte, um Regeln oder Constraints festzulegen, z.B. Prädikatenlogik oder Graphersetzungsregeln.

6.2.1 Datenbeschreibungssprache: Konzeptuelle Graphen

Zur Darstellung von Informationen in der Informatik sind graphische Strukturen sehr gut geeignet, z.B. Zustandsübergangs- oder Datenflußdiagramme. Diese Strukturen erinnern an Graphen aus der Mathematik. Es existieren Knoten und Kanten, wobei sowohl die Knoten als auch die Kanten mit Attributen versehen werden können. Konzeptuelle Graphen sind eine abstrakte Repräsentation von Logik mittels zweier Arten von Knoten, sogenannten "Konzepten" und "konzeptuellen Relationen", die durch Pfeile verbunden sind [CG98]. Die Konzepte sind in Konzept-Hierarchien und die Relationen in Relationen-Hierarchien angeordnet (Partialordnungen).

GRAS

Das Werkzeug GRAS (Graphorientiertes Datenbanksystem) benutzt konzeptuelle Graphen als Informationsmodell. GRAS ([SW92], [KSW92], [Lef94], [NS91], [NS96]) besteht aus einer Datenbank zum Speichern von Graphen und einem Graphersetzungssystem PROGRES (PRogrammiertes GRaphERsetzungssystem) ([Sch91], [Sch94], [Sch96], [SWZ96]). Das Datenmodell, das GRAS benutzt, sind gerichtete attributierte kanten- und knotenmarkierte Graphen. Daraus leitet sich die Datenbeschreibungssprache PROGRES ab. Attributierte Graphgrammatiken dienen hier als formale Basis.

Liegen Informationen der kognitiven Ebene vor, so werden zunächst konzeptuelle Graphen für jede einzelne Repräsentationssprache entworfen, d.h. Typen (Konzepte) von Knoten und Kanten und Beziehung zwischen ihnen werden identifiziert. Diese werden definiert und somit können die Informationen der realen Welt beschrieben werden (ARS-Darstellung).

Mit PROGRES können Konsistenzbedingungen und Attributberechnungsregeln in Form von Graphschemata (in Anlehnung an Schemata der DBS) definiert werden. Dadurch können primitive (Attribut-) Datentypen festgelegt werden und die Prinzipien der Klassifizierung und Vererbung beschrieben werden. Man erhält Typen von Knotentypen, d.h. Typen 2. Ordnung. PROGRES dient dann zur Beschreibung von Schema-treuen Graphen (Syntaxregeln, Integritätsregeln).

Außerdem dient PROGRES zur Beschreibung von Graph-Ersetzungsregeln (prädikatenlogischer Ableitungsbegriff). Die Theorie darunter ist ein logikbasierter Graphersetzungskalkül (Graphen = Menge von prädikatenlogischen Formeln). Dies ist eine Prädikatenlogik erster Stufe mit Gleichheit, es gibt auch Graphgrammatiken, die mengentheoretisch oder kategorientheoretisch sind.

Konzeptuelle Graphen basieren auf der mathematischen Theorie der Graphgrammatiken [CG98]. Zur formalen Beschreibung (ARS-Formal) solcher Strukturen eignen sich attributierte Graphgrammatiken [SWZ95].

Integration von verschiedenen Repräsentationssprachen geschieht durch paarweises Koppeln von Graphgrammatiken. PROGRES [SWZ95] dient als gemeinsame Metamodellierungs-, d.h. Repräsentationssprache. Liegen verschiedene Repräsentationssprachen vor, so werden zunächst konzeptuelle Graphen für jede einzelne Repräsentationssprache entworfen, d.h. Typen (Konzepte) von Knoten und Kanten werden identifiziert (ARS-Sichten) und miteinander in Beziehung gesetzt (ARS-Intra-Regeln). Die konzeptuellen Graphen der einzelnen Repräsentationssprachen können dann durch Verbindungen integriert werden (ARS-Inter-Regeln).

Konsistenzprüfungen (AW-Konsistenz) werden durch Test-Konsistenzanfragen (Vervollständigungsoperator zum Abschluß der Theorie) realisiert. Test-Anfragen, suchen Teilgraphen (regelorientiert) und liefern die Antwort, ob ein passender Teilgraph (Substitution der Knotenvariablen) gefunden worden ist oder nicht. Auswirkungsanalysen (AW-Auswirkungen) sind dadurch nicht realisierbar, da keine Liste mit gefundenen inkonsistenten Graphen zurückgegeben wird. Graph-Ersetzungen dienen als ECA-Regeln, mit denen nur schematreue Graphen erzeugt und verändert werden können (AW-Änderungen). Diese Graph-Ersetzungen können Produktionen (atomar, nichtdeterministisch) oder auch Transaktionen (imperativ) sein. GRAS ist eher auf automatische Ersetzungen ausgelegt und nicht auf Konsistenzprüfung.

Es gibt eine graphische und sichtenbasierte Schnittstelle (AW-Graphisch und AW-Sichtenbasiert), die jedoch nur lesbar ist, d.h. die Graphen können auf der Oberfläche inspiziert werden, aber es können keine Änderungen an den Graphen gemacht werden. Die Anwendbarkeit (AW-Anwendbar) ist nicht gut, da die Graphen-Datenbank langsam ist.

KOGGE

Das Werkzeug KOGGE (KOblenzer Generator für Graphische Entwurfsumgebungen, MetaCASE-Werkzeug) benutzt als Informationsmodell EER/GRAL, d.h. erweiterte ER-Diagramme (EER) und eine Graph-Spec-Language (GRAL), was eine formale Z-ähnliche Sprache zur Definition einschränkender Bedingungen (Constraints, AW-Änderungen) auf Graphen ist ([EC94], [EWD+96], [ES97]). EER-Diagramme sind graphbasiert, und sind in Z definiert (Prädikatenlogik, ARS-Formal). Die zur internen Darstellung benutzte Datenbeschreibungssprache basiert auf TGraphen, welche attributierte angeordnete (gerichtete) Graphen sind.

Meta-Entities sind darstellbar, da das Informationsmodell das Prinzip der Metamodellierung enthält (ARS-Sichten). Die zugrundeliegende Datenbank ist GraLab (GraphenLabor). Es existieren G2QL (graphische) und GReQL (textuelle Anfragesprache), um Anfragen an die Datenbank zu stellen (AW-Auswirkungen). Constraintprüfung ist möglich (AW-Konsistenz). Statecharts beschreiben die gewünschte Funktionalität des Werkzeugs. Die EER-Diagramme und die Statecharts werden in den KOGGE-Interpreter eingegeben (graphisch (AW-Graphisch) und sichtenbasiert (AW-Sichtenbasiert) und das gewünschte CASE-Werkzeug wird erzeugt. Dies funktioniert jedoch noch nicht automatisch (AW-Anwendbar), weshalb KOGGE für uns nicht in Frage kommt. Würde es funktionieren, so könnte jede gewünschte Anforderung an das Werkzeug erfüllt werden (auch AW-Änderungen).

6.2.2 Datenbeschreibungssprache: Objektorientierte Metamodelle

Objektorientierte Metamodelle werden in objektorientierten Datenbanken gespeichert. Wissensmanagementsysteme, die mit Metamodellen arbeiten, verwenden eine spezielle Constraint-Sprache, die Einschränkungen zwischen den verschiedenen Elementen festlegt.

ConceptBase

Das Informationsmodell des Werkzeugs ConceptBase ist eine Erweiterung von früheren Informationsmodellen wie ER-Diagrammen, objektorientierten Diagrammen oder semantischen Netzen [Jar97]. Erweiterungen sind das Prinzip der Zeitlogik, prädikatenlogische Formeln und Regeln, und das Prinzip der Metamodellierung [EJJ+89]. ConceptBase ist ein Wissensbankverwaltungssystem mit einer objektorientierten Datenbank.

Die Datenbeschreibungssprache für ConceptBase ist O-Telos [MBJ+90], eine prädikatenlogische Sprache 1. Ordnung. Die formale Basis ist also die Prädikatenlogik. Durch Formeln ist die Definition von Constraints möglich [NJZ+95]. Deduktive Regeln (Hornlogik vgl. PROLOG) dienen zum logisches Schließen. Dazu wird eine Beweistheorie benötigt. ConceptBase benutzt historische Theorien, die aus Axiomen und Funktionen bestehen.

Informationen der realen Welt sind darstellbar durch Klassen, Attribute und Assoziationen (ARS-Darstellung).

Metamodelle (eine Sicht) sind darstellbar (ARS-Sichten), da das Prinzip der Metamodellierung in ConceptBase zur Verfügung gestellt wird, wie auch die Prinzipien der Instanziierung, Vererbung und Aggregation. Die Darstellung integrierter Metamodelle (mehrere Sichten) ist möglich durch Constraints auf Klassen, Assoziationen und Attributen (ARS-Sichten + Inter-Regeln). Constraints sind Instanzen einer Klasse "Assertion", prädikatenlogische Formeln erster Ordnung.

Überprüfung der Constraints auf Instanzen ist möglich (AW-Konsistenz). Es erfolgt eine Fehlermeldung bei Verletzung der Constraints beim Ändern von Instanzen. Betroffene Instanzen werden zurückgeliefert mit Begründung (verletztes Constraint). Anfragen an die DB sind Instanzen einer Klasse "Query" (query) und werden realisiert durch deduktive Regeln. Regeln sind auch Instanzen einer Klasse "Assertion" (rule) und beruhen auf Hornlogik (PROLOG). Dadurch kann die Auswirkungsanalyse realisiert werden (AW-Auswirkungen). Eine automatische Anpassung an einen neuen Datenbank-

Zustand, so daß alle Constraints wieder gelten, kann durch aktive ECA-Regeln realisiert werden (AW-Änderungen).

Die Schnittstelle steht graphisch und sichtenbasiert zur Verfügung (AW-Graphisch und AW-Sichtenbasiert). Der Graphik-Browser zeigt Super- und Subklassen, Instanzen und Klassen. Das Werkzeug kann problemlos durch Definition von Metamodellen erweitert werden (AW-Erweiterbar). Außerdem können eigene Client-Programme über eine Programm-Schnittstelle angebunden werden. Die Anwendbarkeit ist gut, es gehen keine Daten verloren und das System ist schnell (AW-Anwendbar).

CMSL/TCM

CMSL ist eine konzeptuelle Modellierungssprache für Spezifikationen (Conceptual Model Specification Language). Sie ist objektorientiert, und mit den objektorientierten Strukturierungsprinzipien (Klasse, Attribute, usw.) können Informationen der realen Welt beschrieben werden (ARS-Darstellung). Sie besitzt auch das Metamodellierungskonzept, weshalb Repräsentationssprachen auch beschrieben werden können (ARS-Sichten). Mithilfe von Constaints (systemgesetzte und benutzerdefinierte) können semantische Beziehungen sowohl innerhalb, als auch zwischen Sichten definiert werden (ARS-Intra- + Inter-Regeln). Die formale Grundlage ist Prädikatenlogik (ARS-Formal). CMSL/TCM unterstützt systemgesetzte, jedoch noch nicht benutzerdefinierte Constaintprüfungen über Sichten hinweg (AW-Konsistenz). Integriertes Metamodellieren ist also nicht möglich [Wie97]. Somit sind AW-Auswirkungen und AW-Änderungen auch nicht erfüllt. Die Fragen der Anwendbarkeit (AW-Anwendbar) und Erweiterbarkeit (AW-Erweiterbar) sind unklar. Da CMSL/TCM aber aufgrund der Nichterfüllung von AW-Konsistenz, Auswirkungen, Änderungen nicht als unterstützendes Werkzeug geeignet ist, ist die Frage nach der Anwendbarkeit und Erweiterbarkeit auch nicht mehr relevant.

6.3 Auswahl eines geeigneten Werkzeugs

Um ein geeignetes Werkzeug auszuwählen, betrachten wir zusammenfassend die Bewertungen der einzelnen in Frage kommenden Werkzeuge. Dabei bedeutet ein "?" in der Tabelle, daß es unklar ist, ob die Anforderung erfüllt wird oder nicht. Dies ist an dieser Stelle dann aber nicht mehr wichtig gewesen für die Bewertung des Werkzeugs. Es stellt sich heraus, daß O-Telos/ConceptBase die meisten Anforderungen erfüllt.

Tabelle 30: Bewertung der Werkzeuge

Werkzeuge / Anforderungen	Konzeptuelle Graphen		OO-Metamodelle	
	PROGRES/ GRAS	O-Telos/ ConceptBase	CMSL/ TCM	EER-GRAL/ KOGGE
(ARS-Darstellung)	+	+	+	+
(ARS-Sichten)	+	+	+	+
(ARS-Intra-Regeln)	+	+	+	+

Tabelle 30: Bewertung der Werkzeuge

(ARS-Inter-Regeln)	+	+	+	+
(ARS-Formal)	+	+	+	+
(AW-Konsistenz)	+	+	-	+
(AW-Auswirkungen)	-	+	-	+
(AW-Änderungen)	+	+	-	(+)
(AW-Graphisch)	+/-	+/-	+	+
(AW-Sichtenbasiert)	+	+	+	+
(AW-Erweiterbar)	?	+	?	+
(AW-Anwendbar)	-	+	?	-

PROGRES/GRAS ist eher auf automatische Transformationen fokussiert als auf Konsistenzprüfung und das System ist zu langsam, was auf die Graph-Datenbank zurückzuführen ist. Auf der graphischen Schnittstelle können jedoch nur die Graphen dargestellt werden (Ausgabe), jedoch keine Änderungen durchgeführt werden (Eingabe). Wir teilen den Eintrag bei AW-Graphisch deshalb in Ausgabe/Eingabe auf, um hier eine getrennte Bewertung vornehmen zu können.

CMSL/TCM erfüllt alle Anforderungen an die Sprache, das System kann jedoch noch keine benutzerdefinierten Constraints überprüfen, weshalb es nicht geeignet ist. Die Entwickler arbeiten jedoch daran, Constraintprüfungen möglich zu machen.

KOGGE erfüllt an sich alle Anforderungen, scheitert jedoch an der Anwendbarkeit des Systems. Die Bedienung ist zu kompliziert, und das automatische Erstellen von CASE-Tools funktioniert noch nicht voll automatisch, wie uns die Entwickler mitgeteilt haben. Dadurch wird dann auch AW-Änderungen (automatische Änderungspropagierung) nicht erfüllt. Dieser Ansatz sollte jedoch im Auge behalten werden, da KOGGE auch ein geeignetes Werkzeug für unser Problem und Ziel wäre, wenn es funktionieren würde.

ConceptBase erfüllt alle Anforderungen, die wir an ein geeignetes Werkzeug gestellt haben, der einzige Kritikpunkt ist, daß auf der graphischen Oberfläche keine Eingaben gemacht werden können. Dies ist macht jedoch nur Abstriche an der Komfortabilität des Systems. Deshalb wählen wir O-Telos/ConceptBase aus.

Kapitel 7

Evaluation von O-Telos/ConceptBase

Die Bewertung und Auswahl von O-Telos/ConceptBase wurde größtenteils aufgrund der Literaturrecherche vorgenommen (bis auf kleine Tests zur Anwendbarkeit). Im diesem Kapitel wird O-Telos/ConceptBase anhand zweier Fallstudien, die auf den Beispielen aufbauen, praktisch evaluiert, um zu erkennen, ob die Bewertung anhand der Literaturrecherche korrekt ist.

Wir führen zur praktischen Evaluation zunächst zwei Fallstudien A und B ein. Anschließend evaluieren wir O-Telos/ConceptBase gegen die Anforderungen aus Kapitel 6 und gegen die Fallstudien (siehe Bild 31).

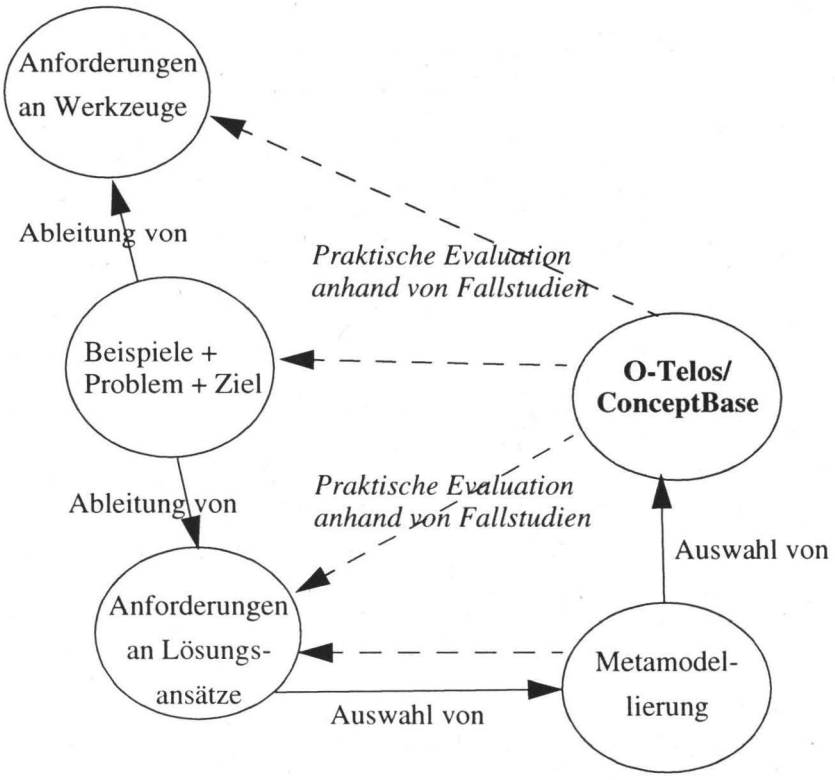

Bild 31: Praktische Evaluation von O-Telos/Concept-Base

Einführung der Fallstudien

Die Fallstudien A und B beziehen sich auf die beiden Beispiele "Entwicklungsaufwand" und "Lichtregelungssystem", die sich durch die gesamte Arbeit hindurchziehen. Beide Beispiele werden wir nun im Rahmen der Fallstudien ausführlich diskutieren.

Die Fallstudie A behandelt den Software Engineering Bereich "Meßplanung mit GQM". Sie bezieht sich auf das Beispiel "Entwicklungsaufwand" und ist in Anhang A zusammengefaßt beschrieben.

Die Fallstudie B behandelt den Software Engineering Bereich "Objektorientierter Entwurf mit UML". Sie bezieht sich auf das Beispiel "Lichtregelungssystem" und ist in Anhang C zusammengefaßt beschrieben.

Die Software-Artefakte und die Sichten innerhalb der Artefakte, die in den beiden Fallstudien betrachtet werden, sind sehr unterschiedlicher Art. In Fallstudie A wird als Artefakt "umfassende Projektpläne" und als Sichten GQM-Pläne, Projektpläne und Qualitätsmodelle betrachtet. In der Fallstudie B wird als Artefakt "UML-Entwurfsdokumente" und als Sichten Klassendiagramme, Sequenzdiagramme und Zustandsdiagramme auf ein Softwaresystem betrachtet.

Das Ziel der Fallstudien ist die Evaluation von ConceptBase anhand der Fallstudien bezüglich der Anforderungen. Dazu wird in Fallstudie A ein Ausschnitt aus dem integrierten Projektplan- und GQM-Plan-Metamodell modelliert, während in Fallstudie B ein Ausschnitt aus dem integrierten UML-Metamodell modelliert wird.

Ein Nebenziel, daß sich dabei bei Fallstudie A ergibt, ist ein Beitrag zur Weiterentwicklung des konzeptuellen Modells (Metamodells) der GQM-Pläne, das noch nicht fest definiert ist. Dies wird bei Fallstudie B nicht angestrebt, es wird das definierte Metamodell von UML inklusive Constraints (OCL) modelliert (Rational, open metamodel).

Evaluation von ConceptBase anhand der Anforderungen

Wir führen die Evaluation von ConceptBase anhand der Anforderungen aus Kapitel 6 jeweils für beide Fallstudien durch (ARS-Darstellung, -Sichten, -Intra- und -Inter-Regeln, AW-Konsistenz, -Auswirkungen, -Änderungen und AW-Graphisch, -Sichtenbasiert, -Erweiterbar, -Anwendbar, siehe Tabelle 29 auf Seite 50). Die Anforderungen unterteilen sich in Anforderungen an die zugrundeliegende Repräsentationssprache (ARS-Darstellung, -Sichten, -Intra- und -Inter-Regeln) und Anforderungen an das Werkzeug (AW-Konsistenz, -Auswirkungen, -Änderungen, -Graphisch, -Sichtenbasiert, -Erweiterbar, -Anwendbar). Beim Werkzeug ConceptBase ist die zugrundeliegende Repräsentationssprache O-Telos. Wir beginnen mit der Evaluation anhand der Anforderungen an die Repräsentationssprache und behandeln anschließend die Anforderungen an das Werkzeug. Der Titel jedes Abschnitts gibt die jeweils behandelte Anforderung an und die Elemente von O-Telos/ConceptBase, die zur Erfüllung dieser Anforderung dienen. Innerhalb der Abschnitte werden wir jeweils beide Fallstudien anhand der gerade behandelten Anforderung betrachten.

7.1 ARS-Formal: Prädikatenlogik

Da in der Zielsetzung Werkzeugunterstützung gefordert wird, muß die Repräsentationssprache eine formale Basis haben. Das Werkzeug ConceptBase ist ein deduktives objektorientiertes Wissensverwaltungssystem für Meta-Datenbanken und implementiert O-Telos einen Dialekt der Sprache Telos (griechisch: Ziel, Absicht). O-Telos ist eine Wissensrepräsentationssprache, die einen axiomatisierten strukturellen objektorientierten Kern mit einer prädikatenlogischen Sprache erster Ordnung mit Sorten (Typen) und mit der Allen'schen Zeitlogik [Ric92] integriert [EJJ+89].

Die Theorie, die der Sprache zugrundeliegt ist eine Prädikatenlogik erster Stufe. Wir werden sie hier einführend erläutern und jeweils nachfolgend bei der Betrachtung der weiteren Anforderungen anhand der Fallstudien vertiefen. Die objektorientierten Strukturierungsprinzipien wie Instanziierung, Vererbung und Aggregation werden durch eine Menge von Axiomen (prädikatenlogische Formeln) realisiert, die fest in ConceptBase implementiert sind. Die Zusicherung dieser Strukturierungsprinzipien basiert auf DATALOG, einer Logik, in der definiert werden kann, welche Aussagen das System als wahr betrachtet. In prädikatenlogischen Formeln können als Sorten (Typen) alle Objekte, die in die Datenbank eingegeben wurden, benutzt werden.

In O-Telos werden Klassen, Beziehungen, Attribute, usw. uniform als "Propositionen" (Behauptungen, Prädikat P) behandelt. Propositionen können instanziiert werden, d.h. sowohl Klassen, Attribute wie auch Beziehungen können instanziiert werden.

Propositionen werden intern im System (ConceptBase) jeweils in folgender Form angegeben [KMS+89]:

P (id, x, l, y, t)

Dies bedeutet, daß die Proposition x eine Beziehung l zur Proposition y zur Zeit t hat [JJQ98]. Der Identifier (id) der Proposition wird jeweils vom System generiert und die Identifier werden im folgenden immer mit vorangestelltem "#" gekennzeichnet.

Die Quellproposition (source) ist die Proposition x, die Benennung (label) der Proposition ist l, die Zielproposition (destination) ist y und t ist die Belief-Zeit (duration), die angibt, in welchem Zeitintervall die Proposition dem System bekannt ist (Zeitlogik).

Klassen, die keine Attribute sind, werden mit "Individuals" bezeichnet und haben als Benennung ihren Klassennamen und als Quelle und Ziel sich selbst.

P (#id, #id, KlassenName, #id, t)

Attribute werden auch als Klassen behandelt, die Propositionen mit anderen Propositionen in Beziehung setzen. Die Benennung der Attribut-Propositionen ist der Attributname.

P (#id, q, AttributName, z, t)

Die Quellproposition q ist die Klasse, zu der das Attribut gehört. Die Zielproposition z ist die Klasse, von der der Attributwert eine Instanz sein muß, sozusagen der Attributtyp des Attributs.

7.2 ARS-Darstellung: Sprachkonzepte von O-Telos

Mithilfe der Repräsentationssprache O-Telos ist es möglich, Informationen der realen Welt darzustellen. Um Informationen zu beschreiben werden Klassen, Attribute und Assoziationen benötigt. Anhand der beiden Fallstudien werden wir jeweils beispielhaft zeigen, wie in O-Telos diese drei Konzepte realisiert werden. Dabei werden wir im folgenden die Belief-Intervalle der Zeitlogik weglassen, da sie in Bezug auf die Fallstudien unwichtig sind, und eine Erläuterung der Zeitlogik den Rahmen dieser Arbeit sprengen würde.

7.2.1 Fallstudie A: Entwicklungsaufwand

Informationen der realen Welt sind in dieser Fallstudie z.B. der GQMPlan *Aufwand-Entwicklungsprozeß*, das Ziel *Entwicklungsaufwand-Ziel* oder die Frage *Q1*, und die Beziehungen zwischen diesen Elementen. Innerhalb des GQM-Plans *Aufwand-Entwicklungsprozeß* (*EffortDevelopmentProcess-GQMPlan*) existiert das Ziel *Entwicklungsaufwand-Ziel* (siehe Anhang A). Dies wird in O-Telos durch eine Instanz *EffortDevelopmentGoal* von Class mit mehreren Attributen realisiert.

```
Individual EffortDevelopmentGoal in Class with
 attribute
   edg_object : GQMDevelopmentProcess
   edg_purpose : Characterization
   edg_viewpoint : Projectmanager
   edg_quality_focus : Effort
   edg_context : SEIIPraktikum
end
```

Bild 32: EffortDevelopmentGoal

Die interne Darstellung der Klasse *EffortDevelopmentGoal* in ConceptBase ist die Proposition:

P (#p1, #p1, EffortDevelopmentGoal, #p1, ...)

Dabei ist #p1 der interne Identifier und *EffortDevelopmentGoal* der Name der Proposition.

Das erste Attribut von *EffortDevelopmentGoal* ist *edg_object* und hat als Zielproposition die Klasse *GQMDevelopmentProcess*. Dies bedeutet, daß der Wert des Attributs *edg_object* eine Instanz der Klasse *GQMDevelopmentProcess* sein muß (Attributtyp). Die interne Darstellung der Klasse GQMDevelopmentProcess ist die Proposition:

P (#p2, #p2, GQMDevelopmentProcess, #p2, ...)

Dabei ist #p2 der interne Identifier und *GQMDevelopmentProcess* der Name der Proposition. Die Schreibweise für Attribute lautet Klasse!attribut. Das Attribut *EffortDevelopmentGoal!edg_object* wird dann intern folgendermaßen realisiert:

P (#p3, #p1, edg_object, #p2, ...)

Dabei ist #p3 der interne Identifier und der *edg_object* Name der Proposition. Weiterhin ist #p1 der interne Identifier der Quellproposition *EffortDevelopmentGoal* und #p2 der interne Identifier der Zielproposition *GQMDevelopmentProcess*.

Die Instanz *EffortDevelopmentGoal* ist eine Aggregation aus ihren Attributen. Intern in ConceptBase ist eine Aggregation eine Liste von Propositionen mit gleichen Quellpropositionen.

Eine Beziehung kann in O-Telos auch als Klasse modelliert werden, die wiederum Attribute besitzt, die als Zielpropositionen die Klassen enthalten, die verbunden werden sollen.

Innerhalb des GQM-Plans *EffortDevelopmentProcessGQMPlan* wird z.B. das Ziel *EffortDevelopment-Goal* durch die Frage *Q1* operationalisiert (siehe Anhang A). Diese Operationalisierungsbeziehung kann folgendermaßen realisiert werden.

```
Individual Q1Operationalizes in Class with
  attribute
    q1_op_goal : EffortDevelopmentGoal
    q1_op_question : Q1
end
```

Bild 33: Q1Operationalizes

Q1Operationalizes ist der Name der Beziehung, das Attribut *q1_op_goal* stellt die Verbindung zu *EffortDevelopmentGoal* her und das Attribut *q1_op_question* stellt die Verbindung zur Frage *Q1* her.

Somit sind Klassen, Attribute und Beziehungen modellierbar, d.h. die Informationen der realen Welt (in diesem Fall der GQM-Plan "EffortDevelopmentProcessGQMPlan") sind darstellbar (siehe Anhang E).

7.2.2 Fallstudie B: Lichtregelungssystem

Betrachtet man das Zustandsdiagramm "Licht", so existiert dort ein Zustand *An*. Dies wird durch eine Klasse *An* realisiert.

```
Individual An in Class
end
```

Bild 34: Zustand An

Die interne Darstellung der Klasse *An* ist die Proposition:

P (#p1, #p1, An, #p1, ...)

Dabei ist #p1 der interne Identifier und *An* der Name der Proposition. Weiterhin existiert in dem Diagramm noch ein Zustand *Aus* und ein Übergang zwischen den Zuständen *An* und *Aus*. Dieser Übergang wird durch eine Klasse *AnAus* realisiert.

```
Individual AnAus in Class
end
```

Bild 35: Übergang AnAus

Die interne Darstellung der Klasse *AnAus* ist die Proposition:

P (#p2, #p2, AnAus, #p2, ...)

Dabei ist #p2 der interne Identifier und *AnAus* der Name der Proposition. Die Verbindung zwischen dem Zustand *An* und dem Übergang *AnAus* wird folgendermaßen realisiert:

```
Individual AnOutgoing in Class with
 attribute
   an_target : An
   an_outgoing : AnAus
end
```

Bild 36: Verbindung zwischen Zustand An und Übergang AnAus

Dies ist so kompliziert, da das Open-Metamodel von UML dies so realisiert. Die interne Darstellung der Klasse *AnOutgoing* ist die Proposition:

P (#p3, #p3, AnOutgoing , #p3, ...)

Die Schreibweise für Attribute ist Klasse!attribut. Das Attribut *AnOutgoing!an_target* wird dann folgendermaßen realisiert:

P (#p4, #p3, an_target, #p1, ...)

Das Attribut *AnOutgoing!an_outgoing* wird folgendermaßen realisiert:

P (#p4, #p3, an_outgoing, #p2, ...)

Dabei ist #p4 der interne Identifier und *an_outgoing* der Name der Proposition. Weiterhin ist #p3 der interne Identifier der Quellproposition *AnOutgoing* und #p2 der interne Identifier der Zielproposition *AnAus*.

AnOutgoing ist eine Aggregation aus den Attributen. Intern ist eine Aggregation eine Liste von Propositionen mit gleichen Source-Propositionen.

Somit sind Klassen, Attribute und Beziehungen modellierbar, d.h. die Informationen der realen Welt (in diesem Fall die Zustandsdiagramme "Licht" und "Raum") sind darstellbar (siehe Anhang G).

7.3 ARS-Sichten: Meta-Klassen

O-Telos besitzt die Fähigkeit, verschiedene Sichten, die in verschiedenen Repräsentationssprachen vorliegen, zu repräsentieren. Dies geschieht mithilfe von Metamodellierung. Das O-Telos zugrundeliegende Informationsmodell besitzt unter anderem die Strukturierungsprinzipien Metamodellierung und Vererbung. Die Repräsentationssprachen werden mittels Meta-Klassen modelliert.

7.3.1 (Meta-) Instanziierung

O-Telos bietet die Möglichkeit, sowohl Klassen als auch Attribute (da diese auch als Klassen behandelt werden) über mehrere Ebenen hinweg zu instanziieren. Es existiert also eine Ebenen-Hierarchie, die von unten nach oben immer abstrakter wird. Auf der untersten Ebene (L_0) befinden sich "Token". Das sind Propositionen, die keine Instanzen haben. Auf der nächsten Ebene (L_1) befinden sich einfache Klassen (Class). Das sind Propositionen, die als Instanzen nur Token haben. Eine Ebene darüber (L_2) befinden sich Meta-Klassen (MetaClass). Das sind Propositionen, die Klassen als Instanzen haben, usw. Außerdem existieren noch sogenannte Omega-Klassen. Dies sind Propositionen, die auf allen Ebenen Instanzen haben können, wie z.B. die Proposition "Proposition" [MBJ+90].

In ConceptBase intern werden die Instanziierungsbeziehungen in folgender Form abgelegt.

*P (#p3, #p1, *instanceof, #p2, ...)*

Dies bedeutet, daß die Proposition mit dem Identifier #p1 eine Instanz der Proposition mit Identifier #p2 ist.

7.3.1.1 Fallstudie A: Entwicklungsaufwand

Ein GQM-Plan besitzt immer ein Ziel (*Goal*) mit den Attributen Objekt (*object*), Zweck (*purpose*), Qualitätsfokus (*quality_focus*), Blickwinkel (*viewpoint*) und Kontext (*context*). Wir definieren eine Meta-Klasse *Goal* mit den entsprechenden Attributen.

```
Individual Goal in MetaClass with
 attribute
    object: GQMObject;
    purpose : Purpose;
    quality_focus : QualityFocus;
    viewpoint : ViewPoint;
    context : Context
end
```

Bild 37: Meta-Klasse Goal

EffortDevelopmentGoal wird nun als Instanz von *Goal* betrachtet, wobei auch die Attribute von Goal instanziiert werden, z.B. *Goal!object* durch *EffortDevelopmentGoal!edg_object*. Auch die Attribute werden instanziiert, da Attribute und Klassen gleichermaßen als Propositionen behandelt werden.

```
Individual EffortDevelopmentGoal in Goal with
 attribute,object
   edg_object : GQMDevelopmentProcess
 attribute,purpose
   edg_purpose : Characterization
 attribute,viewpoint
   edg_viewpoint : Projectmanager
 attribute,quality_focus
   edg_quality_focus : Effort
 attribute,context
   edg_context : SEIIPraktikum
end
```

Bild 38: EffortDevelopmentGoal als Instanz von Goal

Die Meta-Klasse *Goal* kann als Typ ihrer Instanz *EffortDevelopmentGoal* betrachtet werden. Ziele dürfen nur diese Form besitzen. Damit wird die Syntax von Zielen durch die modellinhärenten Konsistenzbedingungen gesichert (siehe 7.3.3).

7.3.1.2 Fallstudie B: Lichtregelungssystem

In der Sicht Zustandsdiagramme existieren Zustände (*State*) und Übergänge (*Transition*). Um Übergänge zu charakterisieren, definieren wir eine Meta-Klasse *Transition*. Sie besitzt keine Attribute.

```
Individual Transition in MetaClass
end
```

Bild 39: Meta-Klasse Transition

AnAus wird nun als Instanz von *Transition* betrachtet.

```
Individual AnAus in Transition
end
```

Bild 40: AnAus als Instanz von Transition

Die Meta-Klasse *Transition* kann als Typ ihrer Instanz *AnAus* betrachtet werden. Übergänge dürfen nur diese Form besitzen. Damit wird die Syntax von Übergängen durch die modellinhärenten Konsistenzbedingungen gesichert (siehe 7.3.3).

7.3.2 Vererbung (Generalisierung/Spezialisierung)

Durch Generalisierung können Attribute vererbt werden. Dabei wird die Klasse, die die Attribute vererbt, Superklasse, und die Klasse, die die Attribute erbt, Subklasse genannt. Dadurch entstehen sogenannte "isA"-Hierarchien. Diese "isA"-Hierarchien sind orthogonal (siehe Bild 41) zu der "in"-Hierarchie (Instanziierung). In O-Telos ist Mehrfachvererbung möglich, d.h. eine Klasse kann von

mehreren unabhängigen Superklassen erben. Die interne Darstellung für isA-Beziehungen ist die folgende Proposition.

*P (#p3, #p1, *isa, #p2, ...)*

Dies bedeutet, daß die Proposition mit dem Identifier #p1 eine Spezialisierung der Proposition mit Identifier #p2 ist.

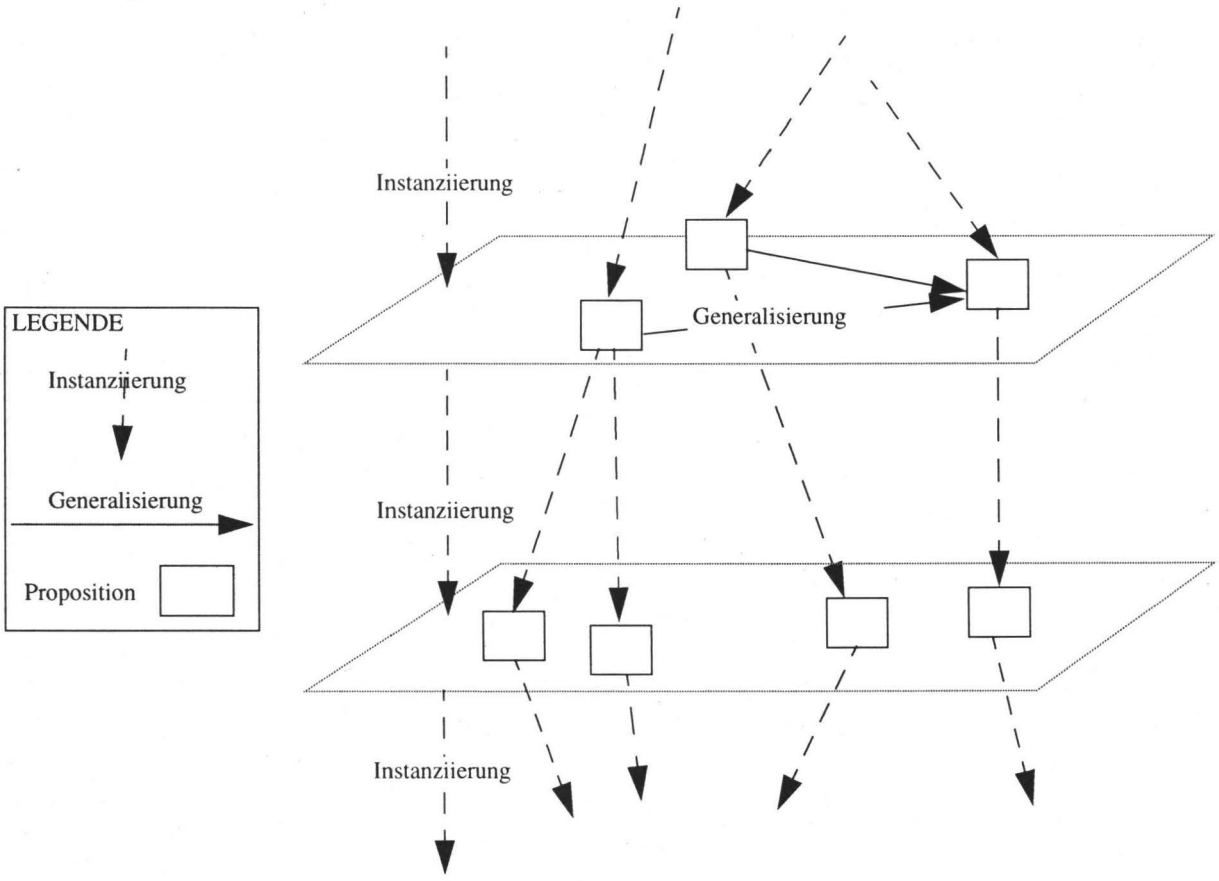

Bild 41: Orthogonale Generalisierungs- und Instanziierungs-Hierarchien

7.3.2.1 Fallstudie A: Entwicklungsaufwand

Die Klassen *Goal*, *Question* und *Measure* haben alle das Attribut *object*. Deswegen ist es sinnvoll, eine Klasse Plan-Element (*PlanElem*) als Superklasse einzuführen, die das Attribut *object* besitzt und an die Klassen *Goal*, *Question* und *Measure* vererbt.

In ConceptBase kann die Klassenstruktur mithilfe eines graphischen Browsers betrachtet werden. Ein Ausschnitt aus dem Metamodell für GQM-Pläne (siehe Bild 42) zeigt die Generalisierung zwischen *PlanElem* und *Goal*, *Question* und *Measure*. *QualityQuestion* ist eine Spezialisierung von Question.

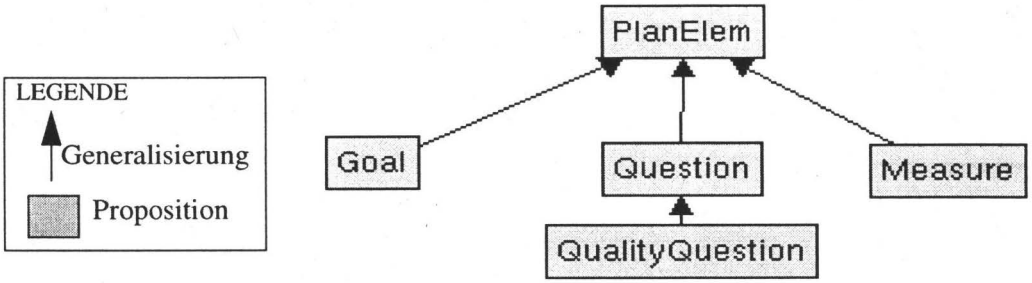

Bild 42: Generalisierung - Graphik

Textuell wird das Prinzip der Generalisierung durch eine "isA"-Beziehung angegeben. Wir definieren zunächst die Superklasse *PlanElem* und anschließend die Subklasse *Goal* von *PlanElem*.

```
Individual PlanElem in MetaClass with
 attribute
    object : GQMObject
end

Individual Goal in MetaClass isA PlanElem with
 attribute
    purpose : Purpose;
    quality_focus : QualityFocus;
    viewpoint : ViewPoint;
    context : Context
end
```

Bild 43: Generalisierung - Text

Somit erbt die Klasse *Goal* das Attribut *object* von der Klasse *PlanElem*. Durch diese Spezialisierung wird jedoch nicht die aktuelle Modellierungsebene verlassen. Diese Ebene wird nur durch Instanziierung ("in"-Beziehung) verlassen (Anhang D: Ausschnitt aus integriertem Metamodell für GQM-und Projektpläne).

7.3.2.2 Fallstudie B: Lichtregelungssystem

State und *Transition* sind Spezialisierungen von *ModelElement*, wobei bei *State* noch eine Klasse *State-Vertex* dazwischensteht. *ModelElement* ist die Wurzel des gesamten integrierten UML-Metamodells (oberste Superklasse). In ConceptBase kann die Klassenstruktur mithilfe eines graphischen Browsers betrachtet werden (siehe Bild 44).

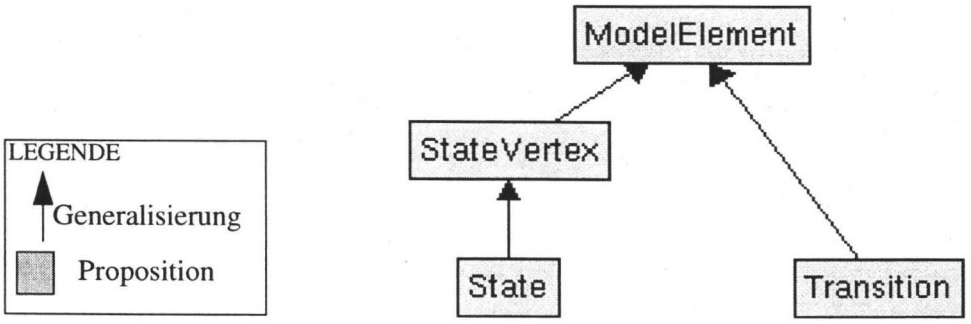

Bild 44: Generalisierung - Graphik

Textuell wird das Prinzip der Generalisierung durch eine "isA"-Beziehung angegeben.

Individual ModelElement in MetaClass
end

Individual Transition in MetaClass isA ModelElement
end

Bild 45: Generalisierung - Text

Wir definieren zunächst die Superklasse *ModelElement* und anschließend die Subklasse *Transition* von *ModelElement* (siehe Bild 45).

7.3.3 Axiome zur Sicherung modellinhärenter Konsistenzbedingungen

Um die modellinhärenten Konsistenzbedingungen, wie Generalisierung, Meta-Instanziierung, usw. zu sichern, werden intern in ConceptBase Axiome benutzt (DATALOG). Axiome sind prädikatenlogische Formeln. Die Zusicherung dieser Axiome wird vom System garantiert. Dazu werden an bestimmte Propositionen die Konsistenzbedingungen als Constraints angehängt (es gibt z.B. Klasse *IsA*, *InstanceOf*, usw.). Diese Constraints sind in ConceptBase fest definiert und gewährleisten z.B. korrekte Instanziierung oder Vererbung.

Hat die Repräsentationssprache, die modelliert werden soll, weitere Strukturierungsprinzipien (z.B. Multiplicity), so werden an die Meta-Klassen, die die Repräsentationssprache repräsentieren, Regeln und Constraints angehängt, die die Axiome (Strukturierungsprinzipien) der betrachteten Sprache zusichern. Dieses Vorgehen wird in 7.4 erläutert.

Die Instanziierung kann als weicher Typmechanismus angesehen werden. Die Klassen legen fest, aus welchen Attributen sich ihre Instanzen zusammensetzen können (nicht müssen). So werden praktisch die Syntaxregeln der Repräsentationssprache definiert. Modellinhärente Konsistenzbedingungen (herrührend von den Modellierungsprinzipien, z.B. Generalisierung, Aggregation, (Meta-)Instanziierung) sichern die korrekte Syntax der Daten. Syntaxregeln werden also durch eine Typüberprüfung realisiert.

7.4 ARS-Intra-Regeln: Die Klasse "Assertion"

Die Verfolgbarkeit innerhalb einer Sicht kann hergestellt werden, d.h. explizite Intra-Sicht-Beziehungen sind in O-Telos formulierbar. Semantische Beziehungen innerhalb einer Sicht werden durch Instanzen der Klasse "Assertion" repräsentiert. Eine Repräsentationssprache wird mithilfe von Meta-Klassen modelliert, deren Regeln und Constraints die Axiome der betrachteten Sprache zusichern. Jeder Proposition können Constraints zugewiesen werden. Constraints sind prädikatenlogische Formeln erster Ordnung, die erfüllt sein müssen. Constraints sind Instanzen einer Klasse "Assertion" und werden als Attributwerte einer Proposition hinzugefügt. Durch diese Constraints können semantische Beziehungen ausgedrückt werden. Dies diskutieren wir im folgenden anhand der Fallstudien.

7.4.1 Fallstudie A: Entwicklungsaufwand

Eine Frage darf höchstens ein Ziel operationalisieren. Soll diese Frage ein weiteres Ziel operationalisieren, so soll diese Operationalisierungsbeziehung zurückgewiesen werden. Dies wird durch folgendes Constraint *c* der Klasse *Operationalizes* gesichert.

```
Individual Operationalizes in MetaClass with
 attribute,constraint
   c : $ not (exists q/QualityQuestion g1,g2/Goal op/Operationalizes
   (this question q) and (this goal g1) and
   (op question q) and (op goal g2)) )$
end
```

Bild 46: Constraint

Wird eine Frage *Q1*, die ein Ziel *G1* operationalisiert (durch *OP1*), durch eine weitere Operationalisierungsbeziehung *OP2* an ein weiteres Ziel *G2* gebunden, so wird das Constraint der Meta-Klasse Operationalizes verletzt. Die Einfüge-Operation von *OP2* wird zurückgewiesen, und das Constraint *c* der Klasse Operationalizes als Begründung der Zurückweisung angegeben. Weiterhin wird *OP1* als schon existierende Operationalisierungsbeziehung angegeben.

7.4.2 Fallstudie B: Lichtregelungssystem

Ein Übergang darf höchstens einen Zustand als Ziel (*target*) haben. Soll ein weiterer Ziel-Zustand eingeführt werden, so soll dies zurückgewiesen werden. Dies wird durch folgendes Constraint *c* der Klasse *Outgoing* gesichert.

```
Individual Outgoing in MetaClass with
 attribute,constraint
   c : $ not (exists s1,s2/State
   (this target s1) and (this target s2) )$
end
```

Bild 47: Constraint

Wird der die Verbindung *AnOutgoing* von Zustand *An* nach Übergang *AnAus* betrachtet, so hat diese Verbindung einen Zielzustand *An*. Wird jetzt versucht, ein weiterer Zielzustand, z.B. *Aus* einzugeben (TELL (*AnOutgoing an_target Aus*)), so können beide Variablen *s1* und *s2* gebunden werden. Dies bedeutet, daß für alle Variablen Werte existieren (nämlich *An* und *Aus*) und somit das Constraint nicht mehr erfüllt ist, da ein *not* vor der Formel steht.

7.5 ARS-Inter-Regeln: Integration der Meta-Klassen

Die Integration von Sichten, die in verschieden Repräsentationssprachen vorliegen, erfolgt durch die Integration der Metamodelle der verschiedenen Sprachen. Dies geschieht durch gemeinsame Metaklassen. Um Verfolgbarkeit zwischen diesen Sichten herstellen zu können, werden weiterhin die semantischen Beziehungen zwischen den Sichten durch explizite Inter-Sicht-Constraints formuliert. Diese Constraints definieren explizit die semantischen Beziehungen zwischen den einzelnen Metamodellen. Mit Hilfe dieser Constraints entsteht dann ein integriertes Gesamt-Metamodell.

7.5.1 Fallstudie A: Entwicklungsaufwand

Es findet eine Verbindung der Konzepte durch Constraints statt. *PlanElem* hat als *Objekt* entweder einen Prozeß oder ein Produkt, dies sind Konzepte aus dem Metamodell der Projektpläne (siehe Bild 48).

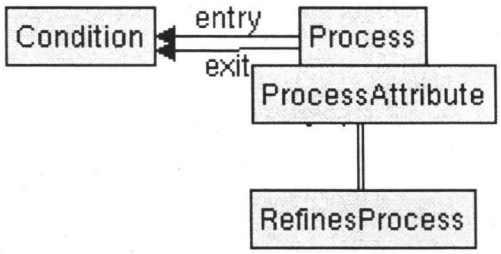

Bild 48: Ausschnitt aus Projektplan-Metamodell

Die Verbindung zwischen dem Metamodell der Projektpläne und dem Metamodell der GQM-Pläne sieht folgendermaßen aus:

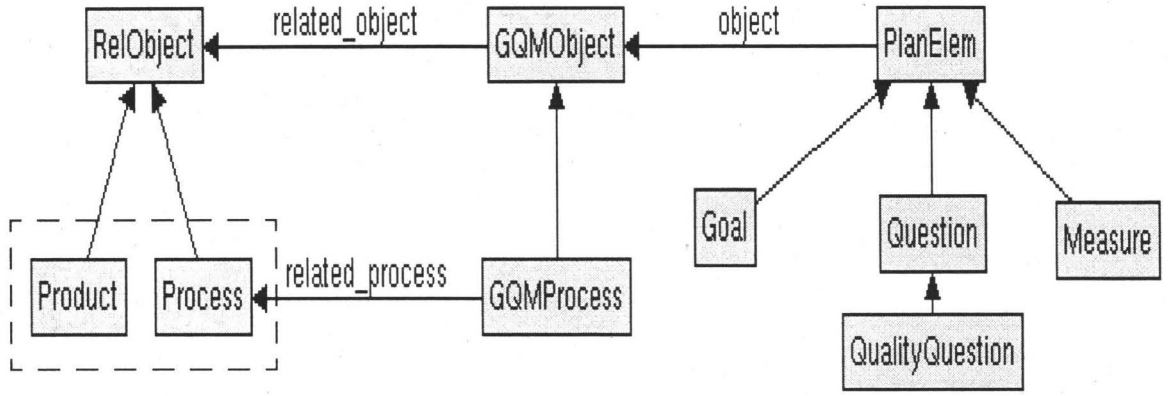

Bild 49: Verbindung - Graphik

Ein Planelement hat als Objekt ein GQMObjekt, daß mit einem *RelObject* in Beziehung gesetzt wird. Ein *RelObject* kann ein Prozeß oder ein Produkt aus dem Metamodell der Projektpläne sein. Wird ein Prozeß gelöscht, der mit einem GQMProcess verbunden ist, so wird das Constraint der Klasse *PlanElem* verletzt.

```
Individual PlanElem in MetaClass with
  attribute,constraint
    c : $ not ( exists go/GQMObject
    ((this object go) and (exists p/Process go related_object p)) or
    ((this object go) and (exists pr/Product go related_object pr)) )$
end
```

Bild 50: NotExistObject-QueryClass

Dieses Constraint besagt, daß für jedes Planelement eines GQM-Plans ein mit diesem Planelement verbundener Prozeß oder ein mit diesem Planelement verbundenes Produkt existieren muß.

7.5.2 Fallstudie B: Lichtregelungssystem

Die Verbindung von verschiedenen Repräsentationssprachen geschieht über gemeinsame Konzepte (Meta-Klassen). Übergänge (Transition) werden von Events angestoßen ("getriggert"). CallEvents sind eine Spezialisierung der Klasse Event, die mit UMLOperationen verbunden sind (siehe Bild 53 auf Seite 71).

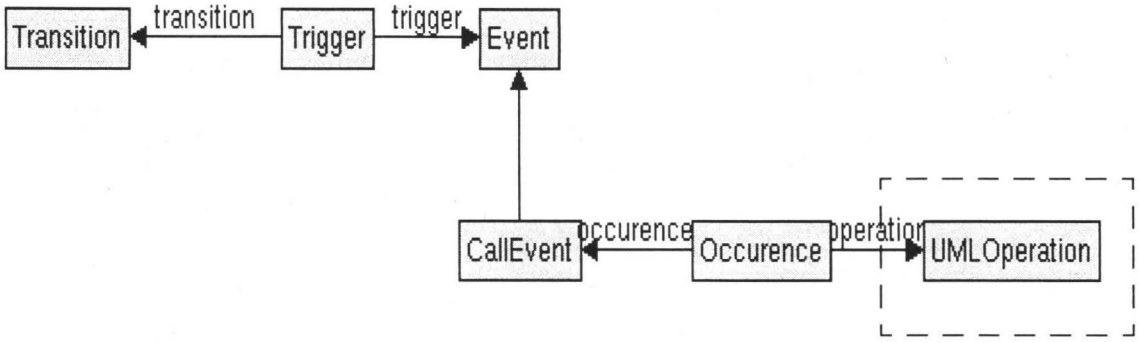

Bild 51: Verbindung - Graphik

UMLOperationen kommen auch im Metamodell der UML-Klassendiagramme vor als BehavorialFeature eines Classifiers.

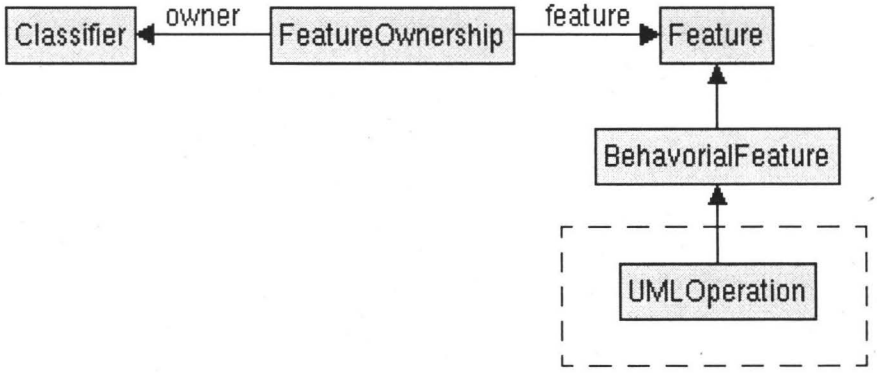

Bild 52: Verbindung - Graphik

Die Definition der Klasse UMLOperation sieht folgendermaßen aus:

Individual UMLOperation in MetaClass isA BehavorialFeature with
attribute
specification : Uninterpreted;
is_polymorphic : Boolean;
concurrency : CallConcurrencyKind
end

Bild 53: Gemeinsames Konzept

Die Attribute der Klasse sind zur besseren Übersichtlichkeit auf den Bildern 51 und 52 nicht dargestellt. Sie sind auch nicht relevant für dieses Beispiel. Ein Constraint wird der Meta-Klasse CallEvent hinzugefügt, das über die Occurence-Assoziation die Verbindung zu den Klassendiagrammen herstellt.

Individual CallEvent in MetaClass isA Event with
attribute,constraint
c: $ exists uo/UMLOperation oc/Occurence
(oc operation uo) and (oc call_event this) $
end

Bild 54: UMLOperation-CallEvent-Constraint

7.6 AW-Konsistenz: Constraint-Prüfung

ConceptBase ist ein Wissensbankverwaltungssystem und hat eine objektorientierte deduktive Datenbank (C++) und eine Inferenzmaschine (DATALOG-basiert), mithilfe der Konsistenzprüfungen durchgeführt werden können. Die Wissensbank enthält eine Sammlung von historischen Theorien erster Ordnung, die mit Belief-Zeiten (Intervalle, Allen'sche Zeitlogik) indiziert sind. Weiterhin enthält sie eine Menge von Konsistenz-Constraints [MBJ+90]. Eine Theorie enthält Typen (alle durch TELL eingegebenen Propositionen = Klassen, Attribute, Constraints, usw.) und Axiome (Constraints für Vererbung, Instanziierung,...). Dann werden Prädikate definiert (z.B. in(x,X) liefert true oder false, je nachdem, ob x eine Instanz von X ist) und Funktionen (z.B. from(P) liefert z.B. source).

Bei einer Operation auf den Daten (TELL, UNTELL, ASK) wird eine neue historische Theorie erzeugt. Um Herleitung von Wissen zu realisieren, wird eine Beweistheorie [KMS+89] benötigt. Zunächst muß ein Abschluß der historischen Theorie gemacht werden, um negative Informationen zu erhalten. Konsistenz bedeutet hier, daß alle Constraints gelten. Die Methode zur Konsistenzprüfung ist eine Version der Methode von [BDM88] zur Constraint-Erfüllung in deduktiven Datenbanken, genauere Erläuterungen führen im Rahmen dieser Arbeit zu weit.

Wird z. B. durch UNTELL eine Proposition gelöscht, so entsteht wieder eine neue historische Theorie, die wieder abgeschlossen werden muß, und auf der die Constraints überprüft werden. Bei einer inkonsistenten Änderung erfolgt eine Fehlermeldung, die Angabe der verletzten Constraints, und die Änderung wird zurückgewiesen. Dabei werden die Constraints auf Klassen definiert und auf den Instanzen der Klassen überprüft.

7.6.1 Fallstudie A: Entwicklungsaufwand

Wir betrachten jeweils die Konsistenzprüfung anhand von Constraints innerhalb einer Sicht und anhand von Constraints zwischen Sichten.

Constraints innerhalb einer Sicht

Eine Frage darf höchstens ein Ziel operationalisieren. Soll diese Frage ein weiteres Ziel operationalisieren, so soll diese Operationalisierungsbeziehung zurückgewiesen werden. Dies wird durch folgendes Constraint c der Klasse Operationalizes gesichert.

```
Individual Operationalizes in MetaClass with
  attribute,constraint
    c : $ not (exists q/QualityQuestion g1,g2/Goal op/Operationalizes
    (this question q) and (this goal g1) and
    (op question q) and (op goal g2)) )$
end
```

Bild 55: Constraint

Wird eine Frage *Q1*, die ein Ziel *G1* operationalisiert (durch *OP1*), durch eine weitere Operationalisierungsbeziehung *OP2* an ein weiteres Ziel *G2* gebunden, so wird das Constraint der Meta-Klasse Operationalizes verletzt. Die Variable q wird an *Q1* gebunden, die Variable g1 wird an *G1* gebunden, die Variable op wird an *OP2* gebunden und die Variable g2 wird an *G2* gebunden. Dies bedeutet, daß für

alle Variablen Werte existieren und somit das Constraint nicht mehr erfüllt ist, da ein *not* vor der Formel steht. Die Einfüge-Operation von *OP* wird zurückgewiesen, und das Constraint *c* der Klasse Operationalizes als Begründung der Zurückweisung angegeben. Weiterhin wird *OP2* als schon existierende Operationalisierungsbeziehung angegeben.

Constraints zwischen Sichten

Wird ein Prozeß gelöscht, der mit einem GQMProcess verbunden ist, so wird das Constraint der Klasse PlanElem verletzt.

```
Individual PlanElem in MetaClass with
 attribute,constraint
   c : $ not ( exists go/GQMObject
   ((this object go) and (exists p/Process go related_object p)) or
   ((this object go) and (exists pr/Product go related_object pr)) )$
end
```

Bild 56: NotExistObject-QueryClass

Dieses Constraint besagt, daß für jedes Planelement eines GQM-Plans ein mit diesem Planelement verbundener Prozeß oder ein mit diesem Planelement verbundenes Produkt existieren muß.

7.6.2 Fallstudie B: Lichtregelungssystem

Constraints innerhalb einer Sicht

Ein Übergang darf höchstens einen Zustand als Ziel (target) haben. Soll ein weiterer Ziel-Zustand eingeführt werden, so soll dies zurückgewiesen werden. Dies wird durch folgendes Constraint c der Klasse Operationalizes gesichert.

```
Individual Outgoing in MetaClass with
 attribute,constraint
   c : $ not (exists s1,s2/State
   (this target s1) and (this target s2) )$
end
```

Bild 57: Constraint

Wird der die Verbindung AnOutgoing von Zustand An nach Übergang AnAus betrachtet, so hat diese Verbindung einen Zielzustand An. Wird jetzt versucht, ein weiterer Zielzustand, z.B. Aus einzugeben (TELL (AnOutgoing an_target Aus)), so können beide Variablen s1 und s2 gebunden werden. Dies bedeutet, daß für alle Variablen Werte existieren (nämlich An und Aus) und somit das Constraint nicht mehr erfüllt ist, da ein *not* vor der Formel steht. Die Einfüge-Operation wird zurückgewiesen, und das Constraint *c* der Klasse Outgoing als Begründung der Zurückweisung angegeben. Weiterhin wird *An* als schon existierender Zielzustand angegeben.

Constraints zwischen Sichten

Wird eine UMLOperation gelöscht, so kann das Constraint der Klasse CallEvent verletzt werden.

```
Individual CallEvent in MetaClass isA Event with
    attribute,constraint
        c: $ exists uo/UMLOperation oc/Occurence
        (oc operation uo) and (oc call_event this) $
end
```

Bild 58: UMLOperation-CallEvent-Constraint

Es werden die inkonsistenten Stellen, nämlich die Instanzen der Klasse CallEvent zurückgeliefert, die das Constraint verletzen.

7.7 AW-Auswirkungen: Die Klasse "QueryClass"

Eine Auswirkungsanalyse kann durch Instanzen der Klasse QueryClass realisiert werden. Anfragen an die Datenbank werden anhand von deduktiven Regeln realisiert (Herleitung durch Inferenzmaschine). Deduktiven Regeln sind wie Constraints Instanzen der Klasse "Assertion" und haben die Form:

Bedingung (Condition) -> Konklusion

Sie basieren auf Hornlogik (PROLOG) und werden mit DATALOG realisiert. Jede Anfrage, die an das System gestellt wird, terminiert und liefert die korrekte Antwort. Die Antworten auf Anfragen sind parametrisierte Views [EJJ+89]. Als Bedingung wird in einer Query-Klasse das negierte Constraint angeben, das freie Variablen enthält [MBJ+90]. Antwort sind dann alle Propositionen, die das Constraint nicht erfüllen, d.h. alle inkonsistenten Stellen. Diese Antworten erhält man durch Substitutionen der freien Variablen in dem Constraint. Die Herleitung geschieht durch einen Backward-Chaining-Algorithmus. Der Vorteil dieser Methode ist, daß Änderungen nicht zurückgewiesen werden, sondern die inkonsistenten Stellen in einer Query-Klasse gesammelt werden, und der Benutzer selbst entscheiden kann, wann er sich die inkonsistenten Stellen anschaut und wie er Änderungen vornimmt, da auch inkonsistente Sichten akzeptiert werden.

7.7.1 Fallstudie A: Entwicklungsaufwand

Constraints innerhalb einer Sicht

Jedes Ziel muß mindestens durch eine Qualitäts-Frage operationalisiert werden. Dies wird durch ein Constraint c der Klasse Goal gesichert. Wird die einzige Qualitätsfrage *Q1*, die das Ziel *EffortDevelopmentGoal* operationalisiert, gelöscht, so wird das Constraint der Meta-Klasse *Goal* verletzt. Die Variable q kann an keine QualityQuestion mehr gebunden werden, für die die angegebenen Bedingungen gelten. Dies bedeutet, daß für die Variablen keine Werte existieren und somit das Constraint erfüllt ist, da ein not vor der Formel steht. Das Ziel *EffortDevelopmentGoal* wird eine Instanz der Query-Klasse NotExistQuestion.

```
Individual NotExistQuestion in QueryClass isA Goal with
 attribute,constraint
   c : $ not (exists q/QualityQuestion op/Operationalizes
   (op question q) and (op goal this))) $
end
```

Bild 59: NotExistQuestion-QueryClass

Das Constraint c ist eine prädikatenlogische Formel mit Sorten (Typen). Es kommen die Variablen q und op vor. Dabei ist q vom Typ QualityQuestion und op vom Typ Operationalizes. Für jedes Goal muß immer ein op des Typs Operationalizes existieren, der als goal dieses Goal hat und als question ein q des Typs QualityQuestion. Dieses Constraint wird vom System jeweils auf den Instanzen von *Goal* überprüft, z.B. bei *EffortDevelopmentGoal*. Das System versucht die Variablen q und op an Instanzen der Klasse QualityQuestion und Operationalizes zu binden. Werden keine solchen Instanzen gefunden, wird das Constraint c nicht mehr erfüllt. Da ein not vor der Formel steht, wird das Ziel in die Query-Klasse aufgenommen.

Constraints zwischen Sichten

Nun betrachten wir die Beziehungen zwischen der Sicht GQM-Pläne und Projektpläne. Das Prozeß-Object-Constraint wird negiert in eine Query-Klasse NotExistObject geschrieben.

```
Individual NotExObject in QueryClass isA PlanElem with
 attribute,constraint
   c : $ not ( exists go/GQMObject
   ((this object go) and (exists p/Process go related_object p)) or
   ((this object go) and (exists pr/Product go related_object pr)) )$
end
```

Bild 60: NotExistObject-QueryClass

Die Antwort auf die Anfrage sind dann die Instanzen dieser Klasse.

7.7.2 Fallstudie B: Lichtregelungssystem

Constraints innerhalb einer Sicht

Ein Zustandsdiagramm (State-Machine) muß immer einen Initial-Zustand (Top-State) haben.

```
Individual StateMachine in MetaClass with
 attribute,constraint
   c : $ exists s/State t/Top
   (t state_machine this) and (t top s)) $
end
```

Bild 61: Constraint

Das Constraint c ist eine prädikatenlogische Formel mit Sorten (Typen). Es kommen die Variablen s und t vor. Dabei ist s vom Typ State und t vom Typ Top. Wird der Initial-Zustand eines Zustandsdia-

gramms gelöscht, so wird das Constraint der Klasse StateMachine verletzt. Die Instanz von StateMachine wird in die Query-Klasse aufgenommen, die das Constraint nun verletzt.

Constraints zwischen Sichten

Das UMLOperation-CallEvent-Constraint wird negiert in eine Query-Klasse InconsistentCallEvent geschrieben.

```
Individual InconsistentCallEvent in QueryClass isA CallEvent with
    attribute,constraint
        c: $ not (exists uo/UMLOperation oc/Occurence
        (oc operation uo) and (oc call_event this)) $
end
```

Bild 62: InconsistentCallEvent-QueryClass

Die Antwort auf die Anfrage sind dann die Instanzen dieser Klasse. Diese Instanzen erfüllen das Constraint nicht.

7.8 AW-Änderungen: ECA-Regeln

Automatische Änderungspropagierung kann durch ECA-Regeln stattfinden. ECA-Regeln sind Instanzen einer Klasse "ECArule" (Event-Condition-Action). Ein "Event" ist dabei ein Trigger, der aus Operationen (TELL, UNTELL, ASK) auf den Daten besteht. Eine "Condition" ist die Bedingung (Anfrage mit freien Variablen), die gelten muß für alle Instanzen, auf denen man den folgenden Aktionsteil ausführen will. "Action" enthält alle Operationen, die auf der Datenbank durchgeführt werden sollen. Diese werden auf allen Antwort-Werten (der freien Variablen) ausgeführt (Einfügen TELL, Löschen UNTELL, Anfragen ASK). Falls die Antwort leer ist, wird der ELSE-Zweig des Aktionsteils genau einmal ausgeführt.

In einigen Situationen ist es sinnvoll, inkonsistente Sichten zu dulden. Ein Beispiel ist, daß Sichten weitgehend erstellt werden, und erst nach einem gewissen Zeitpunkt auf Konsistenz überprüft werden. In diesen Situationen ist automatische Änderungspropagierung durch das Werkzeug absolut nicht erwünscht.

7.8.1 Fallstudie A: Entwicklungsaufwand

Das Event ist „Benutzbares-System-Erstellen einfügen". Die Bedingung (durch eine Anfrage MyECA-Query realisiert) wählt den passenden GQM-Plan aus. Die Aktion fügt die passende Frage (*Question*) ein. Durch das Attribut *active* ist das An- und Ausschalten der ECA-Regeln möglich, je nachdem, ob *my_act true* oder *false* zugewiesen wird.

```
Individual MyECAQuery in QueryClass isA Question with
   attribute,query
     q: $ y,p/Process rp/RefinesProcess gp/GQMProcess q/Question
      (rp process y) and
      (rp top_process p) and
      (gp related_process p) and
      (q object p)
end


Individual MyECA in ECArule with
   attribute, ecarule
     my_eca: $ x,p/Process rp/RefinesProcess gp/GQMProcess q/Question
     ON TELL in(x,Process)
     IF (q in MyECAQuery)
     DO
       TELL (quest in Question)
       TELL (break in BreaksDown)
       TELL (break ur_quality-question q)
       TELL ( break question quest) $
   attribute, active
     my_act: True
end
```

Bild 63: ECA- Entwicklungsaufwand

Nachdem im Bedingungsteil (IF) die passende Frage (*Question*) im passenden GQM-Plan ausgewählt worden ist, wird eine passende Unterfrage dieser Frage eingefügt (siehe Bild 63).

7.8.2 Fallstudie B: Lichtregelungssystem

Das Ereignis ist das Einfügen eines CallEvents. Es ist schwierig, direkt eine passende UMLOperation einem Classifier der Sicht Klassendiagramme zuzuordnen. Deshalb wird zunächst eine passende UML-Operation eingefügt. Durch das Attribut active ist das An- und Ausschalten der ECA-Regeln möglich, je nachdem, ob my_act true oder false zugewiesen wird.

```
Individual MyECA in ECArule with
   attribute, ecarule
     my_eca: $ x/Process
      ON TELL in(x,CallEvent)
      DO
        TELL in(z,UMLOperation)
        TELL (occur in Occurence)
        TELL (occur occurence these)
        TELL (occur operation z)
   attribute, active
     my_act: True$
end
```

Bild 64: ECA- Lichtregelungssystem

Dann kann diese UMLOperation, die jetzt noch keinem Classifier zugeordnet ist, durch eine Query-Klasse entdeckt werden, die alle UMLOperationen auflistet, die keinem Classifier zugeordnet sind. Der Benutzer kann dann von Hand diese Zuordnung vornehmen.

Es ist schwierig, automatische Änderungspropagierungen in Form von ECA-Regeln zu erstellen, da das implizit im Kopf vorhandene Wissen explizit in O-Telos formuliert werden muß. Dazu ist Übung im Umgang mit den Constraints notwendig.

Ein Problem, das auftreten kann, ist das Problem von Zyklen. Wenn zwei Trigger sich z.B. immer gegenseitig aktivieren, so läuft diese Schleife endlos. ConceptBase gibt eine Warnung, falls ein Zyklus entdeckt wird, stoppt ihn aber nicht. Der Benutzer muß selbst dafür sorgen, daß keine solchen Konflikte auftreten.

7.9 AW-Graphisch, -Sichtenbasiert, -Erweiterbar, -Anwendbar

Die rein werkzeugspezifischen, eher technischen Anforderungen behandeln wir hier geschlossen, und nehmen eine Bewertung von ConceptBase anhand der gesammelten Erfahrungen im Rahmen der Fallstudien vor.

Es existieren zwei Benutzungsschnittstellen, d.h. Oberflächen, für ConceptBase, eine X11- und eine Java-Oberfläche. Die Java-Oberfläche ist sehr langsam und verursacht häufiger Systemabstürze. Dies widerspricht der in AW-Anwendbar geforderten Effizienz beim Arbeiten mit dem System. Diese Oberfläche ist noch neu und wird derzeit weiterentwickelt und verbessert. Die X11-Oberfläche ist dagegen stabil. Das System stürzt nicht ab und es gehen keine Daten verloren.

Die X11-Oberfläche stellt eine graphische Benutzungsschnittstelle zur Verfügung (AW-Graphisch). An dieser Schnittstelle werden jedoch die Daten nur dargestellt und können nicht geändert werden, d.h. es existiert nur eine graphische Ausgabe-Schnittstelle, jedoch keine graphische Eingabe-Schnittstelle. Anforderung AW-Graphisch wird daher nur zum Teil erfüllt. Außerdem hat das Fehlen einer graphische Eingabe-Schnittstelle Auswirkungen auf die Erfüllung der geforderten Effizienz in AW-Anwendbar, da alle Eingaben textuell gemacht werden müssen. Dies zieht ein zeitaufwendiges Editieren nach sich, da auch keine Eingabe-Masken zur Verfügung stehen. Dies soll bei der Java-Oberfläche zumindest teilweise verbessert werden durch die Bereitstellung von Masken für Anfragen, welche den Editier-Aufwand verringern sollen. Diese Masken haben zur Zeit allerdings das Problem, daß sie die eingegebenen Constraints nicht annehmen, d.h. eingegebene Daten gehen verloren.

Die graphische Schnittstelle ist sichtenbasiert (AW-Sichtenbasiert). Der Benutzer kann sich nur die Propositionen anzeigen lassen, die ihn interessieren. Dabei kann er sich auch jeweils die Super-, Sub- und Metaklassen, die Attribute, sowie die Instanzen der Proposition anzeigen lassen. Außerdem existieren ein Modul- und ein Viewkonzept. Durch das Modulkonzept sind Sichten auf die Daten möglich. Module teilen die Propositionen sowohl auf Modell-, wie auch auf Token-Ebene auf. Die Klassen in den Modulen erscheinen jedoch nicht mehr auf der graphischen Oberfläche. Das Viewkonzept ist auf Klassen definiert. Es erlaubt, nur Ausschnitte der Klassen zu zeigen. Dies bedeutet, daß nur ausgewählte Attribute dargestellt werden.

Das Werkzeug ist erweiterbar (AW-Erweiterbar). Es können z.B. eigene graphische Notationen für die graphische Oberfläche erzeugt werden, indem Propositionen einer Klasse "GTyp" (graphischer Typ) definiert werden und Regel, die angeben, Instanzen welcher Klasse welchem graphischen Typ zugeordnet werden. ConceptBase ist realisiert in einer Client-Server-Architektur. Client-Programme können mit dem ConceptBase-Server verbunden werden. Es kann ein Datenaustausch über das Internet-Protokoll (interprocess communication) stattfinden. So erlaubt die ConceptBase Programm-Schnittstelle dem Benutzer, eigene Programme in Java, C++ oder Prolog anzubinden.

7.10 Zusammenfassende Bewertung und Erweiterungsvorschläge

Zusammenfassend ist zu sagen, daß die Anforderungen, bis auf die eher technische Anforderungen AW-Graphisch und AW-Anwendbar, erfüllt sind. Eine graphische Schnittstelle (AW-Graphisch) ist zwar vorhanden, aber sie dient nur zur Darstellung der vorhandenen Modelle und Token, jedoch können dort keine Modelle und Token geändert werden. Die Änderungen müssen alle im Texteditor gemacht werden. Dies ist wenig komfortabel. Da das textuelle Editieren zeitaufwendig ist, wird auch AW-Anwendbar nicht vollständig erfüllt.

Ein Vorschlag zur Erweiterung des ausgewählten Ansatzes ist daher, Änderungen in der graphischen Oberfläche möglich zu machen. Es sollte zumindest Masken für Anfragen oder Eingaben geben, in die nur die jeweiligen Propositionen-Namen eingegeben werden müssen, damit nicht soviel Text getippt werden muß. Diese Masken sollten sowohl dem Metamodellierer auf Modell-Ebene, als auch dem Benutzer auf Token-Ebene zur Verfügung stehen. Auf Token-Ebene sollte es z.B., wie in klassischen Datenbanksystemen, tabellenartige Masken für die existierenden Klassen geben. Die Masken würden die Attribute der Klasse enthalten, in die der Benutzer dann nur noch die Attribut-Werte eingeben müßte.

Es existieren viele verschiedene Möglichkeiten, gegebene Informationen in O-Telos zu modellieren. O-Telos setzt keine Grenzen in Bezug auf die Modellierung von Verfolgbarkeit, und das Entdecken von Inkonsistenzen. Die Grenzen sind eher auf der menschlichen Seite, da das Explizitmachen von implizitem (kognitivem) Wissen, das nur unstrukturiert im Kopf des Menschen vorliegt, schwierig ist.

Kapitel 8

Zusammenfassung und Ausblick

Dieses Kapitel enthält abschließend eine Zusammenfassung der Arbeit und einen Ausblick auf zukünftige Tätigkeiten.

8.1 Zusammenfassung

Das Ziel dieser Arbeit war die Auswahl eines Verfolgbarkeitsansatzes, der den Benutzer bei Änderungen unterstützt. Unterstützung bedeutet hierbei, daß der Aufwand zur und die Fehler bei der Durchführung von Änderungen reduziert werden. Da existierende Ansätze zum Ändern von Software-Artefakten (z.B. basierend auf einer manuell erstellten Verfolgbarkeitsmatrix) bei großen Informationsmengen sehr arbeitsintensiv sind, besteht die Annahme, daß diese Unterstützung durch die Verwendung von Werkzeugen erreicht werden kann. Deshalb wurde ein Verfolgbarkeitsansatz ausgewählt, der durch ein Werkzeug unterstützt wird. Der Verfolgbarkeitsansatz sollte allgemein gehalten sein, da er auf verschiedene Bereiche des Software Engineering, wie z.B. Systementwurf oder Meßplanung, anwendbar sein sollte.

Aus dem Problem wurden Anforderungen an einen geeigneten Lösungsansatz abgeleitet. Eine Literaturrecherche ergab, daß das Prinzip der Metamodellierung in Verbindung mit Wissensbankverwaltungssystemen ein geeigneter Lösungsansatz ist. Die Anforderungen an einen Lösungsansatz wurden in Bezug auf konkrete Werkzeuge erweitert. Eine zusätzliche Recherche nach einem den Lösungsansatz unterstützenden Werkzeug ergab, daß das Wissensbankverwaltungssystemen ConceptBase, das auf der Wissensrepräsentationssprache O-Telos basiert, ein geeignetes Werkzeug ist. O-Telos unterstützt das Prinzip der Metamodellierung.

ConceptBase wurde anhand zweier Fallstudien evaluiert. Die Evaluierung bestätigte die Bewertung anhand der Recherche, daß Metamodellierung/Wissensbankverwaltungssysteme als Ansatz und ConceptBase als realisierendes Werkzeug zur Lösung unseres Problems geeignet sind. ConceptBase erfüllt die gestellten Anforderungen, ausgenommen der Anforderung, daß eine graphische Schnittstelle vorhanden sein sollte. Diese steht zwar zur Verfügung, ist jedoch nur eine "Read-Only"-Schnittstelle.

Eine Erweiterung von ConceptBase wäre wünschenswert, so daß Änderungen auf der graphischen Oberfläche durchgeführt werden können und nicht textuell im Texteditor eingegeben werden müssen. Dies würde den Komfort des Systems erheblich erhöhen.

8.2 Ausblick

Der gewählte Verfolgbarkeitsansatz ist sehr allgemein gehalten, da er für die unterschiedlichsten Software-Artefakte aus den verschiedenen Bereichen des Software Engineering anwendbar sein sollte. Sinnvolle nächste Schritte wären:

- Eine intensive Betrachtung von Verfolgbarkeitsbeziehungen von speziellen Bereichen, wie z.B. der Meßplanung oder des UML-Entwurf. Dies war im Rahmen dieser Arbeit nicht möglich, da es, wie schon in 7.10 erwähnt, sehr zeitaufwendig ist, implizites Wissen explizit zu machen.
- Inkonsistenzen, die in diesen speziellen Bereichen auftreten, detailliert zu klassifizieren. Es existieren verschiedene Arten von Inkonsistenzen, für die es wiederum verschiedene Arten von Constraints gibt.
- Verschiedene Lösungsstrategien für verschieden Klassen von Inkonsistenzen zu entwickeln, eventuell mithilfe einer Kombination von ECA-Regeln, von direkt in Klassen definierten Constraints und von Query-Klassen.

Im weiteren Umfeld, z.B. auf dem Gebiet der Datenbank- oder Wissensverwaltungssysteme existieren auch noch einige offene Fragen. Weiterführende Arbeiten wären hier:

- Die Bewältigung der Komplexität bei Anfragen an die Wissensbank. Dieses Problem stellt sich bei sehr komplexen Informationsmengen, wie sie in großen Software-Projekten vorliegen. Bei den Fallstudien im Rahmen dieser Arbeit waren die Anfragen an die Datenbank effizient genug.
- Die Untersuchung von Konflikten bei Triggern, um Zyklen zu vermeiden. Falls sich Trigger gegenseitig aktivieren, können unendlich lange Operationen auf den Daten der Wissensbank angestoßen werden.
- Das Problem der Konsistenz der Konsistenzregeln (Constraints). Werden die Constraints direkt in die Klassen eingefügt, so werden inkonsistente Konsistenzregeln, d.h. Constraints, die anderen schon eingefügten Constraints widersprechen, vom System erkannt. Werden die Constraints jedoch in Query-Klassen definiert, so ist ihre Konsistenz nicht gewährleistet.

Anhang A

Fallstudie A

"Meßplanung mit GQM"

Diese Fallstudie stammt aus dem Software Engineering Bereich "Meßplanung". Es wird der GQM-Ansatz (Goal/Question/Metric) verwendet. Der Aufwand für ein Software-Entwicklungsprojekt soll gemessen werden (Entwicklungsaufwand). Dazu wird ein Meßplan (GQM-Plan) benötigt.

Mithilfe des GQM-Ansatzes wird zunächst ein Ziel (Goal) für den Meßplan entworfen (siehe Bild 65 auf Seite 83). Das Ziel beinhaltet ein Objekt, auf das sich die Messung bezieht (hier Entwicklungsprozeß); einen Zweck, zu dem gemessen wird (hier Charakterisierung); einen Qualitätsfokus, bezüglich dem gemessen wird (hier Aufwand); den Blickwinkel, aus dem gemessen wird (hier Projektmanager) und den Kontext, in dem die Messung stattfindet (hier SEII-Praktikum).

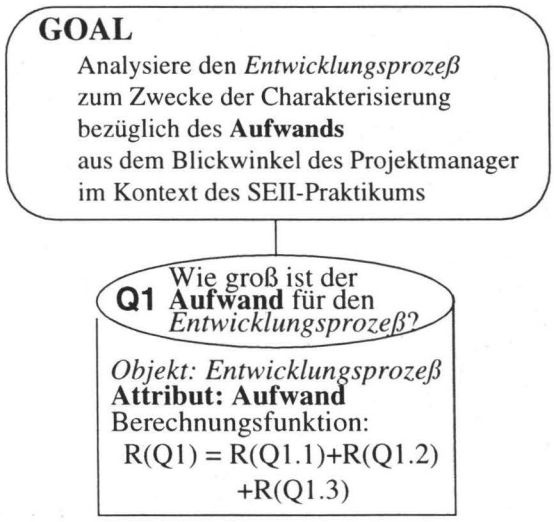

Bild 65: Goal-Question

Dieses Ziel wird durch Fragen operationalisiert (Question). Die Fragen haben wie das Ziel jeweils ein Objekt (siehe Bild 65 auf Seite 83), auf das sie sich beziehen (in Q1 Entwicklungsprozeß), ein Qualitätsattribut, auf das sich die Frage bezieht (in Q1 Aufwand) und eine Berechnungsfunktion, die angibt, wie die Antwort auf die Frage ermittelt wird (hier Summe der Resultate der Unterfragen Q1.i von Q1).

Die Fragen können in Unterfragen verfeinert werden (z.B. Q1 in Q1.1-Q1.3). Falls eine Frage nicht weiter verfeinert wird, wird ein Maß (Measure) bestimmt, das Daten als Antwort für diese Frage zur Verfügung stellt (siehe Bild 66).

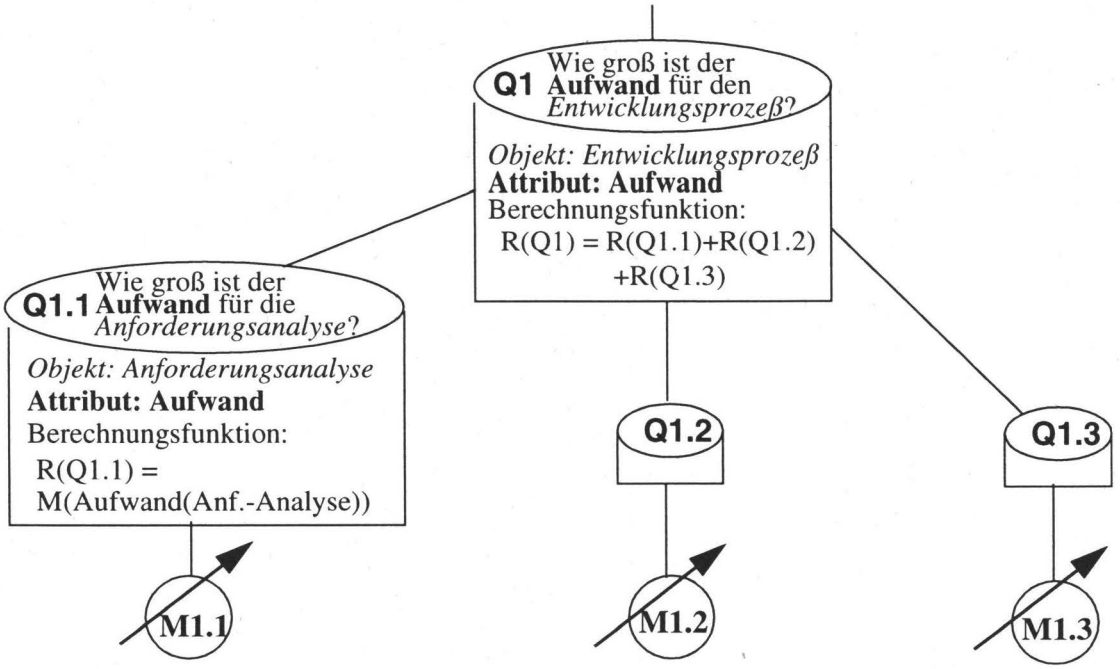

Bild 66: Question-Measure

Diese Betrachtungsweise der Fallstudie ist die Sicht GQM-Pläne. Die Repräsentationssprache ist eine GQM-Plan-Sprache, die im wesentlichen aus Templates für Ziele, Fragen und Maße besteht. Der beschriebene GQM-Plan heißt "Entwicklungsprozeß-Aufwand".

Innerhalb dieser Sichten bestehen semantische Beziehungen zwischen den einzelnen Elementen der Sicht, z.B. muß ein Ziel immer durch mindestens eine Frage operationalisiert werden.

Das Objekt des GQM-Plans "Entwicklungsprozeß-Aufwand" bezieht sich auf ein Prozeßmodell "Entwicklungsprozeß". Dieses befindet sich in der Sicht Prozeßmodelle auf die Fallstudie (siehe Bild 67). Der Entwicklungsprozeß (P) ist demnach eine Aggregation der Prozesse Anforderungsanalyse, Systementwurf und Kodieren (Pi). Die Repräsentationssprache dieser Sicht ist eine Prozeßmodellierungssprache.

Die semantischen Beziehungen zwischen der Sicht Prozeßmodelle und der Sicht GQM-Pläne bestehen darin, daß z.B. die Unterfragen von Q1 gemäß dieses Prozeßmodells erzeugt werden. Für jeden Unterprozeß Pi von P wird jeweils eine Unterfrage Q1.i von Q1 erzeugt, deren Objekt sich wiederum auf Pi bezieht (siehe Bild 66).

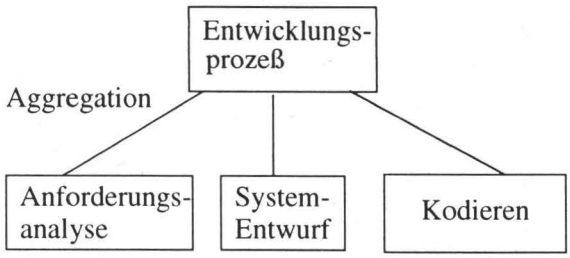

Bild 67: Prozeßmodell "Entwicklungsprozeß"

Der Qualitätsfokus des GQM-Plans "Entwicklungsprozeß-Aufwand" bezieht sich auf ein Qualitätsmodell "Aufwand". Dieses befindet sich in der Sicht Qualitätsmodelle auf die Fallstudie (siehe Bild 68). Danach berechnet sich der Gesamtaufwand eines Prozesses (Proz) falls der Prozeß in Unterprozesse verfeinert wird, rekursiv aus der Summe der Gesamtaufwände dieser Unterprozesse. Falls der Prozeß nicht weiter verfeinert wird, ist der Gesamtaufwand dieses Prozesses gleich dem gemessenen Aufwand für diesen Prozeß. Die Repräsentationssprache dieser Sicht ist eine Programmiersprachen-ähnliche Sprache.

```
GesamtAufwand(Proz) =
  falls Proz verfeinert:
        GesamtAufwand(SubProz)
  sonst:
        M(Aufwand(Proz))
```

Bild 68: Qualitätsmodell "Aufwand"

Die semantischen Beziehungen zwischen der Sicht Qualitätsmodelle und der Sicht GQM-Pläne bestehen darin, daß sich z.B. die Berechnungsfunktion der Fragen aus dem Qualitätsmodell ableitet. Fragen, die nicht weiter verfeinert werden, haben als Berechnungsfunktion den gemessenen Aufwand für das Objekt, auf das sie sich beziehen (ein Maß M). Verfeinerbare Fragen haben als Berechnungsfunktion die Summe der Resultate ihrer Unterfragen (siehe Bild 66 auf Seite 84).

Es liegen also drei Sichten in unterschiedlichen Repräsentationssprachen vor, zwischen denen semantische Beziehungen existieren. Falls ein Element einer Sicht geändert wird, so hat diese Änderung eventuell Auswirkungen auf andere Elemente derselben Sicht, oder auf Elemente der anderen beiden Sichten.

Angenommen, das Prozeßmodell "Entwicklungsprozeß" wird geändert, indem noch ein vierter Subprozeß, nämlich "Benutzbares-System-erstellen" hinzugefügt wird (siehe Bild 69, Sicht Prozeßmodelle). Die syntaktische Intra-Sicht-Konsistenz von der Sicht Prozeßmodelle bleibt erhalten, "Benutzbares-System-erstellen" ist syntaktisch korrekt dargestellt. Die semantische Intra-Sicht-Konsistenz wird auch nicht verletzt, da am Ende eines Entwicklungsprozesses ein benutzbares System erstellt wird.

Bei der Inter-Sicht-Konsistenzprüfung muß nur die Sicht GQM-Pläne (das GOAL wurde aus Platzgründen weggelassen, es bleibt unverändert) betrachtet werden, da keine Überschneidung von der Sicht Prozeßmodelle mit der Sicht Qualitätsmodelle existiert.

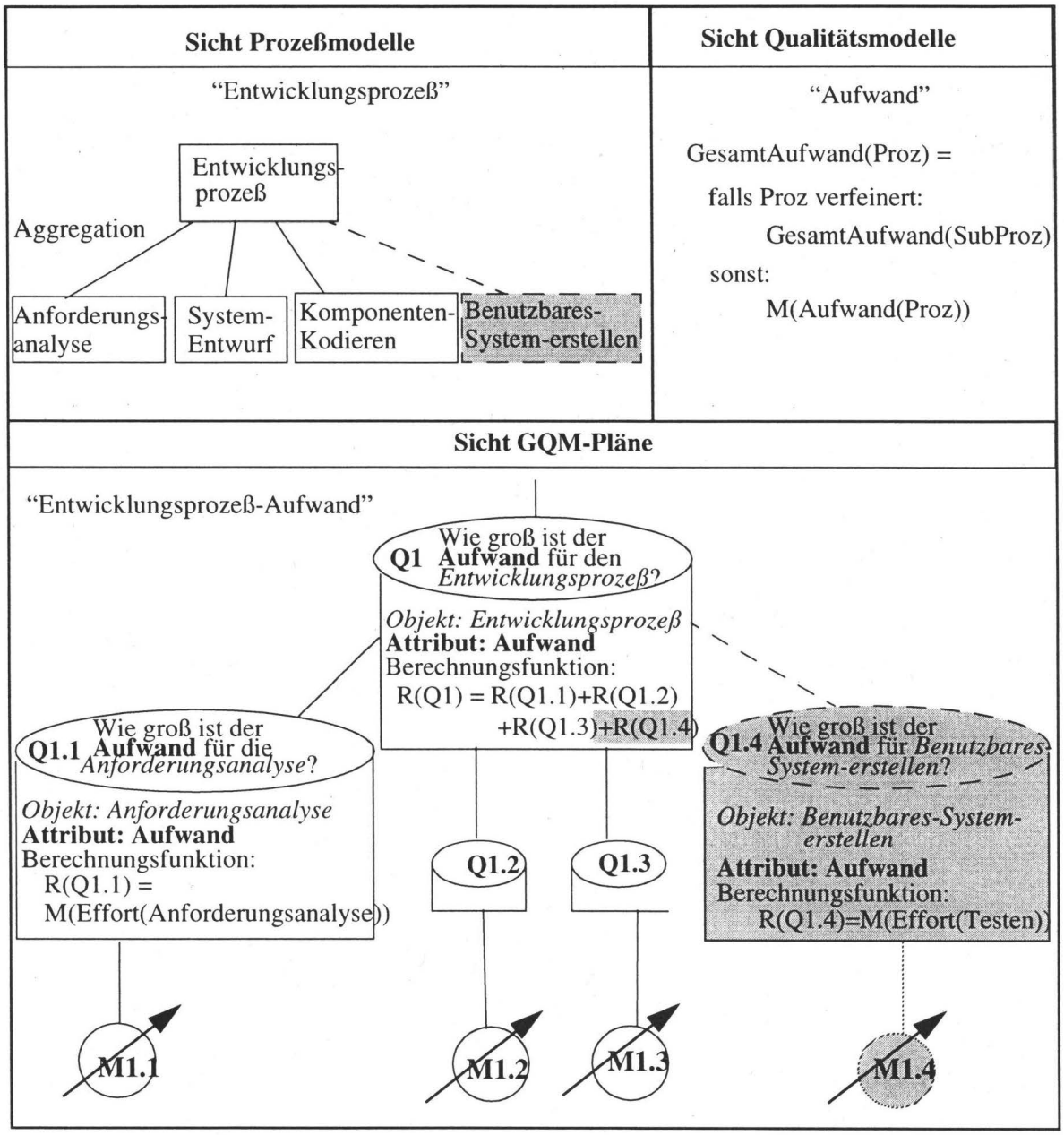

Bild 69: Geänderte, wieder konsistente Sichten - Beispiel Entwicklungsaufwand

Die Inter-Sicht-Konsistenz ist verletzt, da Q1 nun nicht mehr gemäß Prozeßmodell verfeinert ist. Um die Konsistenz wiederherzustellen, kommt in der Sicht GQM-Pläne zunächst eine vierte Unterfrage Q1.4 mit Objekt "Benutzbares-System-erstellen" hinzu, das dazugehörige Maß M1.4, und außerdem muß die Berechnungsfunktion von Q1 um R(Q1.4) ergänzt werden. Da eine Überschneidung von der Sicht GQM-Pläne mit der Sicht Qualitätsmodelle existiert, muß nun auch hier die Inter-Sicht-Konsistenz geprüft werden. Diese wird durch die vorgenommene Änderung nicht verletzt, d.h. es müssen keine Änderungen in der Sicht Qualitätsmodelle ausgeführt werden.

Anhang B

GQM-Plan

Ausschnitte aus [Cha98]

GQM plan
name: "Entwicklungsprozeß-Aufwand"
type: basic
goal
Analyze Entwicklungsprozess
for the purpose of Charakterisierung
with respect to Aufwand
from the viewpoint of Projektmanager
in the context of SEII-Praktikum
result definition: Zweielementiger Vektor, wobei das zweite Element ein 4-elementiger Vektor ist : (Personenstunden, (a, b, c, d))
determine result: Fasse die Ergebnisse von Q_1 und Q_2 zu einem zweielementigen Vektor zusammen
quality focus
quality question Q_1
Which is the Aufwand of the Entwicklungsprozess?
category: quality focus
comment: Wie ist der Aufwand von dem Entwicklungsprozess? Beantwortung der Frage am Ende des Entwicklungsprozesses.
result definition: Personenstunden (rationale Zahl)
determine result: Addiere die Resultate von Q_1.1 - Q_1.4 und dividiere durch 60
result
data origin: measured
value(s):
quality question Q_1.1
Which is the Aufwand of the Prozess System-Anforderungen-bearbeiten?
category: quality focus
comment: Wie ist der Aufwand von dem Prozess System-Anforderungen-bearbeiten?
result definition: Personenminuten
determine result: Addiere die von den einzelnen Personen gelieferten Aufwandsdaten

result
data origin: measured
value(s):
measure M_1.1
attribute: Aufwand
object: Prozess System-Anforderungen-bearbeiten
scale: absolut
range: groesser gleich 0
unit: min
result definition: Personenminuten
determine result: Ermittle den Aufwand jeder an dem Prozess beteiligten Person
result
data origin: measured
value(s):
quality question Q_1.2

Which is the Aufwand of the Prozess System-Entwurf-bearbeiten ?
category: quality focus
comment: Wie ist der Aufwand von dem Prozess System-Entwurf-bearbeiten? Beantwortung der Frage am Ende dieses Prozesses.
result definition: Personenminuten
determine result: Addiere die von den einzelnen Personen gelieferten Aufwandsdaten
result
data origin: measured
value(s):
measure M_1.2.1.1
attribute: Aufwand
object: Prozess System-Entwurf-bearbeiten
scale: absolut
range: groesser gleich 0
unit: min
result definition: Personenminuten
determine result: Ermittle den Aufwand jeder an dem Prozess beteiligten Person
result
data origin: measured
value(s):

quality question Q_1.3
Which is the Aufwand of the Prozess Komponenten-bearbeiten?
category: quality focus
comment: Wie ist der Aufwand von dem Prozess Komponenten-bearbeiten? Beantwortung der Frage am Ende diesesProzesses.
result definition: Personenminuten
determine result: Addiere die von den einzelnen Personen gelieferten Aufwandsdaten
result
data origin: measured
value(s):
measure M_1.3.1.1.1
attribute: Aufwand
object: Prozess Komponenten-bearbeiten
scale: absolut
range: groesser gleich 0
unit: min
result definition: Personenminuten
determine result: Ermittle den Aufwand jeder an den Prozessen beteiligten Person
result
data origin: measured
value(s):

quality question Q_1.4
Which is the Aufwand of the Prozess Benutzbares-System-erstellen?
category: quality focus
comment: Wie ist der Aufwand von dem Prozess Benutzbares-System-erstellen?
result definition: Personenminuten
determine result: Addiere die von den einzelnen Personen gelieferten Aufwandsdaten
result
data origin: measured
value(s):
measure M_1.4.1.1
attribute: Aufwand
object: Prozess Benutzbares-System-erstellen
scale: absolut
range: groesser gleich 0
unit: min
result definition: Personenminuten
determine result: Ermittle den Aufwand jeder an dem Prozess beteiligten Person
result
data origin: measured
value(s):

Anhang C

Fallstudie B

"Objektorientierter Entwurf mit der UML"

Das zweite Beispiel stammt aus dem Bereich "Objektorientierter Entwurf eines Softwaresystems mit der UML" (Unified Modeling Language) [Rat97].

Es sei ein Raum mit einer Lichtquelle gegeben. Das System soll das An- und Ausschalten des Lichts übernehmen. Zunächst existieren informelle Anforderungen, die an das System gestellt werden.

> **Anf1**: Wenn eine Person den Raum betritt (d.h. der Raum ist belegt), und es draußen dunkel ist, soll das System das Licht anschalten.
>
> **Anf2**: Wenn die letzte Person den Raum verläßt (d.h. der Raum ist leer), soll das System das Licht ausschalten.
>
> **Anf3**: Wird es hell draußen, soll das System das Licht ausschalten (wenn es an ist).
>
> **Anf4**: Ist der Raum belegt, und es wird dunkel draußen, soll das System das Licht anschalten.

Bild 70: Informelle Anforderungen

UML ist eine objektorientierte Beschreibungssprache für Anforderungen und Entwurf eines Softwaresystems. Sie ist aus OMT [RBP+91], OOSE [Jac94] und dem Booch-Ansatz [Boo94] entstanden ist. Es existieren verschiedene Diagramme, die verschiedene Sichten auf das System beschreiben. Es gibt unter anderem Use Cases und Sequenzdiagramme, die sich an Szenarien anlehnen, weiterhin Klassendiagramme zur Beschreibung der statischen Aspekte des Systems (wie sieht das Datenmodell aus?) und Zustandsdiagramme, die der Beschreibung der dynamischen Aspekte dienen (welche Zustände durchlaufen Instanzen von Klassen?). Es gibt verschiedene Möglichkeiten, das System auf Entwurfsebene zu strukturieren. Die hier gegebene Struktur führt zu einer Reduktion der Kommunikation im System.

Die Anforderung Anf2 an das Lichtregelungssystem wird durch ein Sequenzdiagramm "Licht aus" (siehe Bild 71) beschrieben. Der Anwesenheitssensor (Instanz der Klasse Sensor) wird alle 5 Sekunden abgefragt. Meldet er "anwesenheit() = leer", ruft die Instanz von Raum die Methode "ausschalten()" der Instanz von Licht auf. Falls das Licht an ist (zustand = an), wird bei der Instanz des Lichtaktors die

Methode "aus()" aktiviert. Die Repräsentationssprache S, die wir hier verwenden, ist die Notation von UML für Sequenzdiagramme.

Eine semantische Beziehung innerhalb der Sicht Sequenzdiagramme ist z.B., daß ein Pfeil nicht ins Leere zeigen darf, sondern immer auf eine Instanz zeigen muß

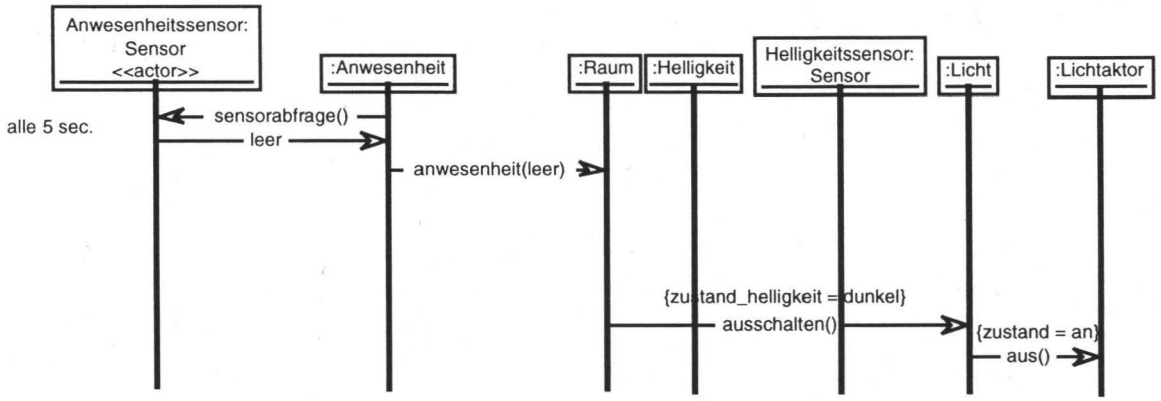

Bild 71: Sequenzdiagramm "Licht aus"

Für das Beispiel existiert ein Klassendiagramm "Lichtregelungssystem" (siehe Bild 72), welches eine Klasse "Raum" umfaßt, die die Attribute "zustand_anwesenheit" und "zustand_helligkeit" und die Methoden "anwesenheit(zustand)" und "helligkeit(zustand)" besitzt. Außerdem existieren die Klassen "Sensor", "Helligkeit" und "Anwesenheit", die zur Überprüfung der Lichtverhältnisse draußen und der Anwesenheit von Personen im Raum benötigt werden. Der "Raum" ist über eine Aggregation mit "Licht" verbunden, d.h. eine Instanz von "Raum" enthält zumindest eine Instanz von "Licht". Weiterhin existiert noch die Klasse "Lichtaktor", die mit ihren Methoden "an()" und "aus()" für die Lichtschaltung zuständig ist.

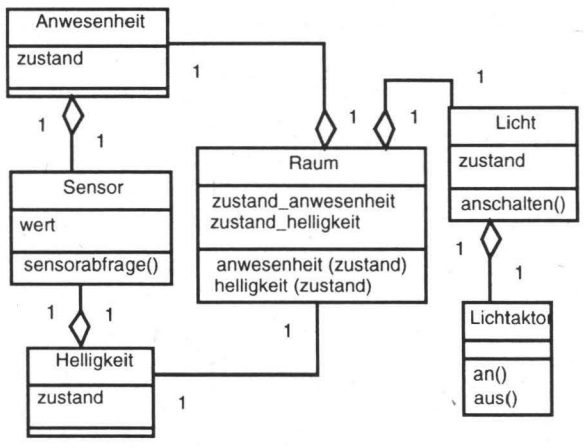

Bild 72: Klassendiagramm "Lichtregelungssystem"

Eine semantische Beziehung zwischen den Sequenzdiagrammen und dem Klassendiagramm, ist z.B. daß Instanzen der Klassen aus dem Klassendiagramm in den Sequenzdiagrammen als Instanzen verwendet werden, z.B. ist Anwesenheitssensor eine Instanz von Sensor. Eine andere Beziehung is beispielsweise, daß Methoden, die eine Klasse im Sequenzdiagramm geschickt bekommt, im Klassendiagramm als Methoden der Klasse definiert sind.

Das dynamische Verhalten einzelner Klassen wird durch Zustandsdiagramme beschrieben. Für das Beispiel gibt es ein Zustandsdiagramm "Raum" (siehe Bild 73) und ein Diagramm "Licht" (siehe Bild 67). Eine Instanz der Klasse "Raum" kann vier verschiedene Zustände annehmen, "leer/hell", "leer/dunkel", "belegt/hell" und "belegt/dunkel" (jeweils "Anwesenheit"/"Lichtverhältnisse draußen"). Der Zustand wird je nach Sensormeldungen gewechselt. z.B. geht eine Instanz von Raum von "belegt/dunkel" nach "leer/dunkel" über, falls von ihr die Methode "anwesenheit(leer)" aufgerufen wird. Außerdem ruft sie selbst die Methode "ausschalten()" von Licht auf. Eine Instanz der Klasse "Licht" kann zwei verschiedene Zustände annehmen, "an" und "aus". Der Übergang zwischen diesen Zuständen geschieht jeweils durch das Aufrufen der Methoden "ausschalten()" und "anschalten()".

Semantische Beziehungen zwischen dem Klassendiagramm und den Zustandsdiagrammen sind z.B.: Übergänge in den Zustandsdiagrammen haben als Label Bedingungen, die Methoden-Aufrufe enthalten. Dieselben Methoden sind im Klassendiagramm definiert, z.B. "ausschalten()" bei "Licht". Klassen im Klassendiagramm haben zugehörige Zustandsdiagramme, falls sie ein dynamisches Verhalten aufweisen, hier für "Raum" und "Licht" dargestellt.

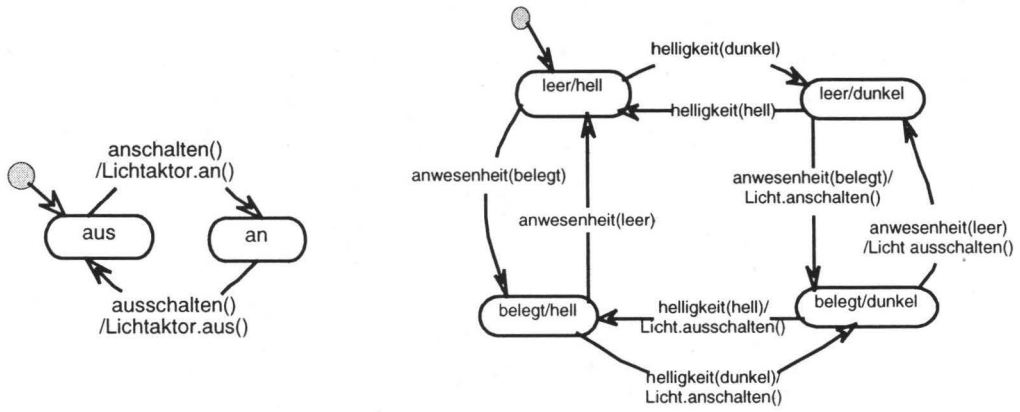

Bild 73: Zustandsübergangsdiagramme "Raum" und "Licht"

Es liegen hier wie in Fallstudie A drei Sichten in unterschiedlichen Repräsentationssprachen vor, zwischen denen semantische Beziehungen existieren: Sequenz-, Klassen- und Zustandsdiagramme. Falls ein Element einer Sicht geändert wird, so hat diese Änderung eventuell Auswirkungen auf andere Elemente derselben Sicht, oder auf Elemente der anderen beiden Sichten.

Falls die Sensoren ständig wechselnde Werte liefern, also z.B. der Helligkeitssensor bei Morgengrauen ständig zwischen "hell" und "dunkel" schwankt, so wird das Licht ständig an- und ausgeschaltet, es flackert. Um dieses Flackern zu vermeiden, werden Anf2 und Anf3 dahingehend geändert, daß, wenn die Person den Raum verläßt (bzw. es draußen hell wird), das Licht erst nach t_1 Minuten ausgeschaltet wird. Wir betrachten hier nur das Sequenzdiagramm "Licht aus" für Anf2. Dort wird eine weitere Instanz "Zeit" benötigt, die jeweils die aktuelle Zeit t_{akt} mitteilt (siehe Bild 74 auf Seite 92). Wird bei "Licht" "ausschalten()" aufgerufen, fragt "Licht" die aktuelle Eintrittszeit t_{entry} mithilfe der Methode "abfrage()" ab (siehe Bild 11, Sicht Sequenzdiagramme). Danach fragt "Licht" alle 30 Sekunden die aktuelle Zeit t_{akt} mithilfe der Methode "abfrage()" ab. Sobald $t_{akt} - t_{entry} > t_1$ Minuten vorbei sind und das Licht vorher an war (zustand = an) und nicht "anschalten()" in der Zwischenzeit aufgerufen wurde, wird die Methode "aus()" von "Lichtaktor" aufgerufen.

Die syntaktische Konsistenz der Sicht Sequenzdiagramme bleibt erhalten, da alle Elemente im Sequenzdiagramm "Licht aus" syntaktisch korrekt dargestellt sind. Die semantische Intra-Sicht-Konsistenz bleibt auch erhalten, da dies eine semantisch sinnvolle Änderung ist und keine Widersprüche im Sequenzdiagramm auftreten.

Die Inter-Sicht-Konsistenz mit der Sicht Zustandsdiagramme wird verletzt, da im Zustandsdiagramm "Licht" immer noch direkt das Licht ausgeschaltet wird. Dies wird behoben, indem ein neuer Zustand "Wartend" eingefügt wird, in dem regelmäßig die Zeit t_{akt} abgefragt wird (Do-Anweisung). Dieser wird erst verlassen (nach "Aus"), falls $t_{akt} - t_{entry} > t_1$ Minuten vorbei sind (siehe Bild 74, Sicht Zustandsdiagramme), oder falls die Methode "anschalten()" aufgerufen wird (nach "An"). Die Intra-Sicht-Konsistenz von der Sicht Zustandsdiagramme wird durch diese Änderung nicht verletzt. Das Zustandsdiagramm "Raum" bleibt unverändert (deshalb in Bild 72 weggelassen).

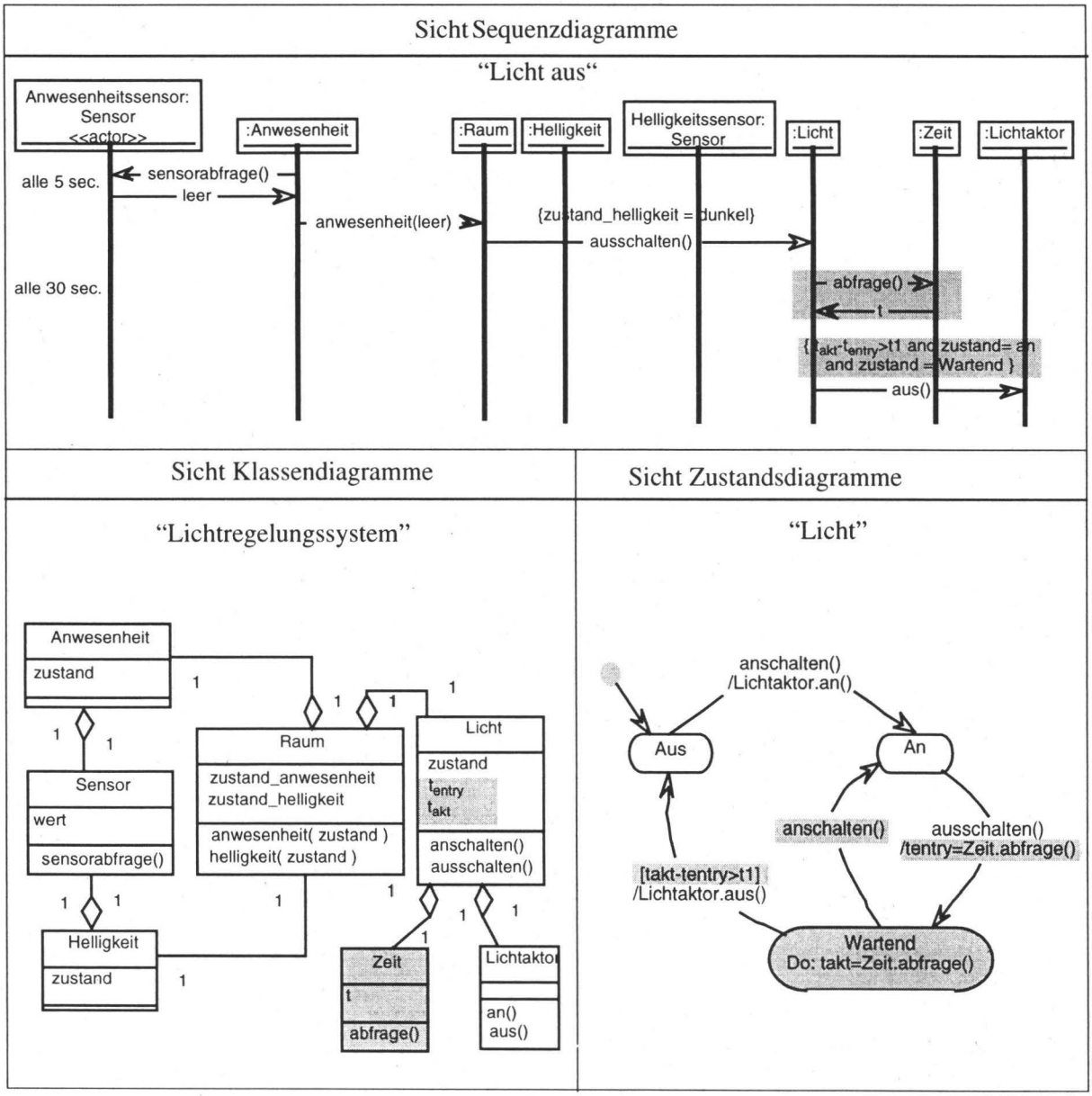

Bild 74: Geänderte, wieder konsistente Sichten - Beispiel Lichtregelungssystem

Da sich die die Sicht Zustandsdiagramme mit der Sicht Klassendiagramme überschneidet, muß zunächst dort die Inter-Sicht-Konsistenz geprüft werden. Sie ist verletzt, da im Klassendiagramm "Lichtregelungssystem" keine Klasse "Zeit" mit der Methode "abfrage()" und einem Attribut "t" existiert. Diese Klasse wird eingefügt (siehe Bild 74, Sicht Klassendiagramme). Außerdem fehlen der Klasse "Licht" die Attribute "t_{entry}" und "t_{akt}". Auch sie werden eingefügt. Die Intra-Sicht-Konsistenz von der Sicht Klassendiagramme wird dadurch nicht verletzt, und die Inter-Sicht-Konsistenz sowohl mit der Sicht Zustandsdiagramme als auch mit der Sicht Sequenzdiagamme ist damit wiederhergestellt.

Anhang D

Telos: Metamodell - GQM

Ausschnitt aus integriertem Metamodell für GQM-und Projektpläne

```
Individual Multiplicity in AssoziationenModul,Class
end

Individual Assoziation in AssoziationenModul,Class with
    attribute
        connection1 : Class;
        connection2 : Class
end

Individual Aggregation in AssoziationenModul,Class isA Assoziation
end

Individual Composite in AssoziationenModul,Class isA Assoziation
end

Individual "(1,1)" in Multiplicity
end

Individual "(1,n)" in Multiplicity
end

Individual "(0,n)" in Multiplicity
end

Individual "(0,1)" in Multiplicity
end

Attribute Assoziation!connection1 in AssoziationenModul!co with
    attribute
        multiplicity : Multiplicity
end

Attribute Assoziation!connection2 in AssoziationenModul!co with
    attribute
        multiplicity : Multiplicity
end

Individual Type in Class
end

Individual Object in Class
end

Individual ResultDefinition in Class
end

Individual GQMPlan in Class with
    attribute
        name : String;
        type : Type
end

Individual DetermineResult in Class
end
```

```
Individual InterpretResult in Class
end

Individual RepresentResult in Class
end

Individual DataOrigin in Class
end

Individual Value in Class
end

Individual Purpose in SimpleClass,GQMModul
end

Individual QualityFocus in Class
end

Individual ViewPoint in Class
end

Individual Context in Assoziation,UML,Class isA Aggregation with
    attribute,connection1
        context : ModelElement
    attribute,connection2
        behavior : StateMachine

end

Individual Category in Class
end

Individual ObjectSetDefinition in Class
end

Individual Scale in Class
end

Individual Range in Class
end

Individual Unit in Class
end

Individual RelatedAspects in Class
end

Individual RangeRestrictions in Class
end

Individual Definition in Class
end

Individual Usage in Class
end

Individual Validity in Class
end

Individual Product in Class isA ProjectPlanElement with
    attribute
        attribute : ProductAttribute
end

Individual Ressource in Class isA ProjectPlanElement with
    attribute
        attribute : RessourceAttribute
end

Individual ContainsG in Assoziation with
    attribute,connection1
        ur_gqm_plan : GQMPlan
    attribute,connection2
        gqm_plan : GQMPlan
end

Individual GQMAttribute in Class
end
```

```
Individual GQMOperation in Class
end

Individual ProcessAttribute in Class
end

Individual ProductAttribute in Class
end

Individual RessourceAttribute in Class
end

Individual Process in Class isA ProjectPlanElement with
    attribute
      attribute : ProcessAttribute;
      entry : Condition;
      exit : Condition
end

Individual RefinesProcess in Assoziation with
    attribute,connection1
      top_process : Process
    attribute,connection2
      process : Process
end

Individual Owns in Assoziation with
    attribute,connection1
      ressource : Ressource
    attribute,connection2
      prozess : Process
end

Individual ConceptProduce in Assoziation with
    attribute,connection1
       process : Process
    attribute,connection2
       product : Product
end

Individual ConceptConsume in Assoziation with
    attribute,connection1
       process : Process
    attribute,connection2
       product : Product
end

Individual Condition in Class
end

Individual RefinesProduct in Assoziation with
    attribute,connection1
       top_product : Product
    attribute,connection2
       product : Product
end

Individual RefinesRessource in Assoziation with
    attribute,connection1
       top_ressource : Ressource
    attribute,connection2
       ressource : Ressource
end

Individual ProjectPlanElement in Class
end

Individual ProjectPlan in Class
end

Individual ProjectPlanContains in Aggregation,Class with
    attribute,connection1
       project_plan : ProjectPlan
    attribute,connection2
```

```
        project_plan_element : ProjectPlanElement
end

Individual PlanElem in Class with
    attribute
        result_definition : ResultDefinition;
        determine_result : DetermineResult;
        interpret_result : InterpretResult;
        object_set_definition : ObjectSetDefinition;
        comment : String;
        object : GQMObject
end

Individual ContainsP in Assoziation with
    attribute,connection1
        gqm_plan : GQMPlan
    attribute,connection2
        plan_element : PlanElem
end

Individual Result in Class with
    attribute
        data_origin : DataOrigin;
        value : Value
end

Individual Has in Assoziation with
    attribute,connection1
        plan_element : PlanElem
    attribute,connection2
        result : Result
end

Individual Measure in Class isA PlanElem with
    attribute
        name : String;
        scale : Scale;
        range : Range;
        unit : Unit
end

Individual Goal in Class isA PlanElem with
    attribute
        purpose : Purpose;
        quality_focus : QualityFocus;
        viewpoint : ViewPoint;
        context : Context
end


Individual Question in Class isA PlanElem with
    attribute
        category : Category;
        attribut : GQMAttribute
end


Individual ProvidesDataFor in Assoziation with
    attribute,connection1
        measure : Measure
    attribute,connection2
        question : Question
end

Individual Operationalizes in Assoziation with
    attribute,connection1
        question : Question
    attribute,connection2
        goal : Goal
end

Individual VariationQuestion in Class isA Question
end
```

```
Individual VBreaksDown in Assoziation with
    attribute,connection1
        ur_variation_question : VariationQuestion
    attribute,connection2
        variation_question : VariationQuestion
end

Individual QualityQuestion in Class isA Question with
    attribute
        related_aspects : RelatedAspects;
        operation : GQMOperation
end

Individual IsUsedBy in Assoziation with
    attribute,connection1
        gqm_plan : GQMPlan
    attribute,connection2
        quality_question : QualityQuestion
end

Individual QBreaksDown in Assoziation with
    attribute,connection1
        ur_quality_question : QualityQuestion
    attribute,connection2
        quality_question : QualityQuestion
end

Individual DependencyRelationship in Assoziation with
    attribute,connection1
        dependent : QualityQuestion
    attribute,connection2
        independent : VariationQuestion
    attribute
        range_restrictions : RangeRestrictions;
        definition : Definition;
        usage : Usage;
        validity : Validity
end

Individual NotExObject in QueryClass,GQMQuery isA PlanElem with
    attribute,constraint
        c : $ not (exists go/GQMObject ((this object go) and (exists p/Process go related_object p))
        or ((this object go) and (exists pr/Product go related_object pr)) )$
end

Individual NoQuestion in QueryClass,GQMQuery isA GQMPlan with
    attribute,constraint
        c : $ not (exists q/QualityQuestion cp/ContainsP (cp plan_element q)
        and (cp gqm_plan this))$
end

Individual NoQuestionRefinement in QueryClass,GQMQuery isA QualityQuestion with
    attribute,constraint
        c : $ not ((exists q/QualityQuestion qbd/QBreaksDown (qbd quality_question this)
        and (qbd ur_quality_question q)) or (exists op/Operationalizes g/Goal (op question this)
        and (op goal g)))$
end

Individual NoVarQuestionRefinement in QueryClass,GQMQuery isA VariationQuestion with
    attribute,constraint
        c : $ not ((exists vq/VariationQuestion vbd/VBreaksDown (vbd variation_question this)
        and (vbd ur_variation_question vq)) or (exists op/Operationalizes g/Goal (op question this)
        and (op goal g)))$
end
```

Anhang E

Telos: GQMPlan

Telos: GQMPlan "Entwicklungsaufwand"

Individual Characterization in Purpose
end

Individual Effort in QualityFocus
end

Individual Projectmanager in ViewPoint
end

Individual SEIIPraktikum in Context
end

Individual M14 in Measure
end

Individual M11 in Measure
end

Individual M12 in Measure
end

Individual M13 in Measure
end

Individual GQMObject in Class
end

Individual GQMProcess in Class isA GQMObject with
  attribute
    related_process : Process
end

Individual GQMProduct in Class isA GQMObject
end

Individual GQMRequirementsAnalysis in Class,GQMProcess with
  attribute,related_process
    gqm_related_process : RequirementsAnalysis
end

Individual RequirementsAnalysis in Process,Class
end

Individual GQMSystemDesign in Class,GQMProcess with
  attribute,related_process
    gqm_related_process : SystemDesign
end

Individual GQMComponentCoding in Class,GQMProcess with
  attribute,related_process
    gqm_related_process : ComponentCoding
end

Individual ComponentCoding in Process,Class
end

```
Individual GQMMakeSystem in Class,GQMProcess with
    attribute,related_process
        gqm_related_process : MakeSystem
end

Individual MakeSystem in Process,Class
end

Individual GQMDevelopmentProcess in Class,GQMProcess with
    attribute,related_process
        gqm_related_process : DevelopmentProcess
end

Individual DevelopmentProcess in Process, Class
end

Individual Q1 in QualityQuestion, with
    attribute,object
        q1_object : GQMDevelopmentProcess
end

Individual MSRefinesProcess in RefinesProcess with
    attribute,top_process
        ms_top_process : DevelopmentProcess
    attribute,process
        ms_process : MakeSystem
end

Individual CCRefinesProcess in RefinesProcess with
    attribute,top_process
        cc_top_process : DevelopmentProcess
    attribute,process
        cc_process : ComponentCoding
end

Individual SDRefinesProcess in RefinesProcess with
    attribute,top_process
        sd_top_process : DevelopmentProcess
    attribute,process
        sd_process : SystemDesign
end

Individual RARefinesProcess in RefinesProcess with
    attribute,top_process
        ra_top_process : DevelopmentProcess
    attribute,process
        ra_process : RequirementsAnalysis
end

Individual Q13 in QualityQuestion with
    attribute,object
        q13_object : GQMComponentCoding
end

Individual Q14 in QualityQuestion with
    attribute,object
        q14_object : GQMMakeSystem
end

Individual Q11 in QualityQuestion with
    attribute,object
        q11_object : GQMRequirementsAnalysis
end

Individual Q12 in QualityQuestion with
    attribute,object
        q12_object : GQMSystemDesign
end

Individual Q11BreaksDown in QBreaksDown with
    attribute,ur_quality_question
        q11_ur_quality_question : Q1
    attribute,quality_question
        q11_quality_question : Q11
end
```

```
Individual Q12BreaksDown in QBreaksDown with
    attribute,ur_quality_question
        q11_ur_quality_question : Q1
    attribute,quality_question
        q11_quality_question : Q12
end

Individual Q14BreaksDown in QBreaksDown with
    attribute,ur_quality_question
        q11_ur_quality_question : Q1
    attribute,quality_question
        q11_quality_question : Q14
end

Individual Q13BreaksDown in QBreaksDown with
    attribute,ur_quality_question
        q11_ur_quality_question : Q1
    attribute,quality_question
        q11_quality_question : Q13
end

Individual M11ProvidesDataFor in ProvidesDataFor with
    attribute,question
        m11_pdf_question : Q11
    attribute,measure
        m11_pdf_measure : M11
end

Individual M12ProvidesDataFor in ProvidesDataFor with
    attribute,question
        m12_pdf_question : Q12
    attribute,measure
        m12_pdf_measure : M12
end

Individual M13ProvidesDataFor in ProvidesDataFor with
    attribute,question
        m13_pdf_question : Q13
    attribute,measure
        m13_pdf_measure : M13
end

Individual M14ProvidesDataFor in ProvidesDataFor with
    attribute,question
        m14_pdf_question : Q14
    attribute,measure
        m14_pdf_measure : M14
end

Individual EffortDevelopmentGoal in Goal with
    attribute,object
        edg_object : GQMDevelopmentProcess
    attribute,purpose
        edg_purpose : Characterization
    attribute,viewpoint
        edg_viewpoint : Projectmanager
    attribute,quality_focus
        edg_quality_focus : Effort
    attribute,context
        edg_context : SEIIPraktikum
end

Individual Q1Operationalizes in Operationalizes with
    attribute,goal
        q1_op_goal : EffortDevelopmentGoal
    attribute,question
        q1_op_question : Q1
end
```

Anhang F

Telos: Metamodell - UML

Telos: Ausschnitt aus dem integriertem UML-Metamodell

```
Individual Context in Assoziation,UML,Class isA Aggregation with
    attribute,connection1
        context : ModelElement
    attribute,connection2
        behavior : StateMachine
end

Individual ModelElement in UML,Class isA Element with
    attribute
        name : Name
end

Individual Feature in UML,Class isA ModelElement with
    attribute
        ownerscope : ScopeKind;
        visibility : VisibilityKind
end

Individual Namespace in UML,Class isA ModelElement
end

Individual Parameter in UML,Class isA ModelElement with
    attribute
        default_value : Expression;
        kind : ParameterDirectionKind
end

Individual BehavorialFeature in UML,Class isA Feature with
    attribute
        is_query : Boolean
end

Individual StructuralFeature in UML,Class isA Feature with
    attribute
        multiplicity : UMLMultiplicity;
        changeable : ChangeableKind;
        target_scope : ScopeKind
end

Individual GeneralizableElement in UML,Class isA Namespace with
    attribute
        is_root : Boolean;
        is_leaf : Boolean;
        is_abstract : Boolean
end

Individual Classifier in UML,Class isA GeneralizableElement
end

Individual VisibilityKind in UML,Class
end
```

```
Individual ElementOwnership in UML,Composite,Class with
    attribute
        visibility : VisibilityKind
    attribute,connection1
        namespace : Namespace
    attribute,connection2
        ownedElement : ModelElement
end

Individual Name in UML,Class
end

Individual ScopeKind in UML,Class
end

Individual ChangeableKind in UML,Class
end

Individual Expression in UML,Class
end

Individual ParameterDirectionKind in UML,Class
end

Individual Uninterpreted in UML,Class
end

Individual CallConcurrencyKind in UML,Class
end

Individual Param in UML,Composite,Class with
    attribute,connection2
        parameter : Parameter
    attribute,connection1
        behavorial_feature : BehavorialFeature
end

Individual FeatureOwnership in UML,Composite,Class with
    attribute,connection2
        feature : Feature
    attribute,connection1
        owner : Classifier
end

Individual ParamType in Assoziation,UML,Class with
    attribute,connection2
        param : Parameter
    attribute,connection1
        type : Classifier
end

Individual StructuralFeatureType in Assoziation,UML,Class with
    attribute,connection2
        feature : StructuralFeature
    attribute,connection1
        type : Classifier
end

Individual UMLMultiplicity in UML,Class
end

Individual UMLOperation in UML,Class isA BehavorialFeature with
    attribute
        specification : Uninterpreted;
        is_polymorphic : Boolean;
        concurrency : CallConcurrencyKind
end

Individual UMLAttribute in UML,Class isA StructuralFeature with
    attribute
        initial_value : Expression
end

Individual UMLClass in UML,Class isA Classifier with
    attribute
```

```
        is_active : Boolean
end

Individual AssociationEnd in UML,Class isA ModelElement with
    attribute
        isNavigable : Boolean;
        isOrdered : Boolean;
        aggregation : AggregationKind;
        multiplicity : UMLMultiplicity;
        changeable : ChangeableKind;
        targetScope : ScopeKind
end

Individual AggregationKind in UML,Class
end

Individual UMLAssociation in UML,Class isA GeneralizableElement
end

Individual AssociationClass in UML,Class isA UMLClass,UMLAssociation
end

Individual Generalization in UML,Class isA ModelElement with
    attribute
        diskriminator : Name
end

Individual Specification in Assoziation,UML,Class with
    attribute,connection1
        specification : Classifier;
        association_end : AssociationEnd
end

Individual SubType in Assoziation,UML,Class with
    attribute,connection1
        generalization : Generalization
    attribute,connection2
        subtype : GeneralizableElement
end

Individual SuperType in Assoziation,UML,Class with
    attribute,connection1
        specialization : Generalization
    attribute,connection2
        supertype : GeneralizableElement
end

Individual Action in UML,Class isA ModelElement with
    attribute
        recurrence : Expression;
        target : ObjectSetExpression;
        is_asynchronous : Boolean;
        script : String
end

Individual Argument in UML,Class isA ModelElement with
    attribute
        value : Expression
end

Individual ReturnAction in UML,Class isA Action
end

Individual Request in UML,Class
end

Individual CallAction in UML,Class isA Action with
    attribute
        mode : SynchronousKind
end

Individual ActArg in Assoziation,UML,Class isA Composite with
    attribute,connection1
        action : Action
    attribute,connection2
```

```
        argument : Argument
end

Individual ActRequ in Assoziation,UML,Class isA Composite with
    attribute,connection1
        action : Action
    attribute,connection2
        request : Request
end

Individual Event in UML,Class isA ModelElement
end

Individual CallEvent in UML,Class isA Event
end

Individual End in Assoziation,UML,Class isA Composite with
    attribute,connection1
        association : UMLAssociation
    attribute,connection2
        connection : AssociationEnd
end

Individual Occurence in Assoziation,UML,Class with
    attribute,connection1
        occurence : CallEvent
    attribute,connection2
        operation : UMLOperation
end

Individual StateMachine in UML,Class isA ModelElement
end

Individual Guard in UML,Class isA ModelElement with
    attribute
        expression : BooleanExpression
end

Individual Transition in UML,Class isA ModelElement
end

Individual StateVertex in UML,Class isA ModelElement
end

Individual BooleanExpression in UML,Class
end

Individual State in UML,Class isA StateVertex
end

Individual SimpleState in UML,Class isA State
end

Individual ActionSequence in UML,Class
end

Individual ExtTrans in Assoziation,UML,Class isA Composite with
    attribute,connection1
        state_machine : StateMachine
    attribute,connection2
        transition : Transition
end

Individual TransGuard in Assoziation,UML,Class isA Composite with
    attribute,connection1
        transition : Transition
    attribute,connection2
        guard : Guard
end

Individual Effect in Assoziation,UML,Class isA Composite with
    attribute,connection1
        transition : Transition
    attribute,connection2
        effect : ActionSequence
end
```

```
Individual Sequ in Assoziation,UML,Class isA Composite with
    attribute,connection1
        action_sequence : ActionSequence
    attribute,connection2
        action : Action
end

Individual Entry in Assoziation,UML,Class isA Composite with
    attribute,connection1
        state : State
    attribute,connection2
        entry : ActionSequence
end

Individual Exit in Assoziation,UML,Class isA Composite with
    attribute,connection1
        state : State
    attribute,connection2
        exit : ActionSequence
end

Individual Top in Assoziation,UML,Class isA Composite with
    attribute,connection1
        state_machine : StateMachine
    attribute,connection2
        top : State
end

Individual Trigger in Assoziation,UML,Class isA Aggregation with
    attribute,connection1
        transition : Transition
    attribute,connection2
        trigger : Event
end

Individual Incoming in Assoziation,UML,Class with
    attribute,connection1
        target : StateVertex
    attribute,connection2
        incoming : Transition
end

Individual Outgoing in Assoziation,UML,Class with
    attribute,connection1
        source : StateVertex
    attribute,connection2
        outgoing : Transition
end

Individual DeferredEv in Assoziation,UML,Class with
    attribute,connection1
        state : State
    attribute,connection2
        deferred_event : Event
end

Individual Aggregate in UML,AggregationKind
end

Individual None in UML,AggregationKind
end

Individual UMLComposite in UML,AggregationKind
end

Individual Collaboration in UML,Class isA Namespace
end

Individual Interaction in UML,Class isA ModelElement
end

Individual AssociationEndRole in UML,Class isA AssociationEnd
end
```

```
Individual AssociationRole in UML,Class isA UMLAssociation with
    attribute
        multiplicity : UMLMultiplicity
end

Individual ClassifierRole in UML,Class isA Classifier with
    attribute
        multiplicity : UMLMultiplicity
end

Individual Message in UML,Class
end

Individual EndRole in Assoziation,UML,Class isA Composite with
    attribute,connection1
        association_role : AssociationRole
    attribute,connection2
        connection : AssociationEndRole
end

Individual CollOwnedElement in Assoziation,UML,Class isA Composite with
    attribute,connection1
        collaboration : Collaboration
    attribute,connection2
        owned_element : AssociationRole
end

Individual ClassOwnedElement in Assoziation,UML,Class isA Composite with
    attribute,connection1
        collaboration : Collaboration
    attribute,connection2
        owned_element : ClassifierRole
end

Individual CollInteraction in Assoziation,UML,Class isA Composite with
    attribute,connection1
        context : Collaboration
    attribute,connection2
        interaction : Interaction
end

Individual InterMess in Assoziation,UML,Class isA Aggregation with
    attribute,connection1
        interaction : Interaction
    attribute,connection2
        message : Message
end

Individual ActionMess in Assoziation,UML,Class with
    attribute,connection1
        action : Action
    attribute,connection2
        message : Message
end

Individual Precedessor in Assoziation,UML,Class with
    attribute,connection1
        precedessor : Message
    attribute,connection2
        message : Message
end

Individual Activator in Assoziation,UML,Class with
    attribute,connection1
        activator : Message
    attribute,connection2
        message : Message
end

Individual Sender in Assoziation,UML,Class with
    attribute,connection1
        sender : ClassifierRole
    attribute,connection2
```

```
        message : Message
end

Individual Receiver in Assoziation,UML,Class with
    attribute,connection1
        receiver : ClassifierRole
    attribute,connection2
        message : Message
end

Individual AssClassRole in Assoziation,UML,Class with
    attribute,connection1
        type : ClassifierRole
    attribute,connection2
        association_end_role : AssociationEndRole
end

Individual BaseEnd in Assoziation,UML,Class with
    attribute,connection1
        association_end : AssociationEnd
    attribute,connection2
        association_end_role : AssociationEndRole
end

Individual Base in Assoziation,UML,Class with
    attribute,connection1
        association : UMLAssociation
    attribute,connection2
        association_role : AssociationRole
end

Individual BaseClass in Assoziation,UML,Class with
    attribute,connection1
        classifier : Classifier
    attribute,connection2
        classifier_role : ClassifierRole
end

Individual RepresentedClassifier in Assoziation,UML,Class with
    attribute,connection1
        represented_classifier : Classifier
    attribute,connection2
        collaboration : Collaboration
end

Individual RepresentedOperation in Assoziation,UML,Class with
    attribute,connection1
        represented_operation : UMLOperation
    attribute,connection2
        collaboration : Collaboration
end

Individual ConstrainingElement in Assoziation,UML,Class with
    attribute,connection1
        constraining_element : ModelElement
    attribute,connection2
        collaboration : Collaboration
end

Individual StateMachine in Meta Class with
    attribute,constraint
        c: $ (exists b/BehavorialFeature this context b) or
        (exists c/Classifier this context c) $
end
```

```
Individual MyECA in ECArule with
    attribute, ecarule
        my_eca: $ x/Process
        ON UNTELL in(x,Process)
        IF in(y,Question)
        DO ASK
        UNTELL in(y,Question) $
    attribute, active
        my_act: True
end
```

Anhang G

Telos - Lichtregelungssystem

Telos: Lichtregelungssystem (UML)

Individual Sensor in UML,UMLClass
end

Individual Anwesenheit in UML,UMLClass
end

Individual Helligkeit in UML,UMLClass
end

Individual Raum in UML,UMLClass
end

Individual Licht in UML,UMLClass
end

Individual Lichtaktor in UML,UMLClass
end

Individual AnwZustand in UML,UMLAttribute
end

Individual HellZustand in UML,UMLAttribute
end

Individual Wert in UML,UMLAttribute
end

Individual ZustandAnwesenheit in UML,UMLAttribute
end

Individual ZustandHelligkeit in UML,UMLAttribute
end

Individual LichtZustand in UML,UMLAttribute
end

Individual AnwStructuralFeatureAnwZustand in UML,StructuralFeatureType with
attribute,feature
anwfeature : AnwZustand
attribute,type
anwtype : Anwesenheit
end

Individual HellStructuralFeatureHellZustand in UML,StructuralFeatureType with
attribute,feature
hellfeature : HellZustand
attribute,type
helltype : Helligkeit
end

Individual SensorStructuralFeatureWert in UML,StructuralFeatureType with
attribute,feature
sensorfeature : Wert
attribute,type

```
        sensortype : Sensor
end

Individual RaumStructuralFeatureZustAnw in UML,StructuralFeatureType with
    attribute,feature
        raum_feature : ZustandAnwesenheit
    attribute,type
        raum_type : Raum
end

Individual RaumStructuralFeatureZustHell in UML,StructuralFeatureType with
    attribute,feature
        raum_feature : ZustandHelligkeit
    attribute,type
        raum_type : Raum
end

Individual LichtStructuralFeatureLichtZust in UML,StructuralFeatureType with
    attribute,feature
        licht_feature : LichtZustand
    attribute,type
        licht_type : Licht
end

Individual SensorAbfrage in UML,UMLOperation
end

Individual AnwesenheitOp in UML,UMLOperation
end

Individual HelligkeitOp in UML,UMLOperation
end

Individual Anschalten in UML,UMLOperation
end

Individual Ausschalten in UML,UMLOperation
end

Individual SensorFeatureOwnershipAbfrage in UML,FeatureOwnership with
    attribute,feature
        sensor_feature : SensorAbfrage
    attribute,owner
        sensor_owner : Sensor
end

Individual RaumFeatureOwnershipAnwOp in UML,FeatureOwnership with
    attribute,feature
        raum_feature : AnwesenheitOp
    attribute,owner
        raum_owner : Raum
end

Individual RaumFeatureOwnershipHellOp in UML,FeatureOwnership with
    attribute,feature
        raum_feature : HelligkeitOp
    attribute,owner
        raum_owner : Raum
end

Individual LichtFeatureOwnershipAnschalten in UML,FeatureOwnership with
    attribute,feature
        licht_feature : Anschalten
    attribute,owner
        licht_owner : Licht
end

Individual LichtFeatureOwnershipAusschalten in UML,FeatureOwnership with
    attribute,feature
        licht_feature : Ausschalten
    attribute,owner
        licht_owner : Licht
end
```

```
Individual LichtaktorFeatureOwnershipAn in UML,FeatureOwnership with
    attribute,feature
        lichtaktor_feature : An
    attribute,owner
        lichtaktor_owner : Lichtaktor
end

Individual LichtaktorFeatureOwnershipAus in UML,FeatureOwnership with
    attribute,feature
        lichtaktor_feature : Aus
    attribute,owner
        lichtaktor_owner : Lichtaktor
end

Individual SensorAnw in UML,UMLAssociation
end

Individual SensorHell in UML,UMLAssociation
end

Individual RaumHell in UML,UMLAssociation
end

Individual RaumAnw in UML,UMLAssociation
end

Individual RaumLicht in UML,UMLAssociation
end

Individual LichtaktorLicht in UML,UMLAssociation
end

Individual SensorAnwEnd1 in UML,AssociationEnd with
    aggregation,attribute
        anw_aggregation : Aggregate
end

Individual SensorAnwEnd2 in UML,AssociationEnd with
    aggregation,attribute
        sensor_aggregation : None
end

Individual SensorHellEnd2 in UML,AssociationEnd with
    aggregation,attribute
        sensor_aggregation : None
end

Individual SensorHellEnd1 in UML,AssociationEnd with
    aggregation,attribute
        hell_aggregation : Aggregate
end

Individual RaumHellEnd1 in UML,AssociationEnd with
    aggregation,attribute
        raum_aggregation : None
end

Individual RaumHellEnd2 in UML,AssociationEnd with
    aggregation,attribute
        hell_aggregation : None
end

Individual RaumAnwEnd2 in UML,AssociationEnd with
    aggregation,attribute
        anw_aggregation : None
end

Individual RaumAnwEnd1 in UML,AssociationEnd with
    aggregation,attribute
        raum_aggregation : Aggregate
end

Individual RaumLichtEnd1 in UML,AssociationEnd with
    aggregation,attribute
        raum_aggregation : Aggregate
end
```

```
Individual LichtaktorLichtEnd2 in UML,AssociationEnd with
    aggregation,attribute
        lichtaktor_aggregation : None
end

Individual LichtaktorLichtEnd1 in UML,AssociationEnd with
    aggregation,attribute
        licht_aggregation : Aggregate
end

Individual RaumLichtEnd2 in UML,AssociationEnd with
    aggregation,attribute
        licht_aggregation : None
end

Individual LLE1 in UML,End with
    association,attribute
        lle1_association : LichtaktorLicht
    attribute,connection
        lle1_connection : LichtaktorLichtEnd1
end

Individual RAE2 in UML,End with
    association,attribute
        rae2_association : RaumAnw
    attribute,connection
        rae2_connection : RaumAnwEnd2
end

Individual RAE1 in UML,End with
    association,attribute
        rae1_association : RaumAnw
    attribute,connection
        rae1_connection : RaumAnwEnd1
end

Individual RHE1 in UML,End with
    association,attribute
        rhe1_association : RaumHell
    attribute,connection
        rhe1_connection : RaumHellEnd1
end

Individual RHE2 in UML,End with
    association,attribute
        rhe2_association : RaumHell
    attribute,connection
        rhe2_connection : RaumHellEnd2
end

Individual RLE2 in UML,End with
    association,attribute
        rle2_association : RaumLicht
    attribute,connection
        rle2_connection : RaumLichtEnd2
end

Individual RLE1 in UML,End with
    association,attribute
        rle1_association : RaumLicht
    attribute,connection
        rle1_connection : RaumLichtEnd1
end

Individual SAE1 in UML,End with
    association,attribute
        sae1_association : SensorAnw
    attribute,connection
        sae1_connection : SensorAnwEnd1
end

Individual SAE2 in UML,End with
    association,attribute
        sae2_association : SensorAnw
    attribute,connection
        sae2_connection : SensorAnwEnd2
end
```

```
Individual SHE2 in UML,End with
    association,attribute
        she2_association : SensorHell
    attribute,connection
        she2_connection : SensorHellEnd2
end

Individual SHE1 in UML,End with
    association,attribute
        she1_association : SensorHell
    attribute,connection
        she1_connection : SensorHellEnd1
end

Individual LLE2 in UML,End with
    association,attribute
        lle2_association : LichtaktorLicht
    attribute,connection
        lle2_connection : LichtaktorLichtEnd2
end

Individual TypeAss in Assoziation,UML,Class with
    attribute,connection1
        type : Classifier
    attribute,connection2
        participant : AssociationEnd
end

Individual AnwSensorAnw in UML,TypeAss with
    attribute,type
        anw_type : Anwesenheit
    attribute,participant
        anw_participant : SensorAnwEnd1
end

Individual HellSensorHell in UML,TypeAss with
    attribute,type
        hell_type : Helligkeit
    attribute,participant
        hell_participant : SensorHellEnd1
end

Individual SensSensorHell in UML,TypeAss with
    attribute,type
        sens_type : Sensor
    attribute,participant
        sens_participant : SensorHellEnd2
end

Individual SensSensorAnw in UML,TypeAss with
    attribute,type
        sens_type : Sensor
    attribute,participant
        sens_participant : SensorAnwEnd2
end

Individual RaumRaumAnw in UML,TypeAss with
    attribute,type
        raum_type : Raum
    attribute,participant
        raum_participant : RaumAnwEnd1
end

Individual AnwRaumAnw in UML,TypeAss with
    attribute,type
        anw_type : Anwesenheit
    attribute,participant
        anw_participant : RaumAnwEnd2
end

Individual LichtRaumLicht in UML,TypeAss with
    attribute,type
        licht_type : Licht
    attribute,participant
```

```
        licht_participant : RaumLichtEnd2
end

Individual RaumRaumLicht in UML,TypeAss with
    attribute,type
        raum_type : Raum
    attribute,participant
        raum_participant : RaumLichtEnd1
end

Individual HellRaumHell in UML,TypeAss with
    attribute,type
        hell_type : Helligkeit
    attribute,participant
        hell_participant : RaumHellEnd2
end

Individual RaumRaumHell in UML,TypeAss with
    attribute,type
        raum_type : Raum
    attribute,participant
        raum_participant : RaumHellEnd1
end

Individual LLichtaktorLicht in UML,TypeAss with
    attribute,type
        licht_type : Licht
    attribute,participant
        licht_participant : LichtaktorLichtEnd1
end

Individual LaLichtaktorLicht in UML,TypeAss with
    attribute,type
        lichtaktor_type : Lichtaktor
    attribute,participant
        lichtaktor_participant : LichtaktorLichtEnd2
end

Individual LeerHell in UML,State
end

Individual LeerDunkel in UML,State
end

Individual BelegtDunkel in UML,State
end

Individual BelegtHell in UML,State
end

Individual Aus in UML,UMLOperation,State

end

Individual An in UML,UMLOperation,State
end

Individual AnAus in UML,Transition
end

Individual AusAn in UML,Transition
end

Individual LichtSM in UML,StateMachine
end

Individual RaumSM in UML,StateMachine
end

Individual AnIncoming in UML,Incoming with
    attribute,target
        an_target : An
    attribute,incoming
        an_incoming : AusAn
end
```

```
Individual AusIncoming in UML,Incoming with
    attribute,target
        aus_target : Aus
    attribute,incoming
        aus_incoming : AnAus
end

Individual AusOutgoing in UML,Outgoing with
    attribute,source
        aus_source : Aus
    attribute,outgoing
        aus_outging : AusAn
end

Individual AnOutgoing in UML,Outgoing with
    attribute,source
        an_source : An
    attribute,outgoing
        an_outging : AnAus
end

Individual TopLichtSM in UML,Top with
    attribute,state_machine
        licht_state_machine : LichtSM
    attribute,top
        licht_top : Aus
end

Individual LHTransLD in UML,Transition
end

Individual LHTransBH in UML,Transition
end

Individual BHTransBD in UML,Transition
end

Individual BHTransLH in UML,Transition
end

Individual BDTransBH in UML,Transition
end

Individual BDTransLD in UML,Transition
end

Individual LDTransLH in UML,Transition
end

Individual LDTransBD in UML,Transition
end

Individual LHOutgoingBH in UML,Outgoing with
    attribute,source
        lh_source : LeerHell
    attribute,outgoing
        lh_outgoing : LHTransBH
end

Individual LHOutgoingLD in UML,Outgoing with
    attribute,source
        lh_source : LeerHell
    attribute,outgoing
        lh_outgoing : LHTransLD
end

Individual LDOutgoingBD in UML,Outgoing with
    attribute,source
        ld_source : LeerDunkel
    attribute,outgoing
        ld_outgoing : LDTransBD
end

Individual LDOutgoingLH in UML,Outgoing with
    attribute,source
```

```
        ld_source : LeerDunkel
    attribute,outgoing
        ld_outgoing : LDTransLH
end

Individual BDOutgoingBH in UML,Outgoing with
    attribute,source
        bd_source : BelegtDunkel
    attribute,outgoing
        bd_outgoing : BDTransBH
end

Individual BDOutgoingLD in UML,Outgoing with
    attribute,source
        bd_source : BelegtDunkel
    attribute,outgoing
        bd_outgoing : BDTransLD
end

Individual BHOutgoingBD in UML,Outgoing with
    attribute,source
        bh_source : BelegtHell
    attribute,outgoing
        bh_outgoing : BHTransBD
end

Individual BHOutgoingLH in UML,Outgoing with
    attribute,source
        bh_source : BelegtHell
    attribute,outgoing
        bh_outgoing : BHTransLH
end

Individual TopRaumSM in UML,Top with
    attribute,state_machine
        raum_state_machine : RaumSM
    attribute,top
        raum_top : LeerHell
end

Individual LHIncomingLD in UML,Incoming with
    attribute,target
        lh_target : LeerHell
    attribute,incoming
        lh_incoming : LDTransLH
end

Individual LDIncomingBD in UML,Incoming with
    attribute,target
        ld_target : LeerDunkel
    attribute,incoming
        ld_incoming : BDTransLD
end

Individual LHIncomingBH in UML,Incoming with
    attribute,target
        lh_target : LeerHell
    attribute,incoming
        lh_incoming : BHTransLH
end

Individual BHIncomingLH in UML,Incoming with
    attribute,target
        bh_target : BelegtHell
    attribute,incoming
        bh_incoming : LHTransBH
end

Individual LDIncomingLH in UML,Incoming with
    attribute,target
        ld_target : LeerDunkel
    attribute,incoming
        ld_incoming : LHTransLD
end
```

```
Individual BDIncomingLD in UML,Incoming with
    attribute,target
        bd_target : BelegtDunkel
    attribute,incoming
        bd_incoming : LDTransBD
end

Individual BDIncomingBH in UML,Incoming with
    attribute,target
        bd_target : BelegtDunkel
    attribute,incoming
        bd_incoming : BHTransBD
end

Individual BHIncomingBD in UML,Incoming with
    attribute,target
        bh_target : BelegtHell
    attribute,incoming
        bh_incoming : BDTransBH
end
```

Literaturverzeichnis

[Atk97] Colin Atkinson: *Meta-Modeling for Distributed Object Environments.* First International Enterprise Distributed Object Computing Workshop (EDOC'97), Brisbane, Australien, Oct. 1997.

[Atk98] C. Atkinson: *Supporting and Applying the UML Conceptual Framework.* UML'98 International Workshop, June, 1998.

[BA96] Shawn Bohner, Robert Arnold: *Software Change Impact Analysis.* IEEE Computer Society Press, Los Alamitos, California, 1996.

[Bas95] Victor Basili: *Applying the Goal/Question/Metric Paradigm in the Experience Factory.* In: Software Quality Assurance and Measurement: A Worldwide Perspective, International Thompson Computer Press, 1995.

[BC85] Joachim Biskup, Bernhard Convent: *A formal View Integration Method.* Universität Dortmund, Informatik Forschungsbericht 208, Sept. 1985.

[BDM88] F. Bry, H. Decker, R. Manthey: *A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases.* Proceedings EDBT'88, p.488-505, Venedig, 1988.

[Boo94] G. Booch: *Object-oriented Analysis and Design. With Applications.* Second Edition, Benjamin/Cummings Publishing Company 1994.

[Bro98] Kerstin Brockhage: *Suche nach einem Verfolgbarkeitsansatz zur Verbesserung von Änderbarkeit.* Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, Oktober 1998.

[CG95] Marsha Chechik, John Gannon: *Automatic Analysis of Consistency between Requirements and Designs.* Technical Report CS-TR-3394, Dept. of CS, University of Maryland, College Park, 1995.

[CG96] Marsha Chechik, John Gannon: *Verification of Consistency between Concurrent Program Designs and their Requirements* Technical Report CS-TR-3549, Dept. of CS, University of Maryland, College Park, 1996.

[CG98] ConceptualGraphs.com: *ConceptualGraphs.* http://ksi.cpsc.ucalgary.ca/~lukose/cseic, Jan. 1998.

[Cha98] Eleni Chaitidou: *Erstellung von Projekt- und GQM-Plänen für ein Software-Entwicklungsprojekt.* Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern, September 1998.

[EC94] Jürgen Ebert, Martin Carstensen: *Ansatz und Architektur von KOGGE Version 1.0.* Interner Projektbericht 2/94, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1994.

[EFK+94] S. Easterbrook, A. Finkelstein, J. Kramer, B. Nuseibeh: *Co-ordinating Distributed View-Points: the anatomy of a consistency check.* Technical Report 94/7, Department of Computing, Imperial College, London, 1994.

[EJJ+89] Stefan Eherer, Matthias Jarke, Manfred Jeusfeld, Andreas Miethsam, Thomas Rose: *A KBMS For Database Software Evolution: ConceptBase V2.0 User Manual.* MIP 8936,

Universität Passau, 1989.

[ES97] J. Ebert, R. Süttenbach: *An OMT Metamodel.* Universität Koblenz-Landau, Fachbericht Informatik 13/97, 1997.

[EWD+96] J. Ebert, A. Winter, P. Dahm, A. Franzke, R. Süttenbach: *Graph Based Modeling and Implementation with EER/GRAL.*. Proceedings of the 15th International Conference on Conceptual Modeling, Cottbus, Oct. 1996.

[Fla98] Rony G. Flatscher: *Specification (and Implementation) of the EIA/CDIF Meta-Meta-Model for Relational Database Management Systems (ORACLE).* http://dustbin.informatik.uni-bremen.de/umlbib/authors/FlatscherRony.html.

[FP92] C. Francalanci, B. Pertini: *Object-oriented view integration to support reuse of requirements specification.* In: Proceedings of the Second International Computer Science Conference, p. 544-552, Dez. 1992.

[Har87] D. Harel: *Algorithmics: the spirit of computing.* Addison-Wesley, 1987.

[Här97] T. Härder: *Datenbanksysteme I.* Script zur Vorlesung WS 1997/98, Fachbereich Informatik, Universität Kaiserslautern, 1997.

[HB98] B. Henderson-Sellers, A. Bulthuis: *Object-Oriented Metamethods.* Springer-Verlag New York, 1998.

[HJL96] Constance L. Heitmeyer, Ralph D. Jefforts, Bruce G. Labaw: *Automated Consistency Checking of Requirements Specifications.* In: ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 3, July 1996, p. 231-261.

[HL96] Mats P. E. Heimdahl, Nancy G. Leveson: *Completeness and Consistency in Hierarchical State-Based Requirements.* in IEEE Transactions on Software Engineering, Vol. 22, No. 6, June 1996.

[HR92] Theo Härder, Joachim Reinert: *Einsatz von GRID Files zur Sicherung der referentiellen Integrität in SQL2.* Interner Bericht, Universität Kaiserslautern, März 1992.

[ISO90] ISO/IEC: *Information technologie - Information Resource Dictionary System (IRDS) framework.* ISO/IEC 10027, 1990.

[Jac94] I. Jacobson: *Object-Oriented Software-Engineering: A Use Case Driven Approach.* Addison-Wesley, MA. 1994.

[Jar97] Matthias Jarke, NATURE Team: *Meta Models for Requirements Engineering.* jhttp:// ksi.cpsc.ucalgary.ca/KAW/KAW96/jarke/Jarke.html.

[JJR89] Matthias Jarke, Manfred Jeusfeld, Thomas Rose: *A Software Process Data Model for Knowledge Engineering in Information Systems.* MIP 8910, Universität Passau, 1989.

[JJQ98] Matthias Jarke, Manfred A. Jeusfeld, Christoph Quix: *ConceptBase V5.0 User Manual.* h http://www-i5.informatik.rwth-aachen.de/CBdoc/userManual, März 1998.

[KMS+89] Manolis Koubrakis, John Mylopoulos, Martin Stanley, Alex Borgida: *Telos: Features and Formalization.* Technical Reports on Knowledge Representation and Reasoning, KRR-TR-89-4, University of Toronto, Feb. 1989.

[KSW92] N. Kiesel, A. Schürr, B. Westfechtel: *Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications* Aachener Informatik-Fachberichte 92-44, RWTH Aachen, 1992.

[Lef94] Martin Lefering: *Software Document Integration using Graph Grammar Specifications.* Aachener Informatik-Fachberichte 92-15, RWTH Aachen, 1994.

[MBJ+90] John Mylopoulos, Alex Borgida, Matthias Jarke, Manolis Koubrakis: *Telos: Representing Knowledge about Information Systems.* ACM Transactions on Information Systems, Vol. 8, No. 4, Oct.1990, p. 325-362.

[NJZ+95] Hans W. Nissen, Matthias Jarke, Georg Zemanek, Harald Huber: *Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology.* MIP 9512, Universität Passau, 1995.

[NS91] Manfred Nagl, Andy Schürr: *A Specification Environment for Graph Grammars* In Proceedings 4th Workshop on Graph Grammars and Their Application to Computer Science, LNCS 532, p. 599-609, Berlin: Springer Verlag, 1991.

[Met98] Metamodel.com: *MetaModel.com.* http://www.metamodel.com, September 1998.

[NS96] Manfred Nagl, Andy Schürr: *Software Integration Problems and Coupling of Graph Grammar Specifications.* Aachener Informatik-Fachberichte 96-5, RWTH Aachen, 1996.

[Nus94] B. Nuseibeh: *A Multi-Perspective Framework for Method Integration.* Dissertation, Department of Computing, Imperial College, London, Oktober 1994.

[Rat97] Rational Software Corporation. UML Ressources. *Unified Modeling Language, version 1.1* Rational Software Corporation, 18880 Homestead Rd, Cupertino, CA 95014, September 1997, http://www.rational.com/uml/documentation.html.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object Oriented Modeling and Design.* Prentice Hall, 1991.

[Rei92] Joachim Reinert: *Sicherung der referentiellen Integrität - Optimierung durch Schemaanalyse.* Interner Bericht, Universität Kaiserslautern, Juli 1992.

[Ric92] Michael M. Richter: *Prinzipien der künstlichen Intelligenz.* Universität Kaiserslautern, Teubner Verlag, Stuttgart, 1992.

[Rom91] Dieter Rombach: *Practical Benefits of Goal-Oriented Measurement.* In: N. Fenton and B. Littlewood, editors, Software Reliability and Metrics, pages 217-235. Elsevier Applied Science, London, 1991.

[Rom97] Dieter Rombach: *Software Engineering I* Script zur Vorlesung WS 97/98, Fachbereich Informatik, Universität Kaiserslautern, 1998.

[Sch96] Andy Schürr: *Graph Transformations and Graph Transformation Units in GRACE.* Aachener Informatik-Fachberichte 96-13, RWTH Aachen, 1996.

[Sch94] Andy Schürr: *PROGRES, A Visual Language and Environment for PROgramming with Graph REwriting Systems.* ?Aachener Informatik-Fachberichte 94-11, RWTH Aachen, 1994.

[Sch91] Andy Schürr: *PROGRES: A VHL-Language Based on Graph Grammars.* n Proceedings 4th Workshop on Graph Grammars and Their Application to Computer Science, LNCS 532, p. 641-659, Berlin: Springer Verlag, 1991.

[SF97] G. Spanoudakis, A. Finkelstein: *Overlaps among Requirements Specifications* Department of Computer Science, City University, London 1997.

[SHA96] Syouri Kouno, Han-Myung Chang, Araki K.: *Consistency Checking between data and process diagrams based on formal methods.* In: Proceedings of 20th International Computer Software and Applications Conference, 1996, COMPSAC'96, p. 261-269.

[SL95] D. Skuce, T. C. Lethbridge: *CODE5: A Unified System for Managing Conceptual Knowledge.* In International Journal of Human-Computer Studies. 42, p. 413-451.

[Spi89] J. M. Spivey: *The Z Notation: A Reference Manual* Prentice-Hall, London, 1989.

[SWZ96] Andy Schürr, Andreas J. Winter, Albert Zündorf: *Spezifikation und Prototyping graphbasierter Systeme* Informatik, Forschung und Entwicklung, Vol. 11, p. 191-202, Springer Verlag, 1996.

[SW92] Andy Schürr, Bernhard Westfechtel: *Graphgrammatiken und Graphersetzungssysteme.* Aachener Informatik-Fachberichte 92-15, RWTH Aachen, 1992.

[SWZ95] Andy Schürr, Andreas J. Winter, Albert Zündorf *Graph Grammar Engineering with PROGRES* Aachener Informatik-Fachberichte 95-02, RWTH Aachen, 1995

[Ver97] Martin Verlage: *Ein Ansatz zur Modellierung großer Software-Entwicklungsprozesse durch Integration unabhängig erfaßter rollenspezifischer Sichten.* Dissertation am FB Infomatik der Universität Kaiserslautern, Februar 1997.

[Wav97] Wave OML: *OML-Ontology* http://wave.eecs.wsu.edu/WAVE/Ontologies/CKML/CKML15.html Markup Language, Juli 1997.

[Web81] Wolfgang Weber: *Ein Subsystem zur Aufrechterhaltung der semantischen Integrität in Datenbanken* Jochem Heinzmann Verlag, Karlsruhe, 1981.

[Wie95] Roel Wieringa: *An Introduction to Requirements Traceability.* Technical Report, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, Nov. 1995.

[Wie92] R. J. Wieringa *:A conceptual model specification language (CMSL Version 2).* Technical Report, Vrije Universiteit, Amsterdam, Juni 1992.

[Wie97] Roel Wieringa: *Toolkit for Conceptual Modeling (TCM).* http://www.cs.vu.nl/~tcm/users-guide/User.html, Dez. 1997.