



Universität Karlsruhe (TH)
Fakultät für Informatik
*Institut für Programmstrukturen und
Datenorganisation (IPD)*



Konzeption und Umsetzung einer Erweiterung des Common Warehouse Metamodel (CWM) zur Beschreibung von imperfekten Daten

Studienarbeit

von

Alexander Haag

30. September 2004

Verantwortlicher Betreuer: Prof. Dr.-Ing. Dr. h.c. Peter C. Lockemann
Betreuender Mitarbeiter: Dipl.-Inform. Heiko Schepperle

Ich erkläre hiermit, die vorliegende Arbeit selbstständig und ohne unzulässige fremde Hilfe angefertigt zu haben. Die verwendeten Literaturquellen sind im Literaturverzeichnis angegeben.

Karlsruhe, den 30. September 2004

Zusammenfassung

Metadaten werden in einem Data-Warehouse mit Hilfe des Common Warehouse Metamodell (CWM) beschrieben. In dieser Studienarbeit wird ein Konzept für eine Erweiterung des CWM zur Beschreibung imperfekter Daten erarbeitet. Da imperfekte Daten in herkömmlichen Data-Warehouses während der ETL-Phase bereinigt werden, muss durch die Erweiterung sichergestellt werden, dass die Imperfektion während des ganzen Warehouseprozesses erhalten bleibt. Im Rahmen dieser Arbeit wird das CWM auf zwei verschiedene Arten erweitert und eine der beiden Erweiterungen mit Hilfe eines neu erstellten Werkzeugs evaluiert.

Inhaltsverzeichnis

I	<i>Einleitung</i>	1
1	Motivation und Zielsetzung	1
2	OVID-Projekt	1
3	Aufgabenstellung	1
4	Vorgehensweise	2
II	<i>Analyse</i>	3
5	Das Common Warehouse Metamodel	3
5.1	Der generelle Aufbau des CWM	3
5.1.1	Das Core-Package	4
5.1.2	Das Relational-Package	4
5.2	Die Verwendung von CWM	4
5.3	Die Techniken zur Erweiterung des CWM	5
5.3.1	Einfache Erweiterung (<i>Simple extension</i>)	6
5.3.2	Modellierte Erweiterung (<i>Modeled extension</i>)	6
6	Imperfekte Daten in OVID	7
6.1	Szenario	7
6.2	Das Kontextmodell	8
7	Das Data-Warehouse-Werkzeug von Cognos	9
7.1	Die Cognos Suite	10
8	Erkenntnisse aus der Analyse	10
III	<i>Konzept einer Erweiterung von CWM und einer Anpassung eines Werkzeuges</i>	11
9	Erste einfache Erweiterung des CWM	11

10 Zweite weitergehende Erweiterung	12
10.1 Das Eclipse Modeling Framework (EMF)	12
10.2 Resultat	13
10.3 Zugehörigkeitsfunktionen aus der Fuzzylogik	13
11 Anforderungen an das neu erstellte Werkzeug	15
IV <i>Umsetzung der Erweiterungen des CWM</i>	16
12 Die Umsetzung der simple Extension	16
13 Die Umsetzung der modeled Extension	18
V <i>Evaluation der Erweiterungen</i>	22
14 Verfahren mit imperfekten Spalten in der Praxis	22
14.1 Speichern in 'Externer Relation'	22
14.2 Speichern innerhalb der Ursprungsrelation	23
15 Das Tool zum Testen der CWM-Erweiterungen	24
15.1 Beispieleingaben	25
15.2 Ausgabe des Werkzeugs	27
16 Ergebnisse der Evaluation	29
17 Ausblick	31

Teil I

Einleitung

1 Motivation und Zielsetzung

Im Rahmen des OVID-Projektes liegen verteilt imperfekte und aggregierte Verkehrsdaten vor. Um den Austausch imperfekter und aggregierter Daten zwischen den verschiedenen Dienstleistern zu gewährleisten, soll das Common Warehouse Metamodel (CWM) benutzt werden. Da die Beschreibung imperfekter Daten in der derzeitigen Definition des CWM nicht vorhanden ist, ist es die Aufgabe dieser Arbeit, ein Konzept für eine Erweiterung des CWM zur Beschreibung von imperfekten Daten zu erstellen und dieses umzusetzen. Das hier erarbeitete Konzept für die Erweiterung stützt sich weitestgehend auf vorangegangene Studien- und Diplomarbeiten aus dem OVID-Projekt. In diesen Arbeiten wird unter anderem das Kontextmodell zur Beschreibung von imperfekten Daten angewandt. Weiter werden die Verkehrsdaten des OVID-Projektes in die verschiedenen Imperfektionsarten klassifiziert. Damit die Erweiterung getestet werden kann, soll diese, in Data Warehouse Werkzeugen evaluiert werden.

Ziel dieser Arbeit ist es somit, die im OVID-Projekt verteilt vorliegenden imperfekten Verkehrsdaten mit Hilfe des CWM auf Metaebene zu beschreiben und in einem Werkzeug umzusetzen, damit alle beteiligten Komponenten diese Daten benutzen und verstehen können.

2 OVID-Projekt

Während die permanente und allumfassende Verfügbarkeit von Information für den Verkehrsteilnehmer von morgen sowohl im Personen- als auch im Güterverkehr eine greifbare Zukunftsperspektive darstellt, gibt es nur unpräzise Vorstellungen darüber, wie die Informationsfülle individuell handlungsrelevant aufbereitet werden kann. Auch ist noch unzureichend durchdacht, wie der Verkehrsteilnehmer oder sein Fahrzeug als aktives Element in den Prozess der Informationserzeugung und -vermittlung eingebunden sein wird. Die letztgenannten Faktoren sind aber entscheidend dafür, inwieweit technische Optionen für die Veränderung von Verhaltensmustern tatsächlich individuell genutzt und sozial wirksam werden.

Das Ziel des vom Bundesministerium für Bildung und Forschung geförderten Verbundprojektes OVID [1] der Universität Karlsruhe (TH) ist es, verkehrsbezogene Probleme zukünftiger Informationswelten verstehen und neue Möglichkeiten nutzen zu lernen. Hierzu erfolgt der Aufbau einer Plattform zur Modellierung und Bewertung von verkehrsinfrastrukturellen, verkehrstelematischen und logistischen Maßnahmen im Verkehrs- und sozio-ökonomischen System.

Das IPD ist im Rahmen des Teilprojektes B1: Verlässliche Datenbanken für die Informationsbereitstellung im Verkehr an diesem Projekt beteiligt. Ein Ziel dieses Teilprojektes ist es Imperfektion und Data-Warehouse-Techniken mit Verkehrsdaten sinnvoll zu verknüpfen.

3 Aufgabenstellung

Das Hauptziel dieser Arbeit liegt in der Aufgabe, imperfekte Daten, wie sie zum Beispiel in den Verkehrsdaten des OVID-Projektes vorliegen, modellieren zu können. Die Modellierung soll mit Hilfe des Common-Warehouse-Metamodells geschehen. Im CWM muss zu diesem Zweck eine Möglichkeit

geschaffen werden, imperfekte Daten modellieren bzw. ausdrücken zu können. Aus den genannten Gründen sollen in dieser Arbeit Erweiterungen des CWM vorgestellt und auf ihre Einsetzbarkeit hin überprüft werden.

4 Vorgehensweise

Zunächst werden die theoretischen Grundlagen erarbeitet. Dabei wird das CWM vorgestellt. Danach folgt ein Überblick über imperfekte Daten und Data-Warehouse-Werkzeuge.

Aus diesen Grundlagen wird dann ein Szenario entwickelt, wie imperfekte Daten mit Hilfe des CWM dargestellt werden können.

Dies beinhaltet zuerst die Konzeption der Erweiterung des CWM zur Darstellung imperfekter Daten und die Konzeption einer Anpassung eines Data-Warehouse-Werkzeuges.

Danach schließt sich die Umsetzung dieser Konzepte und die Erstellung eines Werkzeugs zum Testen der erzeugten Erweiterungen an.

Nachdem die Erweiterungen konzipiert und umgesetzt wurden, werden diese mit Hilfe des selbst erstellten Werkzeugs evaluiert.

Teil II

Analyse

5 Das Common Warehouse Metamodel

5.1 Der generelle Aufbau des CWM

Das Common Warehouse Metamodel (CWM, [6]) ist ein von der OMG propagierter Standard zur Beschreibung von Data-Warehouses. Dieser Standard definiert ein Metamodel, das Metadaten sowohl aus betriebswirtschaftlicher als auch aus technischer Sicht darstellen kann. Das CWM stellt als Metamodel sowohl die Syntax als auch die Semantik bereit, um den kompletten Data-Warehouse-Prozess beschreiben zu können. Es wird dabei als Basis verwendet, um Metadaten zwischen heterogenen Systemen auszutauschen [7]. Das CWM ist in 5 Schichten mit insgesamt 21 Metamodel packages eingeteilt (Abbildung 1), wobei die oberen Schichten auf den unteren aufbauen und diese gegebenenfalls erweitern. Jedes Package korrespondiert dabei mit einem Funktionsbereich einer typischen Data-Warehouse-Anwendung. Dabei verwendet das CWM die grundlegendsten Modellelemente von UML. Kern des Modells ist das *Core*-Package. Alle anderen Packages bauen auf das *Core*-Package auf oder erweitern es. Das CWM ist geeignet um Daten in relationalen, recordorientierten und objektorientierten Datenbanken zu modellieren, bietet aber auch Möglichkeiten für OLAP (Online Analytical Processing) und Data-Mining [11]. Diese Vielfältigkeit erlaubt es, mit dem CWM den ganzen Data-Warehouses-Prozess zu modellieren.

Ein weiterer wichtiger Punkt ist, dass das CWM dem Anwender erlaubt, nur die Packages zu benutzen, die er für seine Anwendung benötigt, und er die restlichen Packages nicht zu beachten braucht. Das heißt, wenn ein Anwender zum Beispiel ein relationales Schema beschreiben will, benötigt er aus dem unten gezeigten Schichtenmodell nur die Packages *Core* und *Relationships* aus der Objektmodell-schicht sowie *Relational* aus der Ressourcenschicht.

Management	Warehouse Process			Warehouse Operation		
Analysis	Transformation	OLAP	Data Mining	Information Visualization	Business Nomenclature	
Resource	Object	Relational	Record	Multidimensional	XML	
Foundation	Business Information	Data Types	Expressions	Keys and Indexes	Software Deployment	Type Mapping
Object Model	Core		Behavioral	Relationships		Instance

Abbildung 1: CWM Schichtenmodell [6]

Zu der Spezifikation von CWM liefert die OMG vier weitere Komponenten [7, Seite 644]:

1. Das komplette Common Warehouse Metamodel in einem Rational-Rose-Modell (also in Form von UML-Diagrammen die das CWM graphisch darstellen)
2. Eine XML-Datei (die eine MOF 1.3 konforme Version des Common Warehouse Metamodels in einem XML-Dokument darstellt)
3. Eine CWM DTD-Datei, mit der Anwender die Richtigkeit eines ausgetauschten XML-Dokuments prüfen können
4. Eine IDL-Repräsentation (CORBA Interface Definition Language) des Common Warehouse Metamodels

Die MOF (Meta Object Facility) vereint unterschiedliche Metamodelle indem sie eine gemeinsame Basis für Metamodelle anbietet. Sie unterstützt alle Metadaten, die mit Hilfe von Objektmodellierungstechniken beschrieben werden können. Die MOF ist ebenfalls ein von der OMG propagierter Standard.

Um einen Einblick in den Aufbau der CWM-Packages zu geben, werden im Folgenden zwei davon beispielhaft aufgegriffen:

5.1.1 Das Core-Package

Das *Core*-Package bildet die Grundlage des gesamten CWM. Wie bereits erwähnt, bauen alle anderen Packages auf dem *Core*-Package auf, das heißt, sie verwenden bzw. erweitern die Klassen aus diesem Package. Eine UML-Darstellung des *Core*-Packages zeigt Abbildung 2. Alle Klassen des CWM sind Unterklassen der im *Core* definierten Element-Klasse. Des weiteren sind fast alle Klassen, außer Support-Klassen wie *TaggedValues*, Unterklassen der Klasse *ModelElement*. Diese bietet eine Reihe von Attributen, wie zum Beispiel *name*, für Ihre Unterklassen an. Von *ModelElement* erben einige aus schon UML bekannte Klassen: *Stereotypes*, *Constraint*, *Dependency*, *Feature* und *Namespace*.

5.1.2 Das Relational-Package

Das *Relational*-Package ist in der Ressourcenschicht eingeordnet. Es stellt alle nötigen Teile bereit, um relationale Datenbankschemata beschreiben zu können. Es unterstützt die Beschreibung von SQL99-konformen relationalen Datenbanken inklusive der objektorientierten Erweiterungen. In diesem Package sind unter anderem Klassen zur Beschreibung von Schemata, Prozeduren, Triggern, Tabellen, Spalten, Views, Schlüssel und Indizes enthalten. Man kann eine UML-Darstellung des ganzen Packages in [7, Seite 48] finden und nachlesen.

5.2 Die Verwendung von CWM

Mit dem CWM kann der komplette Data-Warehouse-Prozess, auch Information Supply Chain (ISC) genannt, auf Metaebene dargestellt werden. Abbildung 3 zeigt die Zusammenhänge im Data-Warehouse-Prozess. Man sieht, wie Datenquellen, ETL(Extract-Transform-Load)-Prozesse, Data-Warehouses, usw. miteinander verknüpft sind. Das heißt also, von der Beschreibung einer einzelnen Datenquelle über den ETL-Prozess, Data-Warehouses, Data-Marts bis hin zu Analyse-Werkzeugen wie OLAP, Data-Mining, usw. ist alles mit dem CWM darstellbar.

Die Metadaten der einzelnen Teile werden dabei im Metadata-Repository zentral gespeichert. Durch die zentrale Speicherung soll Mehrfachverwendung von Metadaten in den verschiedenen Teilen des

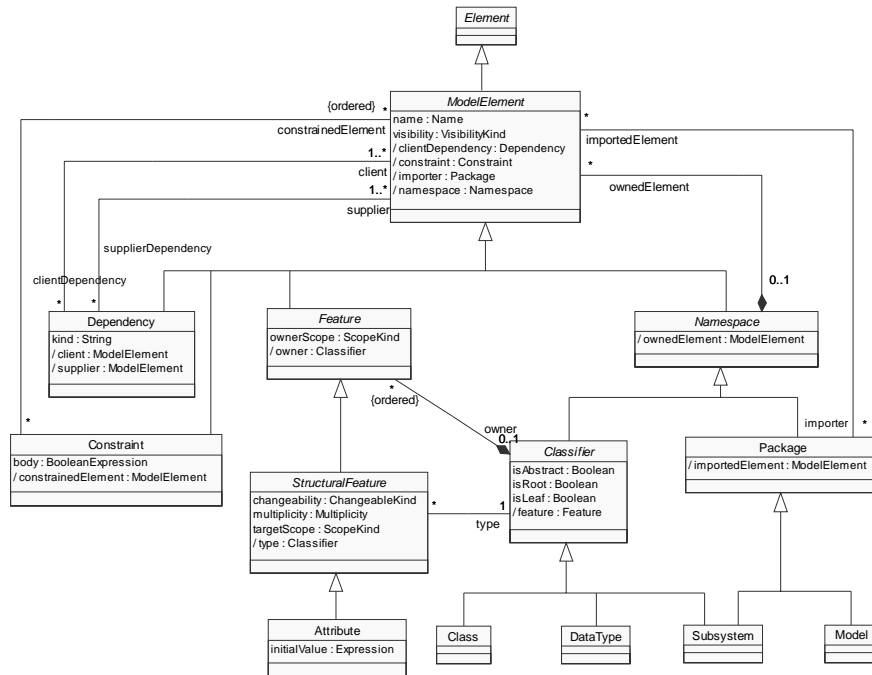


Abbildung 2: CWM Core-Package [6]

Data-Warehouse-Prozesses erreicht werden. Erweiterungen des CWM werden ebenfalls im Repository abgelegt. Durch einen Metadatenmanager wird gewährleistet, dass alle Module, die eine bestimmte Erweiterung benötigen könnten, diese auch als Informationen zur Verfügung gestellt bekommen, oder anders, überhaupt wissen, dass es eine solchen Erweiterung gibt. Durch dieses Wissen können die Module die Daten, welche mit der Erweiterung beschrieben werden, verstehen und weiterverarbeiten.

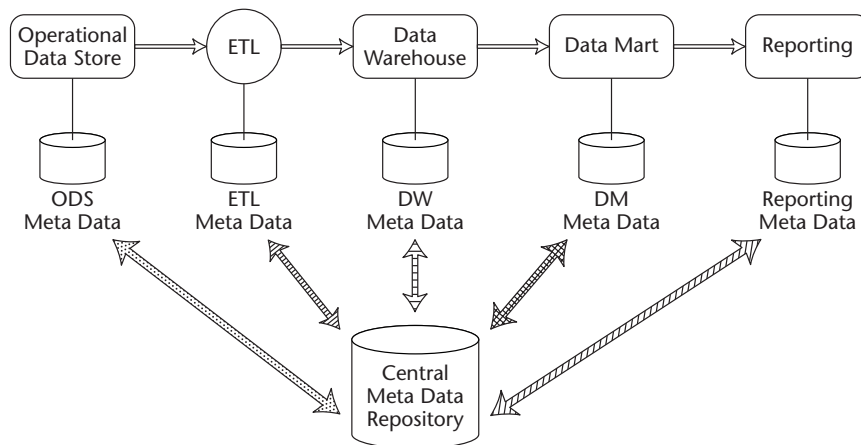


Abbildung 3: ISC mit Metadata-Repository [6]

5.3 Die Techniken zur Erweiterung des CWM

Im Folgenden werden nun die Erweiterungsmöglichkeiten erläutert, die das CWM von sich aus bereit stellt.

Das CWM beinhaltet die gängigsten Eigenschaften und Funktionalitäten einer Vielzahl von Data-Warehouse-Werkzeugen. Dieser Ansatz bietet somit eine breite Basis allgemeiner Definitionen. Das CWM liefert zwar diese breite Basis zur Modellierung von Metadaten, ist aber trotzdem in einigen Fällen nicht ausreichend, um ein spezielles Modell einer Anwendung genau zu beschreiben. Manche Werkzeuge benötigen zur vollen Funktionsfähigkeit eine erweiterte Informationsmenge, die ihnen das CWM in der ursprünglichen Form nicht bieten kann. Diese Nutzer des CWM werden deshalb die gegebenen Definitionen erweitern müssen, um ihr Werkzeug mit den nötigen Informationen auszustatten, die es zum Arbeiten benötigt.

Die Erweiterungstechniken, die von CWM angeboten werden, können in zwei Typen eingeteilt werden[6, Seite 208f], welche im Folgenden erläutert werden.

1. Einfache Erweiterung (*Simple extension*)
2. Modellierete Erweiterung (*Modeled extension*)

5.3.1 Einfache Erweiterung (*Simple extension*)

Dieser in CWM integrierte Erweiterungsmechanismus benutzt *Stereotypes* und *TaggedValues*. Er ermöglicht dem Benutzer, einem Objekt eine beliebige Anzahl von *Stereotypes* oder *TaggedValues* zuzuweisen. Das *Stereotypes* Konzept bietet eine Möglichkeit *ModelElements* zu klassifizieren. *Stereotypes* haben eigentlich nur für den menschlichen Benutzer einen Nutzen, die bessere semantische Verständlichkeit einer Information. *Stereotypes* können zusätzlich *Constraints* und verpflichtende *TaggedValues* spezifizieren, die für ein *ModelElement* gelten[6, Seite 208ff]. Die Anzahl der zu einem *Stereotype* gehörigen *TaggedValues* ist beliebig.

TaggedValues sind Name/Wert-Paare und erlauben es Informationen an ein *ModelElement* anzuhängen. Die Interpretation erfolgt nicht im CWM, sondern muss zwischen den verschiedenen Anwendern bestimmt, ausgetauscht bzw. publiziert werden. Werkzeuge setzen *TaggedValues* ein, um Informationen, die über die Basis des CWM hinausgehen, aber trotzdem gebraucht werden, bereitzustellen. Solche Informationen könnten z.B. Optionen zur Codegenerierung, Modellmanagement Informationen oder Anwender-spezifische Semantik sein. *TaggedValues* sind sicher eine sehr leichte Erweiterungstechnik, aber die Austauschmöglichkeit von Metadaten beschränkt sich auf die Werkzeuge, die das semantische Wissen haben, diese auch zu verstehen.

Diese Erweiterung ist zum Einstieg sicher sinnvoll, wenn man die imperfekten Daten in einem Relationalen Schema mit Hilfe des Kontextmodells beschreiben will. Das Kontextmodell wurde in der Studienarbeit von Andreas Merkel [10] verwendet und wird in Kapitel 6.2 noch näher erläutert. Der Kontext könnte dann als *TaggedValue* an ein *ModelElement* angehängt werden.

5.3.2 Modellierete Erweiterung (*Modeled extension*)

Sollte die einfache Erweiterung (*Stereotypes* und *TaggedValues*) nicht ausreichen, um das erwünschte Modell darzustellen, braucht man eine robustere und potentere Lösung, um das CWM zu erweitern. Um dies zu erreichen, kann ein Anwender alle standard OO-Techniken verwenden, die es gibt. Dieses Kapitel wird sich auf die Vererbung beschränken, da diese recht einfach und verständlich die Möglichkeiten dieser Erweiterung zeigt. Die Vererbung wird auch nicht zuletzt deshalb gewählt, da die OMG selbst als Beispiel das ER-Modell als modellierete Erweiterung publiziert hat. Dieses Beispiel verwendet ebenfalls Vererbung und soll dazu dienen, dem Anwender zu zeigen, wie er das CWM erweitern kann. Diese Erweiterung wurde in der 2. CWM Spezifikation[3] veröffentlicht.

Um diese Ausarbeitung nicht unnötig aufzublähen, wird hier darauf verzichtet, auf diese Erweiterung näher einzugehen und sie zu erläutern.

Zur einfachen Beschreibung der in OVID vorliegenden imperfekten Daten mit Hilfe von Kontexten (siehe [10]) ist diese Art der Erweiterung sicherlich etwas zu mächtig. Will man aber die imperfekten Daten nicht nur beschreiben, sondern auch damit arbeiten und sie weitergeben, wird man um die Verwendung dieses Erweiterungstypes wohl nicht herumkommen.

6 Imperfekte Daten in OVID

In den Arbeiten von E. Koop [9] und A. Merkel [10] geht es um Imperfektion von Verkehrsdaten in OVID. Koop beschreibt dabei welche Daten überhaupt mit Imperfektion behaftet sein können. Des weiteren wird in dieser Arbeit die relationale Umsetzung von imperfekten Daten untersucht. Merkel nutzt wiederum die Arbeit von Koop, um ein Konzept zu entwickeln über imperfekte Daten zu aggregieren, aber die Imperfektion zu erhalten. Er benutzt das Kontextmodell um die Daten in einem relationalen Schema darzustellen.

Bei imperfekten Verkehrsdaten wird in der oben genannten Arbeiten zwischen unsicherem, unscharfem und ungenauen Wissen unterschieden. Unsicher ist Wissen, wenn man (noch) nicht genau sagen kann, ob eine Aussage richtig oder falsch ist. Zum Beispiel eine Aussage über das Wetter in einer anderen Stadt kann erst durch zusätzliches Wissen (evtl. Telefongespräch) validiert werden. Unscharfes Wissen entsteht bei der Anwendung von Kategorien ohne scharfe Abgrenzung gegeneinander. Ein Beispiel ist hier die Körpergröße vom Lebewesen (klein, mittel, groß) oder die Klassifizierung von Verkehr (stau, stockend, frei). Ungenaueres Wissen hingegen entsteht zum Beispiel durch Aggregation. Intervalle wie 'zwischen Bruchsal und Heidelberg' oder eine aggregierte Durchschnittsgeschwindigkeit an einer Autobahnstelle sind Beispiele hierfür.

Es gibt im OVID-Projekt viele verschiedene imperfekte Daten. Welche Art der Imperfektion aber vorliegt hängt immer auch von der Art der Daten und von der Art ab wie die Daten gewonnen werden [10]. In OVID gibt es **Floating-Car-Daten**(FCD), die im Rahmen des Projektes WAYflow gesammelt und in OVID verwendet werden, **Simulationsdaten**, die aus der Simulationsumgebung VISSIM stammen und **Güterverkehrsdaten**, die aus Jahresberichtsdaten von in Deutschland gemeldeten Fahrzeugen extrahiert wurden. In dieser Menge an Daten stecken mitunter imperfekte Daten. Diese sind anhand vieler Relationen aus dem OVID Projekt in der Arbeit von Erik Koop [9] dargestellt. Auf eine erneute Darstellung dieser Tabellen wird hier aus diesem Grund verzichtet.

6.1 Szenario

Da diese Arbeit innerhalb des Verkehrsprojektes OVID angefertigt wird, bietet es sich auch an, ein Szenario aus der Verkehrswelt zu wählen. Im Verkehr fällt eine so große Zahl von Daten an, das diese erst sinnvoll aggregiert werden müssen, um damit zu arbeiten. Ein Data-Warehouse erfüllt genau diese Aufgabe. Es kann große Mengen von Daten aus verschiedensten Quellen vereinigen, aggregieren und bereinigt darstellen. Soll ein Data-Warehouse z.B. in einer Verkehrsleitzentrale eingesetzt werden, so muss das Warehouse auch Anfragen auf aktuellen Messdaten erlauben, um Reaktionen auf kurzfristig veränderte Verkehrsbedingungen zu erlauben. Die dabei erfassten Messdaten können wie bereits erwähnt durchaus unvollkommen sein. Beispielsweise formulieren Anrufer, die freiwillig als „Stau-melder“ für bestimmte Rundfunksender arbeiten, eine natürlichsprachliche Verkehrsbeschreibung. In dieser Verkehrsbeschreibung werden bestimmte Verkehrszustände von Strecken mit unscharfen Begriffen beschrieben. Solche sind beispielsweise *frei*, *lebhaft*, *stockend* oder *Stau*. Des weiteren werden bestimmte Eigenschaften eines Staus, z.B. Staulänge und Stauposition, beschrieben. Da den meisten Meldungen Schätzungen der Anrufer zu Grunde liegen, sind diese ungenau. Anrufern, die das erste Mal etwas melden, wird ein geringeres Vertrauen entgegen gebracht, als solchen, die schon öfters richtige

Angaben gemacht haben oder Meldern, die offiziellen Stellen wie Polizei oder Autobahnwacht angehören. Unterschiedliches Vertrauen lässt sich als unterschiedliche Sicherheit (oder eben als unsicher) ausdrücken. Es ist nun ein Teil dieser Arbeit zu zeigen, dass es wichtig ist, diese Unvollkommenheiten nach ihrer Art im ganzen Warehouse bzw. Informationssystem zu erhalten und zu speichern.

Einkommende Daten werden zum Beispiel in einer wie folgt aussehenden, vereinfachten Tabelle abgelegt.

Tabelle: Verkehrsmeldung					
MeldungsID	Eingangszeit	Störungstyp	Straße	Geschwindigkeit	Verkehrsfluss
1	13:11,24/07/2004	Unfall	A5	30	stockend
2
3
...
...

Abbildung 4: vereinfachte Tabelle Verkehrsmeldung

Die in Abbildung 4 beschriebene Tabelle zeigt eine Vereinfachung einer Relation aus vorangegangenen Arbeiten [9]. Darin werden z.B. die im Radio einkommenden Verkehrsmeldungen eingetragen und jede mit einer eindeutigen 'MeldungsID' versehen. Informationen die gespeichert werden sind der 'Zeitpunkt' an dem eine Information über eine 'Störung' in einer 'Straße' eintrifft auf der gerade im Schnitt mit der 'Geschwindigkeit' XY km/h gefahren wird. Hierbei ist nun beispielsweise die Spalte 'Verkehrsfluss' eine unscharfe (Fuzzy) Spalte. Sie enthält eine natürlichsprachliche Aussage über den Verkehrsfluss (bsp. stau, stockend,...), also unscharfe Werte.

In einer Verkehrsleitzentrale könnte nun beispielsweise jemand auf die Idee kommen nach allen Durchschnittsgeschwindigkeiten von Strecken zu fragen, die zur Zeit als *stockend* eingestuft sind. Diese könnten dann mit dem Durchschnitt auf den Strecken aus dem vergangenen Jahr verglichen werden. Daraus ergibt sich die Anforderung an das Data-Warehouse, auch Anfragen nach imperfekten Informationen oder imperfekte Aggregationsanfragen zuzulassen und diese ausreichend zu beantworten. Will man zum Beispiel den Einsatz von Verkehrsleitsystemen verändern, könnten solche Informationen wichtige Entscheidungshilfen sein.

In den vorangegangenen Diplom-/Studien-Arbeiten [9],[10] wird versucht die Imperfektion der Daten mit dem Kontextmodell zu beschreiben. Auf die Erkenntnisse dieser Arbeiten stützt sich mein weiteres Konzept.

6.2 Das Kontextmodell

In unserem Papier [8] und in der Studienarbeit von Andreas Merkel wird das Kontextmodell als Konzept zur Klassifizierung von Daten nach Schindler [16] vorgestellt. Es stellt eine Erweiterung des Relationalen Datenmodells dar, welche es zum Beispiel ermöglicht Imperfektion auszudrücken mit der Daten evtl. behaftet sein könnten. In unserem Beispiel (OVID) sind die, in einem relationalen Modell repräsentierten Verkehrsdaten zum Teil unsicher, unscharf oder ungenau. Diese Imperfektion wird durch Kontexte modelliert, indem man nicht mehr die exakten Werte, sondern einen Wertebereich, in welchem die Werte liegen, speichert.

Als Kontext eines Attributs in einem relationalen Schema wird eine Einteilung des Wertebereichs dieses Attributs bezeichnet. In dem von Andreas Merkel verwendeten Modell [10] wird dabei zwischen scharfen und unscharfen Einteilungen unterschieden.

Bei der scharfen Einteilung wird der Wertebereich eines Attributs durch den Kontext in Äquivalenzklassen abgebildet bzw. eingeteilt. Imperfektion kann somit durch die Benutzung dieser Äquivalenzklassen ausgedrückt werden, was im Folgenden beschrieben wird. Am Beispiel des OVID Projektes kann dies einfach gezeigt werden. Die einkommenden Floating-Car-Daten werden zwar mit exakten Werten in den Relationen gespeichert, sind aber nicht in der Genauigkeit bekannt, mit der sie gespeichert werden. Es handelt sich bei diesen Daten nämlich teilweise um Mittelwerte oder Hochrechnungen. Ein anderes Beispiel ist die im Verkehrsfunk durchgegebene Länge eines Staus. Es wird zwar die genaue Länge von 5,463 Kilometern gespeichert und angegeben, sie wird aber so nicht genutzt, da der Wert 5,463 nicht exakt gemessen, sondern ein Mittelwert ist. Statt dieses exakten Wertes könnten man eher speichern, dass der Stau zwischen 5 und 6 Kilometer lang ist. Will man nun das Dilemma umgehen, mit den zwar exakt gespeicherten, aber imperfekten Werten arbeiten zu müssen, kann man nun Anfragen benutzen, die anstatt der exakt abgespeicherten Werte mit den Äquivalenzklassen dieser Werte arbeiten. Man braucht also nicht mehr den Anspruch auf Exaktheit der gespeicherten Werte zu erheben, obwohl diese exakt, aber eben nur als Mittelwert, abgespeichert vorliegen.

Will man nun weitere Arten der Imperfektion darstellen, kommt man zur Verwendung unscharfer Kontexte. Anders als bei der scharfen Einteilung in Wertebereiche, bei der jeder Wert zu genau einer Äquivalenzklasse gehört, gibt es bei der unscharfen Einteilung kontinuierliche Übergänge zwischen den Klassen. Andreas Merkel spricht auch von unscharfen Kontexten. Die unscharfe Einteilung ist sinnvoll, denn in der realen Welt gibt es auch immer wieder Objekte, die nicht nur einer Kategorie angehören. Ein Beispiel hierfür sind Körper und ihre Größe, ein 180 cm Mensch kann sowohl in die Kategorie 'mittel' also auch in die Kategorie 'groß' eingeordnet werden. Schindler und Andreas Merkel verwenden in ihren Arbeiten linguistische Variablen, die ein Konzept aus dem Bereich der Fuzzylogik zur Beschreibung unscharfer Mengen darstellen [15], [10]. Eine linguistische Variable besitzt als Werte sprachliche Konstrukte, sogenannte Terme. Jedem Term ist eine Zugehörigkeitsfunktion $\mu_T(x)$ zugeordnet. Die Zugehörigkeitsfunktion gibt für jeden Wert x aus dem Wertebereich eines Attributs, zu welchem die linguistische Variable gehört, an, wie stark der Term auf den Wert zutrifft. Zugehörigkeitsfunktionen werden in Kapitel 10.3 noch weiter erläutert. Die hier gebrauchten Zugehörigkeitsfunktionen können alle Werte zwischen null und eins annehmen. Wobei der Wert null gleichbedeutend mit 'Term trifft nicht zu' und der Wert eins gleichbedeutend mit 'Term trifft voll zu' ist. Durch die verschiedenen Terme einer linguistischen Variable kann so der Wertebereich eines Attributs unscharf, d.h. mit kontinuierlichen Übergängen, eingeteilt werden[10].

Im Papier von Meier, Mezger, Werro & Schindler [14] wird auch ein Kontextmodell für unscharfes Klassifizieren von Tabelleninhalten vorgeschlagen. Darauf aufbauend wurde die Sprache **fCQL** entwickelt, um unscharfe Abfragen an scharfe Basisdatenbanken zu übermitteln.

Anders als bei dem Papier [14] soll in meiner Arbeit Imperfektion durch den ganzen Data-Warehouse-Prozess sichtbar bleiben. Deshalb wird hier das Kontextmodell benutzt, um das CWM so erweitern, dass man imperfekte Daten darstellen kann und alle Teile des Data-Warehouse-Prozesses damit umgehen können.

7 Das Data-Warehouse-Werkzeug von Cognos

Um die Idee aus dem Szenario, eines Verkehrsinformationssystems, mit einem Data-Warehouse umsetzen zu können, braucht man ein Werkzeug, das Data-Warehouses erstellen oder bearbeiten kann. Eine kommerziell erhältliche Data-Warehouse-Anwendung ist die 'Cognos Enterprise Series 7'-Suite. Diese Suite wurde schon in früheren Arbeiten innerhalb des OVID-B1 Teilprojektes eingesetzt.

7.1 Die Cognos Suite

Die Cognos Suite ist in mehrere Teile aufgeteilt. Es gibt Werkzeuge für die Analyse, zum Durchsuchen der Datenbestände und vieles mehr.

Dabei gibt es Cognos Impromptu für Abfragen und Reports, Cognos Powerplay für OLAP-Reports und Analysen, Cognos Query zur Erstellung von Ad-hoc-Anfragen und Navigation durch die Datenbestände, Cognos Scenario zum Data-Mining und einige mehr [13]. Alle Werkzeuge haben als Teil den Cognos Architekt welcher für die Verwaltung, Verarbeitung und Austausch von Metadaten zuständig ist. Dieser erstellt eine Menge an gemeinsamen Metadaten die allen Cognos Werkzeugen zur Verfügung gestellt werden und er wirkt als zentrale Stelle zur Speicherung und Verwaltung der Metadaten im System. Der Cognos Architekt ist vergleichbar mit der Vereinigung des Metadatamanager und Metadatarepository. Bei der Analyse der Suite taten sich mehrere Fragen auf.

'Wie wird das CWM von der Cognos Suite unterstützt?' und 'Ist die Cognos Suite überhaupt CWM-kompatibel?'

Dem ersten Anschein nach unterstützte die Cognos Suite das CWM, weil das Grundmodell der Suite ebenfalls in Schichten aufgeteilt ist. Nach gründlicher Recherche stellte sich aber heraus, dass die Cognos Suite das CWM nicht unterstützt.

Cognos bietet allerdings ein weiteres Werkzeug, Cognos ReportNet, an, welches laut Herstellerangaben das CWM unterstützt und mit dem der Nutzer auch das CWM erweitern kann. Das ReportNet ist kein Teil der untersuchten 'Cognos Enterprise Series 7'-Suite und stand während dieser Studienarbeit noch nicht zur Verfügung. Es konnte deshalb nicht in die Untersuchungen mit einfließen.

8 Erkenntnisse aus der Analyse

In der Analysephase dieser Arbeit wurde bisher das CWM in Aufbau, Verwendung und Erweiterungsmöglichkeiten vorgestellt. Weiter wurden imperfekte Daten beschrieben und in ein Szenario aus dem OVID-Projekt eingebettet. Als Beschreibungsmöglichkeit dient hierbei das Kontextmodell aus den Arbeiten von Andreas Merkel[10] und Erik Koop[9]. Zum Schluss wurde ein Data-Warehouse-Werkzeug, die Cognos Suite, auf ihre Verwendungsmöglichkeiten und ihre CWM-Verträglichkeit hin überprüft.

Da das Werkzeug keinen Zugriff auf das ihm zugrundeliegende CWM-Modell zulässt, wird es im nun folgenden Erweiterungskonzept keine Rolle mehr spielen.

Aufgrund der unzureichenden Unterstützung des CWM durch die Cognos Suite wurde die Aufgabenstellung dieser Arbeit um ein kleines Evaluierungswerkzeug erweitert. Dieses Werkzeug soll mit dem CWM und dessen Erweiterung umgehen und diese verarbeiten können.

Da laut Aufgabenstellung zuerst das CWM an sich erweitert werden soll, bietet sich an, die von der OMG herausgegebene Spezifikation herzunehmen und dort eine Erweiterung anzusetzen. Dabei kann auf die im CWM enthaltenen Zugangspunkte für Erweiterungen (siehe Kapitel 5.3.1) zurückgegriffen werden. Diese sind die *Simple Extension* mit *Stereotypes* und *TaggedValues* und die *Modeled Extension* mit Hilfe der Vererbung. Das Konzept sollte versuchen die Erweiterung so einfach wie möglich zu halten, da sonst die Anpassung an die bestehenden Data-Warehouse-Werkzeuge umso schwieriger wenn nicht sogar fast unmöglich wird, ohne die Erweiterung mit dem selber gebauten Werkzeug komplett auf die nicht erweiterte Definition herunterzubrechen und somit alle Informationen über Imperfektion zu verlieren.

Teil III

Konzept einer Erweiterung von CWM und einer Anpassung eines Werkzeuges

9 Erste einfache Erweiterung des CWM

Nachdem das zuvor beschriebene Kontextmodell (siehe Kapitel 6.2) eine Erweiterung des relationalen Datenmodells darstellt, soll nun das CWM mit *Stereotypes* und *TaggedValues* so erweitert werden, dass man damit imperfekte Daten im relationalen Datenmodell beschreiben kann. Die Erweiterung ermöglicht, bestimmte Informationen über Imperfektion durch die ganze Architektur eines Data-Warehouses hindurch bis zur obersten Analyseschicht zu erhalten. Dadurch soll eine möglichst realistische Sicht für die Analysewerkzeuge dargestellt werden.

Werden ausschließlich Tabellen als Imperfekt gekennzeichnet, wird es schwierig mit *TaggedValues* hervorzuheben, welche der Spalten aus der Tabelle imperfekt sind und um welchen Typ von Imperfektion es sich jeweils handelt. Es bietet sich aber trotzdem an, die Tabellen mit einem allgemeinen *Stereotype* `<<Imperfect Table>>` zu typisieren. Dieser *Stereotype* muss einen *Constraint* enthalten, der sicherstellt, dass mindestens eine der Spalten der Tabelle Imperfektion (also unsichere, ungenaue oder unscharfe Daten) beinhaltet. So wird eine Einteilung in Tabellen, die keinerlei Imperfektion enthalten, und in Tabellen mit Imperfektion erzeugt. Diese Einteilung ermöglicht es dem Benutzer vorab, eine Zusicherung zu treffen, ob das Ergebnis seiner Anfrage eventuell aus imperfekten Daten erstellt wurde oder zu 100% sicher ist, weil es nur aus Tabellen ohne Imperfektion resultiert.

Um genau sagen zu können, welche der Spalten Imperfektion beinhalten, werden die einzelnen Spalten ebenfalls mit einem *Stereotype* versehen. Es werden drei *Stereotypes*, die nach den verschiedenen Arten der Imperfektion benannt sind, und die, je nachdem welche Art vorliegt, Verwendung finden, unterschieden. Der *Stereotype* `<<Uncertain>>` wird mit den verpflichtenden *TaggedValues* 'Uncertain' und 'Wahrscheinlichkeitsfunktion' versehen. Der *Stereotype* `<<Fuzzy>>` benötigt 'Fuzzy' und 'Zugehörigkeitsfunktion' und der *Stereotype* `<<Imprecise>>` die *TaggedValues* 'Imprecise' und 'Zugehörigkeitsfunktion'. Diese drei *Stereotypes* ergänzen dann die entsprechenden Spalten.

Das Bild in Abbildung 5 zeigt wie die einzelnen *Stereotypes* zusammenhängen und wie alle direkt bzw. indirekt von der Klasse *Stereotype* aus dem *Core*-Package erben.

Mit dem *Stereotype* `<<Imprecise>>` können unter anderem Intervalle dargestellt werden. Da scharfe Kontexte sich auch als Intervalle beschreiben lassen, können sie mit dem *Stereotype* `<<Imprecise>>` modelliert werden. Die Zugehörigkeitsfunktion liefert dann nur die Werte null oder eins. Unscharfe Kontexte lassen sich mit dem *Stereotype* `<<Fuzzy>>` beschreiben. Die Zugehörigkeitsfunktion kann dann auch Werte zwischen null und eins liefern.

Mit dieser Erweiterung kann man auch die aus einer Aggregationsoperation entstehenden Tabellen und Spalten stereotypisieren, falls diese imperfekte Daten repräsentieren. Somit wird sichergestellt, dass für alle späteren Operationen auf den Daten, die Art der Imperfektion klar ersichtlich ist.

Es handelt sich hier in der Ausführung um eine erste Erweiterung. Momentan können mit dieser Erweiterung Zugehörigkeitsfunktionen nur in Textform beschrieben werden. Für die automatische Weiterverarbeitung, die es beispielsweise erlaubt, mit den Zugehörigkeitsfunktionen zu rechnen, müssen imperfekte Datentypen und Zugehörigkeitsfunktionen explizit als Klassen modelliert werden. Dazu werden wir auf den mächtigeren Erweiterungstyp *Modeled Extensions* zurückgreifen. Aus diesem Grund folgt nun ein zweiter erweiterter Ansatz, der mit dieser Art der Erweiterung arbeitet.

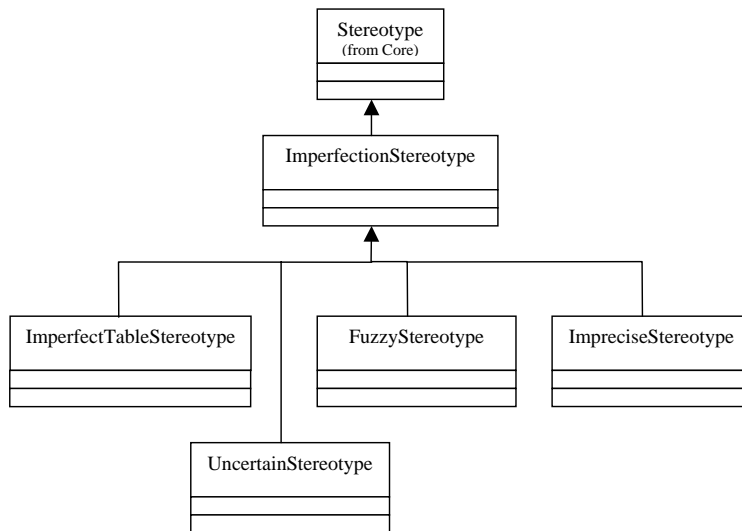


Abbildung 5: Vererbungsstruktur der Stereotypes für Imperfektion

10 Zweite weitergehende Erweiterung

Dieser weitergehende Ansatz verlangt eine tiefgehende Analyse der CWM-Spezifikation. Da die OMG das CWM auch als Rational-Rose-Modell bereitstellt, bietet sich an dieses genauer zu untersuchen und es dann gegebenenfalls zu erweitern. Ein Werkzeug das mit einem Rational-Rose-Modell umgehen und es darstellen kann ist Eclipse.

10.1 Das Eclipse Modeling Framework (EMF)

In Eclipse ist es möglich, mit Hilfe des Zusatzmoduls (Plugins) EMF, aus einem Rational-Rose-Modell Java-Klassen zu erzeugen [4]. Grundlage der Umwandlung eines Rational-Rose-Modells in Javaklassen ist bei EMF die **'Basic Code Generation'**. Dabei erstellt Eclipse Java Interfaces und Klassen, welche die Elemente des vorgegebenen Modells repräsentieren. Zu jedem Package des Modells erstellt Eclipse zwei bis drei Java-Packages. Eines dieser Packages besteht ausschließlich aus Java-Interfaces, welche die API's definieren, um auf Instanzen der Modellklassen zugreifen zu können. Das Package mit den Interfaces beinhaltet zu jeder Klasse des Modells ein Interface. Jedes Interface stellt Methoden bereit, die den Zugriff auf Instanzen einer Klasse ermöglichen, und get/set-Methoden um die Werte aller Features dieser Klasse zu bekommen bzw. zu ändern. Interfaces wurden wohl nicht zuletzt deshalb gewählt, um eine saubere Trennung zwischen Schnittstelle und Implementierung zu erhalten.

Das zweite Package besteht aus Klassen die oben genannte Interfaces implementieren. Das Package, in dem sich die implementierten Klassen befinden, besitzt außer den Klassen des Modells auch noch eine Fabrik-Klasse. Diese stellt Fabrikmethoden bereit, um Instanzen der Klassen dieses Packages zu erzeugen. Die Fabrik-Klasse wurde nach dem Entwurfsmuster *Abstrakte Fabrik* erzeugt. Dieses Entwurfsmuster ist deshalb so geeignet, weil es erlaubt, die Implementierung der einzelnen Klassen zu verstecken und eine zentrale Stelle zur Erzeugung von Instanzen einzelner Klassen anbietet.

Das optionale dritte Package (bei der Umsetzung des CWM-Rose-Modells vorhanden) enthält eine Adapter-Factory und eine Switch Klasse, die nützlich sind um Adapter zu implementieren. Dieses Package wird in dieser Arbeit nicht benötigt.

10.2 Resultat

Nach genauerer Betrachtung der Struktur und der Klassen des CWM blieben zwei Packages übrig, die für eine Erweiterung in Betracht kamen. Das *Core*-Package und das *Relational*-Package.

Das *Core*-Package, weil es die Grundlage des CWM ist und von allen anderen Packages verwendet wird. Hier muss ergänzt werden, wenn Erweiterungen an den grundlegenden Objekten des CWM erzeugt werden sollen. Ein Beispiel wird die Eingliederung von Funktionen in das *Core*-Package sein.

Das *Relational*-Package, weil wir mit Tabellen und Spalten arbeiten und diese mit dem *Relational*-Package modelliert werden. Hier wird die meiste Ergänzungsarbeit geleistet.

Aus den obigen Erkenntnissen kommen wir nun zum zweiten Ansatz.

Zuerst wird das *Relational*-Package um Unterklassen der Klasse *Column* erweitert. Die erste Unterklasse *ImperfectColumn* repräsentiert die generelle Form einer imperfekten Spalte. Davon abgeleitet werden drei Unterklassen, die die jeweilige Form von Imperfektion repräsentieren, also Fuzzy, Imprecise und Uncertain (siehe unten, Abbildung 8). Um der von Eclipse vorgegebenen Struktur gerecht zu werden, werden jeweils Interface und Klasse erstellt. Von diesen werden einige auch nur als Rumpf-Klassen/-Interfaces benötigt, sie beinhalten somit keine Implementierung. Es wurde auch eine Fabrik-Klasse angelegt, um Instanzen der einzelnen Klassen zu erzeugen. Je nach Art der Imperfektion benötigen die imperfekten Spalten auch Funktionen um zum Beispiel eine Zugehörigkeit zu einem Term zu berechnen bzw. auszudrücken (Fuzzy und Imprecise). Diese Funktionen stammen zum Beispiel aus der Fuzzylogik.

10.3 Zugehörigkeitsfunktionen aus der Fuzzylogik

Zugehörigkeitsfunktionen $\mu_A(x)$ können beliebige nicht negative Werte haben (nicht nur zwischen 0 und 1). Diese Arbeit hält sich dabei an die Definition von Zimmermann [15], nach der alle Zugehörigkeitsfunktionen mit $\sup_x \mu_{\tilde{A}} = 1$ als normalisiert bezeichnet werden. Nichtnormale Funktionen können normalisiert werden, indem man den Funktionswert durch das Supremum teilt. Nach Definition Zimmermanns ist eine Fuzzy-Menge eine Erweiterung einer klassischen Menge und eine Zugehörigkeitsfunktion verallgemeinert die 'charakteristische Funktion'. Eine Fuzzy-Menge ist dabei immer eine Menge von geordneten Paaren (Wert, Grad der Zugehörigkeit).

Andreas Merkel zeigt in seiner Studienarbeit[10], dass Zugehörigkeitsfunktionen im Graph beispielsweise als Trapezoid dargestellt werden können. Diesen Trapezoiden kann man sehr einfach durch die vier Eckpunkte beschreiben und er eignet sich auch gut zum implementieren. Geht man davon aus, dass die zu beschreibende Funktion normalisiert ist und der Funktionswert zwischen 0 und 1 linear wächst bzw. fällt, braucht man nur vier Punkte anzugeben um die Funktion korrekt und eindeutig zu beschreiben.

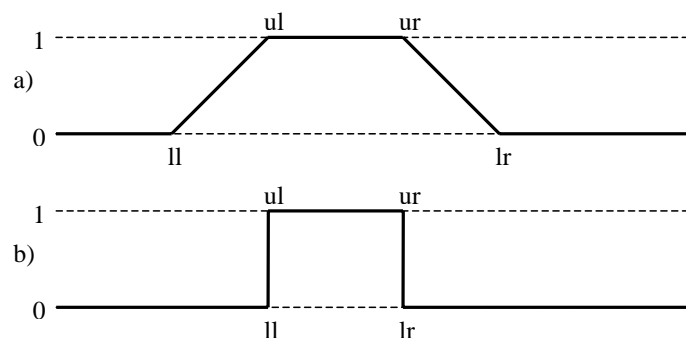


Abbildung 6: TrapezoidMembershipFunction a) Fuzzy b) Scharf

Man benötigt den äußersten linken und rechten Punkt an dem der Funktionswert gerade noch 1 ist und den innersten linken und rechten Punkt an dem der Funktionswert noch 0 ist. Diese Punkte sind in Abbildung 6 dargestellt. Daraus kann man falls nötig auch eine Funktion errechnen, die die Werte der Zugehörigkeitsfunktion zwischen 0 und 1 zurückgibt. Mit einem solchen Trapezoid sind sowohl scharfe als auch unscharfe Einteilungen mittels der Zugehörigkeitsfunktion möglich. Bei scharfen Einteilungen springt der Funktionswert von 0 auf 1 und zurück, wohingegen bei einer unscharfen Einteilung der Funktionswert auf einer Strecke stetig wächst bzw. fällt (siehe Abbildung 6 a und b). Es sind auch andere Steigungen zwischen der unteren und oberen Funktionswertgrenzen vorstellbar, z.B. exponentielle Steigung. Diese müssten dann gegebenenfalls in neuen Funktionsklassen implementiert werden.

Um eine solche Funktion repräsentieren zu können, muss eine neue Vererbungshierarchie (siehe Abbildung 8) aufgebaut werden. Diese wird an das *Core*-Package angehängt. Es gibt dann eine Klasse *Function* von der die zwei hier benötigten Funktionen, *MembershipFunction* für `<<Fuzzy>>` und `<<Imprecise>>` Spalten und *ProbabilityFunction* für `<<Uncertain>>` Spalten, erben. Weiter wurde eine Klasse für eine spezielle *MembershipFunction*, die *TrapezoidMembershipFunction*, in Anlehnung an die Arbeit von Andreas Merkel implementiert. Diese Klassen besitzt als vier Attribute die vier Eckpunkte der Funktion. Ein Term ist das in Kapitel 6.2 beschriebene Konstrukt aus dem Kontextmodell. Wie die Abbildung 7 zeigt, gehören Terme einer Linguistischen Variablen an und zu jedem Term gehört je eine Zugehörigkeitsfunktion und umgekehrt. Das Attribut *Term* ist also nötig um der Funktion später einen Term einer Linguistischen Variablen zuordnen zu können. Die von *Function* geerbte Methode `getValue(double parameter)` gibt den Funktionswert in Abhängigkeit des übergebenen Parameters zurück.

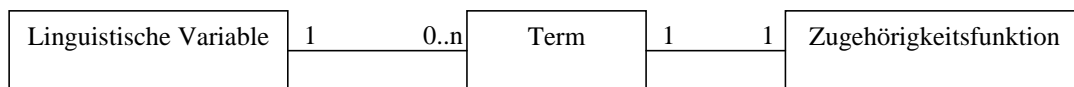


Abbildung 7: Zusammenhang zwischen linguistischen Variablen, Termen und Zugehörigkeitsfunktionen

Klassen für Wahrscheinlichkeitsfunktionen wurden aus zeitlichen Gründen nicht weiter betrachtet.

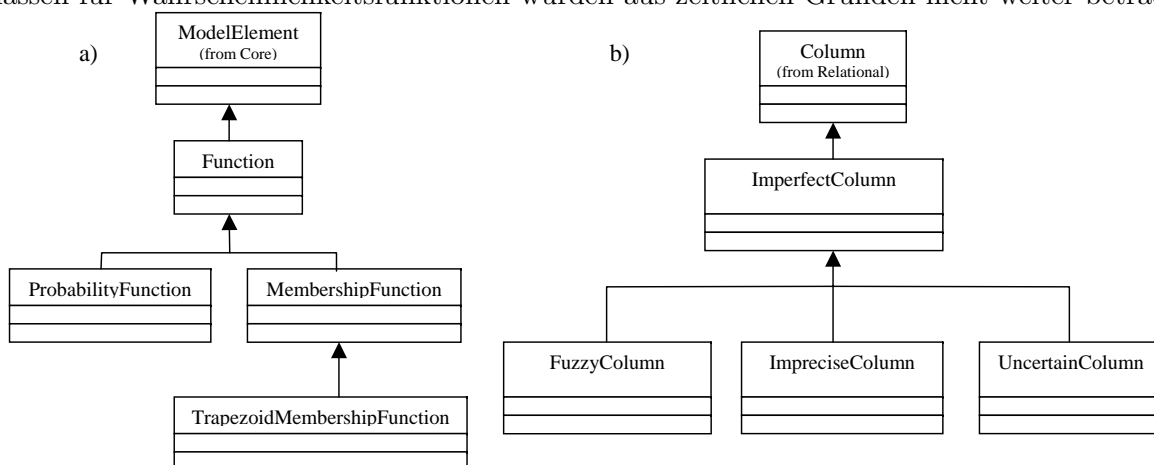


Abbildung 8: Vererbungshierarchien von a) Funktionen und b) Imperfekten Spalten aus dem Package `de.uka.ipd.cwm.imperfect`

Die Abbildung 8 zeigt die Zugangspunkte der erzeugten Klassen/Interfaces in die jeweiligen Packages und die Vererbungsstruktur. Die Funktionen wurden in das *Core*-Package eingeordnet, da es grund-

legende Definitionen sind, die oft gebraucht werden könnten. Eine Eingliederung in das *Relational*-Package stellte sich zuvor als schwierig heraus, da in diesem Package kein geeigneter Punkt zum Anfügen vorhanden ist. Bei der Erweiterung der Spalten hingegen war die Einbindung an das CWM klar. Diese wurden in das *Relational*-Package eingebunden. Dabei wurden die Klassen allerdings nicht direkt in die Original-Packages des CWM geschrieben, sondern es wurde ein neues Package `de.uka.ipd.cwm.imperfect` mit `de.uka.ipd.cwm.imperfect.impl` gebildet und an das CWM angehängt. In dieses Package werden sämtliche bereits beschriebenen Erweiterungen gespeichert, d.h. es finden sich in diesem Package sowohl die Klassen und Interfaces für *Stereotypes* als auch für Funktionen und imperfekte Spalten. Als Vorlage hierfür diente die von der OMG herausgegebene Erweiterung für das ER-Modell, welches ebenfalls in einem eigenen Package angefügt wurde.

11 Anforderungen an das neu erstellte Werkzeug

Nachdem bereits grundlegend geklärt war, dass die Cognos Suite für die weitere Bearbeitung und Evaluation dieser Arbeit nicht ausreichend ist, wurde nach neuen Möglichkeiten gesucht, die bis hier erstellten Erweiterungen zu testen. Die Cognos Suite zu erweitern war aus technischer und zeitlicher Sicht unmöglich. Aus diesen Erkenntnissen wurde beschlossen die Aufgabenstellung dieser Arbeit etwas zu erweitern. Deswegen soll nun ein Werkzeug erstellt werden, das mit dem CWM und den erstellten Erweiterungen umgehen kann.

In den vorherigen Kapiteln konnte man einiges über den Aufbau und die Arbeit mit dem CWM erfahren. Aus diesen Erfahrungen ergeben sich jetzt die Anforderungen an das neu zu erstellend Werkzeug. Obligatorisch ist der Umgang des Werkzeuges mit dem CWM. Des weiteren soll es aber auch mit den Erweiterungen zurecht kommen. Der Austausch von Metadaten in der MOF und im CWM wird über XMI (XML Metadada Interchange) abgewickelt, d.h. das Werkzeug muss auch XMI-Dateien und somit XML verstehen können. Aus diesem Grund muss ein XML-Parser an das Werkzeug angeschlossen werden. In dieser Arbeit wird der XML-Parser **XERCES** von APACHE benutzt. **XERCES** wurde gewählt weil der Parser frei von APACHE.ORG zum Download angeboten wird und er sowohl nach dem SAX- als auch nach dem DOM-Prinzip parsen kann [5]. Es wurden auch leicht verständliche Beispiele mitgeliefert, was die Einarbeitungszeit erheblich verkürzte.

Ziel des Werkzeuges ist es mit der gegebenen Erweiterung so zu verfahren, dass die übergebene Datei nach der Bearbeitung an ein kommerzielles Data-Warehouse-Werkzeug (z.B. Cognos Suite, Oracle Warehouse Builder) weitergegeben werden kann und dieses die, wie auch immer geartete, weitere Verarbeitung erledigt. Eine ausführliche Beschreibung zu den einzelnen Entwurfentscheidungen des Werkzeuges finden sich in Kapitel 15.

Teil IV

Umsetzung der Erweiterungen des CWM

12 Die Umsetzung der simple Extension

Im ersten Konzeptansatz wurde das CWM mit *Stereotypes* und *TaggedValues* erweitert, um Imperfektion sichtbar zu machen. Um der, von Eclipse vorgegebenen, Struktur Rechnung zu tragen, wurden zwei Packages erstellt. In dem ersten Package sind nur Interfaces und im zweiten Package sind die Implementierungen der zugehörigen Klassen. Nun wurden, dem Konzept aus Kapitel 9 entsprechend, Interfaces und Klassen für *ImperfectionStereotype*, *ImperfectTableStereotype*, *FuzzyStereotype*, *ImpreciseStereotype* und *UncertainStereotype* erzeugt. Die erzeugten Java-Klassen und Interfaces erben direkt oder indirekt von dem Interface *Stereotype* aus dem *Core*-Package, wobei in den Interfaces der neu erzeugten Klassen eigentlich keine Information steckt. Alle benötigten Informationen stecken in den implementierten Klassen. Die Vererbungshierarchie wird in Abbildung 5 dargestellt. In Abbildung 9 ist die Erweiterung beispielhaft an der Tabelle *Verkehrsmeldung* aufgezeigt.

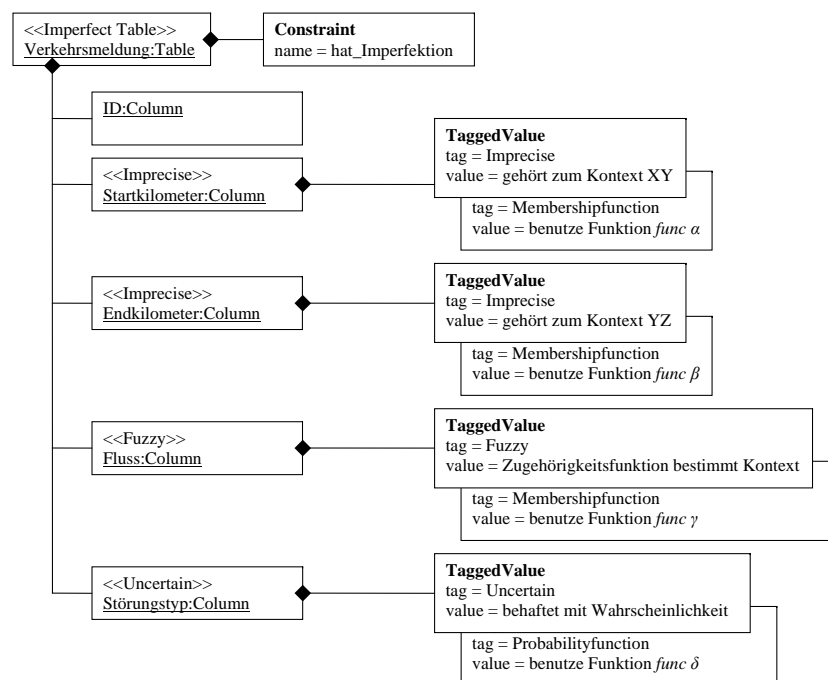


Abbildung 9: Beispiel einer Tabelle mit Imperfektion

Abbildung 9 zeigt die vereinfachte, imperfekte Tabelle *Verkehrsmeldung*. Diese Tabelle wurde mit dem *Stereotype* `<<Imperfect Table>>` behaftet. Sie enthält 5 Spalten, von denen alle bis auf eine imperfekte Daten enthalten. Die Spalten 'Start- und Endkilometer' haben ungenaue Werte, sind also mit dem *Stereotype* `<<Imprecise>>` behaftet. Die Spalte 'Fluss' hat unscharfe Werte und besitzt den *Stereotype* `<<Fuzzy>>`. Die Spalte mit dem Störungstyp ist unsicher und hat `<<Uncertain>>` als *Stereotype*. Es sind also alle drei Arten von *Stereotypes* für Spalten in dieser Tabelle enthalten. Erik Koop beschreibt in seiner Arbeit [9] auch noch Spalten, deren Werte gleichzeitig zwei verschiedene Arten von Imperfektion enthalten, z.B. unsicher UND ungenau. Dies kann in diesem Konzept dadurch gelöst werden, dass neue *Stereotypes* eingeführt werden (z.B. `<<Uncertain&Imprecise>>`). Diese besitzen dann alle

verpflichtenden *TaggedValues* der beiden zugrundeliegenden Imperfektionsarten, in diesen Beispiel von <<Uncertain>> und von <<Imprecise>>.

Am Beispiel der Klasse *FuzzyStereotypeImpl* wird nun die Vorgehensweise bei der Implementierung erläutert.

```
package de.uka.ipd.cwm.imperfect.impl;

import org.omg.cwm.objectmodel.core.CoreFactory;
import org.omg.cwm.objectmodel.core.TaggedValue;
import de.uka.ipd.cwm.imperfect.FuzzyStereotype;

public class FuzzyStereotypeImpl
    extends ImperfectionStereotypeImpl
    implements FuzzyStereotype {

    protected FuzzyStereotypeImpl() {
        super();
        TaggedValue tv;
        //setzt die BaseClass auf Column
        setBaseClass("Column");

        //FuzzyStereotype-Tag setzen
        tv = CoreFactory.eINSTANCE.createTaggedValue();
        tv.setTag("FuzzyStereotype");
        tv.setValue("Benutze Zugehoerigkeitsfunktion!");
        getRequiredTag().add(tv);

        //Zugehrigkeitsfunktion-Tag setzen
        tv = CoreFactory.eINSTANCE.createTaggedValue();
        tv.setTag("MembershipFunction");
        tv.setValue("Beschreibung einer Zugehoerigkeitsfunktion!");
        getRequiredTag().add(tv);
    }
}
```

Wie in Kapitel 10 erwähnt wurde, befinden sich die neu erstellten Interfaces in einem eigenen Package mit Namen *de.uka.ipd.cwm.imperfect* und die dazugehörigen, implementierten Klassen im Unterpaket *de.uka.ipd.cwm.imperfect.impl*.

Im Konstruktor der Klasse *FuzzyStereotypeImpl* wird zunächst die Basisklasse auf 'Column' gesetzt. Dies zeigt an, dass dieser *Stereotype* nur an Spalten angeheftet werden kann. Danach werden Instanzen der verpflichtenden *TaggedValues* mit Hilfe der *CoreFactory* erzeugt, deren Name und Wert gesetzt und mit '*getRequiredTag().add(tv);*' als verpflichtend angehängt. Im Beispiel der Tabelle aus Abbildung 9 sind es zwei *TaggedValues*, es können aber jede beliebige Anzahl *TaggedValues* sein.

Die Implementierungen der anderen *Stereotypes* sind analog aufgebaut. Der *ImperfectTableStereotype* hat allerdings statt der *TaggedValues* nur ein *Constraint*, das sicherstellen soll, dass mindestens eine Spalte dieser Tabelle mit einem *ImperfectionStereotype* behaftet ist.

Ziel dieses Konzeptes war und ist sicherzustellen, dass alle Anwender die gleichen *Stereotypes* beim Umgang mit Imperfektion benutzen (Vereinheitlichung des Wissens und der Semantik!).

Leider ist der eben beschriebene Ansatz zur Automatisierung nicht so gut geeignet, da die Funktionen nur in Textform beschrieben werden können. Deshalb muss eine mächtigere Methode gefunden

werden, um Vorgänge automatisch ablaufen zu lassen. Eine Möglichkeit, einen Schritt näher an Automatisierung heranzukommen, wird im Folgenden umgesetzt. Wir halten uns dabei an das Konzept aus Kapitel 10.

13 Die Umsetzung der modeled Extension

In diesem Teil der Konzeptumsetzung geht es nun um die Erweiterung der im *Relational*-Package beschriebenen *Columns* und um Funktionen. Wie oben besprochen, wird nun die Klasse *Column* so erweitert, das man imperfekte Spalten generieren und diesen dann Funktionen zuordnen können. Um einer Spalte eine Funktion zuordnen zu können muss man diese durch das CWM beschreiben können. Deshalb wurde das Interface *Function* erstellt und daraus *MembershipFunction* und *ProbabilityFunction* abgeleitet. Hier wird, wie in Kapitel 10.3, von normalisierten Funktionen ausgegangen, weshalb man eine Zugehörigkeitsfunktion auch als Trapezoid beschreiben kann. In diesem Beispiel wird also eine spezielle *MembershipFunction*, die *TrapezoidMembershipFunction* (siehe Abbildung 6 aus Kapitel 10.3) implementiert.

Nach Definition sind die obere und die untere Schranke fest auf 1 bzw. 0 gesetzt. Des weiteren wird davon ausgegangen, dass die Zugehörigkeit zu einem Term zwischen 0 und 1 linear wächst bzw. abfällt (siehe [10]). Deshalb benötigt man, um die Funktion zu beschreiben, nur vier Werte. Diese Werte repräsentieren die Punkte auf dem Zahlenstrahl, ab denen die Funktion 0 bzw. 1 annimmt. Das heißt die beiden innersten Punkte an denen die Funktion noch 0 ist und die beiden äußersten Punkte an denen die Funktion den Wert 1 annimmt. Daraus lassen sich mit Hilfe zweier Gleichungen auch die Zwischenwerte einer unscharfen Einteilung ausrechnen:

$$\mu(x) := \begin{cases} 0 & , \text{ falls } x \leq ll \\ (x - ll) * 1 / (ul - ll) & , \text{ falls } ll < x < ul \\ 1 & , \text{ falls } ul \leq x \leq ur \\ 1 - ((x - ur) * 1 / (lr - ur)) & , \text{ falls } ur < x < lr \\ 0 & , \text{ falls } ur \leq x \end{cases}$$

(ll = lowerleft, ul = upperleft, ur = upperright, lr = lowerright, parameter x = Eingabewert)

Die Gleichungen für die Funktionswerte der Steigungen wurden wie oben implementiert, die Werte 1 bzw. 0 werden mit anderen Bedingungen abgefangen und gegebenenfalls ausgegeben.

Stellt die Funktion eine scharfe Einteilung dar, sind die zwei rechten und auch die zwei linken Punkte identisch und der Funktionswert springt an diesen Punkten von 0 nach 1 bzw. 1 nach 0 (siehe Abbildung 6 b aus Kapitel 10.3)).

Die Funktionsklassen wurden, wie die Klassen der ersten Erweiterung auch, in dem neu gebildeten Package `de.uka.ipd.cwm.imperfect` abgelegt und an das *Core*-Package angehängt. Zugangspunkt ist hier die Klasse *ModelElement*, von der alle Funktionen erben.

Nachdem diesem Schritt können nun Funktionen mit CWM beschrieben werden. Als nächstes werden die von *Column* geerbte Klassen und Interfaces erstellt, die imperfekte Columns repräsentieren. In diesen Klassen wird auf die entsprechende Funktion verwiesen bzw. die benötigte Funktion aufgerufen. Beispielhaft wird dies nachfolgend an der Klasse *FuzzyColumnImpl* erläutert.

```
package de.uka.ipd.cwm.imperfect.impl;

import de.uka.ipd.cwm.imperfect.FuzzyColumn;
import de.uka.ipd.cwm.imperfect.Function;
```

```
public class FuzzyColumnImpl
  extends ImperfectColumnImpl
  implements FuzzyColumn {

  protected static final Function FUNCTION_EDEFAULT = null;
  protected Function function = FUNCTION_EDEFAULT;
  protected Function[] functionarray = new Function[10];
  protected int arraypointer = 0;

  protected String[] terms;
  protected String term;
  protected String associatedColumn;

  public FuzzyColumnImpl() {
    super();
  }

  public void setFunction (Function newfunction) {
    functionarray[arraypointer] = newfunction;
    function = newfunction;
    arraypointer++;
  } //setFunction

  public Function getFunction () {
    return function;
  }

  public Function[] getFunctions(){
    return functionarray;
  }

  public void setTerms(String[] argv){
    for(int j=0; j < argv.length ;j++){
      terms[j] = argv[j];
    }
  }

  public void linkTermFunction(String argt, Function argf){
    boolean term_ok = false;
    for (int t = 0;t < terms.length; t++){
      if(terms[t] == argt){
        term_ok = true;
      }
    } //for
    for (int j = 0; j < functionarray.length;j++){
      if (functionarray[j] == argf & term_ok){
        argf.setTerm(argt);
        break;
      }
    }
  }
}
```

```

    }//for
}//linkTermFunction

public double getTermFunctionValue (double x, String term){
    boolean validreturn = false;
    double returnvalue = 0;
    for (int i = 0; i< functionarray.length; i++){
        if (functionarray[i].getTerm() == term){
            returnvalue = functionarray[i].getValue(x);
            validreturn = true;
            break;
        }
    }//for
    return returnvalue ;
}//getTermFunctionValue

public double getFunctionValue (double x) {
    return function.getValue(x);
}

public void setAssotiatedColumn(String column){
    assotiatedColumn = column;
}

public String getAssotiatedColumn(){
    return assotiatedColumn;
}
}

```

In der oben beschriebenen Klasse `FuzzyColumnImpl` werden folgende get/set-Methode beschrieben: Mit der Methode `'setFunction(Function newfunction)'` können der Spalte Funktionen zugeordnet werden. Danach können mit den Methoden `'getFunction()'` bzw. `'getFunctions()'` die zuletzt eingetragenen bzw. alle zugehörigen Funktionen ausgegeben werden. Mit `'setTerms(String[] argv)'` speichert man alle angegebenen Terme `argv`.

Die Methode `'linkTermFunctions(String argt, Function argf)'` setzt den Term `argt` einer Funktion `argf`. Man kann sich den Zugehörigkeitswert eines Wertes `x` zu einem Term `term` mit der Methode `'getTermFunctionValue(double x, String term)'` ausgeben lassen. Des weiteren liefert `'getFunctionValue(double x)'` zu einem Inputwert den Wert der zuletzt eingetragenen Funktion. Mit `'set/getAssotiatedColumn'` kann die Spalte ausgegeben bzw. gesetzt werden, aus der die Inputwerte für die Zugehörigkeitsfunktionen stammen.

Mit den nach obigen Schema erstellten Klassen ist es nun möglich mit dem CWM eine Tabelle zu beschreiben, die unter anderem auch imperfekte Spalten enthält. Diesen Spalten kann dann, je nach Art der Imperfektion, eine oder mehrere Funktionen zugewiesen werden, welche dann ihre Funktionswerte in die jeweilige Spalte eintragen. Am Beispiel des Szenarios aus Kapitel 6.1 würde das ganze wie folgt aussehen. Die Fuzzy-Spalte *Verkehrsfluss* wäre zum Beispiel in eine innere Tabelle mit mehreren Spalten (so viele wie Terme vorhanden) aufgeteilt, also je eine innere Spalte für Frei, Stockend, Zählfließend, usw.. Diesen inneren Spalten wird jeweils eine Zugehörigkeitsfunktion zugeordnet. Die Funktion liefert dann ihren Funktionswert an die Spalte zurück und trägt ihn dort ein (liese sich mit Triggern gut ausführen). Als zugehöriger Term wird dann der Term verwendet, der am ehesten zutrifft, d.h. dessen Zugehörigkeitsfunktion den Wert 1 zurückliefert, oder der 1 am nächsten kommt. Es ist auch

denkbar, dass in die Spalte nur der zugehörige Term eingetragen werden soll. In diesem Fall muss es eine weitere externe Tabelle geben, in der jeder Term in je einer Spalte, mit der jeweiligen Zugehörigkeitsfunktion, aufgeführt ist. So eine Tabelle fungiert dann als eine Art Referenztabelle. Dann muss man mit Triggern oder ähnlichem dafür sorgen, dass der Term, der am ehesten zutrifft, in die ursprüngliche Tabelle eingetragen wird.

Teil V

Evaluation der Erweiterungen

Nachdem das CWM erweitert wurde, soll nun auch gezeigt werden, wie Werkzeuge mit diesen Erweiterungen umgehen könnten. Wie bereits angedeutet, gibt es mehrere Möglichkeiten mit imperfekten Spalten in einer Tabelle zu arbeiten. Diese werden im weiteren Verlauf kurz erläutert und eine davon wird zusätzlich noch an einem Beispiel illustriert.

14 Verfahren mit imperfekten Spalten in der Praxis

Hat man eine imperfekte Spalte in einer Relation, so ist erst nicht klar, wie diese Imperfektion in der Datenbasis dargestellt werden soll. Will man zum Beispiel in einer Spalte die Staulänge natürlichsprachlich beschreiben, so gibt es mehrere Möglichkeiten. Staulänge als Linguistische Variable hat nach [10] zum Beispiel drei Terme: kurz, mittel und lang. Diese Terme besitzen jeweils eine Zugehörigkeitsfunktion.

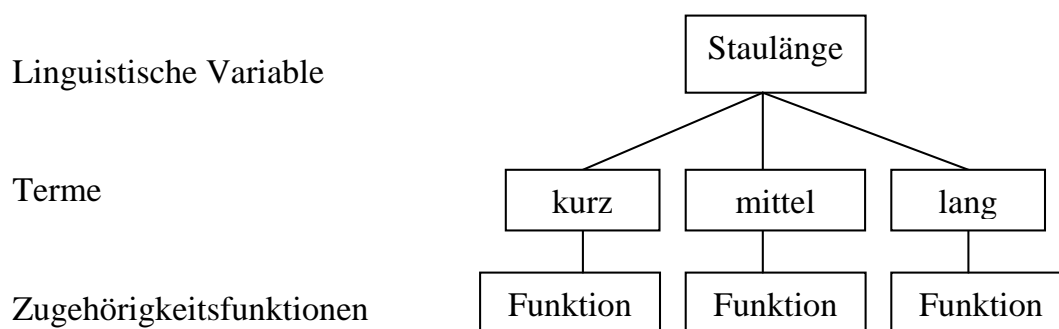


Abbildung 10: Hierarchie einer Linguistischen Variablen

Das Hauptaugenmerk liegt in dieser Arbeit auf Fuzzyspalten, deshalb werden diese auch an Beispielen erläutert, wohingegen die Behandlung anderer Imperfektion (Uncertain und Imprecise) etwas weniger ausführlich erklärt und nur nebenbei erwähnt wird.

14.1 Speichern in 'Externer Relation'

Sei die imperfekte Spalte einer Relation die natürlichsprachliche Aussage über die Staulänge. Diese soll nun mit einer externen Relation abgespeichert werden. In dieser Externen Tabelle stehen alle möglichen Staulängen mit den zugehörigen Werten der Zugehörigkeitsfunktionen zu den Termen. Das heißt, die externe Tabelle *Staulänge* hat soviel Spalten, wie die linguistische Variable *Staulänge* Terme besitzt und eine Spalte mit den Kilometerangaben von 0 bis unendlich (oder bis einer Länge die nie erreicht werden wird, zum Beispiel 150 km). In unserem Beispiel hätte sie vier Spalten: 'kurz', 'mittel', 'lang' mit den jeweiligen Zugehörigkeitswerten und 'Kilometer'. Diese Tabelle wird einmal angelegt und gefüllt. Der Anwender benutzt diese dann nur noch als Referenz und sucht sich mit Hilfe eines Joins den Kilometerwert heraus, der in seinem Fall zutrifft.

Tabelle: Staulänge (Referenz)			
Kilometer	kurz	mittel	lang
1	1	0	0
2	1	0	0
....
6	0.4	0.6	0
7	0.1	0.9	0
8	0	1	0
....
....
150	0	0	1

Abbildung 11: externe Beispielreferenztable für Staulänge

14.2 Speichern innerhalb der Ursprungsrelation

Im oberen Beispiel wurden die Imperfekten Spalten in einer externen Relation abgelegt. Es gibt allerdings noch weitere Möglichkeiten mit imperfekten Spalten umzugehen. Eine davon ist die Speicherung innerhalb der Ursprungsrelation. Dabei fügt man so viele Spalten in die Ausgangsrelation neu ein, wie Terme in der Linguistischen Variablen sind. Diese Spalten repräsentieren dann im Ganzen die Linguistische Variable. Sie werden beim befüllen mit den Zugehörigkeitswerten aufgefüllt, welche aus den jeweiligen Zugehörigkeitsfunktionen errechnet werden. Als zugehöriger Term gilt auch hier der Term mit dem höchsten Zugehörigkeitswert. Die Eintragung der Werte kann mittels eines Triggers geschehen.

Ein Vorteil der Anlegung neuer Spalten in der Ursprungstabelle gegenüber der externen Referenztable ist, das man die Zugehörigkeit direkt aus der Tabelle auslesen kann ohne einen Join zu verwenden. Weiter müssen nur genau die Kilometerwerte in den Zugehörigkeitsfunktionen ausgerechnet werden, die auch wirklich in der Tabelle gespeichert sind. Dies ist je nach Größe der Referenztable ein Speicherplatzvorteil.

Nachteilig ist es die Werte für jeden Eintrag der Tabelle immer wieder neu berechnen zu müssen und nicht einfach abgelesen werden können. Je nach Komplexität der Zugehörigkeitsfunktionen kann eine Referenztable eine beachtliche Einsparnis an Rechenzeit bedeuten.

15 Das Tool zum Testen der CWM-Erweiterungen

Im Rahmen dieser Arbeit wurde auch ein Werkzeug erstellt, welches mit den vorher erzeugten Erweiterungen des CWM umgehen kann. Ziel des Werkzeugs soll eine Umsetzung der Erweiterung des CWM sein, damit Werkzeuge, die mit der Erweiterung nicht umgehen können, mit dem Output dieses Werkzeugs normal weiterarbeiten können. Das Werkzeug wurde in Java geschrieben, da das CWM-Modell komplett in Java vorliegt und die Ergänzungen hierzu ebenfalls in Java verfasst wurden. Zuerst wurde die graphische Oberfläche so einfach wie möglich erzeugt, da dieses Werkzeug nur eine Ergänzung zu kommerziellen Data-Warehouse-Werkzeugen sein soll. Zum Erstellen der Oberfläche wurde Java-Swing eingesetzt, um eine breite Unterstützung der darunterliegenden Hardware zu erreichen. Des Weiteren wurde auf Funktionalität geachtet, weshalb die Applikation nur 5 Buttons und ein Textfeld besitzt (siehe Abbildung 12). Auf ein Menu wurde aus diesem Grunde ebenfalls verzichtet.

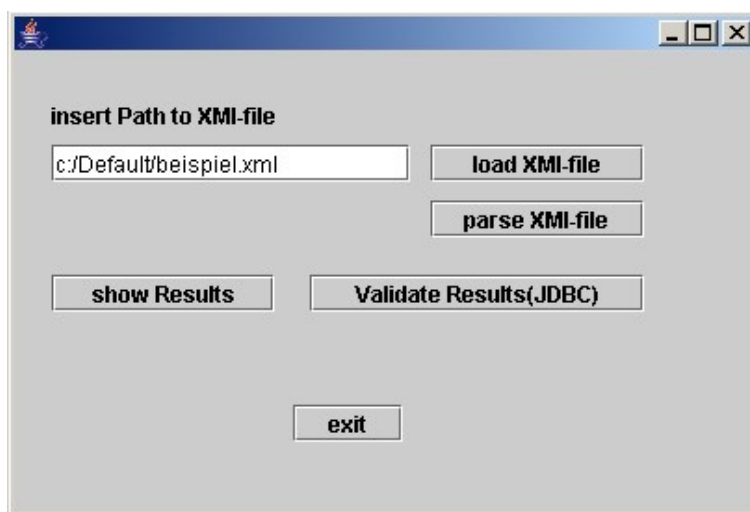


Abbildung 12: Werkzeug zum Testen der CWM-Erweiterungen

Zur Funktionsweise des Werkzeugs ist folgendes zu berichten:

Dem Werkzeug wird eine XML-Datei, in der ein relationales Schema beschrieben wird, übergeben. Diese XML-Datei muss der DTD (Document Type Definition) für XMI (XML Metadata Interchange), welche von der OMG herausgegeben wurde, genügen. Die DTD-Beschreibung für das gesamte CWM findet man unter [3]. Diese DTD enthält alle Packages des CWM-Schichtenmodell und deren Objekte. Da in dieser DTD verständlicherweise die Erweiterungen nicht enthalten waren, wurde sie im Rahmen dieser Arbeit um die CWM-Extension für Imperfektion ergänzt. In der DTD wurde ein neuer Namespace 'CWMEXT' definiert. Unter diesem Namespace finden sich alle Ergänzungen, die mit der CWM-Erweiterung zu tun haben. Dieses Package (CWMEXT) enthält derzeit die Beschreibung für *Fuzzy-Spalten* und *TrapezoidMembershipsFunktionen*. Weitere Ergänzungen wie die Beschreibung für *Imprecise-Spalten* und *Uncertain-Spalten*, sowie weiteren Zugehörigkeitsfunktionen können analog dazu ebenfalls leicht eingefügt werden.

Die an die Applikation übergebene XML-Datei muss also nun mit dieser erweiterten DTD validiert werden können. Die Applikation, wie sie in dieser Arbeit erstellt wurde, liest die übergebene Datei ein und reicht sie weiter an einen externen XML-Parser. Zum Parsen des XML-Files wird der Xerces-XML-Parser verwendet. Der Xerces-XML-Parser ist ein von `Apache.org` entwickelter Parser. Dieser Parser implementiert die W3C XML und DOM (Level 1 und 2) Standards, sowie den SAX (version 2) quasi-standard [5]. In diesem Beispiel wird der DOM-Parser verwendet, da später eventuell auf dem Dokumentenbaum Änderungen vorgenommen werden sollen. Der SAX-Parser lässt nur eine Anfrage

an einen Knoten des Baumes zu, verwirft diesen danach aber wieder, ist also zum verändern des XML-files ungeeignet.

Nachdem die Datei durch den Parser validiert und geparkt wurde, wird der so aufgebaute DOM-Tree genutzt, um einen SQL-String zu generieren. Dieser SQL-String beschreibt das in der XML-Datei codierte SQL-Schema. Dieser Teil der Applikation funktioniert für Schemata mit und ohne Imperfektion. Schemata ohne Imperfektion werden genau so ausgegeben, wie sie in der XML-Datei beschrieben werden. Sollte innerhalb des Schemas Imperfektion enthalten sein, zum Beispiel eine Fuzzy-Spalte, löst die Applikation diese auf.

Das Werkzeug hält sich in der jetzigen Form an die in Kapitel 14.2 beschriebene Variante zum Umgang mit imperfekten Spalten. Die Applikation erzeugt innerhalb der Relation neue Spalten für die Terme einer Fuzzy-Spalte, fügt diese ein und löscht die ursprüngliche Fuzzy-Spalte. Um später noch Aussagen über die Zusammenhänge der Terme und Linguistischen Variablen treffen zu können werden zwei weitere Tabellen erstellt: `CWMX_OVID_LINGVAR` und `CWMX_OVID_COLTERM`. In diesen Tabellen werden die Daten über die Zusammenhänge der Terme zur einer Linguistischen Variablen und deren Zugehörigkeitsfunktionen eingetragen. Die Tabelle `CWMX_OVID_LINGVAR` besteht aus sechs Spalten, eine für die Linguistische Variable, eine für die Terme und je eine für die vier Eckpunkte der *TrapezoidMembershipFunction*. Mit der Tabelle `CWMX_OVID_COLTERM` wird die Verbindung zwischen der Ursprungstabelle und `CWMX_OVID_LINGVAR` hergestellt. Sie enthält fünf Spalten, eine für den Namen der Ursprungstabelle, eine für den Namen der Termspalte innerhalb der Ursprungstabelle. In die weiteren Spalten werden der Name der Inputspalte aus der Ursprungstabelle und der Schlüssel der Tabelle `CWMX_OVID_LINGVAR`, also die Spalten für die Linguistische Variable und den Term, eingetragen.

Die Möglichkeit eine Referenztable anzulegen wird in dieser Arbeit nicht berücksichtigt.

Denkbar wäre auch, das Werkzeug so umzuschreiben, das es die Änderungen im Schema gleich in die ursprüngliche XML-Datei schreibt und diese speichert. Das würde aber den Rahmen dieser Arbeit bei weitem übersteigen.

15.1 Beispieleingaben

Als Testbeispiel wurde die Tabelle aus dem Szenario in Kapitel 6.1 gewählt. Diese wurde in ein Schema und einen Katalog integriert und in XML codiert. Als DTD für die XML-Datei dient die von der OMG herausgegebene Beschreibung des CWM in XMI. Dieses XMI-Datei wurde wie oben beschrieben erweitert und enthält nun auch Definitionen um Imperfektion zu beschreiben. Die so entstandene XML-Datei mit der Beispieltabelle (siehe unten) beinhaltet besagten Relationalen Katalog 'VISSIM' mit einem Schema namens `CWMX-Imperfektion-Schema`. Die Namen hierfür wurden willkürlich gewählt. In diesem Schema gibt es eine Tabelle `Verkehrsmeldungen`. Die Tabelle hat sechs Spalten, eine davon beinhalten unscharfe-oder Fuzzy-Werte. Die Spalte mit der Imperfektion heißt `Verkehrsfluss`. Die Linguistische Variable `Verkehrsfluss` hat die vier im Szenario beschriebenen Terme. Jedem Term ist eine *TrapezoidMembershipFunction* zugeordnet. Des weiteren sind in der Datei die erlaubten Spaltentypen mit `SQLStructuredType` definiert. Die so definierten Typen können einer Spalte als Typ übergeben werden. Beispielsweise hat die Spalte `MeldungsID` den Typ `VARCHAR` und die Spalte `Eingangszeit` den Typ `Date`.

Der folgende Code stammt aus der XML-Datei, die zum Testen dem Werkzeug übergeben und weiter oben in diesem Kapitel beschrieben wurde.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE XMI SYSTEM "cwm02-05-02_ext.dtd">
<XMI xmi.version="1.1" xmlns:CWM="de.uka.ipd.cwm">
  <XMI.header>
```

```

<XMI.metamodel xmi.name="CWM">
  <CWMRDB:Catalog xmi.id="catalog_1" name="VISSIM">
    <CWM:Namespace.ownedElement>
      <CWMRDB:Schema xmi.id="schema_1" name="CMWX-Imperfektion-Schema"
        namespace="catalog_1">
        <CWM:Namespace.ownedElement>
          <CWMRDB:Table xmi.id="tabelle_1" name="Verkehrsmeldungen"
            namespace="schema_1">
            <CWM:Classifier.feature>
              <CWMRDB:Column xmi.id="spalte_1" name="MeldungsID"
                owner="tabelle_1" type="VARCHAR" length="10"/>
            </CWM:Classifier.feature>
            <CWM:Classifier.feature>
              <CWMRDB:Column xmi.id="spalte_2" name="Eingangszeit"
                owner="tabelle_1" type="DATE"/>
            </CWM:Classifier.feature>
            <CWM:Classifier.feature>
              <CWMRDB:Column xmi.id="spalte_3" name="Stoerungstyp"
                owner="tabelle_1" type="VARCHAR" length="30"/>
            </CWM:Classifier.feature>
            <CWM:Classifier.feature>
              <CWMRDB:Column xmi.id="spalte_4" name="Strasse"
                owner="tabelle_1" type="VARCHAR" length="6"/>
            </CWM:Classifier.feature>
            <CWM:Classifier.feature>
              <CWMRDB:Column xmi.id="spalte_5" name="Geschwindigkeit"
                owner="tabelle_1" type="NUMBER"/>
            </CWM:Classifier.feature>
            <CWM:Classifier.feature>
              <CWMEXT:FuzzyColumn xmi.id="fuzzyspalte_1"
                assoatiatedColumn="spalte_3" owner="tabelle_1"
                name="Verkehrsfluss">
                <CWM:Classifier.feature>
                  <CWMEXT:TrapezoidMembershipFunction
                    xmi.id="function_1" ll="80" ul="90" ur="350"
                    lr="350" term="frei"/>
                </CWM:Classifier.feature>
                <CWM:Classifier.feature>
                  <CWMEXT:TrapezoidMembershipFunction
                    xmi.id="function_2" ll="60" ul="70" ur="80"
                    lr="90" term="lebhaft"/>
                </CWM:Classifier.feature>
                <CWM:Classifier.feature>
                  <CWMEXT:TrapezoidMembershipFunction
                    xmi.id="function_3" ll="30" ul="40" ur="60"
                    lr="70" term="stockend"/>
                </CWM:Classifier.feature>
                <CWM:Classifier.feature>
                  <CWMEXT:TrapezoidMembershipFunction

```

```

        xmi.id="function_4" ll="0" ul="0" ur="30"
        lr="40" term="stau"/>
    </CWM:Classifier.feature>
  </CWMEXT:FuzzyColumn>
</CWM:Classifier.feature>
</CWMRDB:Table>
  </CWM:Namespace.ownedElement>
</CWMRDB:Schema>
</CWM:Namespace.ownedElement>
</CWMRDB:Catalog>
<CWMRDB:SQLStructuredType xmi.id="INTEGER" typeNumber="1"/>
<CWMRDB:SQLStructuredType xmi.id="CHARACTER" typeNumber="2"/>
<CWMRDB:SQLStructuredType xmi.id="DATE" typeNumber="3"/>
<CWMRDB:SQLStructuredType xmi.id="FLOAT" typeNumber="4"/>
<CWMRDB:SQLStructuredType xmi.id="NUMBER" typeNumber="5"/>
<CWMRDB:SQLStructuredType xmi.id="BOOLEAN" typeNumber="6"/>
<CWMRDB:SQLStructuredType xmi.id="VARCHAR" typeNumber="7"/>
</XMI.metamodel>
</XMI.header>
</XMI>

```

15.2 Ausgabe des Werkzeugs

Das erstellte Werkzeug übernimmt die im Kapitel 15.1 beschriebene Beispieldatei nachdem von Anwender der 'Load-Button' betätigt wurde. Mit dem 'Parse-Button' wird der Parsevorgang gestartet und der SQL-Ausgabestring generiert. Klickt der Anwender nun den 'Show-Result-Button', bekommt er auf der Konsole alle von dem Werkzeug erzeugten SQL-Befehle ausgegeben. Ein Ablaufdiagramm des Programms wird in Abbildung 13 gezeigt.

Dabei wird nach jedem abgeschlossenen Befehl eine neue Zeile begonnen. Somit hat der Anwender den Überblick wie viele einzelne Befehle generiert wurden und welcher Art diese Befehle sind ohne nach dem Anfang der Befehle suchen zu müssen. Klickt der Anwender den Button 'validate Results', so wird, falls der Rechner online ist, eine Verbindung zur OVID01-Datenbank geöffnet. Danach werden die Befehle nacheinander einzeln an die Datenbank geschickt und ausgeführt. Somit wird sichergestellt, dass die von dem Werkzeug generierten SQL-Befehle auch wirklich der SQL-Syntax entsprechen und auf einer SQL-Datenbank ausführbar sind.

Die SQL-Befehle des Werkzeuges setzen sich wie folgt zusammen: Ist ein Element der XML-Datei eine Tabelle mit Namen *Tabellenname*, so wird an den SQL-String ein 'CREATE Tabellenname(' angefügt.

Da meist innerhalb eines Tabellenelementtags ein oder mehrere Spaltenelemente folgen, werden diese im Anschluss an das 'CREATE'-Statement mit Ihrem Namen und ihrem Typ SQL-konform an den String angehängt. Als Beispiel wird für die Spalte *MeldungsID* der String 'MeldungsID VARCHAR(10)' angefügt. Nachdem alle Spalten angehängt wurden, wird der SQL-Befehl mit Semikolon abgeschlossen. Sollte allerdings eine oder mehrere der Spalten Imperfektion enthalten, ist das Einfügen nicht mehr so einfach möglich. In diesem Fall sucht sich das Werkzeug die Terme der imperfekten Spalte und fügt diese jeweils als Spalte mit dem Typ INTEGER in die Tabelle bzw. den SQL-Befehl ein. Des Weiteren werden beim ersten Vorkommen einer Imperfektion die Tabellen CWMX_OVID_LINGVAR und

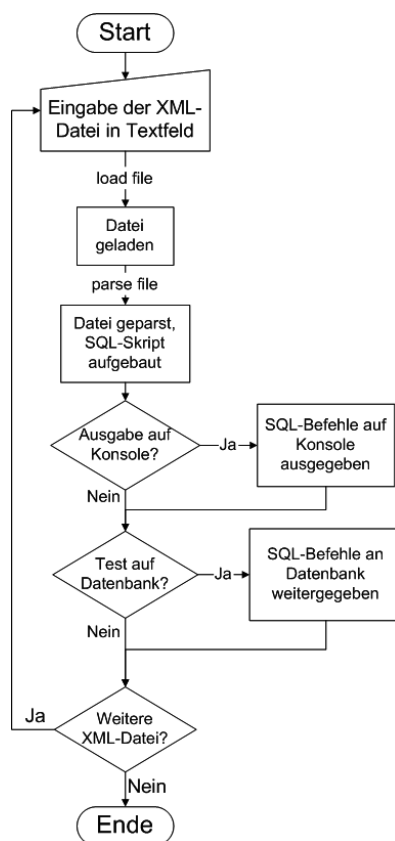


Abbildung 13: Ablaufdiagramm des Werkzeugs zum Testen der CWM-Erweiterungen

CWMX_OVID_COLTERM erstellt. Darin werden, wie in einem früheren Kapitel beschrieben, die Verbindungen zwischen den Linguistischen Variablen (imperfekten Spalten), deren Termen und den Zugehörigkeitsfunktionen abgespeichert. Das Abspeichern der Verbindungen wird mittels eines 'INSERT INTO ...'-Befehls realisiert.

Als nächstes wird hier die Konsolenausgabe des Programmes gezeigt. Zuerst wurde die Datei `cwmbeispiel.xml` geladen. Danach der 'parse-Button' gedrückt. Anschließend wurden die Resultate mit dem 'show Results-Button' ausgegeben.

`cwmbeispiel.xml` geladen.

Parsevorgang korrekt abgeschlossen!

```
DROP Table Verkehrsmeldungen;
```

```
CREATE Table Verkehrsmeldungen (MeldungsID VARCHAR(10),
  Eingangszeit DATE,Stoerungstyp VARCHAR(30),Strasse VARCHAR(6),
  Geschwindigkeit NUMBER,frei INTEGER,lebhaft INTEGER,stockend INTEGER,
  stau INTEGER);
```

```
DROP Table CWMX_OVID_LINGVAR;
```

```
CREATE Table CWMX_OVID_LINGVAR (LingVar VARCHAR(30),
```

```

Terme VARCHAR(30), Funktionswert_1 INTEGER, Funktionswert_2 INTEGER,
Funktionswert_3 INTEGER, Funktionswert_4 INTEGER);

DROP Table CWMX_OVID_COLTERM;

CREATE Table CWMX_OVID_COLTERM (TabellenName VARCHAR(30),
TermSpaltenName VARCHAR(30), InputSpaltenName VARCHAR(30),
LingVar VARCHAR(30), Terme VARCHAR(30));

INSERT INTO CWMX_OVID_LINGVAR VALUES ('Verkehrsfluss','frei',80,90,350,350);

INSERT INTO CWMX_OVID_COLTERM VALUES ('Verkehrsmeldungen','frei','spalte_3',
'Verkehrsfluss','frei');

INSERT INTO CWMX_OVID_LINGVAR VALUES ('Verkehrsfluss','lebhaft',60,70,80,
90);

INSERT INTO CWMX_OVID_COLTERM VALUES ('Verkehrsmeldungen','lebhaft',
'spalte_3','Verkehrsfluss','lebhaft');

INSERT INTO CWMX_OVID_LINGVAR VALUES ('Verkehrsfluss','stockend',30,40,60,
70);

INSERT INTO CWMX_OVID_COLTERM VALUES ('Verkehrsmeldungen','stockend',
'spalte_3','Verkehrsfluss','stockend');

INSERT INTO CWMX_OVID_LINGVAR VALUES ('Verkehrsfluss','stau',0,0,30,40);

INSERT INTO CWMX_OVID_COLTERM VALUES ('Verkehrsmeldungen','stau','spalte_3',
'Verkehrsfluss','stau');
-----

```

Will man nun noch die SQL-Befehle an die Datenbank schicken, muss der Button 'validate Results' gedrückt werden. Die Konsolenausgabe davon ist nicht weiter erwähnenswert.

16 Ergebnisse der Evaluation

Der erste Ansatz einer Erweiterung des CWM aus Kapitel 9, mit dem man mit Hilfe von *Stereotypes* und *TaggedValues* die Imperfektion beschreiben kann, war weit weniger hilfreich als Anfangs erhofft. Zwar kann man mit diesem Ansatz die Imperfektion und die dazugehörigen Funktionen in Textform beschreiben, dies bringt allerdings nicht den erhofften Mehrwert für den Anwender.

Besser war dann der zweite Ansatz, der etwas später in Kapitel 10 geschildert wird. Mit dieser Erweiterung des CWM um ein ganzes Package wurde es möglich weit komplexere und inhaltlich sinnvollere Aufgaben zu bearbeiten.

Das Package besteht zur Zeit aus den oben beschriebenen Klassen für Funktionen, darunter auch die für die *TrapezoidMembershipFunction* und aus den Klassen für imperfekte Spalten. Bei den Klassen für imperfekte Spalten sind alle drei Arten der hier gebrauchten Imperfektion (fuzzy, imprecise und uncertain) implementiert, allerdings fehlen bei der Klasse für *Uncertain-Columns* die zugehörige Funktion. Diese wurde mangels Zeit von der Implementierung ausgenommen.

Diese zweite Erweiterung wurde dann mit dem selbst erstellten Werkzeug auf ihre Funktionsfähigkeit getestet. Dazu wurde eine Beispiel-XML-Datei erstellt, welche ein Relationales Schema mit imperfekten Spalten abbildet. Das Werkzeug generierte aus diesem Schema einige SQL-Befehle, welche dann auch noch gleich an eine Datenbank weitergeleitet werden können. Die Datenbank kann mit Hilfe der ihr übergebenen SQL-Befehle daß komplette Schema aus der XML-Datei aufbauen, in dem die imperfekten Spalten aufgelöst wurden.

Die Auflösung der imperfekten Spalten wurde, wie in Kapitel 14.2 beschrieben, durch einfügen neuer Spalten in die Tabelle realisiert. Die Verbindungen dazu wurden in zwei weiteren Tabellen abgelegt.

Alles in allem ist die zweite, komplexere CWM-Erweiterung eine für unsere Zwecke gelungene Erweiterung. Das Werkzeug zum Verarbeiten der Erweiterung kann allerdings noch um einige zusätzliche Features ergänzt werden. Welche das im einzelnen sind wird im nächsten Kapitel beschrieben.

17 Ausblick

Im Vorfeld der Erstellung der eigentlichen Erweiterung des CWM wurde das CWM in seiner Struktur genauer untersucht und Möglichkeiten der Erweiterung geprüft. Im Anschluss daran kam die Analyse von imperfekten Daten in OVID und die Einarbeitung in ein Data-Warehouse-Werkzeug, die Cognos Suite. Aus diesen Voruntersuchungen wurde dann die erste und zweite Erweiterung entwickelt und umgesetzt. Weiter kam die Erstellung von Anforderungen an ein Werkzeug zur Bearbeitung der Erweiterungen und dessen Implementierung, da die Cognos Suite dazu nicht in der Lage ist. Mit diesem Werkzeug wurde die zweite Erweiterung getestet.

Was in dieser Arbeit leider nicht umgesetzt werden konnte, ist die Fähigkeit des Werkzeuges XML-Dateien selbst zu manipulieren bzw. umzuschreiben oder zu erweitern. Dieser Möglichkeit wurde trotzdem Rechnung getragen, da beim Parsen der XML-Dateien ein DOM-Parser verwendet wurde und kein SAX-Parser und somit der aufgebaute DOM-Tree noch nach dem Parsevorgang erhalten bleibt und verändert werden könnte. Durch diese Wahl des Parsers kann in einem weiteren Schritt das Werkzeug so umgeschrieben werden, das es nicht nur ein Schema aus einer XML-Datei ausliest und als SQL-String zurückliefert, sondern dieses Schema innerhalb der XML-Datei so umändert, dass die Imperfektion, nach der in Kapitel 14.2 beschriebenen Art, aufgelöst bzw. umgesetzt wird.

Neben der ausführlich beschriebenen Erweiterung für Fuzzy-Spalten kann man, nach dem selben Schema, auch noch Imprecise- und Uncertain-Spalten beschreiben. Dies müsste dann in einem nächsten Schritt noch etwas genauer implementiert werden. Denkbar wäre es, diese Schritte in einer weiterführenden Arbeit zu untersuchen und zu implementieren.

Des weiteren kann man sich vorstellen, noch mehr verschiedene Funktionen mit Hilfe der Erweiterung zu realisieren. In der Ausarbeitung wurden nur die *TrapezoidMembershipFunction* exemplarisch herausgestellt und implementiert. Es ist sicher vorstellbar, weit komplexere Zugehörigkeitsfunktionen zu modellieren. Wahrscheinlichkeitsfunktionen wurden hier gar nicht behandelt, sind aber sicher lohnenswert, um in Betracht gezogen zu werden.

Man könnte sich sogar vorstellen, diese Erweiterung, wie die Erweiterung für ER-Modelle, durch die OMG adaptieren zu lassen. Dazu bräuchte man aber sicher noch viele Arbeitsstunden, um die Erweiterung so ausgereift zu bekommen, dass sie ohne Bedenken publiziert und verwendet werden kann.

Literatur

- [1] BUNDESMINISTERIUM für Bildung und Forschung,
„*OVID — Stärkung der SelbstOrganisationsfähigkeit im Verkehr durch I+K-gestützte Dienste*“, Webseite, <http://www.ovid.uni-karlsruhe.de>,
2002
- [2] IPD: Universität Karlsruhe (TH),
„*OVID — Stärkung der SelbstOrganisationsfähigkeit im Verkehr durch I+K-gestützte Dienste: Teilprojekt B1*“, Webseite, <http://www.ipd.uka.de/%7Eovid>,
2003
- [3] Object Management Group,
„*Common Warehouse Metamodel*“, Webseite, <http://www.omg.org/cwm>,
2004
- [4] Eclipse Project,
„*Eclipse Modelling Framework Documents v1.0: Users Guide*“, Webseite,
<http://www.eclipse.org/emf>,
2003
- [5] APACHE XML Project : XERCES2 Java Parser
„*XERCES2 Java Parser Readme*“, Webseite, <http://xml.apache.org>,
2004
- [6] POOLE J., CHANG D., TOLBERT D., and MELLOR D.,
„*Common Warehouse Metamodel, An Introduction*“, OMG-Press,
2002
- [7] POOLE J., CHANG D., TOLBERT D., and MELLOR D.,
„*Common Warehouse Metamodel, Developer's Guide*“, OMG-Press,
2003
- [8] SCHEPPERLE H., MERKEL A., HAAG A.,
„*Erhalt von Imperfektion in einem Data-Warehouse mit Hilfe des Kontextmodells*“, Beitrag zum GI-Symposium 'Data-Warehouse und Data-Mining',
2004
- [9] KOOP Erik Alexander ,
„*Datenbankunterstützung für imperfekte Daten im Verkehrsumfeld*“, Diplomarbeit, Universität Karlsruhe (TH),
2004
- [10] MERKEL Andreas,
„*Konzeption und Umsetzung einer Schemaerweiterung durch Kontexte unter Berücksichtigung von Aggregierungsanfragen*“, Studienarbeit, Universität Karlsruhe (TH),
2004
- [11] IBM, Oracle, Unisys, UBS, NCR, Genesis Dev., and Others,
„*CMW Specification, Proposal to the OMG ADTF RFP*“, OMG Document,
2000

-
- [12] Oracle,
„Oracle Warehouse Builder, Users Guide Release 9.2“, Oracle Document,
2003
- [13] Cognos,
„Cognos Series 7 Dokumentation“, Cognos Document,
2002
- [14] MEIER A. , MEZGER C., WERRO N., SCHINDLER G.,
„Zur unscharfen Klassifikation von Datenbanken mit fCQL“, Universität Fribourg, GI-
Workshopwoche LLWA AKKD,
2003
- [15] ZIMMERMANN H.J.,
Fuzzy Set Theorie – and Its Applications, Sec., rev. edition Kluwer, Boston, Dordrecht,
London,
1992
- [16] SCHINDLER G.,
„Fuzzy Datenanalyse durch kontextbasierte Datenbankanfragen“, Dissertation, TH Aa-
chen,
1998

