Technische Universität Dresden
Fakultät Informatik

Diplomarbeit

Untersuchungen zur Abbildung von OCL-Ausdrücken auf SQL

eingereicht von

Alexander Schmidt
geboren am 16. Mai 1972 in Dresden

Verantwortlicher Hochschullehrer:	Prof. Dr. rer. nat. habil. H. Hußmann
Betreuerin:	Dr. B. Demuth
Institut:	Institut für Softwaretechnik II
Lehrstuhl:	Lehrstuhl Softwaretechnologie
Beginn am:	20.03.1998
Einzureichen am:	21.09.1998

Inhaltsverzeichnis

1 Einleitung

Bei den Programmiersprachen und in der Softwaretechnologie hat in den letzten Jahren das Paradigma der Objektorientierung (objektorientierte Denkweise) eine weite Verbreitung gefunden. Die Modellierung und Implementation von Objekten mit Verhalten und vielen Strukturierungsmöglichkeiten (Vererbung, Assoziationen) zeichnet sich durch ihre Nähe zur natürlichen menschlichen Denkweise aus. Aus diesem Grund ist das objektorientierte Paradigma z.B. sehr gut für die Modellierung von Geschäftsprozessen und für die Implementation von Benutzungsschnittstellen geeignet.

Im Bereich der Datenbanken ist das relationale Paradigma, bei dem alle Informationen auf Tabellen mit Zeilen und Spalten abgebildet werden, noch immer weit verbreitet.
Das liegt vor allem an der Reife der relationalen Datenbanksysteme. Da die Hersteller von relationalen Datenbanksystemen große Umsätze und Rücklagen verzeichnen können, ist ihr Fortbestand gut gesichert. Für die Anwender bedeutet dies Investitionssicherheit. Außerdem haben viele Anwender bereits große Investitionen in die relationale Technologie vorgenommen.
Neben den wirtschaftlichen Vorteilen profitiert das relationale Paradigma aber auch von seinen im Vergleich zum objektorientierten Paradigma wesentlich einfacheren Strukturen und der Existenz allgemein akzeptierter Standards. So ist es bei der Realisierung von Datenbanken nach dem relationalen Paradigma besser möglich, die Aspekte der Sicherheit (Transaktionsschutz und Wiederherstellung im Fehlerfall) intensiv und umfassend zu berücksichtigen. Für relationale Datenbank-Schemata kann leichter sichergestellt werden, daß Fehlbedienungen durch Anwender und Anwendungsprogramme (Fehler des Anwendungsprogrammierers) vom Datenbanksystem automatisch entdeckt, gemeldet und neutralisiert werden und alle Anwendungsprogrammierer und Anwender eine einheitliche Vorstellung von der Semantik des Schemas besitzen. Darauf vertrauen die Anwender in der Wirtschaft, was für diese ein wichtiges Entscheidungskriterium darstellt. Dieses Vertrauen begründet sich auch zu einem wesentlichen Teil in den deskriptiven Integritätsbedingungen in relationalen Datenbankschemata. Sie verhindern auch bei teilweise fehlerhaften Anwendungsprogrammen die Ablage inkonsistenter Daten in der Datenbank mit minimalem Aufwand für den Anwender (bzw. Schema-Designer).

Um die Vorzüge des obejektorientierten und relationalen Paradigmas zu vereinen, werden diese in der Praxis häufig parallel verwendet. So werden oft in Informationssystemen für die Implementierung der Benutzungsschnittstelle objektorientierte Sprachen (mit Klassenbibliotheken für Benutzungsschnittstellenelemente) und für die Implementierung der Datenhaltung relationale Datenbanksysteme eingesetzt. Dabei werden jedoch nicht nur die Vorzüge vereint. Es entsteht auch ein neues Problem. Die Anwendungsprogrammierer müssen zwischen den beiden Paradigmen vermitteln, wobei zusätzlicher Code und neue Fehlerquellen entstehen. Werkzeuge, welche die Anwendungsprogrammierer bei dieser Aufgabe unterstützen bzw. von dieser Aufgabe entlasten sollen, werden als "Relationale Middleware" bezeichnet.

Die für das Vertrauen in Anwendungen auf der Basis relationaler Datenbanken wichtigen Integritätsbedingungen wurden dabei jedoch bis jetzt noch nicht ausreichend berücksichtigt.
Dies liegt zum Teil daran, daß wichtige objektorientierte Notationen (OMT, Booch, UML 1.0) für die Beschreibung über die Darstellungsmittel des Klassendiagramms hinausreichender struktureller Einschränkungen bisher verbale oder anwenderspezifische Notationen empfohlen haben. Mit der UML 1.1 wurde jedoch die OCL zur deskriptiven Beschreibung von Integritätsbedingungen in objektorientierten Modellen eingeführt. In dieser Arbeit soll deshalb untersucht werden, wie "Integritätsbedingungen" aus objektorientierten Modellen (OCL-Invarianten) auf relationale Datenbank-Schemata (SQL-Constraints) abgebildet werden können.

Nach einer detaillierten Untersuchung der OCL (Kapitel 2), deren Vergleich mit ähnlichen Sprachen (Kapitel 3) und einer kurzen Vorstellung der Integritätssicherung (Kapitel 4) soll eine Abbildung von OCL nach SQL gefunden werden. Das Kapitel 5 leitet die Abbildung von OCL nach SQL ein und erklärt zu Beginn den Aufbau der Kapitel 6 bis 10. Die Arbeit wird mit einer Zusammenfassung abgeschlossen (Kapitel 11).

2 Vorstellung der OCL

Die OCL ist Bestandteil der UML 1.1 und wird in [OCL97] beschrieben. Dieses Dokument enthält zwar etwa zu einem Drittel formale Beschreibungen der Sprache (Signatur der Features, Grammatik), wesentliche Elemente wie die Navigationsoperatoren werden jedoch nur verbal beschrieben. Für den in der Arbeit folgenden Vergleich mit ähnlichen Sprachen und vor allem für die Abbildung auf SQL wird jedoch eine vollständige und übertragbare formale Beschreibung der wichtigsten Strukturen der OCL benötigt. Darum hat die Vorstellung der OCL in diesem Kapitel neben der Einstimmung des Lesers vor allem die Beschreibung der OCL mit anderen, geeigneteren formalen Mitteln zum Ziel.
Zunächst sollen die Datentypen und Operatoren der OCL in den ersten beiden Abschnitten möglichst kurz, aber systematisch vorgestellt werden. Der dritte Abschnitt stellt die Aspekte der OCL vor, welche sich nicht in das Schema Datentypen/Operatoren einordnen lassen.
Im vierten Abschnitt wird mit der Untersuchung von Äquivalenzbedingungen der Versuch einer noch intensiveren Strukturierung und Komprimierung der Beschreibung der OCL unternommen.

2.1 Die Datentypen der OCL

Die Datentypen der OCL können grob in Elementardatentypen und Kollektionstypen eingeteilt werden. In [SchaKort98] (Abschnitt 4.1) wird festgestellt, daß die Beziehungen zwischen den Typen in [OCL97] mit der verbalen Beschreibung nicht eindeutig beschrieben sind. Es wurde in dieser Arbeit eine Interpretation gewählt, welche andere Probleme mit der OCL möglichst gut entschärft. So entsteht im Gegensatz zu [SchaKort98] in der gewählten Interpretation kein „Meta-Zyklus“[1], da OclType nicht als Subtyp von OclAny aufgefaßt wurde. Die im folgenden dargestellten Klassendiagramme widersprechen demzufolge der Darstellung in Abschnitt 4.1 in [SchaKort98].

2.1.1 Elementardatentypen (Basic Types)

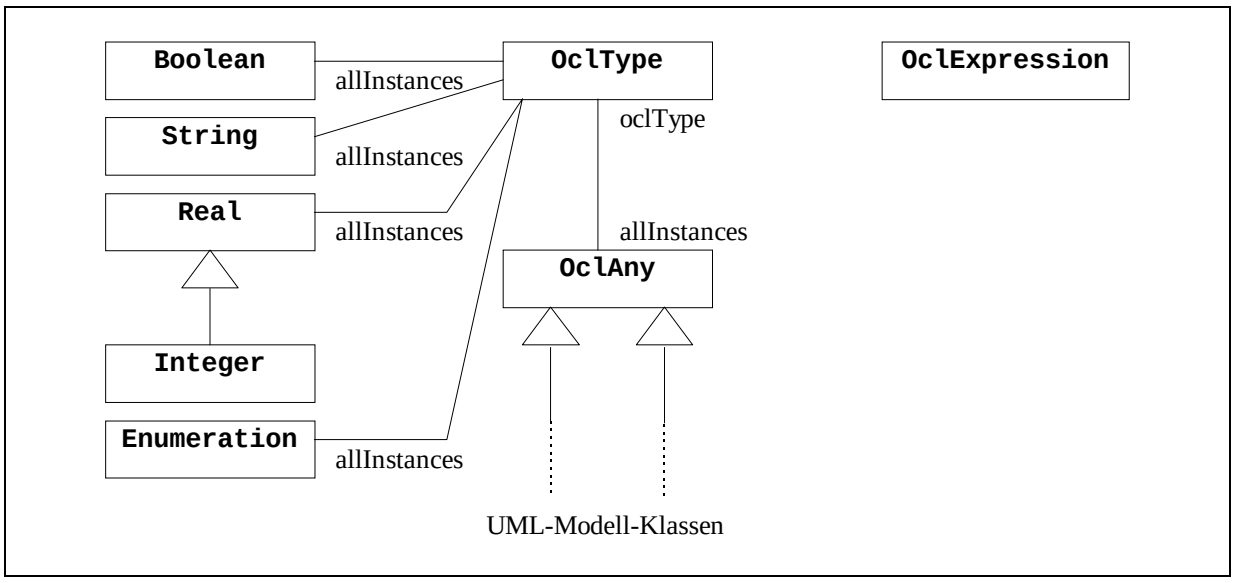


Abbildung 1: Klassendiagramm zu den Elementardatentypen

Die Elementardatentypen umfassen die einfachen Datentypen Boolean, String, Real, Integer und Enumeration für die Konzepte von Wahrheitswerten, ASCII-Zeichenketten, reellen und ganzen Zahlen sowie Aufzählungstypen.
Alle Klassen im UML-Modell erben von der in der OCL definierten Klasse OclAny. Diese Klasse besitzt unter anderem auch eine Methode „oclType“, welche eine Instanz der (Meta-)Klasse OclType als Ergebnis

[1] [SchaKort98]: „But what is the result of OclType.oclType?“

zurückgibt. Diese Instanz von OclType repräsentiert den Typ der betreffenden Instanz von OclAny welcher z.B. die Ermittlung aller Instanzen (allInstances) ermöglicht.
Der Elementardatentyp OclExpression besitzt nur eine Operation (evaluationType). Er wird für die Definition der OCL selbst verwendet, um Funktionen höherer Ordnung behandeln zu können. Da mit der Darstellung der Operatoren durch ihre Signatur in dieser Arbeit eine ausdrucksstärkere Darstellungsform gewählt wurde und ein anderer Zweck von OclExpression nicht gefunden werden konnte, wird auf OclExpression nicht weiter eingegangen (d.h. es wird im folgenden davon ausgegangen, daß OclExpression nicht zu den Elementardatentypen gehört bzw. gar nicht existiert).

2.1.2 Kollektionstypen (Collection-related Types)

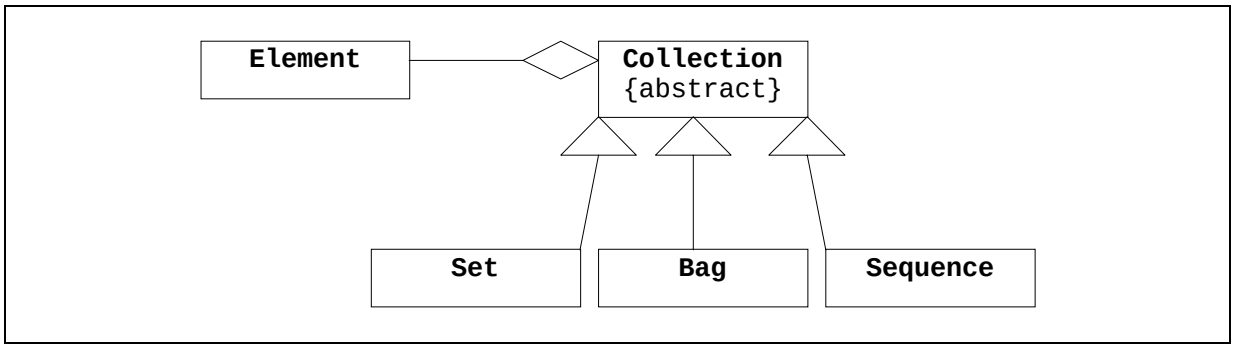


Abbildung 2: Klassendiagramm zu den Kollektionstypen

Für alle Kollektionstypen definiert die OCL einen abstrakten Obertyp „Collection“. Kollektionen enthalten Elemente. Mengen (Set) können ein Objekt höchstens einmal als Element besitzen. Multimengen (Bag) können Objekte mehrfach als Element benennen. Geordnete Multimengen (Sequence) besitzen darüber hinaus eine definierte, von der Anwendung bestimmte Ordnung der Elemente.

2.2 Die Operatoren der OCL

Unter Operatoren der OCL sollen die elementaren Bausteine verstanden werden, aus denen OCL-Ausdrücke zusammengesetzt werden. Jeder Operator übernimmt eine Anzahl von Parametern, aus denen er (ohne irgend etwas in seiner Umgebung zu verändern) ein Ergebnis konstruiert.
In [OCL97] werden die Operatoren im objektorientierten Sinne entsprechend dem Typ des ersten Parameters den Datentypen zugeordnet. Diese Darstellung hat zur Folge, daß C++- oder Java-Programmierer ([OCL97] S. 1: „durchschnittliche Geschäfts- oder Systemmodellierer“) schnell intuitiv OCL-Ausdrücke verstehen und erstellen können.
Für den Vergleich mit anderen Ansätzen und die Abbildung in die nicht-objektorientierte relationale Welt ist diese Gruppierung jedoch wenig hilfreich. Eine Bindung der Operatoren an bestimmte Datentypen ist nicht sinnvoll, da kein Operator den Zustand irgendwelcher Objekte verändert. Die meisten Operatoren haben eine verbindende (umwandelnde) Funktion zwischen den Datentypen (z.B. Prüfen auf Enthaltensein in einer Menge: Menge und Elementardatentyp als Parameter, Wahrheitswert als Ergebnis).

Die gewählte alternative Gruppierung der Operatoren wird in der folgenden Abbildung illustriert und anschließend erläutert.

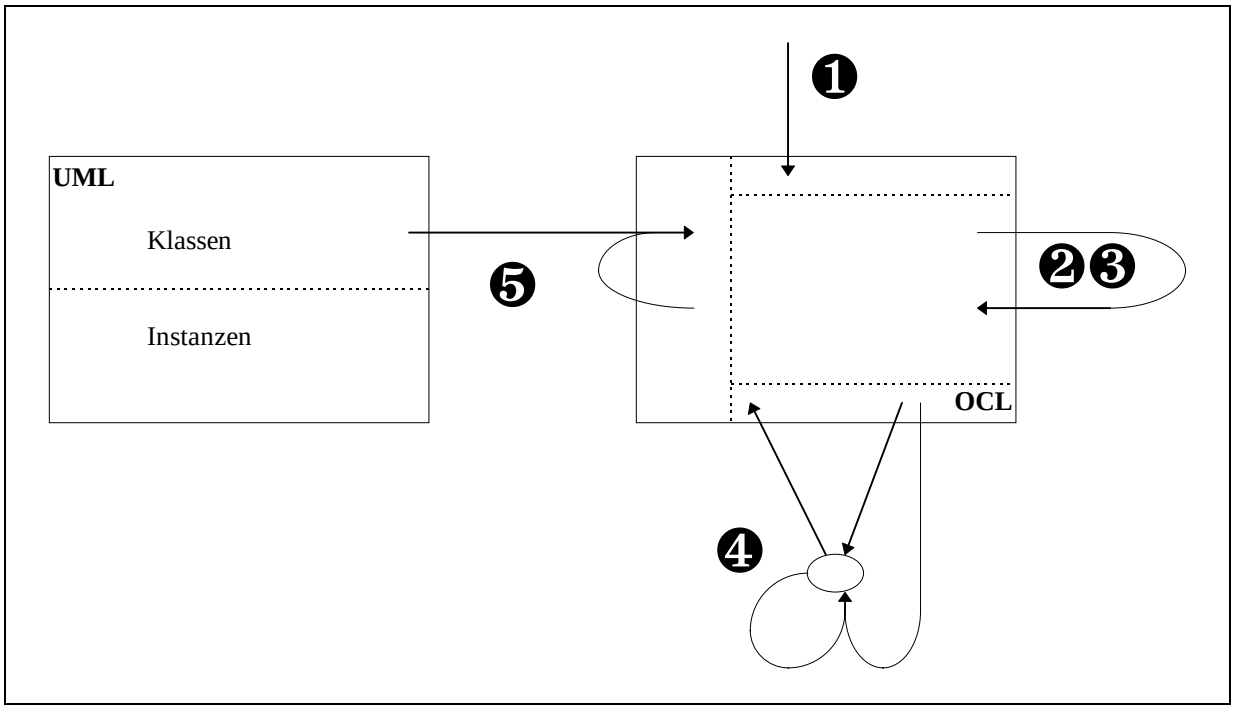


Abbildung 3: Gruppen von OCL-Operatoren

Die Operatoren der OCL können in 5 große Gruppen eingeteilt werden:
❶ Konstanten (z.B. Integerkonstante)
❷ Einfache OCL-Operatoren über Elementardatentypen (z.B. Addition)
❸ Einfache OCL-Operatoren über Kollektionstypen (z.B. Mengendurchschnitt)
❹ OCL-Operatoren vom Typ einer Funktion höherer Ordnung (z.B. select und iterate)
❺ Operatoren mit Zugriff auf Daten und Metadaten des Anwendungsmodells (z.B. Navigation)

Die Graphik illustriert die Gründe für die Einordnung.
Konstanten (❶) erzeugen Ausprägungen von OCL-Datentypen ohne einen Parameter zu benötigen.
Einfache OCL-Operatoren (❷ und ❸) erzeugen Ausprägungen von OCL-Datentypen anhand der Ergebnisse anderer OCL-Teilausdrücke. Sie sind die größte Gruppe und werden deshalb nochmals in einfache OCL-Operatoren über Elementardatentypen (❷) und einfache OCL-Operatoren über Kollektionstypen (❸) unterteilt.
OCL-Operatoren höherer Ordnung (❹) verhalten sich wie einfache OCL-Operatoren. Darüber hinaus übernehmen sie jedoch mindestens einen OCL-Ausdruck als Funktion anstelle seines Auswertungsergebnisses und rufen diese 0..n-mal mit konstruierten Parametern auf.
Operatoren mit Zugriff auf Daten und Metadaten des Anwendungsmodells (❺) übernehmen als Parameter Ergebnisse anderer OCL-Teilausdrücke und beziehen sich bei ihrer Auswertung auf ein UML-Modell, aus dem sie Informationen entnehmen.

2.2.1 Technik der Beschreibung der Operatoren

Während die OCL in [OCL97] durch verbale Beschreibung (Navigationsoperatoren der Gruppe ❺) und programmiersprachenähnliche, objektorientierte Sprache (übrige Operatoren) beschrieben wird, soll die OCL in dieser Arbeit durch die Signatur ihrer Operatoren beschrieben werden. Diese Form wurde aus folgenden Gründen gewählt:

- Es wird eine einfache und kompakte, aber aussagekräftige Beschreibung der OCL ermöglicht.
- Formale Umwandlungen werden besonders gut unterstützt (z.B. Abbildung auf andere Sprachen und allgemeine Formulierung von äquivalenten Ausdrücken).
- Die Vergleichbarkeit mit anderen formalen Sprachen und anderen Ansätzen (z.B. aus dem Wissensgebiet Datenbanken) wird erleichtert.

- Die Darstellung ist frei von den in [SchaKort97] (Abschnitt 4.1) beschriebenen drei Arten von Polymorphismus, welche in [OCL97] verwendet werden.

Ein Nachteil der Darstellungsweise ist, daß nicht alle Aspekte der OCL, welche in [OCL97] (oft nur verbal) beschrieben wurden, in Zusammenhang mit den Operatoren dargestellt werden können. Diese Aspekte werden im Abschnitt 2.3 behandelt.

Die Operatoren werden in der folgenden Form dargestellt:

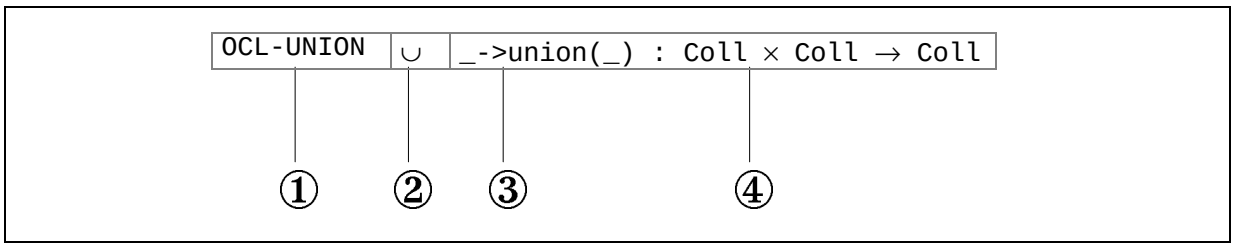


Abbildung 4: Darstellung der Operatoren

Jeder Operator wird mit einer mnemonischen Bezeichnung (①) versehen, um die Bezugnahme zu erleichtern. Außerdem können die überladenen Operatoren (siehe [SchaKort98] 4.1) mit dieser Bezeichnung unterschieden werden. Falls vorhanden, wird das Operationssymbol aus der Mathematik (②) angegeben. Bei der nun folgenden Signatur des Operators beschreibt dessen Name (③) die Syntax in der OCL. An den Stellen mit den Unterstrichen befinden sind in der OCL-Syntax die Parameter. Hinter dem Doppelpunkt folgen die Typen der Parameter (④). Parameter können auch Funktionen sein. Eine solche Funktion höherer Ordnung wird dargestellt, indem der Parameter selbst wieder als Signatur beschrieben und in Klammern gesetzt wird.
Manche Operatoren können eine variable Zahl von Operatoren besitzen. Dafür wird die Darstellungsweise `{ }`^{n} verwendet. Navigationsoperatoren stellen jeweils ein Muster für eine Menge von Operatoren dar. Aus dem Anwendungsmodell müssen die konkreten Operatoren konstruiert werden, indem für den Ausdruck in spitzen Klammern (`<...>`) konkrete Modellelemente eingesetzt werden (z.B. Attributnamen).
Um Platz zu sparen und damit die Übersicht zu erhöhen, werden die Typnamen abgekürzt und Oberbegriffe für Typen verwendet. Die folgende Tabelle stellt die Abkürzungen vor:

Typname	Kurzform des Typnamen
`Basic`	**`Basic`**
`Boolean`	`Bool`
`Integer`	`Int`
`Real`	`Real`
`String`	`Str`
`Enumeration`	`Enum`
`OclType`	`Type`
`OclAny`	`Any`
`Collection`	**`Coll`**
`Set`	`Set`
`Bag`	`Bag`
`Sequence`	`Seq`

Basic ist dabei ein Oberbegriff für die Typen Boolean, Integer, Real, String, Enumeration, OclType und OclAny. Collection ist ein Oberbegriff für Set, Bag und Sequence.
Um die Anzahl der Operatoren zu minimieren, wurden für die Typen der Parameter Mengen und Oberbegriffe von Typen zugelassen. Typmengen werden durch Hintereinanderschreiben der Typnamen dargestellt (z.B.: `IntReal`). `BasicColl` bezeichnet somit einen beliebigen Typ.

2.2.2 Konstanten

Für alle Elementardatentypen mit Ausnahme von OclAny beschreibt die OCL die Darstellung von Konstanten. Die Darstellung von konstanten Werten wird zwar auf Seite 4 (4.) in [OCL97] an Beispielen vorgestellt, eine explizite Definition von Konstanten erfolgt aber nicht. Aus der Verwendung von Konstanten in den Beispielen (z.B. [OCL] S. 13) kann deren Existenz jedoch mit Sicherheit geschlossen werden. Die Darstellung von konstanten Aufzählungswerten wird auf Seite 5 (4.2) eingeführt. Boolean-Konstanten werden auf Seite 24 verwendet. Ein konstanter Typ wird im Beispiel auf Seite 12 (5.10) verwendet (auch in [UML-SEMA] S. 56).

OCL-CONBOOL		_ : → Bool
OCL-CONINT		_ : → Int
OCL-CONREAL		_ : → Real
OCL-CONSTR		'_' : → Str
OCL-CONENUM		#_ : → Enum
OCL-CONTYPE		_ : → Type

Die Zeile
`literal := <STRING> | <number> | "#" <name>`
in der OCL-Grammatik soll offenbar String-, Integer- und Enumeration-Konstanten definieren.
Die Grammatik sollte wie folgt korrigiert werden:

```
literal                 := constant
constant                := integerConstant | realConstant | booleanConstant
                           | stringConstant | enumerationConstant
                           | oclTypeConstant
integerConstant         := <number>
realConstant            := <number> "." <number>
booleanConstant         := "true" | "false"
stringConstant          := <string>
enumerationConstant     := "#" <name>
oclTypeConstant         := <typeName>
```

Die folgende Tabelle mit Beispielen für Konstanten wurde aus [OCL97] (S. 4) übernommen und erweitert.

Typ	**Werte**
Boolean	`true, false`
Integer	`1, 2, 34, 26524, ...`
Real	`1.5, 3.14, ...`
String	`'To be or not to be...'`
Enumeration	`#value1` ([OCL97] S. 5)
OclType	`Person` ([OCL97] S. 12)

2.2.3 Einfache Operatoren über Elementardatentypen

Operatoren der Booleschen Algebra

Die OCL definiert die von der Aussagenlogik bekannten Booleschen Operatoren.

OCL-COND	= {	if _ then _ else _ endif : Bool × BasicColl × BasicColl → BasicColl
OCL-NOT	¬	not _ : Bool → Bool
OCL-AND	∧	_ and _ : Bool × Bool → Bool
OCL-OR	∨	_ or _ : Bool × Bool → Bool
OCL-IMP	⇒	_ implies _ : Bool × Bool → Bool
OCL-EQUB	⇔	_ = _ : Bool × Bool → Bool
OCL-XOR	xor	_ xor _ : Bool × Bool → Bool

Grundlage aller Booleschen Operatoren ist der bedingte Ausdruck (OCL-COND). Die Datentypen des zweiten und dritten Paramters müssen bei diesem Operator mit dem Datentyp des Ergebnisses verträglich sein. Ist der erste Parameter true, so ist das Ergebnis eine Kopie des zweiten Parameters, andernfalls eine Kopie des dritten Parameters. Obwohl der bedingte Ausdruck durch seine äußere Form eine Anweisung vermuten läßt, handelt es sich doch um einen Ausdruck.
Die folgenden beiden Beispiele illustrieren durch Gegenüberstellung, daß die Konzepte dieser Gruppe aus der Mathematik übernommen wurden.

Beispiele:

$$\neg x = \begin{cases} falsch, wenn\ x\ wahr \\ wahr, andernfalls \end{cases}$$

```
not x = if x then false else true endif
```

$$(x \Rightarrow y) \Leftrightarrow (\neg x \vee y)$$

```
(x implies y) = (not x or y)
```

Operatoren über ganzen und reellen Zahlen

Wie in den Ausführungen zu den OCL-Datentypen dargestellt, ist der Typ Integer als Spezialfall von Real definiert. Überall wo ein Parameter vom Typ Real verlangt wird, wird somit auch ein aktueller Wert vom Typ Integer akzeptiert. Da das Subtyping auf der Ebene von SQL-92 später nicht explizit als Konzept vorhanden sein wird, wird es in der folgenden Darstellung umgangen. Die Typen Real und Integer sind als unterschiedliche Typen (ohne Beziehungen) aufzufassen.

OCL-EQU	=	_ = _ : IntReal × IntReal → Bool
OCL-LT	<	_ < _ : IntReal × IntReal → Bool
OCL-GT	>	_ > _ : IntReal × IntReal → Bool
OCL-LEQ	≤	_ <= _ : IntReal × IntReal → Bool
OCL-GEQ	≥	_ >= _ : IntReal × IntReal → Bool
OCL-NEG	–	- _ : IntReal → IntReal
OCL-ADD	+	_ + _ : IntReal × IntReal → IntReal
OCL-SUB	–	_ - _ : IntReal × IntReal → IntReal
OCL-MUL	.	_ * _ : IntReal × IntReal → IntReal
OCL-DIVF	/	_ / _ : IntReal × IntReal → Real
OCL-FLOOR	⌊ ⌋	_.floor : IntReal → Int
OCL-ABS	| |	_.abs : IntReal → IntReal
OCL-MAX	max	_.max(_) : IntReal × IntReal → IntReal
OCL-MIN	min	_.min(_) : IntReal × IntReal → IntReal
OCL-DIVI	div	_.div(_) : Int × Int → Int
OCL-MOD	mod	_.mod(_) : Int × Int → Int

Für den Ergebnistyp von OCL-ADD, OCL-SUB, OCL-MUL, OCL-ABS, OCL-MAX und OCL-MIN, welcher mit IntReal noch nicht exakt bestimmt ist, gilt:
Das Ergebnis ist vom Typ Real wenn einer der Parameter vom Typ Real ist, andernfalls Integer.
OCL-FLOOR ermittelt den ganzzahligen Anteil einer reellen Zahl. OCL-ABS ist der absolute Betrag einer Zahl. OCL-MIN und OCL-MAX ermitteln das Minimum bzw. Maximum zweier Zahlen. OCL-DIVI und OCL-MOD ermöglichen die Bestimmung von Quotient und Rest bei ganzzahliger Division.
Wie bei der vorhergehenden Gruppe sind alle Konzepte dieser Gruppe aus der Mathematik übernommen worden.

Beispiel:

```
x.max(y) = y.max(x)
```

Operatoren über Zeichenketten

Die OCL definiert Operatoren über ASCII-Strings.

OCL-EQSTR		_ = _ : Str × Str → Bool
OCL-SZSTR		_.size : Str → Int
OCL-CONCAT		_.concat(_) : Str × Str → Str
OCL-TOUPP		_.toUpper : Str → Str
OCL-TOLOW		_.toLower : Str → Str
OCL-SUBSTR		_.substring(_,_) : Str × Int × Int → Str

Der Operator OCL-EQSTR vergleicht zwei Strings. OCL-SZSTR ermittelt die Anzahl der Zeichen eines Strings. OCL-CONCAT ermittelt den String, welcher sich durch das Hintereinanderschreiben der beiden Parameter-Strings ergibt. Die Operatoren OCL-TOUPP und OCL-TOLOW erzeugen nach der Vorgabe des Parameters Strings, welche nur aus Groß- bzw. Kleinbuchstaben bestehen. OCL-SUBSTR ermittelt einen Teilstring des Parameter-Strings. Die Zeichenposition wird ab 1 beginnend nummeriert.

Beispiel:
```
('Zeichenkette').substring(2, 7).toUpper.concat('WALD') = 'EICHENWALD'
```

Operatoren über Aufzählungselemente

Für Aufzählungselemente definiert die OCL die Prüfung auf Gleichheit und Ungleichheit.

OCL-EQUENUM	=	_ = _ : Enum × Enum → Bool
OCL-NEQENUM	≠	_ <> _ : Enum × Enum → Bool

Beispiel:
```
(#rot <> #gruen) = true
```

2.2.4 Einfache Operatoren über Kollektionstypen

Konstruktion von Kollektionen

Konstante Kollektionen können durch Aufzählung ihrer Elemente erzeugt werden. Neben Konstanten zur Beschreibung von Elementen kommen mit OCL-Teilausdrücken und Intervallen noch weitere Möglichkeiten in Frage. Die Beschreibung all dieser Alternativen in jeweils einer Signatur wäre umständlich und unübersichtlich. Aus diesem Grunde werden zwei Hilfsoperatoren eingeführt, deren Ergebnisse nur an OCL-CONSET, OCL-CONBAG und OCL-CONSEQ übergeben werden können. Der Hilfsoperator OCLH-SGLELEM hat keine syntaktische Entsprechung in der OCL. Er wandelt einzelne Instanzen in einelementige Kollektionen um. OCLH-INTRVAL erzeugt für ein durch Unter- und Obergrenze beschriebenes Intervall eine entsprechende Kollektion. Die Operatoren OCL-CONSET, OCL-CONBAG und OCL-CONSEQ übernehmen nun eine beliebige Anzahl von (meist einelementigen) Teilkollektionen, welche sie vereinigen. Die folgende formale Darstellung schließt die Möglichkeit parameterloser Konstruktoren nicht mit ein. Diese Möglichkeit zur Erzeugung leerer Kollektionen ist jedoch vorhanden.

OCLH-SGLELEM		Basic → Coll
OCLH-INTRVAL		_.._ : Int × Int → Coll
OCL-CONSET	{ }	Set{_{,_}^{n}} : Set {× Set}^{n} → Set
OCL-CONBAG		Bag{_{,_}^{n}} : Bag {× Bag}^{n} → Bag
OCL-CONSEQ		Sequence{_{,_}^{n}} : Seq {× Seq}^{n} → Seq

Beispiel:
```
Sequence{1..4, (2 * 3) + 1} = Sequence{1, 2, 3, 4, 7}
```

Erzeugung von Kollektionen aus anderen Kollektionstypen

Zu einer Kollektion kann über die folgenden Operatoren eine ähnliche Kollektion anderen Typs erzeugt werden. OCL-ASSET und OCL-ASBAG verwerfen dazu eine evtl. definierte Ordnung. OCL-ASSET entfernt Duplikate. OCL-ASSEQ erzwingt eine undefinierte (zufällige) Ordnung (!, siehe auch [SchaKort98] (4.2)).

OCL-ASSET		_->asSet : BagSeq → Set
OCL-ASBAG		_->asBag : SetSeq → Bag
OCL-ASSEQ		_->asSequence : SetBag → Seq

Beispiel:
```
Sequence{2, 3, 3, 1}->asSet = Set{1, 2, 3}
```

Aggregatfunktionen auf Kollektionen

OCL-SIZE bestimmt die Anzahl der Kollektionselemente. OCL-COUNT bestimmt, wie oft eine bestimmte Instanz als Element in einer Kollektion enthalten ist. Die Summe aller Kollektionselemente (OCL-SUM) kann für Kollektionen von ganzen und reellen Zahlen ermittelt werden.

OCL-SIZE	||	_->size : Coll → Int
OCL-COUNT		_->count(_) : Coll × Basic → Int
OCL-SUM	Σ	_->sum : Coll → IntReal

Beispiel:
```
Bag{1, 2, 1, 3, 2, 4, 2, 2, 2}->count(2) = 5
```

Typische Mengenoperationen auf Kollektionen

Die OCL definiert die Operatoren OCL-INC, OCL-EXC, OCL-APP und OCL-PREP zum Einfügen und Entfernen einzelner Elemente in bzw. aus Kollektionen.
Auf dieser Grundlage können die typischen Mengenoperationen der Mathematik (∪, ∩, Δ, \) definiert werden. Die Anwendbarkeit von Vereinigung und Durchschnitt wird in der OCL über Mengen hinaus auch auf Multimengen ausgedehnt. Der Vereinigungs-Operator kann auch auf geordnete Multimengen angewendet werden, verliert dabei jedoch seine Kommutativität. Die Operandenreihenfolge wird hier zur Bestimmung der Ordnung im Ergebnis verwendet.

OCL-INC	∪ { }	_->including(_) : Coll × Basic → Coll
OCL-EXC	\ { }	_->excluding(_) : Coll × Basic → Coll
OCL-APP	∪ { }	_->append(_) : Seq × Basic → Seq
OCL-PREP	{ } ∪	_->prepend(_) : Seq × Basic → Seq
OCL-UNION	∪	_->union(_) : Coll × Coll → Coll
OCL-INTER	∩	_->intersection(_) : SetBag × SetBag → SetBag
OCL-SYMDIF	Δ	_->symmetricDifference(_) : Set × Set → Set
OCL-DIFFR	\	_ - _ : Set × Set → Set

Beispiele:

```
Set{4, 7}->including(9) = Set{4, 7, 9}

(Set{2, 4}->union(Set{7, 8, 9})) - Set{4, 7} = Set{2, 8, 9}
```

Besondere Operatoren für geordnete Multimengen

Für geordnete Multimengen existieren Operatoren, welche auf die Ordnung (Reihenfolge der Elemente) direkt bezug nehmen. Die Nummerierung der Elemente beginnt bei 1.

OCL-AT		_->at(_) : Seq × Int → Basic
OCL-FIRST		_->first : Seq → Basic
OCL-LAST		_->last : Seq → Basic
OCL-SUBSEQ		_->subSequence(_,_) : Seq × Int × Int → Seq

OCL-AT ermittelt das Element an der angegebenen Position. OCL-FIRST und OCL-LAST ermitteln das erste bzw. letzte Element. OCL-SUBSEQ ermittelt eine Teilsequenz.

Beispiel:
```
Sequence{3, 4, 6}->subSequence(1, 2) = Sequence{3, 4}
```

Relationale Mengenoperatoren auf Kollektionen

Die relationalen Mengenoperatoren werden aus der Mathematik übernommen und ihre Anwendung auf Multimengen und geordnete Multimengen ausgedehnt. Der Operator OCL-SUBSET verhält sich für (geordnete) Multimengen anders, als man u.U. intuitiv erwartet. Sein Verhalten entspricht einer Umwandlung der Parameter in Mengen (durch Entfernen von Ordnung und Duplikaten) und einer anschließenden Teilmengenprüfung für gewöhnliche Mengen.

OCL-ISEMPT	= ∅	_->isEmpty : Coll → Bool
OCL-NOTEMPT	= ∅	_->notEmpty : Coll → Bool
OCL-ELEM	∈	_->includes(_) : Coll × Basic → Bool
OCL-SUBSET	⊆(!)	_->includesAll(_) : Coll × Coll → Bool
OCL-CMPCOLL	=	_ = _ : Coll × Coll → Bool

Beispiel:
```
Bag{1, 2, 3}->includesAll(Bag{1, 2, 2}) =
               Bag{1, 2, 3}->asSet->includesAll(Bag{1, 2, 2}->asSet) = true
```

2.2.5 Operatoren vom Typ einer Funktion höherer Ordnung

Ohne das Konzept einer reinen Ausdruckssprache zu durchbrechen und die OCL mit zu vielen Möglichkeiten einer programmiersprachenähnlichen Verwendung auszustatten, erreichen die Operatoren vom Typ einer Funktion höherer Ordnung eine beträchtliche Erweiterung der Fähigkeiten der OCL. Aus OCL-ITERATE lassen sich alle anderen Operatoren dieser Gruppe herleiten.

OCL-FORALL	∀	_->forAll(_) : Coll × (Basic → Bool) → Bool
OCL-EXISTS	∃	_->exists(_) : Coll × (Basic → Bool) → Bool
OCL-SELECT		_->select(_) : Coll × (Basic → Bool) → Coll
OCL-REJECT		_->reject(_) : Coll × (Basic → Bool) → Coll
OCL-COLLECT		_->collect(_) : Coll × (Basic → Basic) → Coll
OCL-ITERATE		_->iterate(_) : Coll × BasicColl × (Basic × BasicColl → BasicColl) → BasicColl

All- und Existenzquantor sind aus der Mathematik bekannt.
Der Operator OCL-SELECT führt für jedes Element der Quellkollektion (Parameter 1) die übergebene Funktion (Parameter 2) aus. Hat das Ergebnis den Wert „true“, so wird das Element in die Zielkollektion aufgenommen. Die Zielkollektion hat den selben Typ wie die Quellkollektion und ist eine Teilmenge dieser.
Der Operator OCL-REJECT negiert die Bedeutung des Prädikats im Vergleich zu OCL-SELECT.

Der Abbilder OCL-COLLECT erzeugt mit Hilfe der übergebenen Funktion (Parameter 2) aus den Elementen der Quellkollektion (Parameter 1) Instanzen eines anderen Typs und ordnet diese in die Zielkollektion (Ergebnis) ein. Der Kollektionstyp der Zielkollektion ist Sequence wenn die Quellkollektion vom Typ Sequence ist, sonst Bag. Kollektionen kommen als Ergebnis des übergebenen Ausdrucks laut [OCL97] nicht in Frage (siehe hierzu: Vergleich der Notationen).

OCL-ITERATE ist der grundlegende Operator dieser Gruppe. Auf Grund seiner Bedeutung wird dieser auch in [OCL97] sehr ausführlich beschrieben.
Wie bei allen Operatoren dieser Gruppe wird der Funktionsparameter mit einer speziellen Syntax beschrieben. Für OCL-ITERATE kann die Syntax wie folgt dargestellt werden:

```
Quellkollektion->iterate(elem : Elementtyp;
      acc : Akkumulatortyp = <Init-Ausdruck> | <Ausdruck mit elem und acc>)
```

In [OCL97] wird versucht, die Semantik von OCL-ITERATE durch „Java-ähnlichen Pseudocode“ zu erklären. Im folgenden soll dies mit einem Gemisch aus OCL und C++ versucht werden. Da der Operator OCL-AT verwendet wird, ist ggf. die Erzeugung einer der Quellkollektion entsprechenden Sequenz (OCL-ASSEQ) notwendig. Die zufällige Ordnung der Elemente ist unproblematisch, wenn sie auf das Ergebnis keinen Einfluß hat.

```
int index = 1;
Elementtyp elem;
Akkumulatortyp acc = <Init-Ausdruck>;     // Akkumulator intialisieren

while (index <= Quellkollektion->size)   // Index läuft über alle Elemente
{
  elem = Quellkollektion->at(index);      // für jedes Element
  acc = <Ausdruck mit elem und acc>;      // den neuen Akkumulator berechnen
  index = index + 1;
}
```

Der Operator OCL-ITERATE übernimmt als dritten Parameter eine Funktion mit zwei Parametern. Als ersten Parameter dieser Funktion setzt OCL-ITERATE nacheinander die Elemente der Kollektion ein (Iterator). Der zweite Parameter der Funktion ist der Akkumulator des vorhergehenden Iterationsschrittes und das Ergebnis der Akkumulator nach dem aktuellen Iterationsschritt. Der Akkumulator wird durch den zweiten Parameter von OCL-ITERATE initialisiert und sein Wert nach dem letzten Iterationsschritt ist das Ergebnis des gesamten Ausdrucks.
Der *Elementtyp* der *Quellkollektion* muß mit dem Typ des Iterators übereinstimmen. Der *Akkumulatortyp* kann frei gewählt werden. Der Initialisierungsausdruck <Init-Ausdruck> und
die Funktion <Ausdruck mit elem und acc> müssen als Ergebnis den *Akkumulatortyp* besitzen.

Beispiele:

```
Set{1, 2, 3, 4}->select(x | x < 3) =
Set{1, 2} =
Set{1, 2, 3, 4}->iterate(elem : Integer; acc : Set(Integer) = Set{} |
                    if elem < 3 then acc->including(elem) else acc endif)

Set{1, 3, 7, 16}->collect(x | x + 5) =
Bag{6, 8, 12, 21} =
Set{1, 3, 7, 16}->iterate(elem : Integer; Bag : Set(Integer) = Bag{} |
                                                  acc->including(elem + 5))
```

2.2.6 Operatoren mit Zugriff auf Daten und Metadaten des Anwendungsmodells

Die Operatoren mit Zugriff auf Daten und Metadaten des Anwendungsmodells können in drei Untergruppen eingeteilt werden: Navigationsoperatoren, Operatoren über Typen und Operatoren über Instanzen.

Navigationsoperatoren

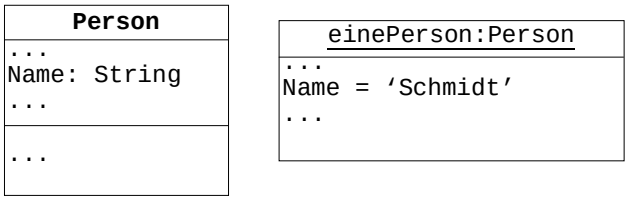
OCL-NAVATTR	_.<Attributname> : Any → BasicSet
OCL-NAVOPER	_.<Operationsname>() : Any → BasicColl _.<Operationsname>(_{,_}^{n}) : Any {× BasicColl}$^{n+1}$ → BasicColl
OCL-NAVASSE	_.<Rollenname> : Any → AnySetSeq
OCL-NAVASOB	_.<assoziationsklassenname> : Any → Set
OCL-NAVAOAE	_.<Rollenname> : Any → Any
OCL-NAVUNQU	_.<Rollenname> : Any → Set
OCL-NAVQUAL	_.<Rollenname>[_{,_}^{n}] : Any × Basic {× Basic}^{n} → AnySetSeq

Die Navigationsoperatoren werden im folgenden einzeln vorgestellt.

Navigation auf Attribute (OCL-NAVATTR)

Zu einer gegebenen Objektinstanz wird der Attributwert (bzw. die Menge der Attributwerte - bei mehrwertigen Attributen) ermittelt.

Beispiel:

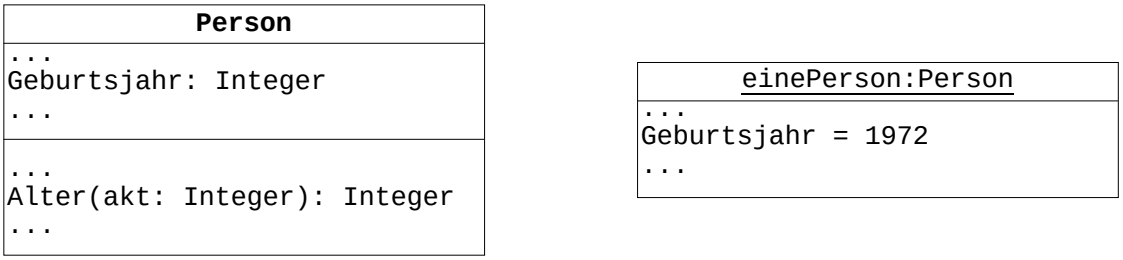



```
einPerson.Name = 'Schmidt'
```

Navigation auf Operationen (OCL-NAVOPER)

Die Operation wird im Kontext der Objektinstanz mit den übergebenen Parametern (0..(n+1)) ausgeführt und das Ergebnis zurückgegeben.

Beispiel:

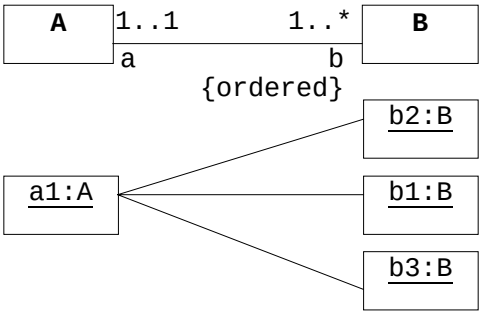



```
Person::Alter(akt: Integer): Integer
  pre:  not (akt < Geburtsjahr)
  post: result = akt - Geburtsjahr

einePerson.Alter(1998) = 26
```

Navigation auf Assoziationsenden (Rollen) (OCL-NAVASSE)

Zu einer gegebenen Objektinstanz werden die verknüpften Objekte ermittelt. Abhängig von der Multiplizitätsobergrenze handelt es sich dabei um ein Objekt oder eine Menge von Objekten. Ist die Rolle mit dem Constraint `{ordered}` versehen, so wird eine geordnete (Multi-)Menge zurückgegeben.

Beispiel:

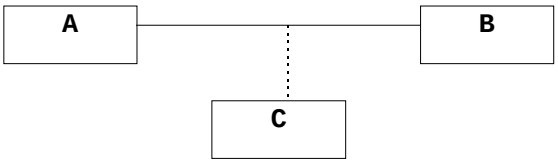



```
a1.b = Seq{b2, b1, b3}
```

Navigation auf Assoziationsobjekte (OCL-NAVASOB)

Zu einem Objekt wird die Menge der Assoziationsobjekte ermittelt, an deren Beziehung das Objekt in der entsprechenden Rolle teilnimmt.

Beispiel:

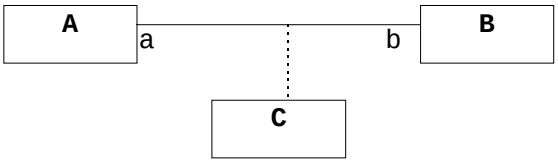



```
einA.C
```

Navigation von Assoziationsobjekten zu Assoziationsenden (Rollen) (OCL-NAVAOAE)

Von einem Assoziationsobjekte ausgehend kann laut Definition über eine Rolle genau ein anderes Objekt erreicht werden.

Beispiel:

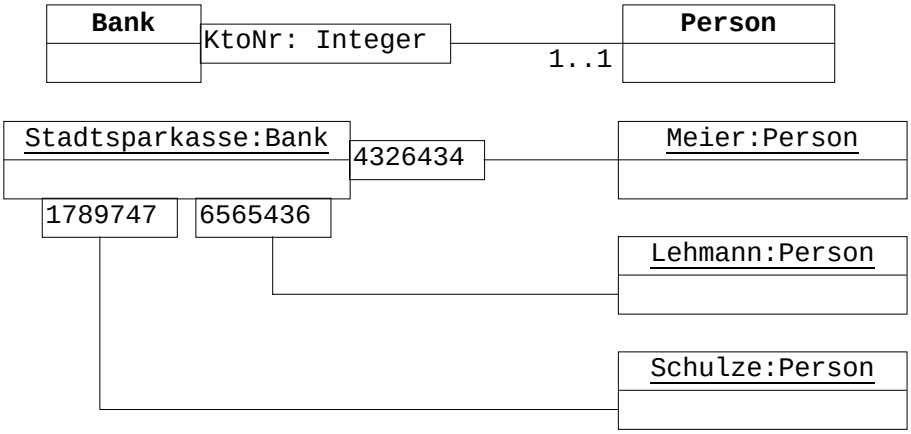



```
einC.a
einC.b
```

Navigation auf qualifizierte Assoziationsenden (Rollen) ohne Qualifikatorangabe (OCL-NAVUNQU)

Wird von einem Objekt auf eine qualifizierte Rolle ohne Berücksichtigung des Qualifikationsattributes navigiert, so erhält man immer eine Menge von Objekten.

Beispiel:

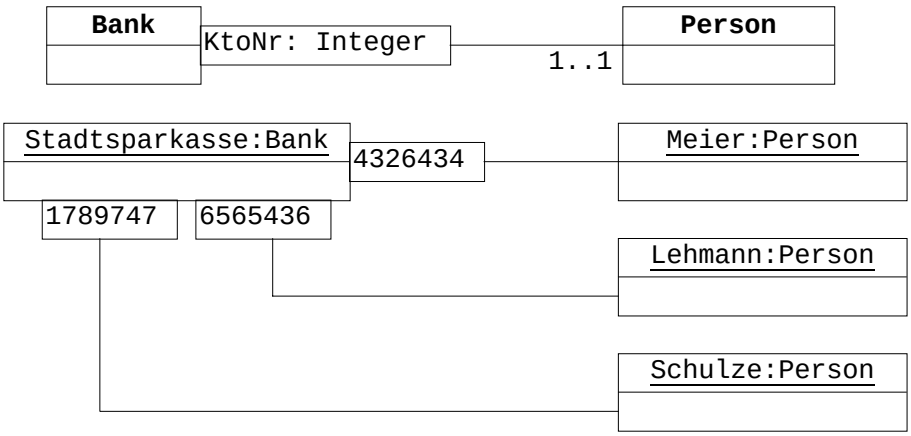



```
Stadtsparkasse.person = Set{Meier, Lehmann, Schulze}
```

Navigation auf qualifizierte Assoziationsenden (Rollen) mit Qualifikatorangabe (OCL-NAVQUAL)

Wird von einem Objekt auf eine qualifizierte Rolle für einen bestimmten Qualifikationswert navigiert, so richtet sich das Ergebnis wie bei der Navigation auf Assoziationsenden (OCL-NAVASSE) nach dem Typ der Rolle.

Beispiel:




```
Stadtsparkasse.person[6565436] = Lehmann
```

Operatoren über Typen

Für Typen (OclType) stehen eine Reihe von Operatoren bereit. Am wichtigsten ist dabei OCL-EXT. Dieser Operator ermittelt alle Instanzen eines Typs einschließlich der Instanzen der Subtypen des Typs (Extension). OCL-TYPENAME ermittelt den Namens des Typs, OCL-ATTRIBS die Menge der Namen der Attribute, OCL-ASSOENDS die Menge der Namen der Rollen und OCL-OPERATNS die Menge der Namen der Operationen.
OCL-SUPERTPS ermittelt die Menge der direkten Supertypen und OCL-ALLSUPTS die Menge aller Supertypen (transitiv).

OCL-EXT	_.allInstances : → Set
OCL-TYPENAME	_.name : Type → Str
OCL-ATTRIBS	_.attributes : Type → Set
OCL-ASSOENDS	_.associationEnds : Type → Set
OCL-OPERATNS	_.operations : Type → Set
OCL-SUPERTPS	_.supertypes : Type → Set
OCL-ALLSUPTS	_.allSupertypes : Type → Set

Operatoren über Instanzen

Alle Instanzen eines UML-Klassendiagramms erben von dem in der OCL definierten Typ OclAny und somit die auf ihm definierten Features. OCL-OBJEQU und OCL-OBJNEQ dienen zum Vergleich zweier Objektidentitäten. Mit OCL-OCLTYPE kann der Typ einer Instanz ermittelt werden. OCL-ISKINDOF wird zu „true“ ausgewertet, wenn der Typ des Objekts Subtyp (transitiv) des übergebenen Typs ist. OCL-ISTYPEOF wird zu wahr ausgewertet, wenn das Objekt den selben Typ hat wie der übergebene Typ. OCL-ASTYPE repräsentiert das Konzept eines „Casts“, d.h. es wird das selbe Objekt zurückgegeben welches übergeben wurde, jedoch können nun die Features des übergebenen Typs benutzt werden.

OCL-OBJEQU	_ = _ : Any × Any → Bool
OCL-OBJNEQ	_ <> _ : Any × Any → Bool
OCL-OCLTYPE	_.oclType : Any → Type
OCL-ISTYPEOF	_.oclIsTypeOf(_) : Any × Type → Bool
OCL-ISKINDOF	_.oclIsKindOf(_) : Any × Type → Bool
OCL-ASTYPE	_.oclAsType(_) : Any × Type → Any

2.3 Weitere Aspekte der OCL

In diesem Abschnitt werden die Aspekte der OCL beschrieben, welche nicht in das Schema Datentypen/Operatoren passen.

2.3.1 Invarianten

Die OCL wird vorrangig zur Spezifikation von Invarianten verwendet. Eine Invariante ist Teil eines UML-Anwendungsmodells und einem Typ (Classifier) zugeordnet. Eine Invariante wird notiert, indem der Name des Typs unterstrichen dargestellt wird und in der nächsten Zeile ein OCL-Ausdruck folgt.

```
<Typ>
  <Boolean-Ausdruck von self>
```

Der OCL-Ausdruck muß den Ergebnistyp Boolean besitzen und im Kontext einer Instanz des Typs (self) formuliert sein. Der kontextuellen Instanz kann auch ein anderer Name als self zugewiesen werden:

```
<kontextuelleInstanz> : <Typ>
   <Boolean-Ausdruck von <kontextuelleInstanz> >
```

Die Bedeutung einer Invariante ist, daß der Ausdruck für alle (korrekten) Instanzen des Typs zu true ausgewertet werden muß. In der Prädikatenlogik würde dies wie folgt ausgedrückt werden:

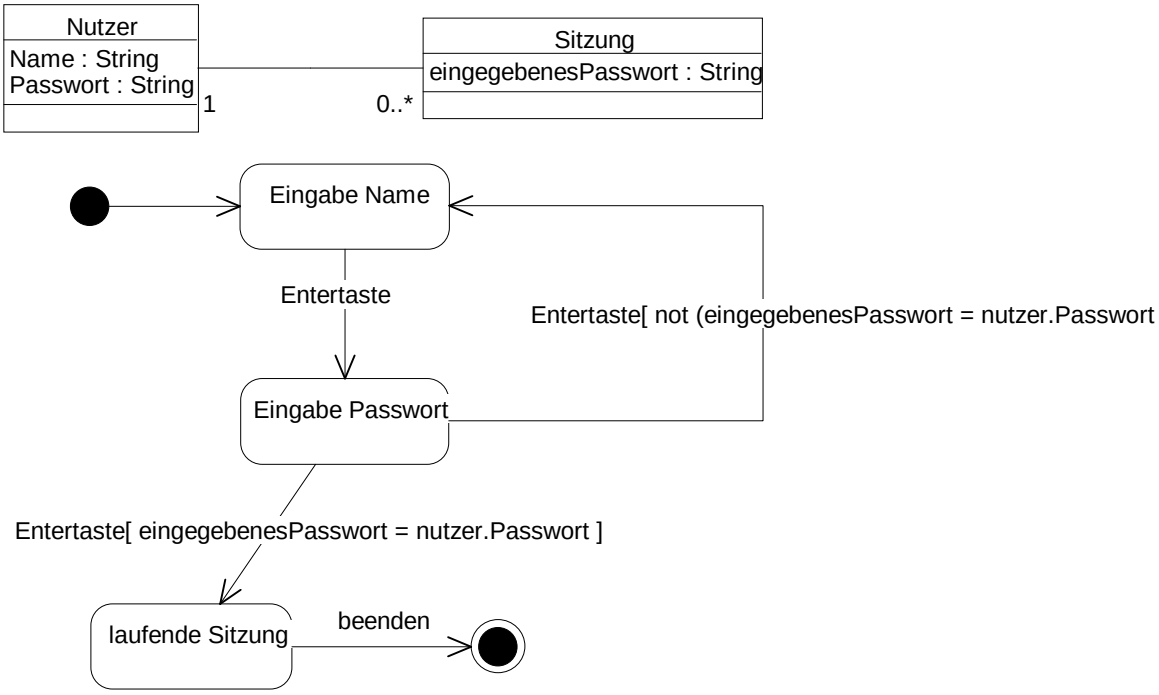
```
∀self:<Typ>.     ...
```

Dies könnte in der OCL auch explizit formuliert werden:

```
<Typ>.allInstances->forAll(self | ... )
```

2.3.2 OCL-Ausdrücke in Wächter- und Änderungsbedingungen

Wie das folgende Beispiel illustriert, können OCL-Ausdrücke auch zur Spezifikation von Wächter- und Änderungsbedingungen eingesetzt werden.



2.3.3 Vor- und Nachbedingungen für Operationen und Methoden

OCL-Ausdrücke können zur Formulierung von Vor- und Nachbedingungen für Operationen und Methoden eingesetzt werden.

```
<Typname>::<Operationsname>(<Signatur>):<Ergebnistyp>
  pre: <Boolean-Ausdruck über self und Parameter>
  post: <Boolean-Ausdruck über self, result und Parameter>
```

In Nachbedingung sind zusätzlich die Werte vor Operationsbeginn ermittelbar (`<property>@pre`).

2.3.4 „Zusätzliche Operationen“

In [UML-SEMA] werden „zusätzliche Operationen“ wie folgt mit der OCL beschrieben:

```
<Operationsname>(<Signatur>):<Ergebnistyp>;
<Operationsname>(<Signatur>) = <Ausdruck über Ergebnistyp>
```

Dies ist eine Kurzform von:

```
<Typname>::<Operationsname>(<Signatur>):<Ergebnistyp>
  post: result = <Ausdruck über Ergebnistyp>
```

Bei den „zusätzlichen Operationen“ wird das Ergebnis der Operation durch OCL-Ausdrücke direkt beschrieben. Durch die Möglichkeit des rekursiven Aufrufs ergibt sich eine Erweiterung der Anwendungsmöglichkeiten. Da „zusätzliche Operationen“ nur einen Wert zurückgeben und keine Änderungen vornehmen können, können sie von anderen OCL-Ausdrücken aufgerufen werden.

2.3.5 Typübereinstimmung

Die OCL ist eine getypte Sprache. Die Übereinstimmung der Typen kann für OCL-Ausdrücke geprüft werden. Die Regeln für die Typübereinstimmung werden in [OCL97] auf den Seiten 5 und 6 verbal beschrieben. Daß das Fehlen einer vollständigen formalen Beschreibung der Typübereinstimmung die korrekte Implementation der Überprüfung sehr erschwert, wird durch die unvollständige Realisierung des OCL-Parsers (siehe [OCL-PARSER]) unterstrichen.

2.3.6 Vorrangregeln

Die Vorrangregeln sind auf den Seiten 6 und 7 in [OCL97] beschrieben.

2.3.7 Kommentare

Laut [OCL97] Seite 7 werden Kommentare durch das Symbol ‘`--`‘ abgegrenzt.

2.3.8 Undefinierte Werte

Die Verwendung undefinierter Werte wird in [OCL97] auf Seite 7 verbal beschrieben. Einige Nachbedingungen zur Beschreibung der Operatoren beschreiben formal das Auftreten undefinierter Werte durch Rückgabe von `undefined`.

2.3.9 Benennungskonventionen

Auf Benennungskonventionen (wie in [OCL97] auf Seite 9) wird im Abschnitt 3.2 dieser Arbeit eingegangen.

2.3.10 Implizite Mengeneigenschaft

Auf Seite 9 von [OCL97] wird beschrieben, daß auf einzelne Objekte Mengenoperationen angewendet werden können. Ein undefinierter Wert wird als leere Menge und ein einzelnes Objekt als einelementige Menge aufgefaßt.

2.3.11 Kombination der Operatoren und Auswertungsrichtung

In [OCL97] S. 10 wird ausgesagt, daß OCL-Operatoren beliebig zu Ausdrücken zusammengesetzt werden können, solange die Typübereinstimmung gewährleistet ist. Die Auswertungsrichtung ist von links nach rechts.

2.3.12 Pfadnamen

Wenn Namen nicht eindeutig sind, kann laut [OCL97] S. 11 eine Pfadbezeichnung mit dem `::`-Symbol zur näheren Bestimmung verwendet werden.

2.3.13 Mengensysteme ([BronSem91] S. 545)

Laut [OCL97] S. 13 werden in OCL-Ausdrücken auftretende Mengensysteme sofort verebnet. In der in dieser Arbeit vorangegangenen Beschreibung der Operatoren wurden Mengensysteme vermieden, indem kein Operator Mengensysteme zum Ergebnis haben darf.

2.3.14 Abkürzung für OCL-COLLECT

Auf Seite 17 in [OCL97] wird beschrieben, daß ein Ausdruck der Form
`collection->collect(propertyname)`
abkürzend in der folgenden Weise dargestellt werden kann:
`collection.propertyname`
Dies gilt auch bei vorhandenen Parametern.

2.3.15 Erweiterte Variante von OCL-FORALL

Eine erweiterte Variante von `OCL-FORALL`, welche die Prüfung von Bedingungen über Kreuzprodukte von Mengen vereinfacht, wird in [OCL97] auf Seite 18 beschrieben.

2.4 Äquivalenzbedingungen für OCL-Ausdrücke

Äquivalenzbedingungen definieren, wann zwei unterschiedliche OCL-Ausdrücke unter allen Umständen das selbe Ergebnis haben.

2.4.1 Zweck der Betrachtung von Äquivalenzbedingungen

Die OCL benutzt in ihrer Definition Nachbedingungen, welche den Ergebnisraum nicht nur einschränken sondern das Ergebnis genau bestimmen. Diese Nachbedingungen sind mit Äquivalenzbedingungen identisch und haben die Beschreibung der Semantik der OCL zum Ziel, d.h. die Semantik eines Operators, welche sich aus Äquivalenzbedingungen herleiten läßt braucht nicht mehr mit anderen (aufwendigeren oder weniger formalen) Mitteln beschrieben werden.
Eine umfangreiche Aufstellung möglichst vieler Äquivalenzbedingungen würde formal begründete Aussagen über Struktur und Eigenschaften der Sprache OCL ermöglichen. Zur Verdichtung von Darstellungen und für die Abstraktion wurden die Äquivalenzbedingungen in dieser Arbeit verwendet.
Die von den Äquivalenzbedingungen ausgehende Möglichkeit der Umformung von OCL-Ausdrücken (analog zur Umformung von Termen in der Mathematik) eröffnet Perspektiven für die Qualitätsverbesserung bei der Implementation bzw. Abbildung der OCL, wie dies z.B. von der Anfrageoptimierung in Datenbanksystemen bekannt ist.

2.4.2 Darstellungstechnik für Äquivalenzbedingungen

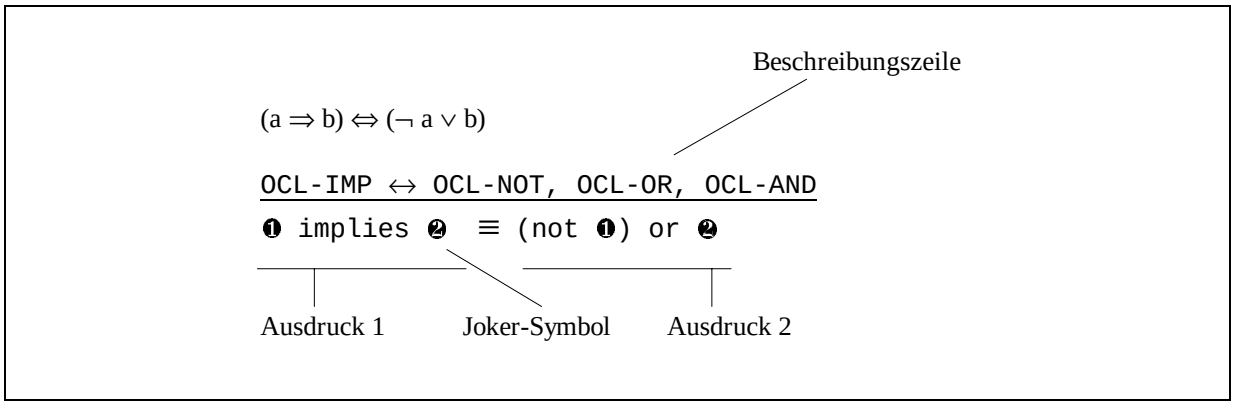


Abbildung 5: Darstellung von Äquivalenzbedingungen in der Mathematik vs. in dieser Arbeit

Jede Äquivalenzbedingung wird durch eine unterstrichene Beschreibungszeile eingeleitet. Die darunter folgende Äquivalenzbedingung enthält zwei Ausdrücke, welche mit einem Identitätssymbol verbunden sind

(. . . ≡ . . .). Die Beschreibungszeile nennt die mnemonischen Kurzbezeichnungen der Operatoren zum Zwecke der besseren Übersicht und der Vermeidung von Fehlinterpretationen (z.B. = für Integer und Boolean mit unterschiedlichen Äquivalenzbedingungen).
Die Äquivalenzbedingungen enthalten Joker-Symbole (z.B. ❷) für beliebige Zeichenfolgen. Das mehrfache Auftreten ein und des selben Joker-Symbols fordert die Gleichheit der entsprechenden Zeichenfolgen. Sind die beiden Ausdrücke korrekte OCL-Ausdrücke, so ist ihr Ergebnis unter allen Umständen gleich. Joker-Symbole mit Parameter (z.B. 3()) treten bei Funktionen höherer Ordnung auf, wenn der Name des Paramters bei der Ersetzung berücksichtigt werden muß.
Fett und Kursiv dargestellte Teile der Äquivalenzbedingungen müssen aus dem Kontext mit den korrekten Zeichenketten ergänzt werden. Dies wird durch die zusammenfassende Betrachtung der Kollektionstypen nötig, mit der die Anzahl der Operatoren begrenzt wird. Kursiv sind auch die Parameternamen für Funktionen höherer Ordnung dargestellt, da sie zur Vermeidung eventueller Namenskonflikte eindeutige Namen zugewiesen bekommen sollten.

Abbildung 6: Darstellung von primären Operatoren

Die „primären Operatoren“, auf deren Grundlage die anderen Operatoren durch Äquivalenzbedingungen definiert sind, sind anstelle von Äquivalenzbedingungen durch eine Beschreibungszeile in Fettdruck in der Auflistung enthalten. Einige Operatoren, welche sich durch andere darstellen lassen, sind ebenfalls durch Fettdruck hervorgehoben. Diese gehören zu einem minimalen Satz von Operatoren, welcher als Basis verwendet werden könnte. Dieser minimale Satz könnte aber auch anders aussehen, es handelt sich somit um ein Beispiel eines minimalen Satzes.

2.4.3 Reihenfolge der Äquivalenzbedingungen

Im folgenden werden die gefundenen Äquivalenzbedingungen entsprechend der im Abschnitt 2.2 eingeführten Gruppierung und Reihenfolge der Operatoren aufgelistet. Aus den Nachbedingungen in [OCL97] entnommene Äquivalenzbedingungen sind durch die Quellenangabe gekennzeichnet.

2.4.4 Äquivalenzbedingungen der Konstanten

Für Konstanten können keine Äquivalenzbedingungen formuliert werden, da keine Umformungsoperatoren (Uminterpretieren von Bitmustern) zwischen den Elementardatentypen definiert sind.

OCL-CONINT

OCL-CONREAL

OCL-CONBOOL

OCL-CONSTR

OCL-CONENUM

OCL-CONTYPE

2.4.5 Äquivalenzbedingungen der einfachen Operatoren über Elementardatentypen

Operatoren der Booleschen Algebra

Für diese Gruppe sind aus der Mathematik eine Vielzahl von Äquivalenzbedingungen bekannt. Aus OCL-COND und OCL-CONBOOL lassen sich alle anderen Operatoren herleiten. Es ist auch möglich, aus OCL-NOT und einem der binären Operatoren alle anderen Operatoren der Gruppe (außer OCL-COND) herzuleiten.

OCL-COND

OCL-NOT ↔ OCL-COND, OCL-CONBOOL ([OCL] S. 24)

```
not ❶   ≡   if ❶ then false else true end
```

OCL-AND

```
❶ and ❷   ≡   if not ❶ then
                  false
              else
                  if ❷ then true else false endif
              endif
```

OCL-OR ↔ OCL-NOT, OCL-AND

```
❶ or ❷   ≡   not ((not ❶) and (not ❷))
```

OCL-IMP ↔ OCL-NOT, OCL-OR, OCL-AND ([OCL] S. 24)

```
❶ implies ❷   ≡   (not ❶) or (❶ and ❷)
```

OCL-XOR ↔ OCL-OR, OCL-AND, OCL-NOT, OCL-EQUB ([OCL] S. 24)

```
❶ xor ❷   ≡   (❶ or ❷) and not (❶ = ❷)
```

OCL-EQUB ↔ OCL-NOT, OCL-XOR

```
❶ = ❷   ≡   not (❶ xor ❷)
```

Operatoren über ganzen und reellen Zahlen

Auch für diese Gruppe sind aus der Mathematik eine Vielzahl von Äquivalenzbedingungen bekannt. Es wäre möglich, andere minimale Sätze von („primären“) Operatoren zu finden, aus denen die restlichen Operatoren hergeleitet werden können.

OCL-EQU ↔ OCL-AND, OCL-LEQ, OCL-GEQ

```
❶ = ❷   ≡   (❶ <= ❷) and (❶ >= ❷)
```

OCL-LT ↔ OCL-NOT, OCL-GEQ

```
❶ < ❷   ≡   not (❶ >= ❷)
```

OCL-GT ↔ OCL-NOT, OCL-LEQ ([OCL] S. 22)

```
❶ > ❷   ≡   not (❶ <= ❷)
```

OCL-LEQ ↔ OCL-EQU, OCL-OR, OCL-LT ([OCL] S. 22)

```
❶ <= ❷   ≡   (❶ = ❷) or (❶ < ❷)
```

OCL-GEQ ↔ OCL-EQU, OCL-OR, OCL-GT ([OCL] S. 22)

```
❶ >= ❷   ≡   (❶ = ❷) or (❶ > ❷)
```

OCL-NEG ↔ OCL-CONINT/REAL, OCL-SUB

```
- ❶   ≡   0 - ❶
```

OCL-ADD ↔ OCL-SUB, OCL-NEG

```
❶ + ❷   ≡   ❶ - (- ❷)
```

OCL-SUB ↔ OCL-ADD, OCL-NEG

```
❶ - ❷   ≡   ❶ + (- ❷)
```

OCL-MUL

OCL-DIVF

OCL-FLOOR

OCL-ABS ↔ OCL-COND, OCL-LT, OCL-NEG, OCL-CONINT/REAL ([OCL] S. 21)

```
❶.abs   ≡   if ❶ < 0 then -❶ else ❶ endif
```

OCL-MAX ↔ OCL-COND, OCL-GEQ ([OCL] S. 21)

```
❶.max(❷)  ≡   if ❶ >= ❷ then ❶ else ❷ endif
```

OCL-MIN ↔ OCL-COND, OCL-LEQ ([OCL] S. 22)

```
❶.min(❷)   ≡   if ❶ <= ❷ then ❶ else ❷ endif
```

OCL-DIVI ↔ OCL-DIVF, OCL-FLOOR

```
❶.div(❷)   ≡   (❶/❷).floor
```

OCL-MOD ↔ OCL-SUB, OCL-DIVI, OCL-MUL ([OCL] S. 23)

```
❶.mod(❷)   ≡   ❶ - (❶.div(❷) * ❷)
```

Operatoren über Zeichenketten

Für diese Gruppe können keine Äquivalenzbedingungen beschrieben werden. Die Operatoren dieser Gruppe würden sich auf `OCL-COND`, `OCL-ITERATE` sowie Operationen über einzelne Zeichen und für den Zugriff auf einzelne Zeichen von Zeichenketten zurückführen lassen, wenn letztere durch die OCL definiert wären.

OCL-EQSTR

OCL-SZSTR

OCL-CONCAT

OCL-TOUPP

OCL-TOLOW

OCL-SUBSTR

Operatoren über Aufzählungselemente

Für die beiden Operatoren der Gruppe existiert genau eine Äquivalenzbedingung.

OCL-EQUENUM

```
❶ == ❷   ≡   not (❶ <> ❷)
```

OCL-NEQENUM ↔ OCL-EQUENUM, OCL-NOT ([OCL] S. 24)

```
❶ <> ❷   ≡   not (❶ = ❷)
```

2.4.6 Äquivalenzbedingungen der einfachen Operatoren über Kollektionstypen

Konstruktion von Kollektionen

OCL-CONSET, OCL-CONBAG und OCL-CONSEQ sind zwar durch OCLH-SGLELEM, OCLH-INTRVAL und OCL-UNION darstellbar, OCL-UNION basiert aber über OCL-INC wieder auf diesen Operatoren. Es ist zu beachten, daß die Parameterkollektionen der Konstruktionsoperatoren von den Hilfsoperatoren OCLH-SGLELEM und OCLH-INTRVAL erzeugt werden müssen und nirgendwo anders eingesetzt werden dürfen. Die Symbole für die leeren Kollektionen können nicht ersetzt werden.

OCLH-SGLELEM

OCLH-INTRVAL

OCL-CONSET ↔ OCL-UNION, (OCLH-SGLELEM, OCLH-INTRVAL)

```
Set{❶, ❷, ...} = ❶->union(❷->union(...))
```

OCL-CONBAG ↔ OCL-UNION, (OCLH-SGLELEM, OCLH-INTRVAL)

```
Bag{❶, ❷, ...} = ❶->union(❷->union(...))
```

OCL-CONSEQ ↔ OCL-UNION, (OCLH-SGLELEM, OCLH-INTRVAL)

```
Sequence{❶, ❷, ...} = ❶->union(❷->union(...))
```

Erzeugung von Kollektionen aus anderen Kollektionstypen

Diese Operatoren haben im wesentlichen nur eine Uminterpretation zum Inhalt. Das Entfernen von Duplikaten bei OCL-ASSET läßt sich wie folgt durch OCL-COND und OCL-ITERATE erreichen:

```
❶->removeDuplicates   ≡
     ❶->iterate(elem; acc : Set(type) = Set{} |
         if acc->includes(elem) then acc else acc->including(elem) endif)
```

Obwohl die Aktivitäten dieser Operatoren somit nachgebildet werden könnten, können diese Operatoren selbst nicht durch äquivalente Ausdrücke ersetzt werden.

OCL-ASSET

OCL-ASBAG

OCL-ASSEQ

Aggregatfunktionen auf Kollektionen

Alle Operatoren dieser Gruppe können nachgebildet werden.

OCL-SIZE ↔ OCL-ITERATE, OCL-CONINT, OCL-ADD ([OCL] S. 25)

```
❶->size   ≡   ❶->iterate(elem; acc : Integer = 0 | acc + 1)
```

OCL-COUNT ↔ OCL-ITERATE, OCL-CONINT, OCL-COND, OCL-*EQ*, ... ([OCL] S. 25)

```
❶->count(❷)   ≡   ❶->iterate(elem; acc : Integer = 0 |
                              if elem = ❷ then acc + 1 else acc endif)
```

OCL-SUM ↔ OCL-ITERATE, OCL-CONINT, OCL-ADD ([OCL] S. 25)

```
❶->sum   ≡   ❶->iterate(elem; acc : Type = 0 | acc + elem)
```

Typische Mengenoperationen auf Kollektionen

Alle Operatoren der Gruppe können auf die Definition von OCL-INC oder OCL-UNION zurückgeführt werden. Der Operator OCL-INC wurde auf Grund seiner einfacheren Semantik als „primärer Operator“ ausgewählt.

OCL-INC ↔ OCL-UNION, OCL-CONCOLL

```
❶->including(❷)   ≡   ❶->union(Coll{❷})
```

OCL-EXC ↔ OCL-REJECT, OCL-*EQ*

```
❶->excluding(❷)   ≡   ❶->reject(e | e = ❷)
```

OCL-APP ↔ OCL-INC

```
❶->append(❷)   ≡   ❶->including(❷)
```

OCL-PREP ↔ OCL-CONCOLL, OCL-UNION

```
❶->including(❷)   ≡   Coll{❷}->union(❶)
```

OCL-UNION ↔ OCL-ITERATE, OCL-INC

```
❶->union(❷)   ≡
        ❷->iterate(elem; acc : Coll(type) = ❶ | acc->including(elem))
```

OCL-INTER ↔ OCL-UNION, OCL-ITERATE, OCL-CONCOLL, OCL-COUNT, ...

```
❶->intersection(❷)   ≡
       (❶->union(❷))->iterate(elem; acc : Coll(type) = Coll{} |
                              if ❶->count(elem) < ❷->count(elem) then
                                  acc->union(❶->select(e | e = elem)
                              else
                                  acc->union(❷->select(e | e = elem)
                              endif)
```

OCL-DIFFR ↔ OCL-SELECT, OCL-NOT, OCL-ELEM

```
❶ - ❷   ≡   ❶->select(e | not ❷->includes(e))
```

OCL-SYMDIF ↔ OCL-DIFFR, OCL-UNION

```
❶->symmetricDifference(❷)   ≡   (❶ - ❷)->union(❷ - ❶)
```

Besondere Operatoren für geordnete Multimengen

Der Zugriff auf einzelne Elemente einer Sequenz (OCL-AT) kann nicht auf andere Operatoren zurückgeführt werden.

OCL-AT

OCL-FIRST ↔ OCL-AT, OCL-CONINT ([OCL] S. 30)

```
❶->first   ≡   ❶->at(1)
```

OCL-LAST ↔ OCL-AT, OCL-SIZE ([OCL] S. 30)

```
❶->last   ≡   ❶->at(❶->size)
```

OCL-SUBSEQ ↔ OCL-CONSEQ, OCL-INTER, OCL-APP, OCL-AT

```
❶->subSequence(❷, ❸)   ≡
        Seq{❷..❸}->iterate(elem; acc : Seq(type) = Seq{} |
                                                acc->append(❶->at(elem))
```

Relationale Mengenoperationen auf Kollektionen

Die relationalen Mengenoperationen lassen sich vollständig auf OCL-COND und OCL-ITERATE zurückführen.

OCL-ELEM ↔ OCL-COUNT, OCL-GT, OCL-CONINT ([OCL] S. 25)

```
❶->includes(❷)   ≡   ❶->count(❷) > 0
```

OCL-SUBSET ↔ ([OCL] S. 25)

```
❶->includesAll(❷)  ≡  ❷->forAll(elem | ❶->includes(elem))
```

Achtung! Intuitiv könnte auch folgendes erwartet werden:

```
❶->includesAll(❷)  ≡  ❷->forAll(elem | ❶->count(elem) >= ❷->count(elem))
```

Dies trifft jedoch nicht zu!

OCL-ISEMPT ↔ ([OCL] S. 25)

```
❶->isEmpty   ≡   ❶->size = 0
```

OCL-NOTEMPT ↔ ([OCL] S. 25)

```
❶->notEmpty   ≡   ❶->size <> 0
```

OCL-CMPCOLL { ❶ und ❷ vom Typ Set }

```
❶ = ❷   ≡  ❶->includesAll(❷) and ❷->includesAll(❶)
```

OCL-CMPCOLL { ❶ und ❷ vom Typ Bag }

```
❶ = ❷   ≡  (❶->union(❷))->forAll(elem | ❶->count(elem) = ❷->count(elem))
```

OCL-CMPCOLL { ❶ und ❷ vom Typ Sequence }

```
❶ = ❷   ≡  (❶->size = ❷->size) and
                Seq{1..❶->size}->forAll(elem | ❶->at(elem) = ❷->at(elem))
```

2.4.7 Äquivalenzbedingungen der Operatoren vom Typ einer Funktion höherer Ordnung

Die Operatoren dieser Gruppe lassen sich aus `OCL-ITERATE` herleiten.

OCL-ITERATE

OCL-FORALL ↔ OCL-ITERATE, OCL-CONBOOL, OCL-AND ([OCL] S. 26)

```
❶->forAll(❷)   ≡   ❶->iterate(elem; acc : Boolean = true | acc and ❷)
```

OCL-EXISTS ↔ OCL-ITERATE, OCL-CONBOOL, OCL-OR ([OCL] S. 26)

```
❶->exists(❷)   ≡   ❶->iterate(elem; acc : Boolean = false | acc or ❷)
```

OCL-SELECT ↔ OCL-ITERATE, OCL-CON**COLL**, OCL-COND, ... ([OCL] S. 27, 28, 30)

```
❶->select( 2(name) )   ≡
        ❶->iterate(name, acc : Coll(type) = Coll{} |
                if 2(name) then acc->including(name) else acc endif)
```

OCL-REJECT ↔ OCL-SELECT, OCL-NOT ([OCL] S. 27, 28, 30)

```
❶->reject(❷)   ≡   ❶->select(not ❷)
```

OCL-COLLECT ↔ OCL-ITERATE, OCL-CON**COLL**, OCL-INC ([OCL] S. 27, 29 (30!?))

```
❶->collect( 2(name) )   ≡
      ❶->iterate(name, acc : Coll(type) = Coll{} | acc->including( 2(name) )
```

2.4.8 Äquivalenzbedingungen der Operatoren mit Zugriff auf Daten und Metadaten des Anwendungsmodells

Navigationsoperatoren

Die einzelnen Navigationsoperatoren sind formal nicht durch andere Operatoren darstellbar. (Enthält ein UML-Klassendiagramm redundante Information (z.B. abgeleitete Assoziationen) so können jedoch u.U. äquivalente Ausdrücke unter Verwendung der Modellsemantik konstruiert werden.)

OCL-NAVATTR

OCL-NAVOPER

OCL-NAVASSE

OCL-NAVASOB

OCL-NAVAOAE

OCL-NAVUNQU

OCL-NAVQUAL

Operatoren über Typen

Die Operatoren über Typen können nicht aus anderen Operatoren hergeleitet werden. `OCL-ALLSUPTS` ist keine Äquivalenzbedingung sondern die rekursive Definition in der Darstellungsweise von Äquivalenzbedingungen (die Musterersetzung würde nicht terminieren), welche zum Zwecke des Überblicks mit in die Auflistung aufgenommen wurde.

OCL-EXT

OCL-TYPENAME

OCL-ATTRIBS

OCL-ASSOENDS

OCL-OPERATNS

OCL-SUPERTPS

OCL-ALLSUPTS

```
❶.allSupertypes   ≡   (❶.supertypes)->union(❶.supertypes.allSupertypes)
```

Operatoren über Instanzen

In dieser Gruppe fällt auf, daß die Nachbedingung für `OCL-ISKINDOF` in [OCL97] falsch formuliert ist.

OCL-OBJEQU ↔ OCL-OBJNEQ, OCL-NOT

```
❶ == ❷   ≡   not (❶ <> ❷)
```

OCL-OBJNEQ ↔ OCL-OBJEQU, OCL-NOT ([OCL] S. 20)

```
❶ <> ❷   ≡   not (❶ = ❷)
```

OCL-OCLTYPE

OCL-ISTYPEOF ([OCL] S. 20)

```
❶.oclIsTypeOf(❷)  ≡ ❶.oclType = ❷
```

OCL-ISKINDOF ([OCL] S. 20 ist falsch!)

```
❶.oclIsKindOf(❷)  ≡ ❶.oclType.allSupertypes->includes(❷) or ❶.oclType = ❷
```

OCL-ASTYPE

2.4.9 Weitere Äquivalenzbedingungen

Es lassen sich eine Reihe weiterer Äquivalenzbedingungen finden, welche nicht einen Operator durch andere erklären. Einige von ihnen können u.U. tiefgreifende Umstrukturierungen der Ausdrücke vornehmen. Bei Bedarf (z.B. für einen „OCL-Optimierer“) könnten aus Literatur zu Mathematik und Logik entsprechende Äquivalenzbedingungen entnommen werden. Im folgenden sind drei Beispiele aufgezeigt.

OCL-ADD ↔ OCL-ADD

```
❶ + ❷   ≡   ❷ + ❶                              -- Kommutativgesetz
```

OCL-MUL, OCL-ADD ↔ OCL-MUL, OCL-ADD

```
❶ * (❷ + ❸)   ≡   (❶ * ❷) + (❶ * ❸)          -- Assoziativgesetz
```

OCL-UNION, OCL-FORALL ↔ OCL-FORALL, OCL-AND

(❶->union(❷))->forAll(3()) ≡ ❶->forAll(3()) and ❷->forAll(3())

2.4.10 Zusammenfassung und Schlußfolgerungen

Die Gruppen Konstanten und Konstruktion von Kollektionen sowie die Zeichenkettenoperatoren, die Operatoren mit Zugriff auf die UML-Modell-Daten und die Kollektionstypwandler bestehen fast nur aus primären Operatoren.
Bei den Algebren (Boolean, Real und Integer) können verschiedene Untermengen der Operatoren bestimmt werden, aus denen dann die übrigen Operatoren der Gruppe hergeleitet werden können.
Die Operatoren vom Typ höherer Ordnung und die Operatoren über Kollektionstypen lassen sich aus dem Operator `OCL-ITERATE` sowie den vorher genannten Operatoren herleiten.
Insgesamt läßt sich eine große Zahl von Operatoren direkt oder indirekt durch `OCL-COND` und `OCL-ITERATE` erklären, welche somit als die leistungsfähigsten und zentralen Konzepte der OCL angesehen werden dürfen.
Der ausgewählte Satz an „primären Operatoren“, auf deren Grundlage sich die anderen Operatoren erklären lassen, umfaßt etwas mehr als die Hälfte der dargestellten Operatoren. Es ergibt sich somit keine besonders starke Reduktion des Umfangs (z.B. bei der Beschreibung der OCL oder der Abbildung auf SQL), wenn man die Betrachtungen auf die „primären Operatoren“ beschränkt.

3 Vergleich der OCL mit ähnlichen Sprachen

Bevor die OCL mit ähnlichen Sprachen detailliert verglichen wird, sollen diese zunächst im ersten Abschnitt kurz vorgestellt werden. Der wichtigste Anhaltspunkt für den Vergleich ist die unterschiedliche Gestaltung der Syntax für die Navigationsausdrücke, welche im zweiten Abschnitt untersucht wird. Der dritte Abschnitt faßt den Vergleich verschiedener Details zusammen, welche mit den Methoden der beiden vorhergehenden Abschnitte nicht erfaßt werden konnten.

3.1 Struktur und Eigenschaften der Sprachen

3.1.1 Struktur und Eigenschaften der OCL

Im Kapitel 2 wurde die OCL bereits eingehend vorgestellt. Sie dient in diesem Vergleich als Referenz. Die im folgenden wiederholt dargestellte Strukturierung wurde ebenfalls in Kapitel 2 eingeführt und soll hier den Darstellungen zu den anderen Notationen gegenübergestellt werden.

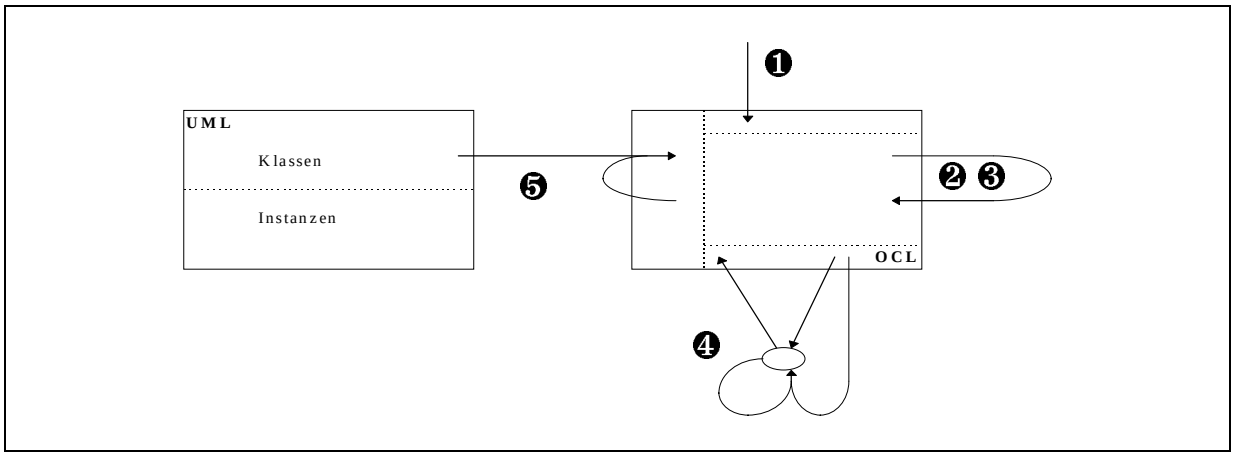


Abbildung 7: Strukturierung der OCL in 5 Gruppen von Operatoren

Die OCL definiert neben den Operatoren mit Zugriff auf Daten und Metadaten des Anwendungsmodells (Gruppe ❺) auch eine Vielzahl von Operatoren zur Transformation von Daten und Wahrheitswerten. Die beiden anderen Sprachen definieren diese Operatoren zur Transformation nicht selbst.
Da es sich bei [OCL97] um einen Standard-Entwurf handelt, ist dieser vor allem in Bezug auf die Syntax sehr exakt.

3.1.2 Struktur und Eigenschaften der ONN

Die „Object Navigation Notation“ (ONN) wird in [BlahPrem98] beschrieben. Im Vergleich zu den beiden anderen Sprachen ist die Definition der ONN wesentlich weniger formal. Die ONN ist Teil des funktionalen Modells der OMT, welches die Datenverarbeitung eines Softwaresystems unter Verwendung von Pseudocode beschreibt. Im Gegensatz zu den anderen beiden Sprachen können hier Zustandsänderungen der Instanzen des Anwendungsmodells direkt beschrieben werden, anstatt nur das Ergebnis der Zustandsänderung zu beschreiben.

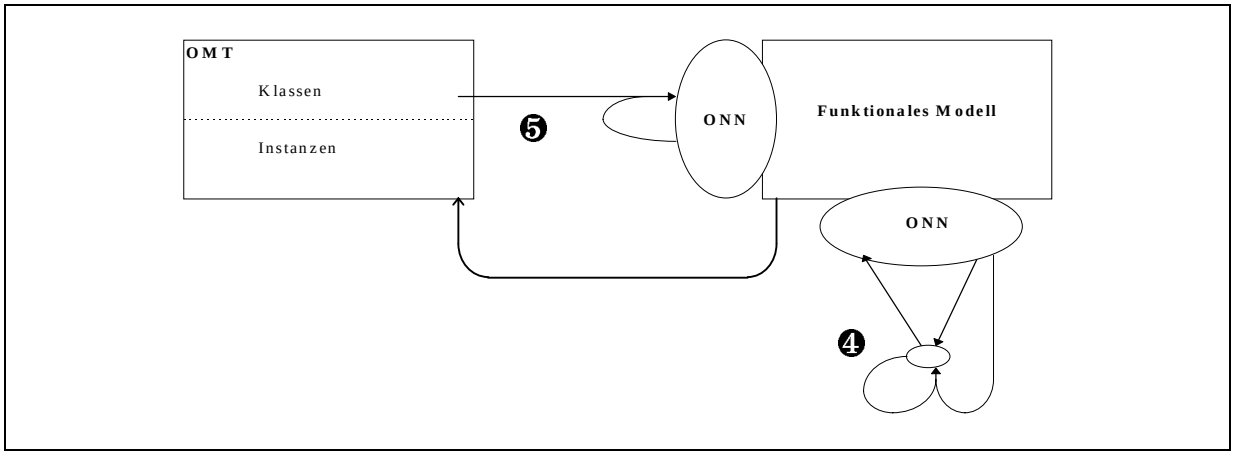


Abbildung 8: Struktur des Funktionalen Modells der OMT

In [BlahPrem98] wird keine vollständige und verbindliche Sprache festgelegt, mit der die Funktionalität des zu spezifizierenden Systems beschrieben wird. Nur für die Grundkonzepte, Pseudocode (Abfolge, Verzweigung und Iteration - nicht im Fokus dieser Arbeit) und Navigation in Objektmodellen, werden Vorschläge gemacht.

3.1.3 Struktur und Eigenschaften der NE

Der Artikel [HHK] führt *nicht* schwerpunktmäßig eine eigenständige Notation für Navigationsausdrücke („Navigation Expressions“ - NE) ein. Er soll *vielmehr* die Semantik von Navigationsausdrücken im allgemeinen vorstellen. In der exakten, auf der formalen Sprache Larch (LSL) basierenden Definition der Semantik unterscheidet sich dieser Ansatz von den beiden anderen.

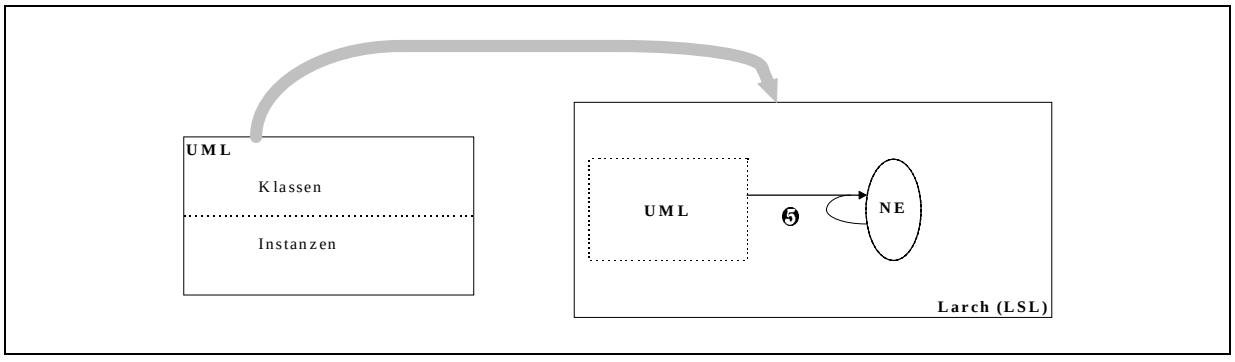


Abbildung 9: Struktur der NE

Die Definition der Semantik erfolgt, indem die wichtigsten Konzepte des UML-Objektmodells auf Larch-Sorten abgebildet werden. Larch-Ausdrücke können somit die Informationen von Instanzen verarbeiten. Die Semantik von Navigationsausdrücken wird durch ihre Abbildung auf Larch-Ausdrücke definiert. Die Transformations-Operatoren (Boolesche und Mengenoperatoren etc.), welche für die Formulierung vollständiger Bedingungen über Objektmodelle nötig sind, werden von der Sprache Larch übernommen.

3.2 Vergleich der wichtigsten Operatoren

Der Vergleich der wichtigsten Operatoren orientiert sich an den Operatoren der OCL. Es wird untersucht, inwieweit die Operatoren bei den anderen Sprachen auch vorhanden sind und gegebenenfalls ihre Syntax gegenübergestellt. Operatoren, auf die in diesem Vergleich nicht eingegangen wird, sind nur für die OCL im Rahmen der Sprache definiert und bei den anderen Sprachen nicht explizit vorhanden.

3.2.1 Navigation auf Attribute

`OCL-NAVATTR : Any → BasicSet`

Notation	Syntax	Bemerkung
OCL	`_.<Attributname>`	Def.: [OCL97] S. 8 (5.2)
ONN	`_.<Attributname>`	Def.: [BlahPrem98] S. 104 (5.3.7)
NE	`_.<Attributname>`	Def.: [HHK] S. 5

3.2.2 Navigation auf Operationen

`OCL-NAVOPER : Any → BasicColl`
`OCL-NAVOPER : Any {× BasicColl}`^{n} `→ BasicColl`

Notation	Syntax	Bemerkung
OCL	`_.<Operationsname>()` `_.< Operationsname>(_{,_}`$^{n-1}$`)`	Def.: [OCL97] S. 8 (5.3) `( )` nicht optional nur Query-Methoden!
ONN	`_#<Operationsname>` kein Ausdruck!	Def.: [BlahPrem98] S. 97 („Method invocation“) kein Ausdruck sondern Anweisung
NE	-	laut [HHK] S. 14 in Arbeit

3.2.3 Namenskonventionen für Rollennamen

Während für Klassen, Attribute und Operationen die Vergabe von in einem bestimmten Namensraum eindeutigen Bezeichnern von allen hier untersuchten Methoden zwingend vorgeschrieben wird, ist dies für die Konstrukte im Bereich der Objektbeziehungen mehr oder weniger nicht der Fall (wohl um die Klassendiagramme nicht mit Rollennamen zu überfrachten).
Um in Navigationsausdrücken dennoch den Gegenstand der Navigation bezeichnen zu können, bieten die Notationen unterschiedliche Möglichkeiten der Benennung. Auf die Semantik der Navigations-Operatoren haben die Benennungsvorschriften keinen Einfluß. Um die vielen Kombinationsmöglichkeiten mit den Navigationsausdrücken nicht einzeln behandeln zu müssen, werden die Benennungsvorschriften unabhängig von den eigentlichen Operatoren verglichen.
Achtung: Eventuelle Mehrdeutigkeiten der Rollen-Spezifikation müssen ausgeschlossen werden, wenn nicht anders möglich durch Vergabe eines eindeutigen Rollennamens im Klassendiagramm!

Notation	Benennung von Rollen	Benennung von Assoziationsklassen
OCL	• Expliziter Zielrollenname ([OCL97] S. 8 (5.4)) • „zielklassenname“ (mit erstem Buchstaben als Kleinbuchstaben) ([OCL97] S. 9 (5.4.1)) • „Pfadname::RollenBenennung“ bei Mehrdeutigkeiten in der Vererbungsstruktur ([OCL97] S. 11 (5.8))	• „assoziationsklassenname“ (mit erstem Buchstaben als Kleinbuchstaben) ([OCL97] S. 10 (5.5)) • „Pfadname::assoziationsklassenname“ bei Mehrdeutigkeiten in der Vererbungsstruktur ([OCL97] S. 11 (5.8))
ONN	• Expliziter Zielrollenname ([BlahPrem98] S. 98) • Zielklassenname ([BlahPrem98] S. 98)	identisch mit Benennung von Rollen (Grund: siehe 3.3.2)

	• „~Quellrollenname“ (Kennzeichnung durch Tilde) ([BlahPrem98] S. 99) • „~Quellklassenname“ (Kennzeichnung durch Tilde) ([BlahPrem98] S. 99)	
NE	• Rollenname ([HHK] S. 5) • Aus Beispielen in ([HHK] S. 4, 5) geht hervor, daß die Einzahl (0..1, 1..1) bzw. Mehrzahl (*) des Klassennames mit kleinem Anfangsbuchstaben bei nicht spezifiziertem Rollennamen verwendet werden kann. Eine Definition dafür konnte nicht gefunden werden.	Assoziationsklassen nicht berücksichtigt

3.2.4 Navigation auf Assoziationsenden

`OCL-NAVASSE : Any → AnySetSeq`

Notation	Syntax	Bemerkung
OCL	`_.<Rollenname>`	Def.: [OCL97] S. 8-9 (5.4)
ONN	`_.<Rollenname>`	Def.: [BlahPrem98] S. 98 (5.3.1) Nicht für Sequenzen definiert
NE	`_.<Rollenname>`	Def.: [HHK] S. 5(SET), 11(SEQ), 13(ANY)

3.2.5 Navigation auf Assoziationsobjekte

`OCL-NAVASOB : Any → Set`

Notation	Syntax	Bemerkung
OCL	`_.<Assoziationsklassenname>`	Def.: [OCL97] S. 10 (5.5) sehr knapp beschrieben
ONN	`_@<Rollenname>` `{<Quellrollenname>=_, <Assoziationsname>}`	Def.: [BlahPrem98] S. 103 Konzept von Links (siehe 3.3.2)
NE	-	Assoziationsklassen nicht berücksichtigt

3.2.6 Navigation von Assoziationsobjekten zu Assoziationsenden

`OCL-NAVAOAE: Any → Any`

Notation	Syntax	Bemerkung
OCL	`_.<Rollenname>`	Def.: [OCL97] S. 10 (5.6)
ONN	`_.<Rollenname>`	Def.: [BlahPrem98] S. 102 (5.3.4)
NE	-	Assoziationsklassen nicht berücksichtigt

3.2.7 Navigation auf qualifizierte Assoziationsenden ohne Qualifikatorangabe

`OCL-NAVUNQU : Any → Set`

Notation	Syntax	Bemerkung
OCL	`_.<Rollenname>`	Def.: [OCL97] S. 11 (5.7)
ONN	`_.<Rollenname>`	Def.: [BlahPrem98] S. 100 (5.3.2)
NE	-	Qualifizierte Assoziationen nicht berücksichtigt

3.2.8 Navigation auf qualifizierte Assoziationsenden mit Qualifikatorangabe

`OCL-NAVQUAL : Any × Basic {× Basic}`^{n} `→ AnySetSeq`

Notation	Syntax	Bemerkung
OCL	`_.<Rollenname>[_,{_}`^{n}`]`	Def.: [OCL97] S. 11 (5.7) Anzahl und Reihenfolge der Werte durch Qualifikationsspezifikation fest vorgegeben
ONN	`_.<Rollenname>[<qualifier>=_,{<qualifier>=_}`^{n}`]`	Def.: [BlahPrem98] S. 98 (5.3.1) nicht für Sequenzen definiert
NE	-	Qualifizierte Assoziationen nicht berücksichtigt

3.2.9 Selektion

`OCL-SELECT : Coll × (Basic → Bool) → Coll`

Notation	Syntax	Bemerkung
OCL	`_->select(_)`	Def.: [OCL97] S. 15 (6.1) und S. 27, 29, 31
ONN	`_[_]`	Def.: [BlahPrem98] S. 103 (5.3) nur für Mengen definiert
NE	`_[_]`	Def.: [HHK] S. 5, nur für Mengen definiert Bag und Seq prinzipiell berücksichtigt

3.2.10 Objekttypabbildung

`OCL-ASTYPE : Any × Type → Any`

Notation	Syntax	Bemerkung
OCL	`_.oclAsType(_)`	Def.: [OCL97] S. 21 (7.1.2)
ONN	`_:_`	Def.: [BlahPrem98] S. 101 (5.3.3) siehe auch 3.3.5
NE	-	Verweis auf [HH-97]

3.2.11 Zusammenfassung an einem Beispiel

Die Syntax der einzelnen Notationen für die wichtigsten Operatoren wird im folgenden in einer Tabelle zusammenfassend gegenübergestellt. Die zusammenfassende Übersicht verwendet Beispielinstanzen aus folgendem Beispiel:

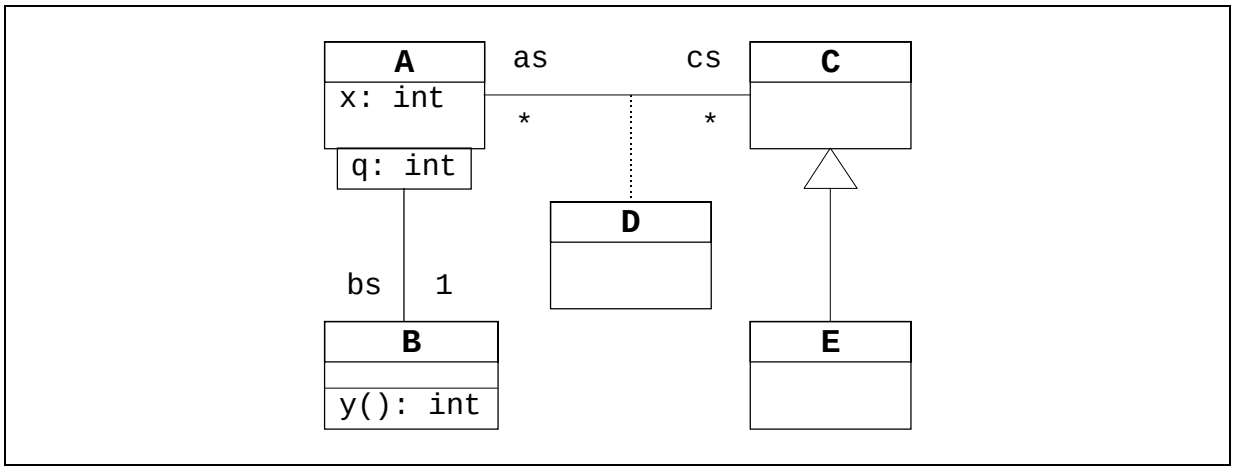


Abbildung 10: Abstrakter Diskursbereich für Zusammenfassung

Definition von Variablen:

```
mengeA : Set(A)          einA : A
mengeB : Set(B)          einB : B
mengeC : Set(C)          einD : D
mengeD : Set(D)          einE : E
einInt : int
```

	OCL	ONN	NE
Navigation auf Attribute	`einInt = einA.x`	`einInt = einA.x`	`einInt = einA.x`
Navigation auf Operationen	`einInt = einB.y()`	`einInt = einB#y`	-
Navigation auf Assoziationsenden	`mengeC = einA.cs`	`mengeC = einA.cs`	`mengeC = einA.cs`
Navigation auf Assoziationsobjekte	`mengeD = einA.d`	`mengeD = einA@cs`	-
Navigation von Assoziationsobjekten zu Assoziationsenden	`einA = einD.as`	`einA = einD.as`	-
Navigation auf qualifizierte Assoziationsenden ohne Qualifikatorangabe	`mengeB = einA.bs`	`mengeB = einA.bs`	-
Navigation auf qualifizierte Assoziationsenden mit Qualifikatorangabe	`einB = einA.bs[835]`	`einB = einA.bs[q=835]`	-
Selektion	`mengeA = mengeA->select (x>100)`	`mengeA = mengeA[x>100]`	`mengeA = mengeA[x>100]`
Objekttypabbildung	`einE = einD.oclAsType(E)`	`einE = einD:E`	-

Anzahl der Operatoren

Es fällt auf, daß NE nach [HHK] deutlich weniger Operatoren unterstützt. Dies hat seine Ursache darin, daß der im Vergleich zu den anderen Publikationen kleine Artikel nicht auf die Konzepte der Qualifikation und der Assoziation als Klasse eingeht.

Stil der Syntax

Obwohl gerade die OCL das Konzept einer reinen Ausdrucks-Sprache konsequent einhält, vermittelt die Syntax der OCL den Eindruck einer Programmiersprache. Dies ist wahrscheinlich der Grund dafür, daß die Autoren für die OCL den Anspruch der leichten Erlernbarkeit für „durchschnittliche Geschäfts- oder Systemmodellierer“ erhebt (siehe [OCL97] S. 1 (1.1)).
Bei tiefgreifenden Untersuchungen (wie im Abschnitt 2.4) ist jedoch eine aus der mathematischen Formelsprache hervorgegangene Notation (über Larch) wie von NE übersichtlicher und besser handhabbar.
Die Syntax der ONN verwendet für die Navigationsausdrücke eine Vielzahl von Sonderzeichen (außer . auch #, @ und :). Dies ermöglicht kurze Ausdrücke und eine sichere Unterscheidung.

3.3 Vergleich weiterer Details

Zum Abschluß sollen Details, welche durch den Vergleich der Operatoren noch nicht erfaßt werden konnten, verbal verglichen werden.

3.3.1 Kollektionsdatentypen

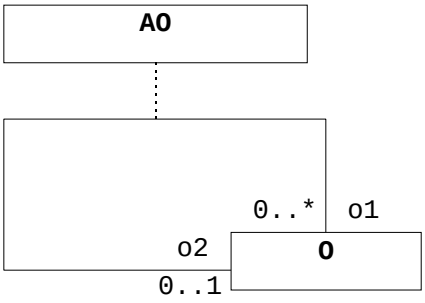
Die ONN berücksichtigt keine geordneten Kollektionen. Die Ordnung von Rollen (Constraint `{ordered}`) kann somit nicht berücksichtigt werden. Die Definition wird dadurch zwar deutlich einfacher, die praktische Anwendung in den meisten Fällen jedoch erschwert. NE geht auf geordnete Kollektionen nur am Rande ein, wogegen diese bei der Definition der OCL umfassend berücksichtigt wurden.

3.3.2 Konzept der Links

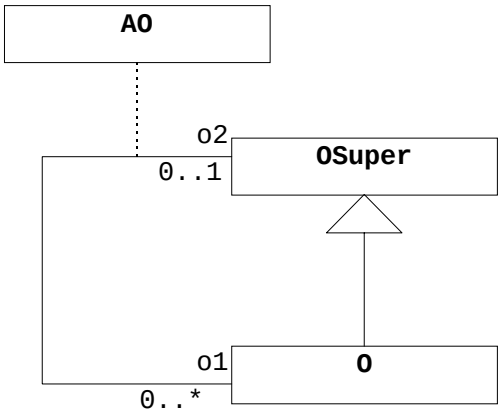
Die ONN schließt das Konzept der Links ganzheitlich ein. Während ein Link in der ONN grundsätzlich als ein Tupel der beteiligten Klassen dargestellt werden und Gegenstand von Ausdrücken sein kann, sind Links in der OCL und NE körperlos, d.h. sie können nur zum Navigieren benutzt werden. Bei der OCL können Links Entitäts-Charakter besitzen, wenn im zugrunde liegenden Modell dem Link diese Körperlichkeit durch das Konstrukt „Assoziation als Klasse“ explizit zugewiesen wurde (und sie somit selbst Objekte sind und auch eine Identität besitzen). Reine Beziehungsattribute können also nur von der ONN berücksichtigt werden.

3.3.3 Zugriff auf Assoziationsobjekte über Rollennamen

Die ONN verwendet im Gegensatz zur OCL für die Navigation zu Assoziationsobjekten den Namen der Rolle anstatt des Namens des Assoziationsobjekts. Am folgenden Beispiel wird klar, warum dies zweckmäßig ist.



In der OCL ist das Ergebnis von `einO.AO` unbestimmt. Es kann sich sowohl um die Instanzen von AO handeln, welche O in der Rolle o1 oder o2 besitzen. In der ONN können die beiden Varianten mit `einO@o1` bzw. `einO@o1` unterschieden werden.
Für die OCL muß das zugrunde liegende UML-Modell wie folgt geändert werden:



Nun kann der Zugriff über `einO.OSuper::AO` bzw. `einO.AO` erfolgen, da die beiden Rollen und damit die Navigation auf das Assoziationsobjekt über das Pfadnamenkonzept unterschieden werden können.

3.3.4 Abbilder und automatisches Verebnen

Die OCL definiert den Abbilder-Operator (`OCL-COLLECT`) und die Punktnotation als Kurzform dafür. Dadurch wird es innerhalb der OCL möglich, auch von Kollektionen aus zu navigieren, obwohl die Navigationsoperatoren nur von einzelnen Objekten ausgehend definiert sind.
Laut Definition ist es in der OCL jedoch nicht möglich, von einer Menge aus auf mehrwertige Attribute und Rollen sowie Operationen mit Kollektionen als Ergebnis zu navigieren.
Die beiden anderen Sprachen definieren die Navigation sowohl von Objekten als auch von Mengen aus. Dadurch besitzen diese nicht die Einschränkung wie die OCL. Eine exakte Definition wie bei der OCL würde dadurch jedoch stark an Umfang zunehmen.
Die ONN definiert die Navigation über Objekte und Objektmengen gemeinsam mit dem Konzept des automatischen Verebnens (Umwandeln von Mengensystemen in Mengen):
„(1) iteriere über alle Objekte der Menge,
(2) finde für jedes Objekt die assoziierten Objekte und
(3) gebe die Vereinigungsmenge zurück“ ([BlahPrem98] S. 98 unten).

3.3.5 Objekttypabbilder

Während die NE nicht auf die Wandlung des bekannten Typs eines Objekts eingeht, besitzt die ONN mit dem Doppelpunkt-Operator ein wesentlich leistungsfähigeres Konzept als die OCL. Anstelle der Pfadangaben der OCL und des Operators `OCL-ASTYPE` (welcher nur für einzelne Objekte definiert ist) existiert bei der ONN mit dem Doppelpunkt-Operator ein einheitliches Konzept. Einzelne Objekte und Objektmengen können zu Sub- oder Supertypen gewandelt werden. Ist die Umwandlung zu einem Subtyp für ein bestimmtes Objekt nicht möglich, so wird (anstelle der Produktion eines undefinierten Wertes) das Element nicht in die Zielmenge eingefügt.

3.3.6 Zeitpunkt in Nachbedingungen

OCL und NE eröffnen für Nachbedingungen die Möglichkeit des Zugriffs auf die Werte vor der Ausführung der Operation. Da die ONN nicht für die Beschreibung von Nachbedingungen vorgesehen ist, ist diese Eigenschaft bei ihr nicht ausgeprägt.

3.3.7 Umgang mit undefinierten Werten

Die OCL geht oberflächlich (verbale Beschreibung) auf undefinierte Werte ein. Die Operatoren der ONN sind im Gegensatz dazu so konstruiert, daß sie immer definierte Werte als Ergebnis haben (dies wird aber nur durch das Weglassen von geordneten Kollektionen und Verarbeitungs-Operatoren möglich). [HHK] geht nicht auf undefinierte Werte ein.

4 Integritätssicherung und SQL

In diesem Kapitel soll keine umfassende Beschreibung der Sprache SQL (wie im Kapitel 2 zur OCL vorgenommen) erfolgen. Vielmehr soll nur eine tiefere Betrachtung der in dieser Arbeit im Mittelpunkt stehenden Integritätssicherung erfolgen.

Die Integrität von relationalen Datenbanken wird durch Integritätsregeln, welche aus Integritätsbedingungen und integritätserhaltenden Aktionen bestehen, sichergestellt.

`not Integritätsbedingung ⇒ Aktion`

Integritätsbedingungen sollen Integritätsverletzungen feststellen. Wird eine Integritätsbedingung zu false ausgewertet (d.h. Integritätsverletzung liegt vor), so muß das Datenbanksystem eine integritätserhaltende Aktion ausführen. In der Regel ist die integritätserhaltende Aktion das Zurücksetzen der aktuellen Transaktion. Es können aber auch andere Aktionen definiert werden (z.B. CASCADE (Fortpflanzung des Löschens) oder Trigger-Aktionen).

Weil die auszuführenden integritätssichernden Aktionen mit der OCL nicht beschrieben werden können (fehlende Datenmanipulation) und eine automatische Ermittlung der geeigneten Aktion nur in Ausnahmefällen möglich ist, soll in dieser Arbeit nur die Aktion "Zurücksetzen der aktuellen Transaktion" berücksichtigt werden. Da nur Integritätsregeln mit der integritätserhaltenden Aktion "Zurücksetzen der Transaktion" betrachtet werden, sollen die Begriffe Integritätsregel und Integritätsbedingung im folgenden synonym verwendet werden.

Die Integritätsbedingungen können laut [Reuter87] u.a. nach folgenden Kriterien eingeteilt werden:

- Reichweite (die Bedingung betrifft ...)
    - ein Attribut
    - mehrere Attribute einer Satzausprägung
    - mehrere Ausprägungen derselben Satzart (Relation)
    - mehrere Ausprägungen aus verschiedenen Relationen
- Zeitpunkt der Überprüfbarkeit
    - unverzögerte Integritätsbedingungen
    - verzögerte Integritätsbedingungen
- Art der Überprüfbarkeit
    - Zustandsbedingungen
    - Zustandsübergangsbedingungen

Zustandsübergangsbedingungen könnten mit den Mitteln der OCL formuliert werden. Das dafür nötige Konstrukt `@pre` zur Berücksichtigung alter Werte ist jedoch auf Nachbedingungen beschränkt. Diese Beschränkung wurde wahrscheinlich von den Desingern der OCL vorgenommen, um keine Alternative zu den Zustandsübergangsdiagrammen zu schaffen (was u.U. zu redundaten oder wenig anschaulichen Analysedokumenten führen würde). Da die Materie der dynamschen Modellierung noch wesentlich weniger exakt erfaßt ist als die statische Modellierung und eine relativ geringe Bedeutung für Datenbankanwendungen besitzt, soll in dieser Arbeit nicht weiter auf diesen Aspekt eingegangen werden. Es werden somit nur Zustandsbedingungen behandelt.

4.1 Integritätsbedingungen in SQL-92

Die Integritätsbedingungen von SQL-92 lassen das gesamte Spektrum von Reichweite und Überprüfungszeitpunkten zu. Dies wird besonders beim Konstrukt ASSERTION deutlich. Mit

```
CREATE ASSERTION ...
  CHECK (...) INITIALLY DEFERRED
```

können beliebige Integritätsbedingungen deskriptiv formuliert werden. Das Datenbanksystem sorgt dann dafür, daß die spezifizierte Integritätsbedingung für die gesamte Datenbank von jeder abgeschlossenen Transaktion eingehalten wird, indem integritätsverletzende Transaktionen zurückgesetzt werden.
Mit dem Schlüsselwort IMMEDIATE kann die Integritätsbedingung unverzögert überprüft werden. Das Datenbanksystem garantiert deren Einhaltung dann nach jeder Datenmanipulationsoperation.

4.2 Integritätsbedingungen in Sybase™ 11 und Oracle8™

In den kommerziellen Implementierungen von SQL wird die allgemeinste Form der Integritätsbedingung (ASSERTION) meist nicht realisiert. So verhält sich dies auch bei Sybase™ und Oracle™.
Mit Triggern können jedoch unverzögerte Integritätsbedingungen realisiert werden. Erstreckt sich die Reichweite auf mehrere Ausprägungen aus verschiedenen Relationen, so muß für alle beteiligten Relationen ein Trigger realisiert werden, da Datenmanipulationen in allen beteiligten Relationen zu einer Verletzung der Bedingung führen können.

```
CREATE TRIGGER ON <Tabelle>
  ON INSERT, UPDATE, DELETE
  AS
   IF (NOT <Bedingung>)
   BEGIN
     ROLLBACK TRANSACTION
   END
```

Außerdem wird eine eingeschränkte Variante des CHECK-CONSTRAINTs angeboten. Die wesentliche Einschränkung besteht darin, daß keine Subqueries verwendet werden dürfen. Die Reichweite wird somit auf ein oder mehrere Attribute einer Satzausprägung beschränkt. Nach jeder Datenmanipulation können die Datenbanksysteme solche Bedingungen demzufolge mit minimalem Aufwand überprüfen.

```
CREATE TABLE <Name>
(
   ...
   CHECK <Bedingung>,
   ...
)
```

5 Überblick über die Abbildung von OCL auf SQL

Dieses Kapitel leitet die Abbildung von OCL auf SQL ein und vermittelt einen Überblick über Struktur und Eigenschaften der Abbildung. Die konkreten Abbildungsregeln (bzw. -muster) werden dann in den folgenden Kapiteln eingehend beschrieben.

5.1 Aufteilung der Abbildungsmuster auf drei Kapitel

Die Grundlage der Aufteilung der Abbildungsmuster ist die in Kapitel 2 eingeführte Struktur der OCL. Die Illustration zur Struktur der OCL (siehe Abschnitt 2.2) wird an dieser Stelle nochmals abgebildet, da die in Bezug auf die Abbildung von OCL nach SQL wesentlichsten Eigenschaften durch sie illustriert werden.

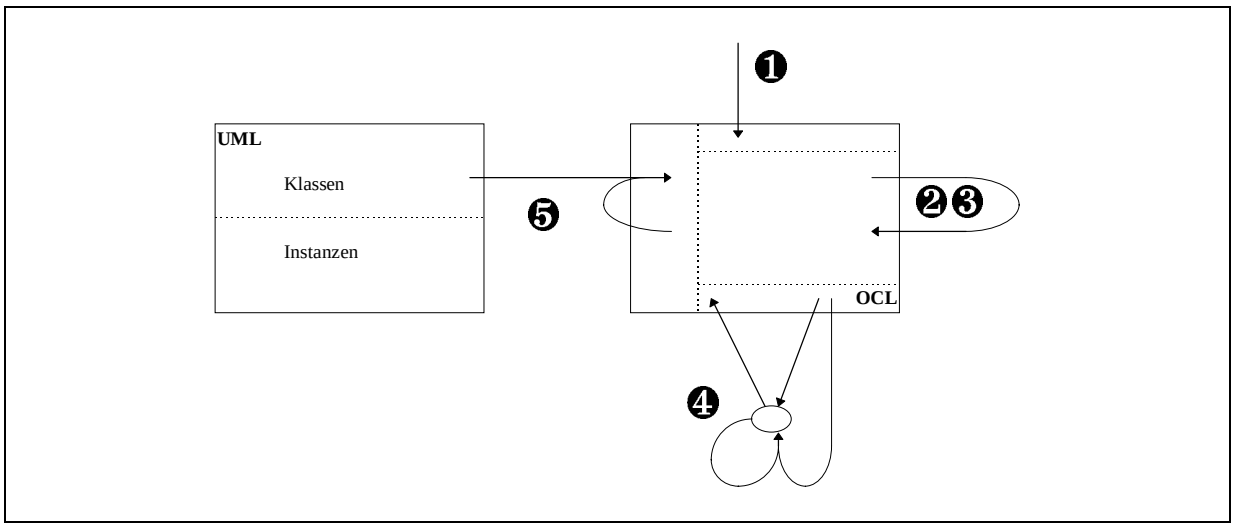


Abbildung 11: Unterschiedliche Eigenschaften der Gruppen von Operatoren der OCL

Diese Gruppierung der Operatoren wurde in Kapitel 2 eingeführt um einen besseren Überblick über die Vielzahl der Operatoren der OCL zu vermitteln. Darüber hinaus hat sie aber ab Kapitel 5 eine noch bedeutendere Funktion. Durch die Gruppierung wird es möglich, Abbildungen von OCL nach SQL mit unterschiedlichen Eigenschaften getrennt zu betrachten.
So ergeben sich Probleme bei der Realisierung der Abbildungsmuster nur in den Gruppen ❸ und ❹ sowie bei den „weiteren Aspekten der OCL“.
Mit der Gruppe ❺ wurde ein minimaler Satz von Operatoren abgegrenzt, welcher von der Abbildung der Klassendiagramme nach SQL abhängig ist. Da es viele Vorschläge und praktische Realisierungen der Abbildung von Klassendiagrammen auf SQL (z.B. bei relationaler Middleware) gibt, brauchen nur die Abbildungsmuster für diese Operatoren überarbeitet werden, wenn eine andere als die zugrunde gelegte Abbildung von Klassendiagrammen auf SQL verwendet werden soll.
Die Abbildung der Operatoren der Gruppen ❶ und ❷ ist problemlos und kann damit mit weniger Aufwand als bei den anderen Gruppen beschrieben werden.

Die Gruppen haben in Bezug auf die Abbildung nach SQL folgende Eigenschaften:
Die Operatoren der Gruppen ❶ und ❷ sind in SQL fast ohne Unterschiede vorhanden. Kapitel 6 beschreibt die entsprechenden Abbildungen.

Die Abbildung der Operatoren der Gruppen ❸ und ❹ enthält die größten Probleme. Einige Operatoren sind nur mit großem Aufwand in SQL nachbildbar. Das Konzept der geordneten Multimengen (Sequence) bringt dabei einen besonders hohen Aufwand, welcher die Kernkonzepte von SQL teilweise überfordert (`OCLH-INTRVAL`). Das Konzept der freien Iteration (`OCL-ITERATE`) kann durch SELECT-Anweisungen nicht nachgebildet werden. Hier kann nur noch eine Abbildung auf die prozeduralen Elemente von SQL3 oder Sybase™ wie Cursor und Schleifen vorgenommen werden. Die Abbildungen werden in Kapitel 7 detailliert beschrieben.

Die Gruppen ❶ - ❹ beziehen sich auf OCL-Zwischenergebnisse. Es werden somit Abbildungen der OCL-Datentypen nach SQL als Grundlage benötigt. Die Abbildungen der OCL-Operatoren der Gruppe ❺ basieren darüber hinaus auf einer Abbildung von UML-Klassendiagrammen auf SQL. Für die Abbildung von OO-Klassen auf SQL-Tabellen existieren in der Literatur und in Implementierungen (relationale Middleware) zahlreiche Alternativen. Die Abbildungen der OCL-Operatoren der Gruppe ❺ hängen von der jeweiligen Alternative ab. Da mit der Gruppe ❺ eine vergleichsweise geringe Zahl von Operatoren aus der Gesamtmenge der OCL-Operatoren abgespalten wurde, wäre auch die Betrachtung von Alternativen möglich. In Kapitel 8 wird eine Möglichkeit der Abbildung OO-Klassen auf SQL-Tabellen und die zugehörigen Abbildung von OCL-Operatoren auf SQL vorgestellt.

In Kapitel 9 wird die Abbildung der „weiteren Aspekte der OCL“ auf SQL beschrieben.

In Kapitel 10 werden Probleme mit den Abbildungen identifiziert und Wege zu deren Lösung skizziert.

5.2 Technik der Darstellung der Abbildung OCL auf SQL

In Kapitel 6 wird die Abbildung der OCL-Operatoren der Gruppen ❶ und ❷, in Kapitel 7 die Abbildung der OCL-Operatoren der Gruppen ❸ und ❹ und in Kapitel 8 die Abbildung der OCL-Operatoren der Gruppe ❺ beschrieben.
Zu Beginn der Kapitel werden jeweils die Grundlagen für die Abbildungen beschrieben. Für Kapitel 6 ist dies die Abbildung der OCL-Elementardatentypen auf SQL, für Kapitel 7 die Abbildung der OCL-Kollektionstypen auf SQL und für das Kapitel 8 die gewählte Abbildung von OO-Klassen auf SQL.
Anschließend folgen die Abbildungsmuster für die jeweiligen Operatoren. Die Abbildungsmuster werden hierbei immer nach dem selben Schema beschrieben. Dieses Schema orientiert sich an der Beschreibung der Operatoren in Abschnitt 2.2 und der Äquivalenzbedingungen in Abschnitt 2.4. Es übernimmt die dort eingeführten Darstellungstechniken und paßt diese an die speziellen Anforderungen an.

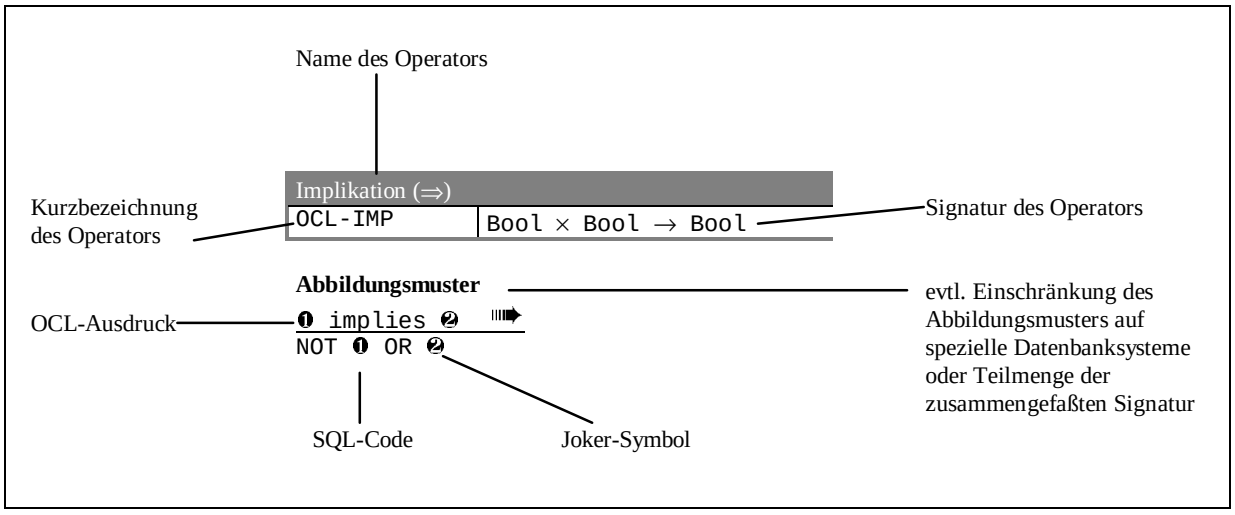


Abbildung 12: Darstellung der Abbildungsmuster

Zunächst wird der Operator, für den die folgenden Abbildungsmuster vorgesehen sind, durch Name, Kurzbezeichnung und Signatur benannt.
Nun folgen ein oder mehrere Abbildungsmuster. Bei mehreren Abbildungsmustern wird hinter dem Wort „Abbildungsmuster“ der Gültigkeitsbereich für die einzelnen Alternativen auf Teilmengen der gesamten Signatur eingeschränkt. Ein Abbildungsmuster besteht aus einer unterstrichenen Zeile mit einem OCL-Ausdruck, einem Abbildungssymbol (Pfeil) und ein oder mehreren Zeilen mit dem SQL-Code, welcher den OCL-Ausdruck implementiert. Die Joker-Symbole stehen auf Seiten des OCL-Ausdrucks für beliebige OCL-Teilausdrücke (soweit die Syntax der OCL dies zuläßt). Im SQL-Code repräsentieren diese den SQL-Code, welcher sich aus der Abbildung der jeweiligen OCL-Teilausdrücke auf SQL ergibt. Da sich SQL (abhängig von der Version des Standards bzw. Datenbanksystems) im Gegensatz zur OCL nicht immer für eine nahezu unbegrenzte Kombinierbarkeit der Operatoren eignet, wird auf die sich daraus ergebenden Probleme im Kapitel 10 eingegangen.

5.3 Abbildung vollständiger OCL-Ausdrücke auf SQL

Mit dem vorhergehend beschriebenen Formalismus wird nur dargestellt, wie einzelne OCL-Operatoren auf SQL abgebildet werden. Es ist aber im Allgemeinen möglich, daß an Stelle der Joker-Symbole (welche in der OCL für beliebige OCL-Teilausdrücke stehen) der für diese Teilausdrücke generierte Code in den zu generierenden SQL-Code eingesetzt wird. Somit lassen sich beliebig große OCL-Ausdrücke mit den beschriebenen Abbildungsmustern auf SQL abbilden.
Implementationen von SQL legen jedoch meist eine mehr oder weniger starke Einschränkung der Kombinationsmöglichkeiten fest. Außerdem treten „prozedurale Abbildungsmuster“ auf, bei denen das genannte Verfahren zur Kombination grundsätzlich nicht funktioniert.

5.4 Varianten und Alternativen

Für alle Abbildungen existieren Varianten für die verschiedenen Versionen des SQL-Standards und verschiedene SQL-Dialekte und oft zahlreiche Alternativen für einen bestimmten Dialekt. Die Untersuchung dieser verschiedenen Möglichkeiten kann genutzt werden, um besonders effiziente und „schöne“ Abbildungen zu finden und die Leistungsfähigkeit der einzelnen Dialekte gegenüberzustellen. Auf die Darstellung der Alternativen wurde bei der Beschreibung der Abbildung OCL auf SQL in den folgenden Kapiteln verzichtet, da dies die Lesbarkeit der Arbeit enorm beeinträchtigt hätte.
Auf der CD zur Diplomarbeit befindet sich eine Zwischenversion zur Diplomarbeit als Textdatei, welche eine Abbildung unter Berücksichtigung von Alternativen und unter Angabe von Beispielen beschreibt. Durch ihren Umfang von 118 Seiten war diese Darstellung jedoch für das Erfassen von wichtigen Eigenschaften und eine anschauliche Beschreibung ungeeignet.
Zusammenfassend kann festgestellt werden, daß die Abbildung von OCL auf SQL um so einfacher ist, je neuer die Version des SQL-Standards ist. Oracle8™ implementiert bereits mehr Features von SQL-92 und SQL3 als Sybase™ 11, und ist damit wesentlich besser für eine praktische Realisierung der Abbildung geeignet.

5.5 SQL-Dialekt für die folgenden Abbildungsmuster

Die folgenden Abbildungsmuster sind schwerpunktmäßig in Transact-SQL von Sybase™ realisiert. Diese Abbildungsmuster wurden weitestgehend an Beispielen überprüft (siehe CD zur DA). Wo mit SQL-92 oder SQL3 deutlich bessere Möglichkeiten bestanden, wurden die Abbildungsmuster entsprechend erstellt. In der Kopfzeile der Abbildungsmuster befindet sich dann ein Hinweis.

6 Abbildung der OCL-Operatoren der Gruppen 1 und 2 auf SQL

In diesem Kapitel werden die Abbildungsmuster für die Abbildung der OCL-Operatoren aus den im Kapitel 2 eingeführten Gruppen ❶ (Konstanten) und ❷ (einfache Operatoren über Elementardatentypen) auf SQL vorgestellt. Dafür muß zunächst die Abbildung der OCL-Elementardatentypen nach SQL beschrieben werden.

6.1 Abbildung der OCL-Elementardatentypen nach SQL

Für die OCL-Elementardatentypen existieren in SQL weitgehend direkte Entsprechungen, wie die folgende Tabelle illustriert:

OCL ➠	SQL
`Integer`	`INTEGER`
`Real`	`REAL`
`String`	`VARCHAR(200)`
`Enumeration`	`VARCHAR(200)`
`OclAny`	`INTEGER`
`OclType`	`VARCHAR(64)`

Für Integer und Real existieren direkte Entsprechungen. Für String ergibt sich in Transact-SQL eine Begrenzung auf 255 Zeichen. Durch die Längenbegrenzung der Indexe muß eine noch kleinere Zahl gewählt werden (200). Bei Enumeration werden die einzelnen Aufzählungselemente durch die entsprechenden Zeichenketten repräsentiert. Für OclAny wird der Datentyp INTEGER gewählt. Jedes Objekt wird somit durch eine eindeutige Nummer identifiziert (Objektidentifikator - OID). Die Klassen (Typen - OclType) werden durch ihren Namen repräsentiert.
Boolean ist der einzige Typ, welcher keine direkte Entsprechung in SQL besitzt. Er wird im folgenden auf die Wahrheitswerte, welche von der WHERE-Klausel und anderen Operatoren in SQL verarbeitet werden, abgebildet. Ein Typ BIT existiert zwar in Transact-SQL, dieser ist jedoch mit den Wahrheitswerten nicht zuweisungskompatibel.
Die Elementardatentypen treten oft auch als einzeilige Tabellen auf, z.B.:

```
TABLE (elem INTEGER)
```

Transact-SQL erlaubt an den meisten Stellen, wo Elementardatentypen vorgesehen sind, auch einzeilige Tabellen. Aus diesem Grund wird im folgenden auf den Unterschied nicht eingegangen.
In der bereits in Abschnitt 5.4 genannten Zwischenversion der Diplomarbeit beschriebenen Abbildung werden die verschiedenen Alternativen exakt unterschieden. Im Interesse einer übersichtlichen und einfachen Darstellung wird im folgenden darauf jedoch nicht eingegangen.

6.2 Eigenschaften der Abbildungsmuster

Nahezu alle in diesem Kapitel betrachteten OCL-Operatoren besitzen eine direkte Entsprechung in SQL. Nur die Operatoren `OCL-MIN` und `OCL-MAX` (Minimum bzw. Maximum zweier Zahlen) besitzen in SQL keine direkte Entsprechung. Hier können die in Abschnitt 2.4 vorgestellten Äquivalenzbedingungen verwendet werden. Alle Abbildungsmuster sind einfach und bringen keine technischen Probleme mit sich. Oft werden OCL-Teilausdrücke direkt in SQL übernommen. Aus diesem Grund besteht der Rest des Kapitels nur noch aus nicht kommentierten Abbildungsmustern.

6.3 Abbildung der OCL-Konstanten auf SQL

Konstante von Wahrheitswerttyp	
OCL-CONBOOL	→ Bool

Abbildungsmuster

OCL ⇛	SQL
false	(0=1)
true	(1=1)

Konstante von Ganzzahltyp	
OCL-CONINT	→ Int

Abbildungsmuster

❶ ⇛
❶ -- direkte Übernahme

Konstante von Gleitkommazahltyp	
OCL-CONREAL	→ Real

Abbildungsmuster

❶ ⇛
❶ -- direkte Übernahme

Konstante von Zeichenkettentyp	
OCL-CONSTR	→ Str

Abbildungsmuster

'❶' ⇛
'❶' -- direkte Übernahme

Konstante von Aufzählungstyp	
OCL-CONENUM	→ Enum

Abbildungsmuster

#❶ ⇛
'❶'

Erzeugung eines Typs	
OCL-CONTYPE	→ Type

Abbildungsmuster

❶ ⇛
'❶'

6.3.1 Abbildung der Operatoren der Booleschen Algebra

Bedingter Ausdruck (= {)	
OCL-COND	Bool × BasicColl × BasicColl → BasicColl

Abbildungsmuster (SQL-92)

```
if ❶ then ❷ else ❸ endif  ⇛
CASE WHEN ❶ THEN ❷ ELSE ❸ END
```

Negation (¬)	
OCL-NOT	Bool → Bool

Abbildungsmuster

```
not ❶  ⇛
NOT ❶                    -- direkte Übernahme, Kleinbuchstaben auch möglich
```

Konjunktion (∧)	
OCL-AND	Bool × Bool → Bool

Abbildungsmuster

```
❶ and ❷  ⇛
❶ AND ❷                  -- direkte Übernahme, Kleinbuchstaben auch möglich
```

Disjunktion (∨)	
OCL-OR	Bool × Bool → Bool

Abbildungsmuster

```
❶ or ❷  ⇛
❶ OR ❷                   -- direkte Übernahme, Kleinbuchstaben auch möglich
```

Implikation (⇒)	
OCL-IMP	Bool × Bool → Bool

Abbildungsmuster

```
❶ implies ❷  ⇛
NOT ❶ OR ❷
```

Äquivalenz (⇔)	
OCL-EQUB	Bool × Bool → Bool

Abbildungsmuster

```
❶ = ❷  ⇛
(❶ AND ❷) OR (NOT ❶ AND NOT ❷)
```

Exklusives Oder (xor)	
OCL-XOR	Bool × Bool → Bool

Abbildungsmuster

❶ xor ❷ ⇛
(❶ OR ❷) AND NOT (❶ AND ❷)

6.3.2 Abbildung der Operatoren der Algebra der Ganzen und Reellen Zahlen

Vergleich zweier Zahlen (=)	
OCL-EQU	IntReal × IntReal → Bool

Abbildungsmuster

❶ = ❷ ⇛
❶ = ❷ -- direkte Übernahme

Kleiner-Als-Relation (<)	
OCL-LT	IntReal × IntReal → Bool

Abbildungsmuster

❶ < ❷ ⇛
❶ < ❷ -- direkte Übernahme

Größer-Als-Relation (>)	
OCL-GT	IntReal × IntReal → Bool

Abbildungsmuster

❶ > ❷ ⇛
❶ > ❷ -- direkte Übernahme

Kleiner-Gleich-Relation (≤)	
OCL-LEQ	IntReal × IntReal → Bool

Abbildungsmuster

❶ <= ❷ ⇛
❶ <= ❷

Größer-Gleich-Relation (≥)	
OCL-GEQ	IntReal × IntReal → Bool

Abbildungsmuster

❶ >= ❷ ⇛
❶ >= ❷ -- direkte Übernahme

Negation (unäres -)	
OCL-NEG	IntReal → IntReal

Abbildungsmuster

```
- ❶
- ❶                                -- direkte Übernahme
```

Addition (+)	
OCL-ADD	IntReal × IntReal → IntReal

Abbildungsmuster

```
❶ + ❷
❶ + ❷                              -- direkte Übernahme
```

Subtraktion (-)	
OCL-SUB	IntReal × IntReal → IntReal

Abbildungsmuster

```
❶ - ❷
❶ - ❷                              -- direkte Übernahme
```

Multiplikation (·)	
OCL-MUL	IntReal × IntReal → IntReal

Abbildungsmuster

```
❶ * ❷
❶ * ❷                              -- direkte Übernahme
```

Division (/)	
OCL-DIVF	IntReal × IntReal → Real

Abbildungsmuster

```
❶ / ❷
❶ / CONVERT(REAL, ❷)
```

(Wenn nicht beide Parameter vom Typ Integer sind, kann die Konvertierung des 2. Parameters entfallen.)

Abrunden auf ganze Zahl (⌊ ⌋)	
OCL-FLOOR	IntReal → Int

Abbildungsmuster

```
❶.floor
FLOOR(❶)
```

Absoluter Betrag (| |)	
OCL-ABS	IntReal → IntReal

Abbildungsmuster

❶.abs ⇛
ABS(❶)

Maximum zweier Zahlen (max)	
OCL-MAX	IntReal × IntReal → IntReal

Abbildungsmuster (SQL-92)

❶.max(❷) ⇛
CASE WHEN ❶ > ❷ THEN ❶ ELSE ❷ END

Minimum zweier Zahlen (min)	
OCL-MIN	IntReal × IntReal → IntReal

Abbildungsmuster (SQL-92)

❶.min(❷) ⇛
CASE WHEN ❶ < ❷ THEN ❶ ELSE ❷ END

Ganzzahlige Division ohne Rest (div)	
OCL-DIVI	Int × Int → Int

Abildungsmuster

❶.div(❷) ⇛
❶ / ❷

Rest einer ganzzahligen Division (mod)	
OCL-MOD	Int × Int → Int

Abildungsmuster

❶.mod(❷) ⇛
❶ - ((❶ / ❷) * ❷)

6.3.3 Abbildung der Operatoren von Zeichenketten

Vergleich zweier Zeichenketten	
OCL-EQSTR	Str × Str → Bool

Abbildungsmuster

❶ = ❷ ➠
❶ = ❷ -- direkte Übernahme

Länge einer Zeichenkette	
OCL-SZSTR	Str → Int

Abbildungsmuster

❶.size ➠
LENGTH(❶) -- CHAR_LENGTH laut Handbuch !

Verknüpfung zweier Zeichenketten	
OCL-CONCAT	Str × Str → Str

Abbildungsmuster

❶.concat(❷) ➠
❶ + ❷

Umwandlung aller Buchstaben einer Zeichenkette in Großbuchstaben	
OCL-TOUPP	Str → Str

Abbildungsmuster

❶.toUpper ➠
UPPER(❶)

Umwandlung aller Buchstaben einer Zeichenkette in Kleinbuchstaben	
OCL-TOLOW	Str → Str

Abbildungsmuster

❶.toLower ➠
LOWER(❶)

Ermittlung einer Teilzeichenkette aus einer Zeichenkette	
OCL-SUBSTR	Str × Int × Int → Str

Abbildungsmuster

❶.substring(❷, ❸) ➠
SUBSTRING(❶, ❷, ❸ - ❷ + 1)

6.3.4 Abbildung der Operatoren von Aufzählungstypen

Vergleich zweier Aufzählungswerte (=)	
OCL-EQUENUM	Enum × Enum → Bool

Abbildungsmuster

❶ = ❷ ⇛
❶ = ❷ -- direkte Übernahme

Prüfen zweier Aufzählungswerte auf Ungleichheit (≠)	
OCL-NEQENUM	Enum × Enum → Bool

Abbildungsmuster

❶ <> ❷ ⇛
NOT ❶ = ❷

7 Abbildung der OCL-Operatoren der Gruppen 3 und 4 auf SQL

Die Abbildung der OCL-Operatoren der Gruppen ❸ und ❹ auf SQL beinhaltet die größten Schwierigkeiten der gesamten Abbildung OCL auf SQL.

Das größte Problem ist die Tatsache, daß für die Operatoren `OCLH-INTRVAL` und `OCL-ITERATE` keine deskriptiven Realisierungsmöglichkeiten in SQL bestehen. Nur unter Verwendung prozeduraler Erweiterungen (Oracle™: PL/SQL; Sybase™: Stored Procedures; SQL3: Persistent Stored Modules) lassen sich diese Operatoren auf SQL abbilden.
Diese prozeduralen Elemente bringen eine Reihe von Problemen mit sich:

- die Optimierung dieses SQL-Codes kann nicht vom Datenbanksystem durchgeführt werden, da der Algorithmus zur Berechnung des Ergebnisses fest vorgegeben wird
- die Abbildungsmuster für die anderen OCL-Teilausdrücke müssen u.U. angepaßt werden oder müssen auch prozedural realisiert werden („Übergreifen der Prozeduralität“)
- der generierte SQL-Code ist umfangreich und unübersichtlich
- die Kombinationsmechanismen (Übergabe der Zwischenergebnisse) sind zu komplex, um diese mit vertretbarem Aufwand formal darstellen zu können

Das zweite Problem ist die Komplexität einer großen Anzahl weiterer Abbildungsmuster. Diese verwenden übergebene Parameter mehrfach, wodurch sehr umfangreicher Code generiert wird. Viele dieser Abbildungsmuster stellen SELECT-Anweisungen mit hohem Aufwand dar, welche höchstwahrscheinlich von den verbreiteten RDBMS nicht optimiert werden bzw. mit den Mitteln von SQL nicht aufwandsärmer realisierbar sind (z.B. Rekonstruktion der dichten Ordnung von Sequenzen; s.u.).

Bei der Abbildung von OCL auf einen konkreten SQL-Dialekt kommt es zu zusätzlichen Problemen durch von Herstellern eingeführten Beschränkungen der SQL-Implementationen. Da deren Berücksichtigung eine umfangreichere und um (überflüssige) Details angereicherte Darstellung erfordert, wurden diese Beschränkungen in der folgenden Darstellung nicht berücksichtigt.

7.1 Abbildung der Kollektionstypen auf SQL

Die Abbildung der Kollektionstypen von OCL auf SQL wird wie folgt vorgenommen:

OCL ➠	SQL
`Set(<Type>)`	`TABLE (elem <Type>)`
`Bag(<Type>)`	`TABLE (elem <Type>)`
`Sequence(<Type>)`	`TABLE (elem <Type>, snr INTEGER)`

Die Bags (Multimengen) der OCL sind direkt auf Tabellen in SQL abbildbar. Die Sets (Mengen) der OCL würden den Mengen der Relationenalgebra bzw. Tabellen mit Primärschlüssel entsprechen. Da für Zwischenergebnisse von Anfragen keine Primärschlüssel definiert werden können, wird dasselbe Abbildungsmuster wie für Bags verwendet. Sequenzen (geordnete Multimengen) erhalten zur Darstellung der Reihenfolge in SQL eine Sequenz-Nummer. Diese sollte jede Zeile der Tabelle eindeutig identifizieren und von 0 bis (Anzahl der Elemente - 1) dicht vergeben sein (`MAX(snr)=COUNT(*)-1`). Durch SQL-Constraints kann dies aus oben genannten Gründen nicht sichergestellt werden.
Die Abbildungsmuster für die Operatoren sind so erstellt, daß sie die genannten Bedingungen für ihre Ergebnisse erfüllen.

7.1.1 Rekonstruktion der dichten Ordnung über Sequenzen

Einige Operatoren zerstören die dichte Ordnung der Sequenz-Nummern von Sequenzen. Dies ist z.B. beim Operator `OCL-EXC` der Fall, da das Entfernen von Elementen einer Sequenz entsprechende „Lücken“ hinterläßt:

```
Seq{4, 6, 7, 8}->excluding(7) = Seq{4, 6, 8}
```

elem	snr
4	0
6	1
8	3

Diese Lücken müssen wieder geschlossen werden, da einige Operatoren (z.B. `OCL-AT`) die dichte Ordnung benötigen. Dazu ist folgende SELECT-Anweisung geeignet:

```
SELECT elem, snr = (SELECT COUNT(*) FROM ❶ WHERE snr < s.snr)
FROM ❶ s
```

elem	snr
4	0
6	1
8	2

Überall, wo diese Nachbearbeitung notwendig ist, wird der Kommentar
`-- anschließend Sequenz-Nummern verdichten`
angehängt.

7.2 Abbildung der einfachen Operatoren über Kollektionstypen auf SQL

Die Abbildungsmuster sind teilweise mit Anmerkungen versehen. Zu beachten ist, daß das prozedurale Abbildungsmuster für `OCLH-INTRVAL` nicht für eine formale Kombination der Teilausdrücke (durch einsetzen des Codes) geeignet ist.

7.2.1 Abbildung der Konstruktion von Kollektionen auf SQL

Umwandlung eines einzelnen Elements in eine einelementige Kollektion	
`OCLH-SGLELEM`	`Basic12 → Coll`

Die Elementardatentypen müssen für diese Abbildungsmuster als einzeilige Tabellen vorliegen. Wenn dies nicht der Fall ist, so müssen diese zunächst via
`SELECT elem = ❶`
in eine solche einzeilige Tabelle überführt werden.

Abbildungsmuster für `Basic → SetBag`

```
❶
❶                                -- keine Aktionen nötig
```

Abbildungsmuster für `Basic → Seq`

```
❶
SELECT elem, snr=0
FROM ❶
```

Erzeugen einer Kollektion aus einer Intervallbeschreibung	
OCLH-INTRVAL	Int × Int → Coll (P)

Abbildungsmuster für Set **und** Bag

❶..❷ ➠

```
-- Deklarationsteil --
DECLARE @actAcc INTEGER

-- Anweisungsteil --
SELECT @actAcc = ❶

WHILE (NOT @actAcc > ❷)
BEGIN
  INSERT INTO [--Ergebnismenge--]
    VALUES (@actAcc)
  SELECT @actAcc = @actAcc + 1
END
-- Ergebnis liegt in @actAcc
```

Abbildungsmuster für Seq

❶..❷ ➠

```
-- Deklarationsteil --
DECLARE @actAcc INTEGER
DECLARE @actSnr INTEGER

-- Anweisungsteil --
SELECT @actAcc = ❶
SELECT @actSnr = 0

WHILE (NOT @actAcc > ❷)
BEGIN
  INSERT INTO [--Ergebnismenge--]
    VALUES (@actAcc, @actSnr)
  SELECT @actAcc = @actAcc + 1
  SELECT @actSnr = @actSnr + 1
END
-- Ergebnis liegt in @actAcc
```

Konstruktion einer Menge	
OCL-CONSET	Set {× Set}^{n} → Set

Abbildungsmuster

Set{❶, ❷, ... } ➠

```
❶
UNION
❷
UNION
...
```

Konstruktion einer Mulitmenge	
OCL-CONBAG	Bag {× Bag}^{n} → Bag

Abbildungsmuster

Bag{❶, ❷, ... } ➠

```
❶
UNION ALL
❷
UNION ALL
...
```

Konstruktion einer geordneten Multimenge	
OCL-CONSEQ	Seq {× Seq}^{n} → Seq

Abbildungsmuster (SQL-92)

Sequence{❶, ❷, ... } ➠
```
SELECT elem, snr
FROM ❶
UNION
SELECT elem, snr + (SELECT COUNT(*) FROM ❶)
FROM ❷
UNION
SELECT elem, snr + (SELECT COUNT(*) FROM (❶ UNION ALL ❷))
FROM ❸
...
```

7.2.2 Abbildung der Operatoren zur Erzeugung von Kollektionen aus anderen Kollektionstypen

Erzeugung der Menge, welche die Elemente wie die vorgegebene Kollektion besitzt	
OCL-ASSET	BagSeq → Set

Abbildungsmuster

❶ ➠
```
SELECT DISTINCT elem
FROM ❶
```

Erzeugung der Multimenge, welche die Elemente wie die vorgegebene Kollektion besitzt	
OCL-ASBAG	SetSeq → Bag

Abbildungsmuster

❶ ➠
```
SELECT elem
FROM ❶
```

Erzeugung der geordneten Multimenge, welche die Elemente wie die vorgegebene Kollektion besitzt	
OCL-ASSEQ	SetBag → Seq

Anmerkung

Die Ordnung der geordneten Multimenge wird aus der Speicherungsreihenfolge hergeleitet. Damit kann das Ergebnis in Abhängigkeit vom Datenbanksystem und der Geschichte der Datenmanipulationsoperationen (Zustandsübergänge der Objekte) unterschiedlich sein. Der Operator sollte demzufolge nicht verwendet werden.

Abbildungsmuster (ORACLE)

❶ ➠
```
SELECT elem, ROWNUM - 1 AS "snr"
FROM ❶
```

7.2.3 Abbildung der Aggregationfunktionen über Kollektionen

Anzahl der Elemente einer Kollektion (Kardinalität)	
OCL-SIZE	Coll → Int

Abbildungsmuster

❶->size ➠
```
SELECT ISNULL(COUNT(*), 0)
FROM ❶
```

Häufigkeit des Vorkommens einer Instanz als Element einer Kollektion	
OCL-COUNT	Coll × Basic → Int

Abbildungsmuster

❶->count(❷) ➠
```
SELECT ISNULL(COUNT(*), 0)
FROM ❶
WHERE elem = ❷
```

Summe aller Elemente einer Kollektion (Σ)	
OCL-SUM	Coll → IntReal

Abbildungsmuster

❶->sum ➠
```
SELECT SUM(elem)
FROM ❶
```

7.2.4 Abbildung der typischen Mengenoperationen über Kollektionen

Konstruktion einer um das Element erweiterten Kollektion (∪ { })	
OCL-INC	Coll × Basic → Coll

Abbildungsmuster für Set

❶->including(❷) ➠
```
❶
UNION
SELECT elem = ❷
```

Abbildungsmuster für Bag

❶->including(❷) ➠
```
❶
UNION ALL
SELECT elem = ❷
```

Abbildungsmuster für Seq

❶->including(❷) ➠
```
SELECT * FROM ❶
UNION
SELECT elem=❷, snr=(SELECT MAX(snr)+1 FROM ❶)
```

Konstruktion einer um das Element verminderten Kollektion (∪ { })	
OCL-ExC	Coll × Basic → Coll

Abbildungsmuster

❶->excluding(❷) ➠
```
SELECT *
FROM ❶
WHERE NOT elem = ❷
-- bei Sequenzen anschließend Sequenz-Nummern verdichten
```

Konstruktion einer um das Element erweiterten Kollektion (∪ { })	
OCL-APP	Seq × Basic → Seq

laut Äquivalenzbedingung identisch mit OCL-INC

Konstruktion einer um das Element erweiterten Kollektion ({ } ∪)	
OCL-PREP	Seq × Basic → Seq

Abbildungsmuster

❶->including(❷) ➠
```
SELECT elem=❷, snr=0
UNION
SELECT elem, snr=snr+1 FROM ❶
```

Mengenvereinigung (∪)	
OCL-UNION	Coll × Coll → Coll

Abbildungsmuster für Set × Set → Set

❶->union(❷) ➠
```
❶ UNION ❷
```

Abbildungsmuster für SetBag × BagSet → Bag

❶->union(❷) ➠
```
❶ UNION ALL ❷
```

Abbildungsmuster für Seq × Seq → Seq

❶->union(❷) ➠
```
SELECT elem, snr
FROM ❶
UNION
SELECT elem, snr = snr + (SELECT MAX(snr) + 1 FROM ❶)
FROM ❷
```

Mengendurchschnitt (∩)	
OCL-INTER	SetBag × SetBag → SetBag

Abbildungsmuster für Set × Set → Set **(SQL-92)**

❶->intersection(❷) ➠
❶ INTERSECT ❷

Abbildungsmuster für SetBag × SetBag → SetBag **(SQL-92)**

❶->intersection(❷) ➠
❶ INTERSECT ALL ❷

Symmetrische Differenz zweier Mengen (Δ)	
OCL-SYMDIFF	Set × Set → Set

Abbildungsmuster (SQL-92)

❶->symmetricDifference(❷) ➠
(❶ EXCEPT ❷) UNION (❷ EXCEPT ❶)

Differenz zweier Mengen (\)	
OCL-DIFFR	Set × Set → Set

Abbildungsmuster (SQL-92)

❶ - ❷ ➠
❶ EXCEPT ❷

7.2.5 Abbildung der besonderen Operatoren für geordnete Multimengen

Erstes Element einer geordneten Multimenge	
OCL-FIRST	Seq → Basic

Abbildungsmuster

❶->first ➠

```
SELECT elem
FROM ❶
WHERE snr = 0
```

Letztes Element einer geordneten Multimenge	
OCL-LAST	Seq → Basic

Abbildungsmuster

❶->last ➠

```
SELECT elem
FROM ❶
WHERE snr = (SELECT MAX(snr) FROM ❶)
```

Element einer geordneten Multimenge an bestimmter Position	
OCL-AT	Seq × Int → Basic

Abbildungsmuster

❶->at(❷) ➠
```
SELECT elem
FROM ❶
WHERE snr = (❷ - 1)
```

Ausschnitt aus einer geordneten Multimenge	
OCL-SUBSEQ	Seq × Int × Int → Seq

Abbildungsmuster

❶->subSequence(❷, ❸) ➠
```
SELECT elem, snr = s.snr - (❷ - 1)
FROM ❶ s
WHERE s.snr >= (❷ - 1) AND s.snr <= (❸ - 1)
```

7.2.6 Abbildung der relationalen Mengenoperatoren über Kollektionen

Test, ob Kollektion kein Element enthält	
OCL-ISEMPT	Coll → Bool

Abbildungsmuster

❶->isEmpty ➠
```
NOT EXISTS (❶)
```

Test, ob Kollektion mindestens ein Element enthält	
OCL-NOTEMPT	Coll → Bool

Abbildungsmuster

❶->notEmpty ➠
```
EXISTS (❶)
```

Ist-Element-Relation (∈)	
OCL-ELEM	Coll × Basic → Bool

Abbildungsmuster

❶->includes(❷) ➠
```
❷ IN (SELECT elem FROM ❶)
```

Teilmengenrelation (⊆)	
OCL-SUBSET	Coll × Coll → Bool

Achtung: ❶ ist die Obermenge!

Abbildungsmuster

```
❶->includesAll(❷)   ➠
NOT EXISTS
  (SELECT elem
   FROM ❷
   WHERE elem NOT IN (SELECT elem FROM ❶))
```

Vergleich zweier Kollektionen	
OCL-CMPCOLL	Coll × Coll → Bool

Abbildungsmuster für Set × Set → Bool3

```
❶ = ❷   ➠
(SELECT COUNT(*) FROM ❶) = (SELECT COUNT(*) FROM ❷) AND
(SELECT COUNT(*) FROM ❶ m1, ❷ m2
 WHERE m1.elem = m2.elem) = (SELECT COUNT(*) FROM ❶)
```

Abbildungsmuster für Bag × Bag → Bool3

```
❶ = ❷   ➠
NOT EXISTS
  (SELECT elem
   FROM ❶ b2
   GROUP BY elem
   HAVING COUNT(elem)>(SELECT COUNT(b1.elem)
                       FROM ❷ b1
                       WHERE b1.elem = b2.elem))
AND
NOT EXISTS
  (SELECT elem
   FROM ❷ b2
   GROUP BY elem
   HAVING COUNT(elem)>(SELECT COUNT(b1.elem)
                       FROM ❶ b1
                       WHERE b1.elem = b2.elem))
```

Abbildungsmuster für Seq × Seq → Bool3 **(SYBASE)**

```
❶ = ❷   ➠
(SELECT COUNT(*) FROM ❶) = (SELECT COUNT(*) FROM ❷) AND
 NOT EXISTS
   (SELECT elem FROM ❶ m1, ❷ m2
    WHERE m1.snr = m2.snr AND NOT m1.elem = m2.elem)
```

7.3 Abbildungen der Operatoren höherer Ordnung

7.3.1 Abbildung von OCL-ITERATE

Der Operator OCL-ITERATE wurde bereits im Abschnitt 2.4 als einer der vielseitigsten und ausdrucksstärksten Operatoren herausgestellt. Der Aufwand für seine Umsetzung nach SQL ist jedoch auch am höchsten. Da sich 7 Abbildungsmuster über je eine Seite ergeben würden, wird die Abbildung von OCL-ITERATE nicht mit Abbildungmustern sondern in einer zusammengefaßten Form dargestellt.
Eine nichtprozedurale Abbildung nach SQL existiert nicht.

Das Ergebnis des hier vorgestellten Abbildungsmusters kann für einen Teilausdruck nicht direkt in das Abbildungsergebnis eines anderen Teilausdrucks eingesetzt werden. Die Kombinationstechnik wird nicht formal beschrieben.

Freie Iteration	
`OCL-ITERATE`	`Coll × BasicColl × (Basic × BasicColl → BasicColl) → BasicColl`

Der Operator `OCL-ITERATE` hat folgende Semantik:

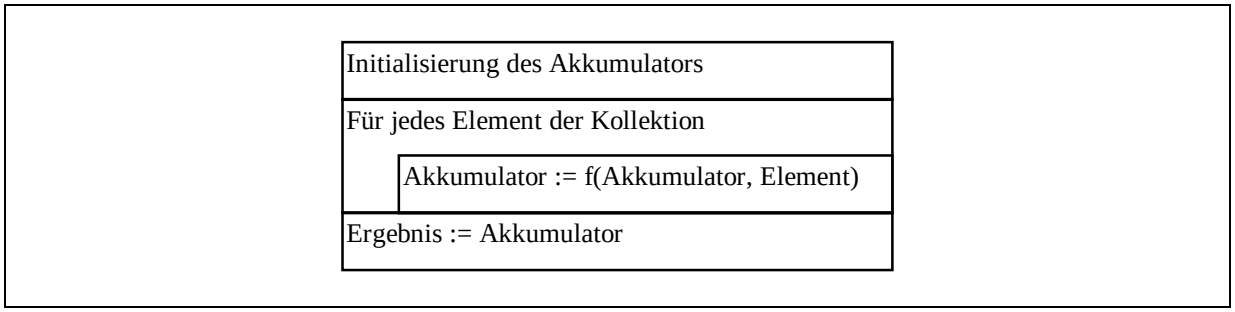


Abbildung 13: Struktogramm für OCL-IERTATE

Eine „Hilfsvariable“ vom Typ des Ergebnisses wird mit dem übergebenen Initialisierungswert belegt. Anschließend wird für jedes Element der Kollektion der neue Wert des Akkumulators berechnet. Der neue Wert des Akkumulators ist das Ergebnis der übergebenen Funktion im Kontext des aktuellen Akkumulators und des aktuellen Elements. Das Gesamtergebnis des Operators ist der Akkumulator nach dem letzten Iterationsschritt. Diese Berechnungsvorschrift kann in SQL nur mit prozeduralen Erweiterungen umgesetzt werden (hier: Transact-SQL). Um die einzelnen Elemente der Kollektion verarbeiten zu können, muß ein Cursor definiert und durchlaufen werden.

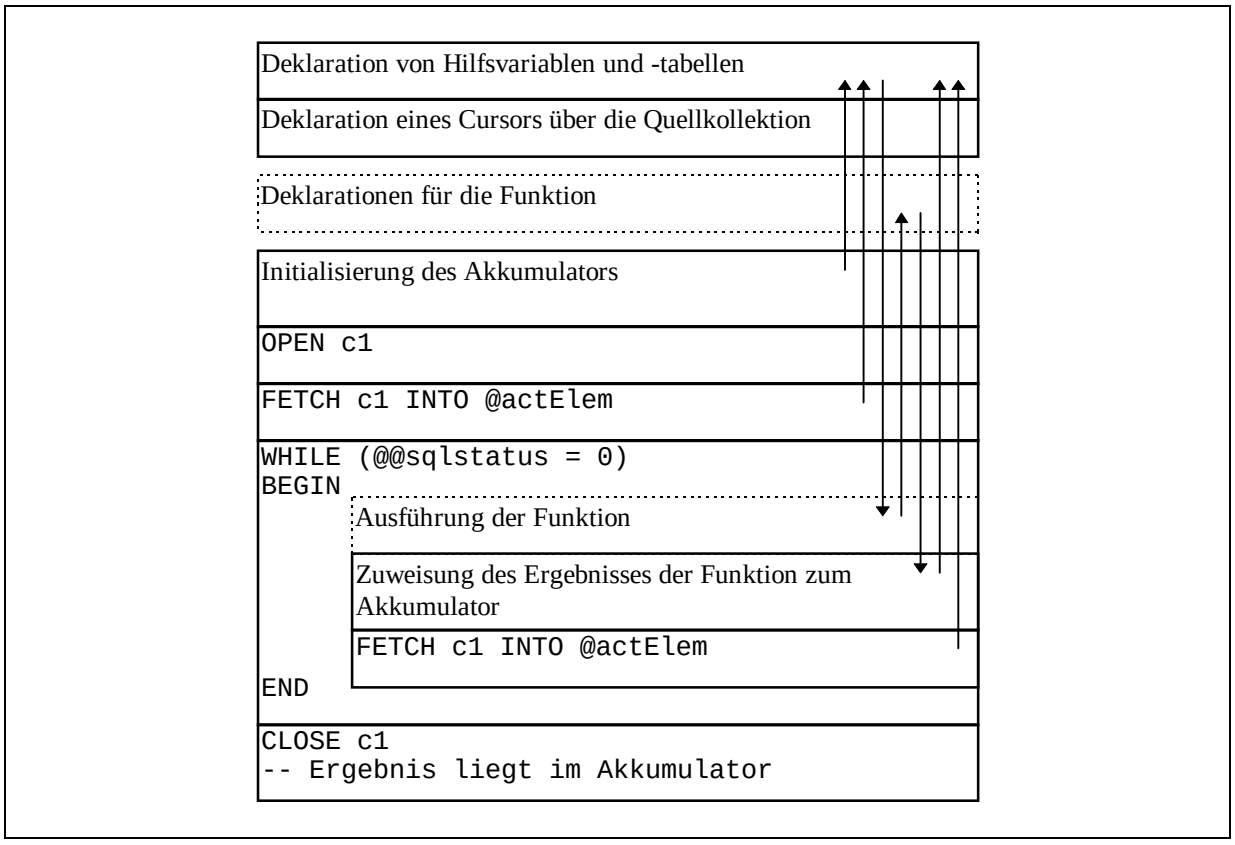


Abbildung 14: Prinzipdarstellung der Abbildung von OCL-ITERATE auf SQL

Deklaration der Hilfsvariablen und -tabellen

Es werden Hilfsvariablen für das aktuelle Element und den Akkumulator benötigt. Der Typ dieser Hilfsvariablen richtet sich nach der konkreten Signatur. Die Hilfsvariable für das aktuelle Element hat den Typ, welcher sich aus der Abbildung des Typs der Elemente der Quellkollektion (Parameter 1) auf SQL ergibt. Ist z.B. die Quellkollektion eine Menge von reellen Zahlen, so sieht die Deklaration der Hilfsvariablen für das aktuelle Element wie folgt aus:

```
DECLARE @actElem REAL
```

Die Hilfsvariable für den Akkumulator hat den Typ des Akkumulators, welcher mit dem Ergebnis des gesamten Iterate-Operators identisch ist. Dies kann ein Elementardatentyp oder ein Kollektionstyp sein. Für einen Elementardatentyp, z.B. String, wird eine Transact-SQL-Variable deklariert. Im konkreten Fall:

```
DECLARE @actAcc VARCHAR(200)
```

Für Kollektionstypen wird eine temporäre Tabelle erzeugt. Ist der Ergebnistyp `Bag(Boolean)`, so wird folgender SQL-Code generiert:

```
CREATE TABLE #actAcc (elem BOOLEAN)
```

Ist die Zielkollektion eine geordnete Multimenge, so muß eine dichte Nummernvergabe für die Elemente vorgenommen werden. Dafür wird eine Zählvariable deklariert, welche vom Typ Integer ist.

```
DECLARE @actSnr INTEGER
```

Deklaration eines Cursors über die Quellkollektion

Um auf die Elemente der Quellkollektion einzeln zugreifen zu können, muß ein Cursor deklariert werden. Dieser soll nur die Elemente der Quellkollektion liefern. Seine Deklaration sieht somit wie folgt aus:

```
DECLARE c1 CURSOR FOR
  SELECT elem
  FROM ❶
  FOR READ ONLY
```

Ist die Quellkollektion eine Sequenz, so kommt es darauf an, daß die Elemente in der Reihenfolge der Sequenz-Nummer verarbeitet werden. Dies wird durch den Zusatz von `ORDER BY snr ASC` hinter der FROM-Klausel erreicht.

Initialisierung des Akkumulators

Ist der Akkumulator von einem Elementardatentyp, so kann ihm der Initialisierungswert direkt zugewiesen werden. Hat der Akkumulator den Typ Integer so wird folgender SQL-Code generiert:

```
SELECT @actAcc = ❷
```

Die Tabelle für einen Akkumulator von einem Kollektionstyp muß zunächst gelöscht werden um dann den Inhalt von Parameter 2 des Operators `OCL-ITERATE` einzufügen:

```
TRUNATE TABLE #actAcc
INSERT INTO #actAcc
  SELECT * FROM ❷
```

Ist das Ergebnis von `OCL-ITERATE` eine geordnete Multimenge, so muß auch noch die Hilfsvariable `@actSnr` initialisiert werden:

```
SELECT @actSnr = 0
```

Zuweisung des Ergebnisses der Funktion zum Akkumulator

Für Akkumulatoren von Elementardatentypen geschieht diese Zuweisung wie folgt:

```
SELECT @actAcc = -- Ergebniszwischenspeicher der Funktion
```

Bei Kollektionstypen muß der alte Akkumulatorinhalt zunächst gelöscht werden, bevor eingefügt werden kann:

```
TRUNATE TABLE #actAcc
INSERT INTO #actAcc
  SELECT * FROM -- Ergebniszwischenspeicher der Funktion
```

Es sei folgendes Beispiel in OCL gegeben:

```
Seq{1,2,3}->iterate(elem : Integer; acc : Integer = 5 | acc*2 + elem * elem) - 3
= (((5 * 2 + 1 * 1) * 2 + 2 * 2) *2 + 3 * 2) - 3 = 58
```

Der folgende SQL-Code müßte dafür generiert werden:

```
CREATE TABLE Quellkollektion (elem INTEGER, snr INTEGER)

DECLARE @endErgebnis INTEGER
DECLARE @iterateErgebnis INTEGER
DECLARE @funktionErgebnis INTEGER
DECLARE @actElem INTEGER
DECLARE @actAcc INTEGER

DECLARE c1 CURSOR FOR
  SELECT elem
  FROM Quellkollektion
  ORDER BY snr ASC
  FOR READ ONLY

-- SQL-Code für Seq{1, 2, 3}

TRUNCATE TABLE Quellkollektion

INSERT INTO Quellkollektion
  SELECT elem = 1, snr = 0
  UNION
  SELECT elem = 2, snr = 1
  UNION
  SELECT elem = 3, snr = 2

SELECT @actAcc = 5

OPEN c1
FETCH c1 INTO @actElem

WHILE (@@sqlstatus = 0)
BEGIN
  -- SQL-Code für acc * 2 + elem * elem
  SELECT @funktionErgebnis = @actAcc * 2 + @actElem *  @actElem
  SELECT @actAcc = @funktionErgebnis
  FETCH c1 INTO @actElem
END

CLOSE c1

SELECT @iterateErgebnis = @actAcc
```

```
-- SQL-Code für ... - 3

SELECT @endErgebnis = @iterateErgebnis - 3

SELECT @endErgebnis -- Anzeige
```

Diese Abbildung wurde mit Sybase™ überprüft. Sie soll die Abbildungsregeln für OCL-ITERATE illustrieren.

7.3.2 Abbildung der übrigen Operatoren vom Typ einer Funktion höherer Ordnung

Wie bereits im Abschnitt 2.4 dargestellt, lassen sich alle Operatoren der Gruppe ❹ durch OCL-ITERATE darstellen. Da OCL-ITERATE jedoch nur prozedural auf SQL abgebildet werden kann, soll für die restlichen Operatoren noch nach nichtprozeduralen Abbildungen gesucht werden. Die prozeduralen Abbildungen werden im Folgenden nicht aufgeführt, da sich diese aus OCL-ITERATE und den Äquivalenzbedingungen herleiten lassen. Die nichtprozeduralen Abbildungsmuster können nur dann angewendet werden, wenn die übergebene Funktion vollständig nichtprozedural auf SQL abgebildet werden kann, da der SQL-Code direkt in die WHERE- bzw. SELECT-Klauseln eingefügt werden muß und eine Übergabe durch Zwischenergebnisse ausscheidet (da die Ausführung von den Operatoren angestoßen wird und nicht vorweggenommen werden kann). Letztere Aussagen gelten für Sybase™ 11.

Selektion	
OCL-SELECT	Coll × (Basic → Bool) → Coll

Abbildungsmuster

❶->select(2(elem))

```
SELECT *
FROM ❶
WHERE 2(elem)
-- anschließend Sequenz-Nummern verdichten (nur für Seq)
```

Selektion (mit invertiertem Prädikat)	
OCL-REJECT	Coll × (Basic → Bool) → Coll

Durch die entsprechende Äquivalenzbedingung kann OCL-REJECT leicht durch OCL-SELECT und OCL-NOT ersetzt werden.

Abbilder	
OCL-COLLECT	Coll × (Basic → Basic) → Coll

Abbildungsmuster für Set und Bag

❶->collect(2(elem))

```
SELECT elem = 2(elem)
FROM ❶
```

Abbildungsmuster für Seq

❶->collect(2(elem))

```
SELECT elem = 2(elem), snr
FROM ❶
```

Existenzquantor (∃)	
OCL-EXISTS	Coll × (Basic → Bool) → Bool

Abbildungsmuster

❶->exists(2(elem)) ➠
EXISTS (SELECT elem FROM ❶ WHERE 2(elem))

Allquantor (∀)	
OCL-FORALL	Coll × (Basic → Bool) → Bool

Abbildungsmuster 189

❶->forAll(2(elem)) ➠
NOT EXISTS (SELECT elem FROM ❶ WHERE NOT 2(elem))

7.4 Qualitätsmängel der Abbildungsmuster

Die schwerwiegendsten Qualitätsmängel resultieren aus prozeduralen Abbildungsmustern. Im Gegensatz zu deskriptiven Abbildungsmustern kann der Optimierer eines Datenbanksystems die Ergebnisse prozeduraler Abbildungsmuster nicht optimieren, da der Berechnungsweg fest vorgegeben wird. Der resultierende SQL-Code ist umfangreich. Prozedurale Abbildungsmuster sind nicht in der Lage, ihr Ergebnis direkt zurückzugeben. Hierfür sind Hilfsvariablen oder -tabellen nötig. Exakte, formale Abbildungsmuster, welche die Codegenerierung mit allen Details beschreiben, konnten deshalb mit vertretbarem Aufwand nicht erstellt werden.

Glücklicherweise sind jedoch nur die Operatoren `OCLH-INTRVAL` und `OCL-ITERATE` von diesem Qualitätsmangel betroffen.
Neben `OCLH-INTRVAL` sind jedoch auch die anderen Abbildungsmuster für Sequenzen mit Problemen behaftet. Inwieweit die „Collections“ von SQL3, welche in Oracle8™ bereits implementiert sind, eine direktere Entsprechung für OCL-Sequenzen darstellen und somit eine problemlosere Abbildung ermöglichen, wurde nicht untersucht. Es ist jedoch zu vermuten, daß die Probleme mit OCL-Sequenzen durch SQL-Collections prinzipiell gelöst werden können.
Die größten Probleme bei der Abbildung von OCL auf SQL bringt der Operator `OCL-ITERATE` mit sich. Dieser Operator ermöglicht es, komplexe Berechnungsvorschriften zu formulieren, die mit der SELECT-Anweisung von SQL nicht mehr realisierbar sind. Durch den Operator `OCL-ITERATE` ist es in der OCL möglich, reine Ausdrücke für Algorithmen zu formulieren, welche sich in SQL und Programmiersprachen nur durch Kontrollflußanweisungen (für Schleifen) beschreiben lassen.
Solch ein Iterate-Operator könnte aber auch in einer SQL-ähnlichen Syntax über Relationen formuliert werden:

```
SELECT ITERATIVE [--Typbeschreibung der Ergebnismenge--]
COMPUTE [--Funktion von Kollektionselement und Ergebnismenge--]
FROM [--Quellkollektion--]
INITIALISE WITH [--initialer Wert der Ergebnismenge--]
```

Wie im Kapitel 2 gezeigt wurde, können für Ausdrücke dieser Art Äquivalenzbedingungen formuliert werden, womit eine Optimierung durch das Datenbanksystem möglich wäre.
Wie für die OCL wäre die Existenz eines Iterations-Ausdrucks auch für SQL eine große Bereicherung.

Ein weiterer Qualitätsmangel ist die mehrfache Verwendung eines Parameters in einem Abbildungsmuster. Bei der Abbildung vollständiger OCL-Ausdrücke auf SQL werden die Abbildungsergebnisse der Teilausdrücke dann mehrfach als SQL-Code eingefügt. Es ergeben sich meist große redundante SQL-Code-Teile. Auch wenn diese Redundanz u.U. bei der Optimierung oder Ausführung der Anfragen eliminiert werden kann, vermittelt der automatisch generierte SQL-Code nicht viel Ästhetik.
Für die Kollektionstypen Set und Bag sind nur `OCL-SYMDIFF` und `OCL-CMPCOLL` (also Symmetische Differenz und Vergleich von Kollektionen) davon betroffen. Eine explizite Realisierung dieser Operatoren in SQL würde das Problem lösen, da keine Nachbildung mit anderen Operatoren mehr nötig wäre.
Wie bereits angedeutet, betrifft dieser Qualitätsmangel vor allem Operatoren über geordneten Multimengen (Verdichten von Sequenzen, `OCL-CONSEQ`, `OCL-LAST`, `OCL-SUBSEQ`, `OCL-INC` für Sequenzen und `OCL-UNION` für Sequenzen). Eine Verwendung von SQL-Collections könnte hier eventuell Abhilfe schaffen.

8 Abbildung der Operatoren der Gruppe 5 auf SQL

In diesem Kapitel wird zunächst ein veränderter Ausschnitt aus dem UML-Metamodell vorgestellt, welcher auf die für die statische Struktur relevanten Konstrukte reduziert und für die Code-Generierung optimiert ist. Auf der Basis dieses Meta-Modells wird anschließend die Generierung der SQL-DDL beschrieben. Daß die beschriebene Generierung nicht etwa eine willkürlich ausgewählte Variante ist, wird nachgewiesen, indem andere Arten der Abbildung als Performance-verbessernde Modifikationen der beschriebenen Abbildung dargestellt werden.

8.1 Meta-Modell der statischen Strukturbeschreibung

Obwohl in [UML-SEMA] bereits ein graphisches Meta-Modell der UML definiert wird, soll an dieser Stelle ein abweichendes Meta-Modell angegeben werden. Die wichtigsten Gründe sind folgende:

- Für dieses Kapitel dieser Diplomarbeit ist nur ein kleiner Ausschnitt aus dem UML-Metamodell relevant. Dieser Ausschnitt muß klar eingegrenzt und deshalb an dieser Stelle nochmals dargestellt werden. Es wurden nur die für die Datenstrukturen relevanten UML-Konstrukte berücksichtigt. Elemente, welche sich auf das Verhalten beziehen, wurden entfernt.
- Die Bezeichnungen der Meta-Modell-Elemente wurden ins Deutsche übertragen.

Des weiteren waren folgende spezielle Änderungen notwendig:

- Die Datentypen der UML-Definition, der OCL-Definition und des SQL-Standards sind unterschiedlich, insbesondere definiert die OCL nur einen Teil der Datentypen der UML mit der notwendigen Exaktheit. Die berücksichtigten Datentypen beschränken sich demzufolge auf die im Dokument zur OCL exakt definierten. Dies sind Real, Integer, String, Boolean und Aufzählungstypen.
- Binäre und n-äre Assoziationen wurden getrennt modelliert. Die Besonderheiten der Qualifikation, Aggregration und Ordnung, welche zwar für die binäre, nicht aber für die n-äre Assoziation definiert sind, sind in diesem Meta-Modell im Gegensatz zum Meta-Modell der UML eindeutig (d.h. direkt im Modell ersichtlich) nur der binären Assoziation zugeordnet. Die Unterschiede zwischen der binären und der n-ären Assoziation sind derart groß, insbesondere auch im Hinblick auf die graphische Darstellung und die SQL-Generierung, daß die explizite Benennung zweier Subklassen der allgemeinen Assoziationen im Meta-Modell sinnvoll ist.
- Das Meta-Modell-Element Assoziation als Klasse wurde nicht wie beim Meta-Modell der UML durch multiple Vererbung sondern durch eine (Meta-)Assoziation zwischen der Meta-Klasse Assoziation und der Meta-Klasse Klasse modelliert. Dies wird durch die getrennte Modellierung von binären und n-ären Assoziationen nötig.
- Während die Generalisierung im Meta-Modell der UML durch eine eigenständige Meta-Klasse modelliert wird, kann sie bei Beschränkung auf die strukturell relevanten Konzepte durch eine Assoziation modelliert werden (da Interface als Verhaltens-Element entfällt und Datentypen eine weniger umfassende Definition erfahren).
- Während im UML-Standard Multiplizitäten als Menge von disjunkten Ganzzahl-Intervallen beschrieben werden, wird in dieser Arbeit nur von einem Intervall ausgegangen. Der seltene Sonderfall von mehr als einem Intervall kann im Bedarfsfall durch OCL-Constraints modelliert werden und rechtfertigt somit eine Berücksichtigung bei den strukturellen Aspekten nicht.
- Statische Attribute und parametrisierbare Klassen sollen als Implementations-Spezifika aufgefaßt werden, welche zwar z.B. bei der Implementation des Zugriffs auf ein RDBMS von C++ aus unbedingt eingesetzt werden sollten, jedoch keine Konzepte des Datenmodells einer Anwendung sind (bzw. sein sollten).
- Attribute können nur einen primitiven Datentyp haben. Attribute vom Typ einer Klasse sind nicht zulässig. Dieser Sachverhalt muß als Aggregation modelliert werden (wie bei OMT).

Im folgenden sind die Meta-Modell-Diagramme abgebildet und ihr Inhalt verbal wiedergegeben.

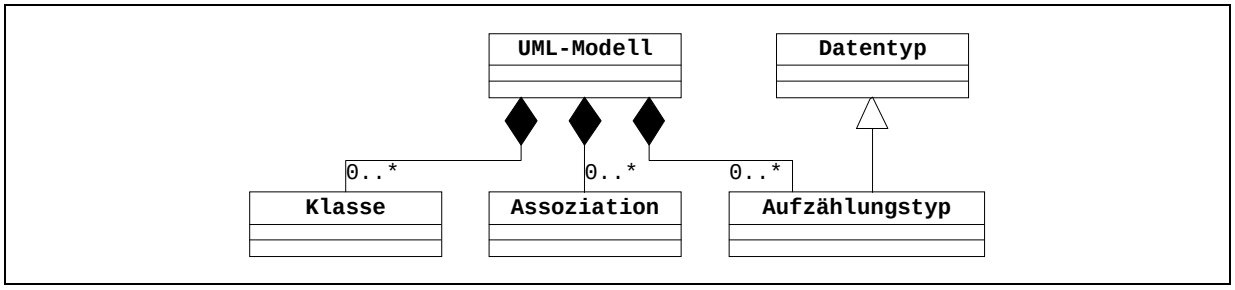


Abbildung 15: Meta-Modell-Diagramm Überblick

Ein UML-Modell besteht aus Klassen, Assoziationen und Aufzählungstypen. Alle Datentypen außer den Aufzählungstypen werden für alle UML-Modelle gemeinsam definiert.

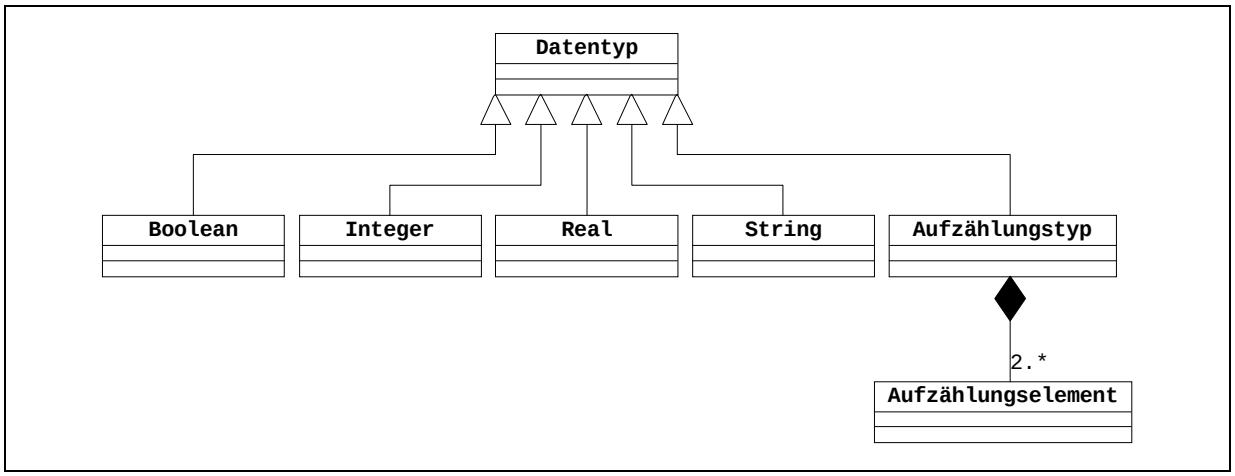


Abbildung 16: Meta-Modell-Diagramm Datentypen

Die allgemeinen primitiven Datentypen sind Real, Integer, String und Boolean. Aufzählungstypen besitzen mindestens zwei Aufzählungselemente.

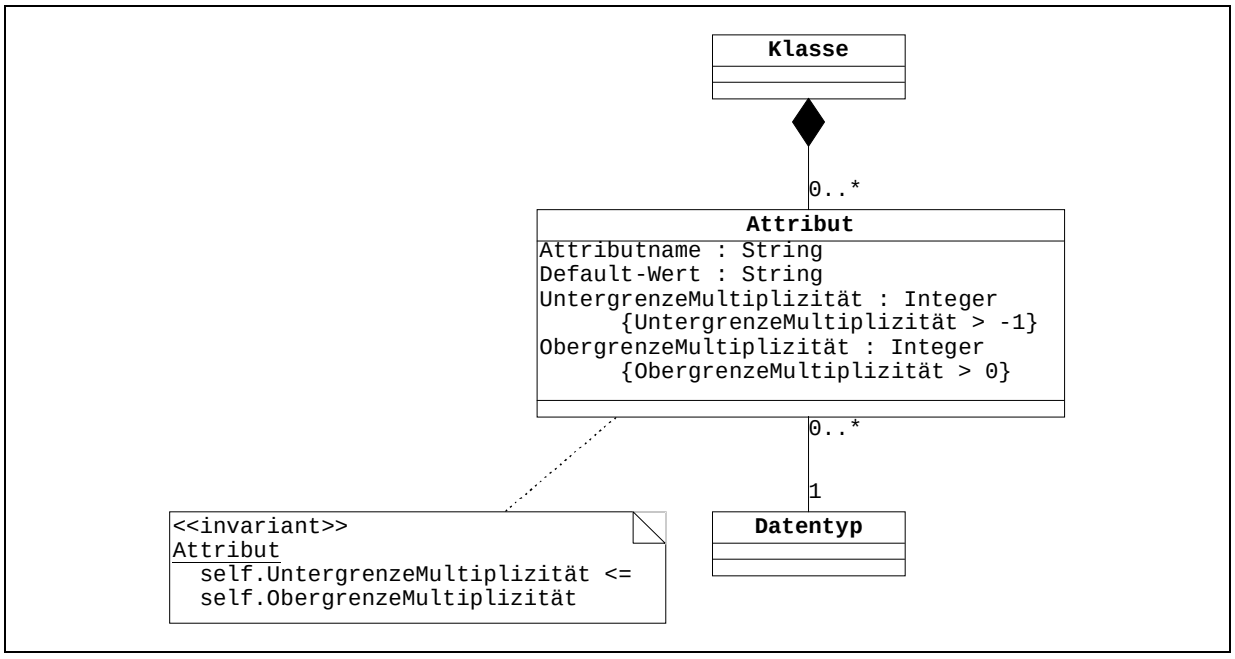


Abbildung 17: Meta-Modell-Diagramm Attribute

Ein Attribut ist genau einer Klasse zugeordnet, eine Klasse kann beliebig viele Attribute besitzen. Ein Attribut hat einen Namen, einen Defaultwert und ist genau einem Datentyp zugeordnet. Ein Attribut hat eine Multiplizität welche durch Unter- und Obergrenze spezifiziert ist. Die Untergrenze muß mindestens null und die Obergrenze mindestens eins sein. Die Untergrenze kann nicht größer als die Obergrenze sein.

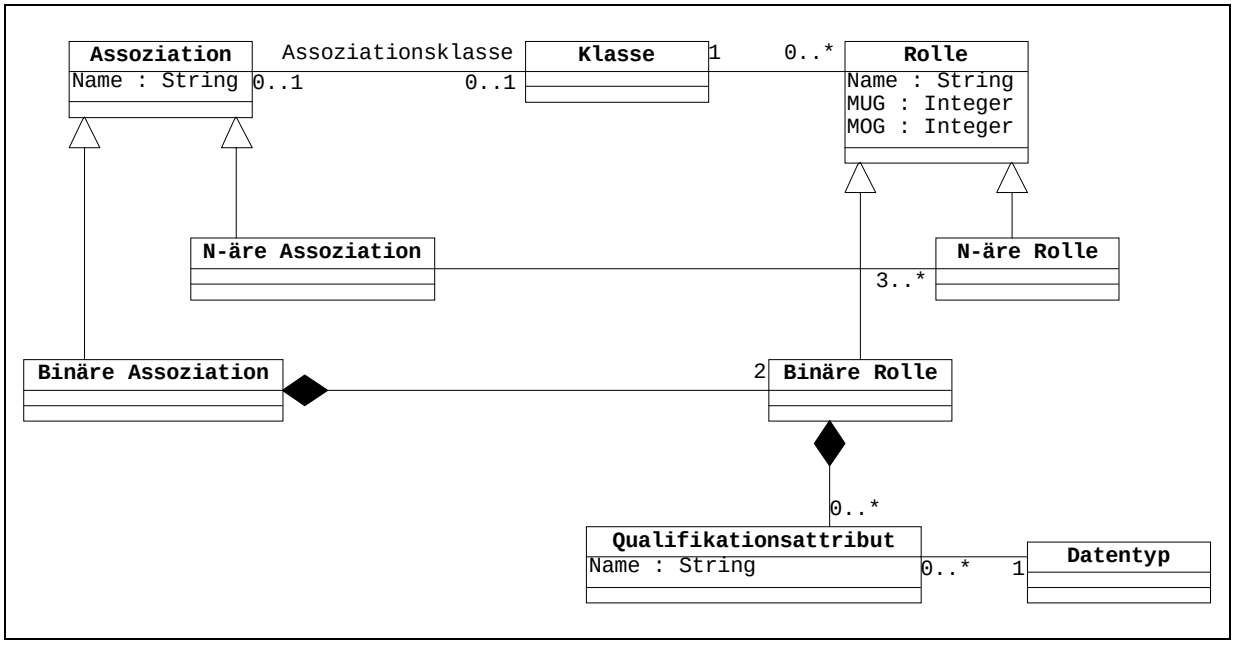


Abbildung 18: Meta-Modell-Diagramm: Assoziationen

Eine Assoziation kann im Rahmen des Konstrukts „Assoziation als Klasse“ durch eine Klasse, die Assoziationsklasse, als Entität repräsentiert werden. Eine n-äre Assoziation besitzt drei oder mehr Rollen, eine binäre Assoziation genau zwei. Eine Rolle hat einen Namen und eine Multiplizität (bestehend aus Unter- und Obergrenze) und ist genau einer Klasse zugeordnet. Eine Klasse kann an beliebig vielen Assoziationen über Rollen teilnehmen. Eine Rolle, welche einer binären Assoziation zugeordnet ist (Binäre Rolle) kann gegenüber der allgemeinen Rolle die Eigenschaft einer Ordnung und einer Qualifikation besitzen. Komposition und Aggregation sind Spezialfälle der binären Assoziation welche in diesem Meta-Modell nicht tiefgreifender berücksichtigt werden.

Im Gegensatz zu den binären Assoziationen ist die Qualifikation für n-äre Assoziationen nicht definiert (gilt auch für Komposition und Aggregation). Wie bei der Generierung von Code aus der binären Assoziation hervorgeht, münden sowohl Qualifikation als auch Ordnung (`{ordered}`) in Qualifikationsattributen bzw. Reihenfolgenattributen, welche einer Rolle zugeordnet sind. Die Probleme, welche sich bei der Qualifikation ergeben, und die OO-Autoren dazu veranlassen, Qualifikation bei n-ären Assoziationen nicht zu beachten, ergeben sich auch für die Ordnungen. Diese werden aus diesem Grunde für n-äre Assoziationen nicht betrachtet.

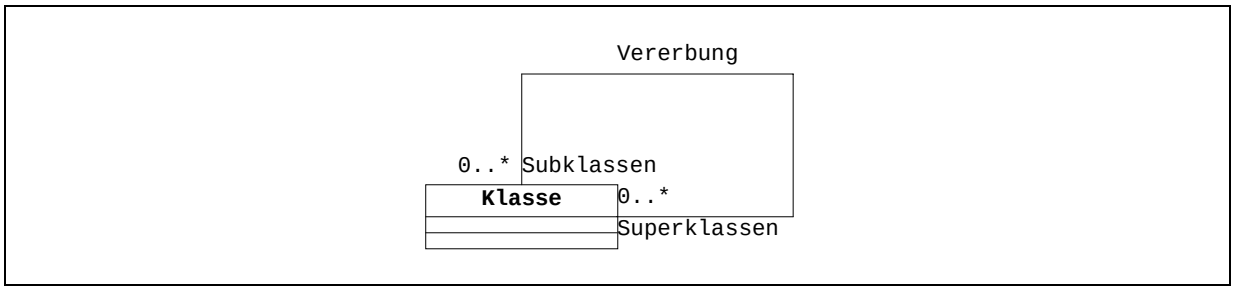


Abbildung 19: Meta-Modell-Diagramm Vererbung

Eine Klasse kann beliebig viele Sub- und Superklassen besitzen.

8.2 Generierung der SQL-DDL

In diesem Abschnitt werden die für diese Arbeit ermittelten Abbildungsregeln ohne Alternativen beschrieben. Die Abbildung von OCL nach SQL setzt die im folgenden beschriebene Abbildung von UML-Anwendungsmodellen nach SQL voraus.
Abbildungen für effizientere Datenbankschemata, welche mehr Eigenschaften des Datenmodells berücksichtigen und Vergleiche mit anderen Abbildungen werden im Abschnitt 8.4 behandelt.
Neben der Definition der reinen Datenstrukturen werden auch Integritätsregeln definiert. Sie stellen sicher, daß die über viele Relationen verteilten Daten der einzelnen Objekte korrekte Instanzen der Modellklassen darstellen. Außerdem definieren sie formal, wie die Speicherung objektorientierter Daten in der relationalen Datenbank nach dem beschriebenen Verfahren zu erfolgen hat und dienen somit dem Verständnis des Verfahrens in dieser Beschreibung.
Als Zielsprache wurde Transact-SQL des RDBMS Sybase™ gewählt, um die Ergebnisse der Abbildung praktisch testen zu können. Transact-SQL ist in den wesentlichen Punkten zu Standard-SQL konform. Auf Standard-SQL (SQL-92, SQL3) wird nur dann eingegangen, wenn im Vergleich zu Transact-SQL deutlich bessere Möglichkeiten der Abbildung bestehen. Dies gilt auch für die anderen Kapitel dieser Arbeit.

8.2.1 Generierung der SQL-DDL für die Meta-Daten aus dem Anwendungsmodell

Die Meta-Daten des Anwendungsmodells werden in den folgenden Abschnitten indirekt auf SQL abgebildet und sind somit als Schema-Information in der Datenbank enthalten. Für viele praktische Anwendungsfälle werden die Meta-Daten auch nicht direkt benötigt.
Die Meta-Daten sollen dennoch in der Datenbank direkt in Tabellen abgelegt werden, da die Beschreibung eines Teils der folgenden Abbildungsmuster (vor allem für OCL-Operatoren) dadurch mit weniger Aufwand exakt und verständlich vorgenommen werden kann. Anstelle des Zugriffs auf die Tabellen könnten die Meta-Daten auch von einem Werkzeug in den zu generierenden SQL-Code mit eingebaut werden. Dies ist jedoch formal schwieriger zu beschreiben und verbale Konstruktionen könnten zu Fehlinterpretationen führen. Für eine kommerzielle Realisierung eines Werkzeugs für die Abbildung UML-Klassendiagramm mit OCL auf SQL würde dieses Vorgehen jedoch Performance-Gewinne ermöglichen.

Die folgenden Tabellen müssen angelegt werden, um die Metainformationen über die Klassen und ihre Generalisierungsbeziehungen in der Datenbank ablegen zu können. Die generierte Tabellenstruktur ist dabei für jedes Anwendungsmodell gleich. Den Namen wird zum Vermeiden von Namenskonflikten die Vorsilbe „Meta“ vorangestellt.

```
CREATE TABLE MetaAnwendungsmodellKlassen
(
  name VARCHAR(64) NOT NULL PRIMARY KEY,
  abstrakt BIT NOT NULL
)

CREATE TABLE MetaGeneralisierungen
(
  Elter VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  Kind VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  Schritte INTEGER NOT NULL
)

CREATE TABLE MetaAttribute
(
  klasse VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  name VARCHAR(64) NOT NULL
)

CREATE TABLE MetaAssoziationsenden
(
  klasse VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  name VARCHAR(64) NOT NULL
)
```

```
CREATE TABLE MetaOperationen
(
  klasse VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  name VARCHAR(64) NOT NULL
)
```

Für jedes Anwendungsmodell müssen die Metadaten in die vorbereiteten Tabellen eingetragen werden. Dazu wird für jede Klasse der Name der Klasse und eine Information darüber, ob die Klasse abstrakt ist, in der Datenbank abgelegt (zweites Attribut 1 für abstrakte Klassen):

```
INSERT INTO MetaAnwendungsmodellKlassen VALUES ('<Name der Klasse>', 0|1)
```

Außerdem müssen alle Klassen, von denen die betreffende Klasse erbt, zusammengetragen werden:

```
INSERT INTO MetaGeneralisierungen VALUES
                      ('<Elternteilname>', '<Klassenname>', <Schritte?>)
```

Die reflexiven Eltern (Klasse selbst - nur zur Erleichterung von Anfragen) und die transitiven Eltern (Vererbung nur über einen oder mehrere transitive Schritte) werden mit aufgenommen. Für `<Schritte?>` wird die Anzahl der Generalisierungsbeziehungen eingetragen, über welche die Elternklasse erreicht wird. Für die Klasse selbst (reflexiv) ergibt sich die Schrittzahl 0. Für die im Diagramm eingetragenen direkten Beziehungen ergibt sich die Schrittzahl 1. Alle anderen Vererbungsbeziehungen haben eine größere Schrittzahl.

Die Namen von Attributen, Assoziationsenden und Operationen müssen ebenfalls in die entsprechenden Tabellen eingetragen werden.

Ein Beispiel für die beschriebene Generierung kann im Anhang A eingesehen werden.

8.2.2 Generierung der SQL-DDL für ein UML-Modell

Für ein UML-Modell, d.h. für alle Klassen gemeinsam, wird eine Relation folgender Form generiert:

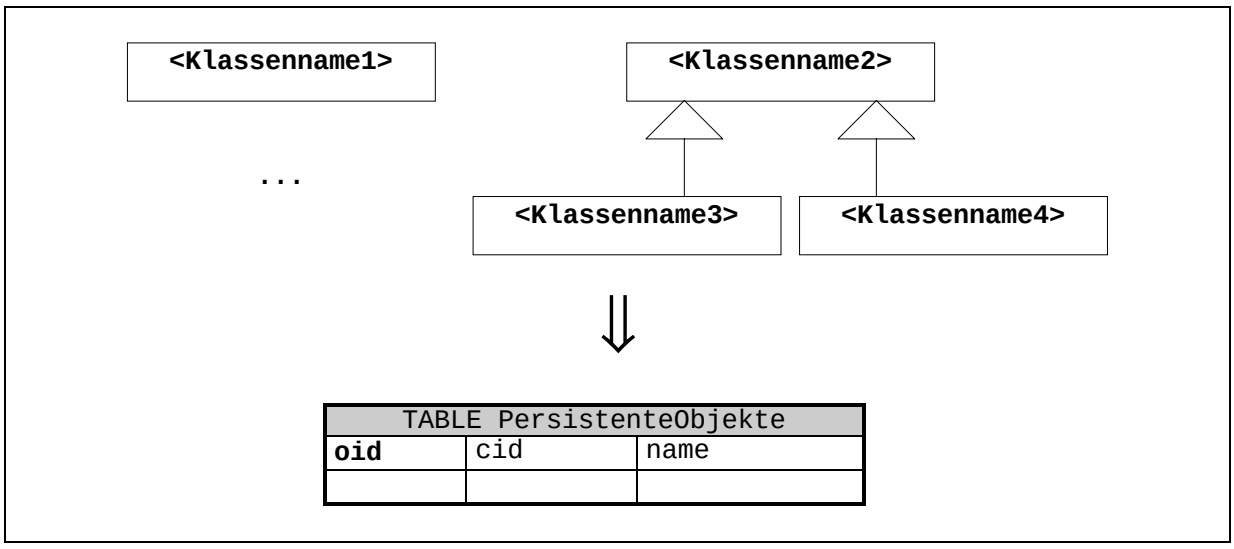


Abbildung 20: Generierung der Instanzentabelle für ein UML-Modell

```
CREATE TABLE PersistenteObjekte
(
  oid INTEGER NOT NULL PRIMARY KEY,   -- Objektidentifikator
  klasse VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  name VARCHAR(64) NULL               -- optionaler Objekt-Name
)
```

Jedes Objekt wird in dieser Relation durch ein Tupel aus einem eindeutigen Objektidentifikator (oid), dem Klassennamen (klasse), welcher die Klassenzugehörigkeit des Objekts spezifiziert und einem optionalen Namen charakterisiert (name). Im Vergleich zum ODMG-Standard ([ODMG-2.0]) kann einem Objekt nur ein Name zugeordnet werden, welcher aber nicht als eindeutiger Einstieg in die Datenbank verwendet werden kann (da hier mehrfache Verwendung eines Namens für unterschiedliche Objekte möglich ist).

Die nächste freie OID kann z.B. über die Anfrage

```
SELECT MAX(oid)+1
FROM PersistenteObjekte
```

ermittelt werden. Dies ist die einfachste Form (aber ausreichend). Da diese bei der Datenmanipulation sehr häufig ist, sollte eine Datenbank-Prozedur dafür definiert werden:

```
CREATE PROCEDURE ErmittleFreieOID (@neueOid INTEGER OUTPUT)
  AS
    IF ((SELECT COUNT(*) FROM PersistenteObjekte) < 1)
    BEGIN
      SELECT @neueOid = 1
    END
    ELSE
    BEGIN
      SELECT @neueOid = MAX(oid)+1
      FROM PersistenteObjekte
    END
```

Eine Änderung des Objektidentifikators und der Klassenzugehörigkeit ist nach dem SQL-Standard für die oben dargestellte Tabelle möglich. Unter Berücksichtigung der im folgenden definierten Constraints für die anderen Konstrukte könnte auch die Integrität der objektorientierten Daten sichergestellt werden (Korrektur der Fremdschlüsselbezüge bei Änderung des Objektidentifikators und korrekte Umwandlung der Objektstruktur bei Änderung der Klassenzugehörigkeit). Wie in [ODMG-2.0] beschrieben, sind diese Operationen für die objektorientierte Welt jedoch nicht zulässig (bzw. wünschenswert). Sie werden demzufolge verboten:

```
CREATE TRIGGER KonstanteOIDuKlasse
  ON PersistenteObjekte
  FOR UPDATE AS
  IF UPDATE(oid)
  BEGIN
    ROLLBACK TRANSACTION
    RAISERROR 17011
        'Fehler: Aenderung des Objektidentifikators (OID) ist unzulaessig!'
  END
  IF UPDATE(klasse)
  BEGIN
    ROLLBACK TRANSACTION
    RAISERROR 17012
        'Fehler: Aenderung der Klassenzugehoerigkeit ist unzulaessig!'
  END
```

Außerdem muß sichergestellt werden, daß keine Instanzen für abstrakte Klassen in der Datenbank verzeichnet werden können:

```
CREATE TRIGGER AbstrakteKlassenOhneInstanzen
  ON PersistenteObjekte
  FOR INSERT AS
  IF EXISTS (SELECT oid
             FROM INSERTED i, MetaAnwendungsmodellKlassen m
             WHERE i.klasse = m.name AND m.abstrakt = 1)
  BEGIN
    ROLLBACK TRANSACTION
    RAISERROR 17014
        'Fehler: Instanzen für abstrakte Klassen unzulaessig!'
  END
```

8.2.3 Generierung von SQL-DDL für Datentypen

Welcher SQL-Code die Datentypen aus UML/OCL realisiert, kann aus der folgenden Tabelle entnommen werden:

UML/OCL-Typ	<SQL-DDL-TYP>
Real	REAL
Integer	INTEGER
String	VARCHAR(200)
Boolean	BIT
Aufzählungstyp nach Definition enum{value1, value2, ..., valueN}	VARCHAR(200) ergänzt um Integritätsregel check (Attributname in ('value1', 'value2', ..., 'valueN'))

Die Länge der Strings ist mit 200 willkürlich festgelegt. Längere Strings können beim RDBMS Sybase™ u.U. nicht in einen Index einbezogen werden (gesamter Index max. 256). Bestimmte Konsistenzbedingungen können dann evtl. nicht (performant) überprüft werden.
In Standard-SQL können Aufzählungstypen eleganter und effizienter mit dem Domain-Konzept realisiert werden.

8.2.4 Generierung von SQL-DDL für Attribute

Für jedes Attribut jeder Klasse wird eine eigene Tabelle generiert. Damit wird der allgemeinste wenn auch seltene Fall eines Attributs mit Mehrfachwerten umgesetzt („Sonderfälle“ im Abschnitt 8.4).

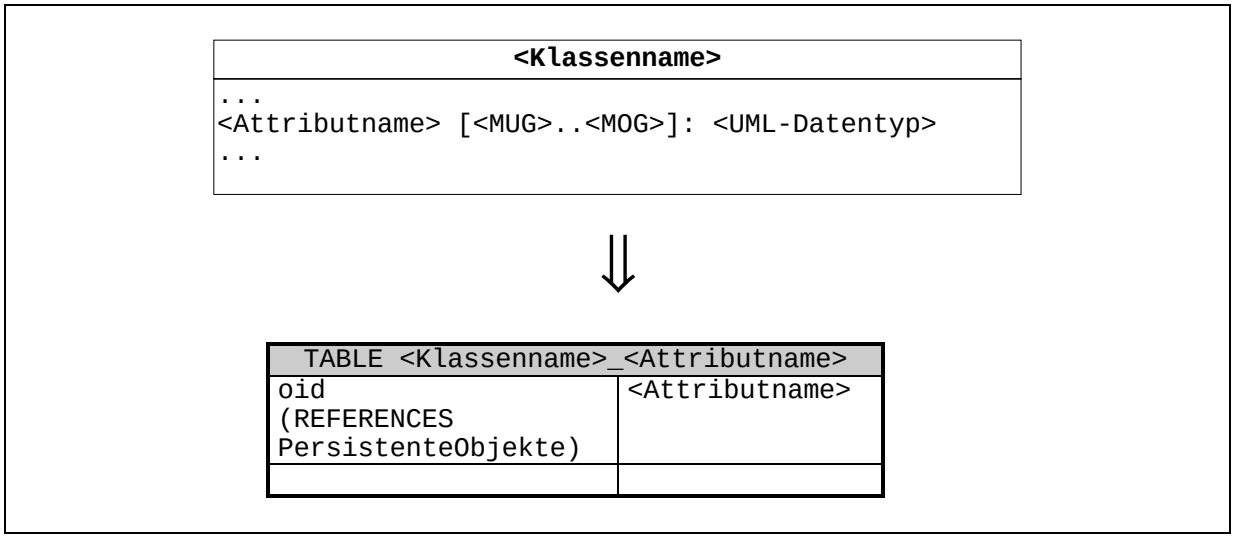


Abbildung 21: Abbildung von Attributen auf SQL

```
CREATE TABLE <Klassenname>_<Attributname>
(
  oid INTEGER NOT NULL REFERENCES PersistenteObjekte,
  <Attributname> <SQL-DDL-TYP> NOT NULL
)
```

Die Änderung der Objektzugehörigkeit eines Attributwertes („Verschieben“ von einer Objektinstanz zur andern) wäre zwar konsistent möglich, ist in der objektorientierten Welt jedoch nicht definiert und wird demzufolge durch den folgenden Trigger unterbunden:

```
CREATE TRIGGER KonstanteOID_<Klassenname>_<Attributname>
  ON <Klassenname>_<Attributname>
  FOR UPDATE AS
  IF UPDATE(oid)
  BEGIN
    ROLLBACK TRANSACTION
    RAISERROR 17021
        'Fehler: Aenderung der Objektzugehoerigkeit eines Attributs ist
         unzulaessig!'
  END
```

Default-Werte, wenn spezifiziert, können durch folgenden Trigger gesetzt werden:

```
CREATE TRIGGER SetzeDefaultWerte
  ON PersistenteObjekte
  FOR INSERT AS
    INSERT INTO <Klassenname>_<Attributname>
    SELECT oid, <Default-Wert>
    FROM inserted
    WHERE klasse IN (SELECT Kind
                     FROM MetaGeneralisierungen
                     WHERE Elter = '<Klassenname>')

    ...
```

Bei mehreren Attributen mit Default-Wert im Modell ist eine einzige CREATE-TRIGGER-Anweisung jeweils um die INSERT-INTO-Anweisungen für die einzelnen Default-Werte zu erweitern, da für eine Bedingung jeweils nur ein Trigger definiert werden darf.

Die bisher beschriebene Abbildung ermöglicht die Speicherung von Daten durch das RDBMS, welche in der objektorientierten Welt nicht sinnvoll sind. Beliebigen Objekten können somit beliebig viele Attributwerte zugeordnet werden. Es muß also zusätzlich sichergestellt werden, daß die oid nur ein Objekt einer Klasse referenziert, welches gemäß dem Objektmodell dieses Attribut auch besitzen kann (Instanz der definierenden Klasse und deren Subklassen (transitiv)). Des weiteren müssen die spezifizierten Multiplizitätsgrenzen eingehalten werden.

Es muß geprüft werden, ob die in der Attributtabelle verzeichneten Objekte dieses Attribut ihrer Klassenzugehörigkeit zufolge überhaupt besitzen dürfen. Nach dem SQL-Standard kann folgende Bedingung formuliert werden:

```
CREATE ASSERTION Test1_<Klassenname>_<Attributname>
  CHECK
    (NOT EXISTS
      (SELECT p.oid
       FROM <Klassenname>_<Attributname> a, PersistenteObjekte p
       WHERE a.oid=p.oid AND klasse NOT IN (SELECT Kind
                                            FROM MetaGeneralisierungen
                                            WHERE Elter = '<Klassenname>')
      )
    )
```

Es darf keine Objekte geben, welche nicht die geforderte Mindestanzahl (MUG) des Attributwertes besitzen. Ist die MUG gleich null, so ist die Formulierung dieser Bedingung überflüssig, da sie immer erfüllt wird.

```
CREATE ASSERTION Test1_<Klassenname>_<Attributname>
  CHECK
    (NOT EXISTS
       (SELECT p.oid
        FROM PersistenteObjekte p, <Klassenname>_<Attributname> a
        WHERE p.cid IN (SELECT Kind
                        FROM MetaGeneralisierungen
                        WHERE Elter = '<Klassenname>')
          AND p.oid *= a.oid
        GROUP BY p.oid
        HAVING COUNT(a.oid) < <MUG>
       )
     )

SET CONSTRAINT Test1_<Klassenname>_<Attributname> DEFERRED
```

Ebenso darf die Multiplizitätsobergrenze (MOG) nicht überschritten werden. Für * könnte die größte mögliche Zahl eingesetzt werden, welche COUNT() zurückgeben kann. Sinnvoller ist es natürlich auch in diesem Fall die Bedingung nicht zu formulieren.

```
CREATE ASSERTION Test1_<Klassenname>_<Attributname>
  CHECK
    (NOT EXISTS
       (SELECT oid
        FROM <Klassenname>_<Attributname>
        GROUP BY oid
        HAVING COUNT(*) > <MOG>
       )
     )
```

Die vorangehend dargestellte Formulierung der Bedingungen ist zwar nach dem SQL-Standard möglich, daß Datenbanksystem Sybase™ unterstützt dies jedoch nicht.

Lösung mit Datenbank-Prozeduren

Um die vorangehend beschriebenen Bedingungen dennoch überprüfen zu können, wird im folgenden eine Datenbank-Prozedur definiert, welche die Bedingungen überprüft und bei Nichterfüllung Fehler meldet. Damit wird die Datenintegrität jedoch nicht von Server-Seite aus garantiert. Der Nutzer bzw. Anwendungsprogrammierer muß die Prüf-Prozeduren explizit aufrufen, um den Datenbestand zu prüfen und im Fehlerfalle selbst reagieren (bzw. Korrektur programmieren).

```
CREATE PROCEDURE Pruefe_<Klassenname>_<Attributname>
  AS
  BEGIN
    -- Attributwerte duerfen nur Objekten zugeordnet werden, welche dieses
    -- Attribut entsprechend ihrer Klassenzugehoerigkeit auch besitzen
    -- koennen.
    IF EXISTS
      (SELECT p.oid
       FROM <Klassenname>_<Attributname> a, PersistenteObjekte p
       WHERE a.oid=p.oid AND klasse NOT IN (SELECT Kind
                                           FROM MetaGeneralisierungen
                                           WHERE Elter = '<Klassenname>'))
    BEGIN
      RAISERROR 17022 'Attribute zu falschen Klassen zugeordnet!'
    END

    -- Keinem Objekt duerfen weniger Attributwerte zugeordnet werden, als
    -- die MUG spezifiziert.
    -- Die Ueberpruefung sollte entfallen wenn <MUG> = 0.
    IF EXISTS
      (SELECT p.oid
       FROM PersistenteObjekte p, <Klassenname>_<Attributname> a
       WHERE p.klasse IN (SELECT Kind
                          FROM MetaGeneralisierungen
                          WHERE Elter = '<Klassenname>')
         AND p.oid *= a.oid
       GROUP BY p.oid
       HAVING COUNT(a.oid) < <MUG>)
    BEGIN
      RAISERROR 17023 'Attribute verletzen MUG!'
    END

    -- Keinem Objekt duerfen mehr Attributwerte zugeordnet werden, als
    -- die MOG spezifiziert.
    -- Die Ueberpruefung sollte entfallen wenn <MOG> = *.
    IF EXISTS
      (SELECT oid
       FROM <Klassenname>_<Attributname>
       GROUP BY oid
       HAVING COUNT(*) > <MOG>)
    BEGIN
      RAISERROR 17024 'Attribute verletzen MOG!'
    END
END
```

Lösung mit Triggern

Eine Lösung, welche die Datenintegrität automatisch sicherstellt und fehlerhafte Dateneingaben zurückweist, könnte mit Triggern realisiert werden. Die im folgenden beschriebene Lösung sollte auch durch eine Optimierung der ASSERTIONs erreicht werden. Dafür müßten jedoch auch die anderen bereits als Trigger beschriebenen Integritätsbedingungen deskriptiv formuliert werden (bzw. fomulierbar sein), da die im folgenden beschrieben Trigger auf Grundlage dieser Vorgaben optimiert sind.

Da weder die Klassenzugehörigkeit eines Objekts noch die Objektzugehörigkeit eines Attributs geändert werden können (durch Trigger durchgesetzt), braucht die Korrektheit der Zuordnung eines Attributs zu einem Objekt in Bezug auf die Konsistenz zum Modell nur unmittelbar nach dem Einfügen überprüft werden.

```
CREATE TRIGGER Richtige_Klasse_<Klassenname>_<Attributname>
  ON <Klassenname>_<Attributname>
  FOR INSERT AS
    IF EXISTS
      (SELECT p.oid
       FROM inserted a, PersistenteObjekte p
       WHERE a.oid=p.oid AND klasse NOT IN (SELECT Kind
                                            FROM MetaGeneralisierungen
                                            WHERE Elter = '<Klassenname>'))
    BEGIN
      ROLLBACK TRANSACTION
      RAISERROR 17022 'Attribute zu falschen Klassen zugeordnet!'
    END
```

Die Multiplizitätsuntergrenze (MUG) kann nach dem Einfügen neuer Objekte verletzt werden. Demzufolge muß folgender Trigger die Konsistenz prüfen.

```
CREATE TRIGGER KorrekteMUG1_<Klassenname>_<Attributname>
  ON PersistenteObjekte
  FOR INSERT AS
    IF EXISTS
      (SELECT p.oid
       FROM inserted p, <Klassenname>_<Attributname> a
       WHERE p.klasse IN (SELECT Kind
                          FROM MetaGeneralisierungen
                          WHERE Elter = '<Klassenname>')
         AND p.oid *= a.oid
       GROUP BY p.oid
       HAVING COUNT(a.oid) < <MUG>)
    BEGIN
      ROLLBACK TRANSACTION
      RAISERROR 17023 'Attribute verletzen MUG!'
    END
```

Die Ausführung dieses Triggers müßte an das Ende einer Transaktion verschoben werden, da unmittelbar nach der Einfüge-Operation für das Objekt bei Vorhandensein einer MUG>0 die Bedingung nie erfüllt sein kann (bei Sybase™ 11 nicht festgestellt - funktioniert also nicht).

Da die Änderung der Objektzugehörigkeit eines Attributs nicht möglich ist, können weitere Konsistenzverletzungen nur noch beim Löschen von Attributen auftreten, welche durch den folgenden Trigger abgefangen werden:

```
CREATE TRIGGER KorrekteMUG2_<Klassenname>_<Attributname>
  ON <Klassenname>_<Attributname>
  FOR DELETE AS
    IF EXISTS
      (SELECT p.oid
       FROM PersistenteObjekte p, <Klassenname>_<Attributname> a,
                    deleted d
       WHERE p.klasse IN (SELECT Kind
                          FROM MetaGeneralisierungen
                          WHERE Elter = '<Klassenname>')
         AND p.oid = d.oid
         AND p.oid *= a.oid
       GROUP BY p.oid
       HAVING COUNT(a.oid) < <MUG>)
    BEGIN
      ROLLBACK TRANSACTION
      RAISERROR 17023 'Attribute verletzen MUG!'
    END
```

Auch dieser Trigger müßte an das Ende einer Transaktion verschiebbar sein.

Die Multiplizitätsobergrenze kann beim Einfügen von Attributen überschritten werden, weshalb dies durch den folgenden Trigger unterbunden werden muß:

```
CREATE TRIGGER KorrekteMOG_<Klassenname>_<Attributname>
  ON <Klassenname>_<Attributname>
  FOR INSERT AS
    IF EXISTS
      (SELECT oid
       FROM inserted
       GROUP BY oid
       HAVING COUNT(*) > <MOG>)
    BEGIN
      ROLLBACK TRANSACTION
      RAISERROR 17024 'Attribute verletzen MOG!'
    END
```

Da nur die Lösung mit Datenbank-Prozeduren mit dem RDBMS Sybase™ korrekt umgesetzt werden kann, konnte auch nur diese Lösung mit ausreichender Sicherheit überprüft werden. Aus diesem Grund werden im folgenden nur noch Konsistenzprüfungen mit Datenbank-Prozeduren vorgestellt.

8.2.5 Generierung von SQL-DDL für binäre Assoziationen

Ein Link kann als Tupel, welches die beteiligten Instanzen nennt, aufgefaßt werden. Eine Assoziation ist somit per Definition (nach Rumbaugh) eine Relation im eigentlichen Sinne (Menge von Tupeln).

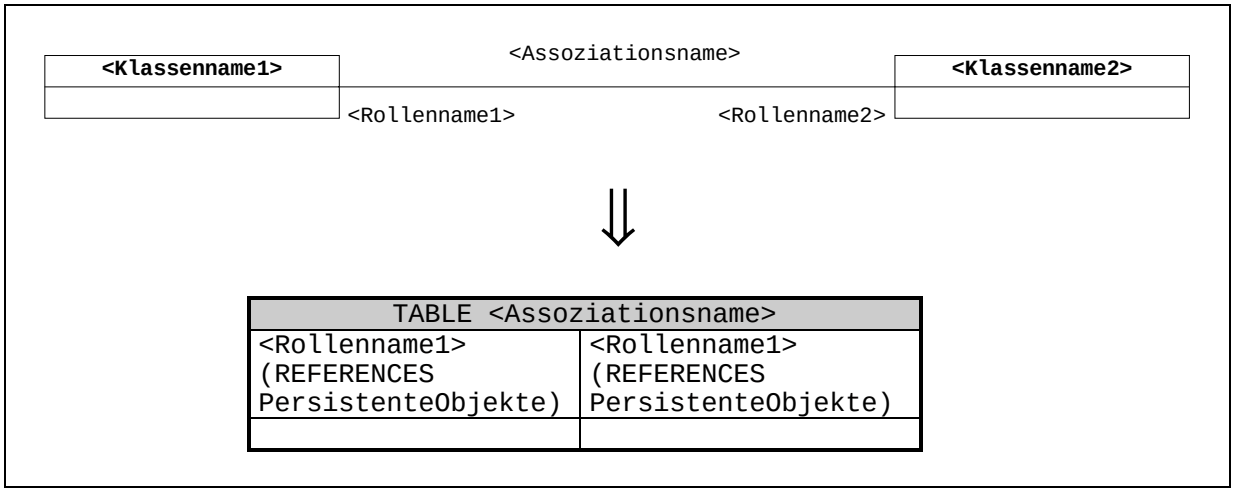


Abbildung 22: Abbildung von binären Assoziationen auf SQL

```
CREATE TABLE <Assoziationsname>
(
  <Rollenname1> INTEGER NOT NULL REFERENCES PersistenteObjekte,
  <Rollenname2> INTEGER NOT NULL REFERENCES PersistenteObjekte,
  UNIQUE(<Rollenname1>, <Rollenname2>)
)
```

Für jede der beiden Rollen müssen die Eigenschaften sichergestellt werden, welche bereits für Attribute beschrieben wurden (korrekte Klassenzugehörigkeit und Multiplizität). Für die Assoziation ist also folgende Datenbank-Prozedur zu generieren:

```
CREATE PROCEDURE Pruefe_<Assoziationsname>
  AS
  BEGIN
    EXECUTE Pruefe_<Assoziationsname>_<Rollenname1>
    EXECUTE Pruefe_<Assoziationsname>_<Rollenname2>
  END
```

Diese ruft die Datenbank-Prozeduren auf, welche die Korrektheit der jeweiligen Rollen sicherstellen.

```
CREATE PROCEDURE Pruefe_<Assoziationsname>_<Rollenname1>
  AS
  BEGIN
    -- Die in der Assoziationstabelle referenzierten Objekte muessen der
    -- Klasse, fuer welche die Rolle definiert ist oder einer ihrer
    -- Kindklassen angehoeren.
    IF EXISTS
      (SELECT oid
       FROM <Assoziationsname> a, PersistenteObjekte p
       WHERE a.<Rollenname1>=p.oid
         AND klasse NOT IN (SELECT Kind
                            FROM MetaGeneralisierungen
                            WHERE Elter = '<Klassenname1>'))
    BEGIN
      RAISERROR 17003 'Rollen zu falschen Klassen zugeordnet!'
    END

    -- Keinem Objekt duerfen weniger Objekte der Zielrolle zugeordnet
    -- werden, als die MUG spezifiziert.
    -- Die Ueberpruefung sollte entfallen wenn <MUG> = 0.
    IF EXISTS
      (SELECT p.oid
       FROM PersistenteObjekte p, <Assoziationsname> a
       WHERE p.klasse IN (SELECT Kind
                          FROM MetaGeneralisierungen
                          WHERE Elter = '<Klassenname2>')
         AND p.oid *= a.<Rollenname2>
       GROUP BY p.oid
       HAVING COUNT(a.<Rollenname1>) < <MUG>)
    BEGIN
      RAISERROR 17004 'Rollen verletzen MUG!'
    END

    -- Keinem Objekt duerfen mehr Objekte der Zielrolle zugeordnet
    -- werden, als die MOG spezifiziert.
    -- Die Ueberpruefung sollte entfallen wenn <MUG> = *.
    IF EXISTS
      (SELECT <Rollenname2>
       FROM <Assoziationsname>
       GROUP BY <Rollenname2>
       HAVING COUNT(*) > <MOG>)
    BEGIN
      RAISERROR 17005 'Rollen verletzen MOG!'
    END
END
```

8.2.6 Generierung von SQL-DDL für n-äre Assoziationen

Für n-äre Assoziationen wird ähnlicher Code wie für binäre Assoziationen generiert. Die Anzahl der Rollen ist hierbei jedoch von drei an nach oben offen.

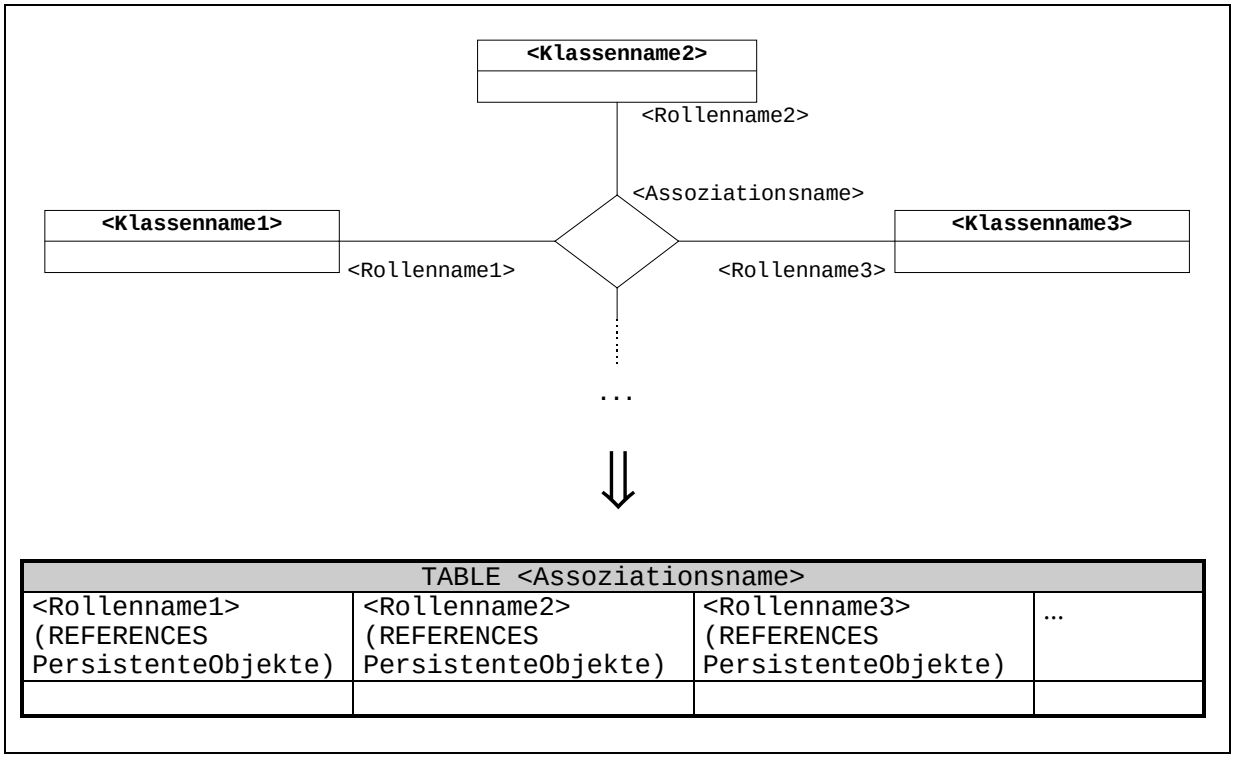


TABLE <Assoziationsname>			
<Rollenname1> (REFERENCES PersistenteObjekte)	<Rollenname2> (REFERENCES PersistenteObjekte)	<Rollenname3> (REFERENCES PersistenteObjekte)	...

Abbildung 23: Abbildung von n-ären Assoziationen auf SQL

```
CREATE TABLE <Assoziationsname>
(
  <Rollenname1> INTEGER NOT NULL REFERENCES PersistenteObjekte,
  <Rollenname2> INTEGER NOT NULL REFERENCES PersistenteObjekte,
  <Rollenname3> INTEGER NOT NULL REFERENCES PersistenteObjekte,
  ...
  UNIQUE(<Rollenname1>, <Rollenname2>, <Rollenname3>, ...)
)

CREATE PROCEDURE Pruefe_<Assoziationsname>
  AS
  BEGIN
    EXECUTE Pruefe_<Assoziationsname>_<Rollenname1>
    EXECUTE Pruefe_<Assoziationsname>_<Rollenname2>
    EXECUTE Pruefe_<Assoziationsname>_<Rollenname3>
    ...
END
```

Während sich fast alle Erweiterungen analog zur binären Assoziation vornehmen lassen, ist bei der Überprüfung der Multiplizitäten folgendes zu beachten: Die Multiplizität wird gegen das (N-1)-Tupel geprüft.

```
CREATE PROCEDURE Pruefe_<Assoziationsname>_<Rollenname1>
  AS
  BEGIN
    -- Die in der Assoziationstabelle referenzierten Objekte muessen der
    -- Klasse, fuer welche die Rolle definiert ist oder einer ihrer
    -- Kindklassen angehoeren.
    IF EXISTS
      (SELECT p.oid
       FROM <Assoziationsname> a, PersistenteObjekte p
       WHERE a.<Rollenname1>=p.oid
         AND klasse NOT IN (SELECT Kind
                            FROM MetaGeneralisierungen
                            WHERE Elter = '<Klassenname1>'))
    BEGIN
      RAISERROR 17003 'Rollen zu falschen Klassen zugeordnet!'
    END

    -- Keinem Objekt duerfen weniger Objekte der Zielrolle zugeordnet
    -- werden, als die MUG spezifiziert.
    -- Die Ueberpruefung sollte entfallen wenn <MUG> = 0.
    IF EXISTS
      (SELECT p.oid
       FROM <Assoziationsname> a, PersistenteObjekte p1,
                                  PersistenteObjekte p2, ...
       WHERE p1.klasse IN (SELECT Kind
                           FROM MetaGeneralisierungen
                           WHERE Elter = '<Klassenname2>')
         AND p2.klasse IN (SELECT Kind
                           FROM MetaGeneralisierungen
                           WHERE Elter = '<Klassenname3>')
         AND p1.oid *= a.<Rollenname2>
         AND p2.oid *= a.<Rollenname3>
         AND ...
       GROUP BY p1.oid, p2.oid, ...
       HAVING COUNT(a.<Rollenname1>) < <MUG>)
    BEGIN
      RAISERROR 17004 'Rollen verletzen MUG!'
    END

    -- Keinem Objekt duerfen mehr Objekte der Zielrolle zugeordnet
    -- werden, als die MOG spezifiziert.
    -- Die Ueberpruefung sollte entfallen wenn <MUG> = *.
    IF EXISTS
      (SELECT COUNT(*)
       FROM <Assoziationsname>
       GROUP BY <Rollenname2>, <Rollenname3>, ...
       HAVING COUNT(*) > <MOG>)
    BEGIN
      RAISERROR 17005 'Rollen verletzen MOG!'
    END
END
```

8.2.7 Generierung von SQL-DDL für Assoziationen als Klasse

Einer Assoziation (binär oder n-är) kann eine Assoziationsklasse zugeordnet sein. In diesem Fall wird die zu generierende Tabelle der Assoziation um die Spalte Assoziationsklasse erweitert.

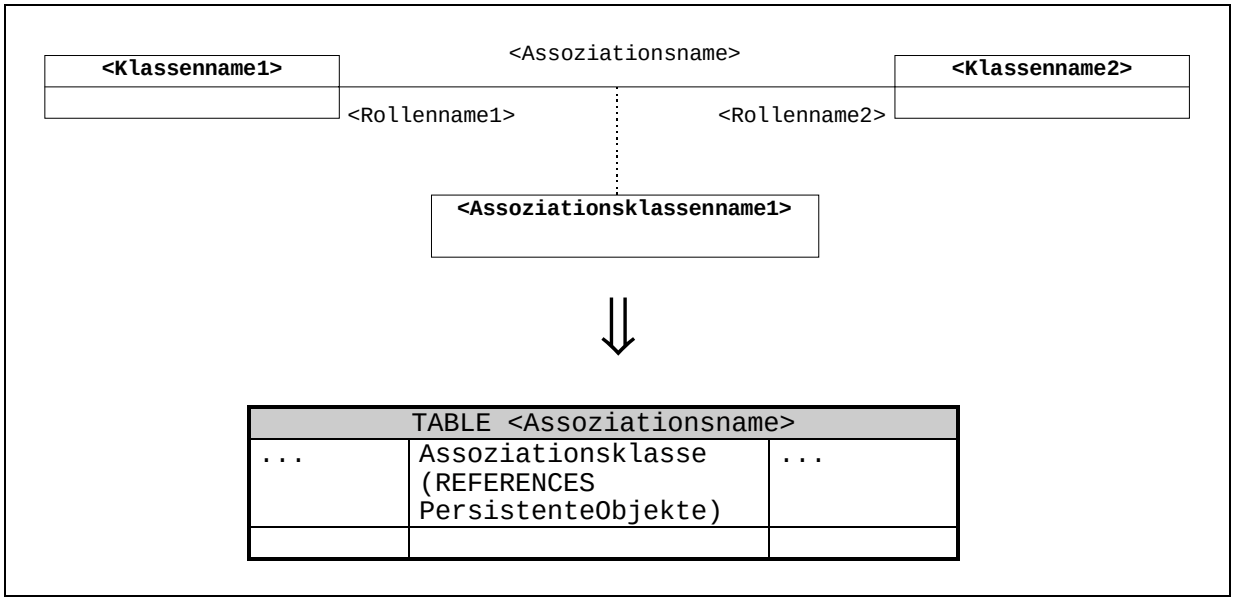


Abbildung 24: Abbildung von Assoziationsklassen auf SQL

```
CREATE TABLE <Assoziationsname>
(
  ...
  Assoziationsklasse INTEGER NOT NULL REFERENCES PersistenteObjekte,
  ...
)
```

Um zu überprüfen, ob das Assoziationsobjekt von korrekter Klassenzugehörigkeit ist und jedes Objekt der entsprechenden Klassen genau einem Link zugeordnet ist, wird die Prozedur Pruefe_<Assoziationsname> um folgende Anweisungen erweitert:

```
CREATE PROCEDURE Pruefe_<Assoziationsname>
  AS
  BEGIN

    ...

    -- Die in der Assoziationstabelle als Assoziationsobjekt referenzierten
    -- Objekte muessen der Klasse, welche als Assoziationsklasse definiert
    -- ist oder einer ihrer Kindklassen angehoeren.
    IF EXISTS
      (SELECT oid
       FROM <Assoziationsname> a, PersistenteObjekte p
       WHERE a.Assoziationsklasse=p.oid
         AND klasse NOT IN (SELECT Kind
                            FROM MetaGeneralisierungen
                            WHERE Elter = '<Assoziationsklassenname>')
    BEGIN
      RAISERROR 17003 'Assoziationen zu falschen Assoziationsklassen
                       zugeordnet!'
    END

    -- Links und Linkobjekte muessen eineindeutig zugeordnet sein.
    IF EXISTS
      (SELECT p.oid
       FROM PersistenteObjekte p, <Assoziationsname> a
       WHERE p.klasse IN (SELECT Kind
                          FROM MetaGeneralisierungen
                          WHERE Elter = '<Assoziationsklassenname>')
         AND p.oid *= a.Assoziationsklasse
       GROUP BY p.oid
       HAVING COUNT(a.Assoziationsklasse) <> 1)
    BEGIN
      RAISERROR 17004 'Zuordnung Assoziation-Klasse falsch!'
    END
  END
```

8.2.8 Generierung von SQL-DDL für qualifizierte Rollen

Qualifikatoren sind Zusatzinformationen zu einer Rolle. Diese Zusatzinformationen werden in der Tabelle der entsprechenden Assoziation gespeichert. Hierbei ist es möglich, daß der Qualifikator aus mehreren Qualifikationsattributen besteht. Dann werden entsprechend mehrere Spalten hinzugefügt.

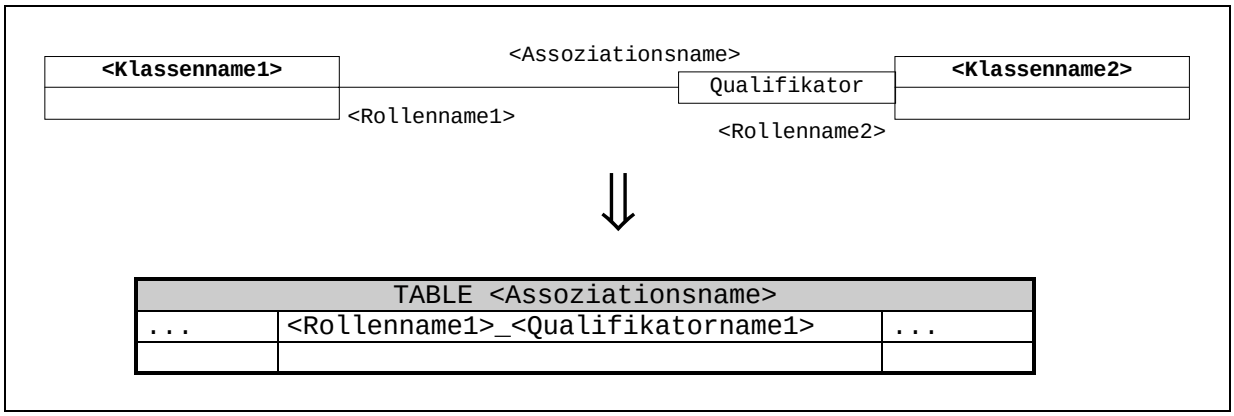


Abbildung 25: Abbildung von Qualifikation auf SQL

```
CREATE TABLE <Assoziationsname>
(
  ...
  <Rollenname1>_<Qualifikatorname1> <SQL-DDL-TYP> NOT NULL,
  ...
)
```

Die Integritätsbedingungen von qualifizierten Rollen unterscheiden sich dahingehend von unqualifizierten Rollen, daß die Reduktion der Multiplizität durch die Qualifikation berücksichtigt werden muß. Die Qualifikatoren werden bei der Überprüfung der Multiplizität in die GROUP-BY-Klausel mit einbezogen und brechen die Multiplizität somit in Teilbereiche auf.

```
CREATE PROCEDURE Pruefe_<Assoziationsname>_<Rollenname1>
  AS
  BEGIN
    -- Die in der Assoziationstabelle referenzierten Objekte muessen der
    -- Klasse, fuer welche die Rolle definiert ist oder einer ihrer
    -- Kindklassen angehoeren.
    IF EXISTS
      (SELECT oid
       FROM <Assoziationsname> a, PersistenteObjekte p
       WHERE a.<Rollenname1>=p.oid
         AND klasse NOT IN (SELECT Kind
                            FROM MetaGeneralisierungen
                            WHERE Elter = '<Klassenname1>')
    BEGIN
      RAISERROR 17003 'Rollen zu falschen Klassen zugeordnet!'
    END

    -- Keinem Objekt duerfen weniger Objekte der Zielrolle zugeordnet
    -- werden, als die MUG spezifiziert.
    -- Die Ueberpruefung sollte entfallen wenn <MUG> = 0 (<= 1!!!).
    IF EXISTS
      (SELECT p.oid
       FROM PersistenteObjekte p, <Assoziationsname> a
       WHERE p.klasse IN (SELECT Kind
                          FROM MetaGeneralisierungen
                          WHERE Elter = '<Klassenname2>')
         AND p.oid *= a.<Rollenname2>
       GROUP BY p.oid, a.<Rollenname1>_<Qualifikatorname1>, ...
       HAVING COUNT(a.<Rollenname1>) < <MUG>)
    BEGIN
      RAISERROR 17004 'Rollen verletzen MUG!'
    END

    -- Keinem Objekt duerfen mehr Objekte der Zielrolle zugeordnet
    -- werden, als die MOG spezifiziert.
    -- Die Ueberpruefung sollte entfallen wenn <MUG> = *.
    IF EXISTS
      (SELECT <Rollenname2>
       FROM <Assoziationsname>
       GROUP BY <Rollenname2>, <Rollenname1>_<Qualifikatorname1>, ...
       HAVING COUNT(*) > <MOG>)
    BEGIN
      RAISERROR 17005 'Rollen verletzen MOG!'
    END

END
```

8.2.9 Generierung von SQL-DDL für geordnete Rollen

Ist für Rollen eine vom Anwendungsumfeld bestimmte Ordnung gefordert (Constraint {ordered}), so kann diese über eine Sequenz-Nummer in der Datenbank repräsentiert werden. Die Sequenz-Nummer wird der für die Assoziation generierten Tabelle als Spalte hinzugefügt.

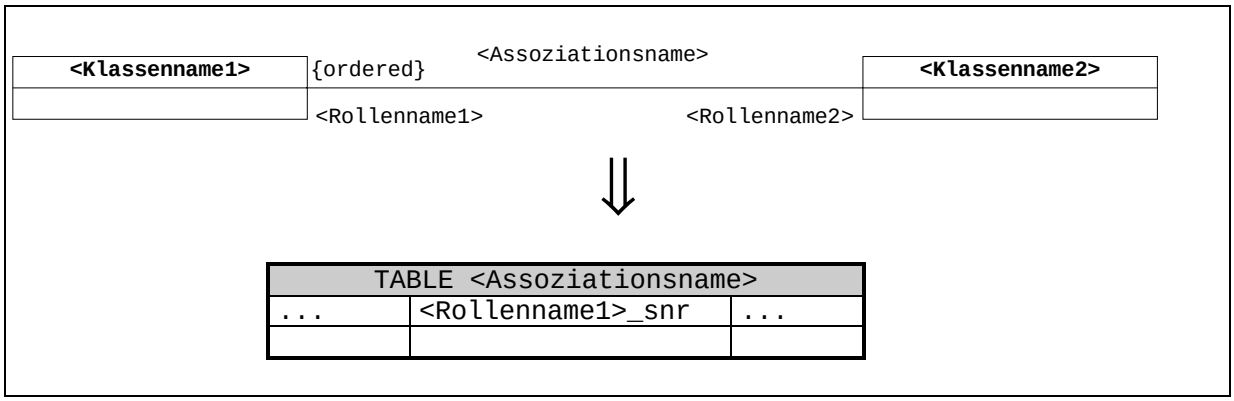


Abbildung 26: Abbildung der Ordnung auf SQL

```
CREATE TABLE <Assoziationsname>
(
  ...
  <Rollenname1>_snr INTEGER NOT NULL,
  UNIQUE(<Rollenname2>, <Rollenname1>_snr,
                          <Rollenname1>_<Qualifikatorname1>, ... ),
  ...
)
```

Wie aus dem SQL-Code ersichtlich ist, muß einem Element der geordneten Menge (Liste bzw. Sequenz) eindeutig eine Position in der Liste (Sequenz-Nummer) zugeordnet sein. Eine evtl. vorhandene Qualifikation spaltet die Räume für die Eindeutigkeit auf.

Für Operationen, Anfragen und Bedingungen über geordneten Rollen kann es nötig sein, daß die Sequenz-Nummern dicht vergeben werden, d.h. daß sie die Zahlen von Null bis zur Größe der Liste abzüglich eins umfassen. Zu diesem Zweck kann die prüfende Datenbankprozedur für die Rolle wie nachfolgend dargestellt erweitert werden.

```
CREATE PROCEDURE Pruefe_<Assoziationsname>_<Rollenname1>
  AS
  BEGIN

    ...

    -- Die Sequenz-Nummern duerfen nur Zahlen von 0..COUNT-1 enthalten.
    IF EXISTS
      (SELECT <Rollenname2>
       FROM <Assoziationsname>
       GROUP BY <Rollenname2>, <Rollenname1>_<Qualifikatorname1>, ...
       HAVING MAX(<Rollenname1>_snr) >= COUNT(<Rollenname1>))
    BEGIN
      RAISERROR 17006 'Listensortierung fehlerhaft!'
    END

    ...

END
```

In der Group-By-Klausel werden die Qualifikatoren der Rolle angegeben (da die Ordnung genau wie die Multiplizität durch die Qualifikation in Teilmengen aufgespaltet wird), gegebenenfalls keine.

8.2.10 Darstellung der Semantik von Komposition und Aggregation durch Invarianten

Die Konzepte der Komposition und Aggregation wurden im Meta-Modell nicht aufgenommen. Der Hinweis, daß diese Konzepte nur Spezialfälle der Assoziation sind, wurde als Begründung angegeben. Im folgenden soll dargestellt werden, wie die Zusatzsemantik von Komposition und Aggregation mit OCL-Invarianten beschrieben werden kann.

Einmaligkeit des physischen Enthaltenseins

Ist eine Klasse Komponente von mehr als einer anderen Klasse, so ergibt sich das Problem, daß die zugehörigen Instanzen nur Komponente einer anderen Instanz sein können.

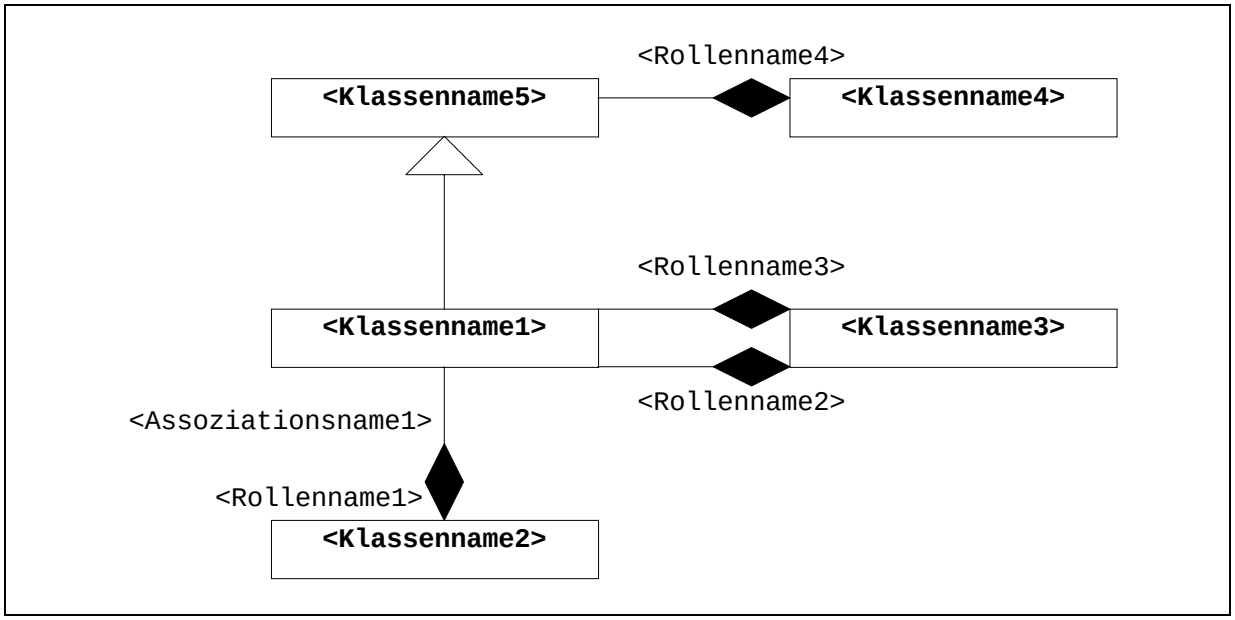


Abbildung 27: Beispiel zum physischen Enthaltensein (Komposition)

Zunächst ist die Menge aller einer Klasse zugeordneten Komponentengruppenklassen zu ermitteln, einschließlich der Komponentengruppenklassen, denen die Elternklassen der betreffenden Klasse untergeordnet sind. Jede Instanz der Klasse `<Klassenname1>` muß in genau einer der Assoziationen mit Aggregationscharakter in der entsprechenden Rolle vorkommen.
Diese Forderung kann durch folgendes OCL-Constraint formuliert werden:

```
<Klassenname1>
  ( self.<Rollenname1>->size() +
    self.<Rollenname2>->size() +
    ... +
    self.<Rollenname4>->size() )      = 1
```

Zyklenfreiheit

Bestehen in einem Klassendiagramm Aggregationszyklen, welche u.U. erst durch Generalisierung geschlossen werden, so sind entsprechende Zyklen auf Instanzenebene auf Grund der Semantik des Enthaltenseins nicht korrekt, sondern nur Heterarchien bzw. Hierarchien.

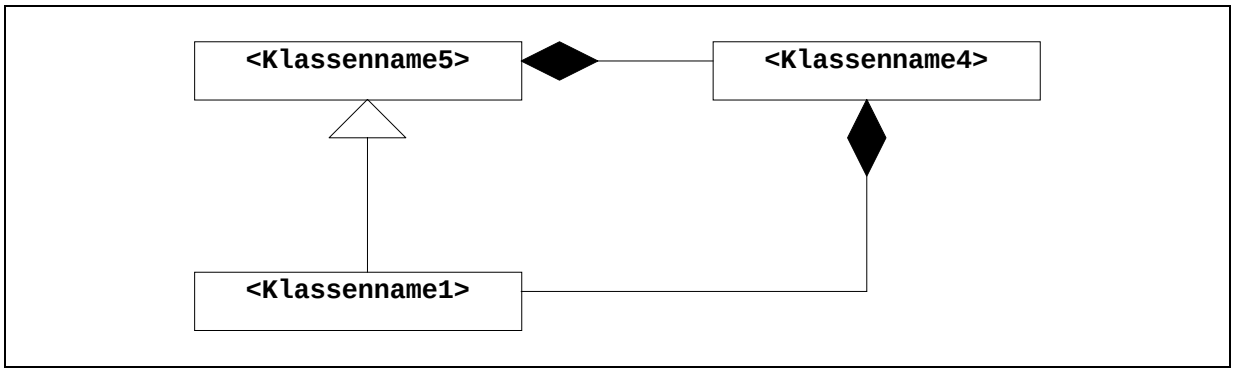


Abbildung 28: Beispiel zu Aggregationszyklen

Die Zyklenfreiheit für einen direkten Zyklus wird durch das folgende OCL-Constraint gefordert:

```
<Klassenname1>
  not (self.<klassenname4>.<klassenname5> = self)
  -- nur für einen direkten Zyklus
```

Zum unterdrücken von längeren Zyklen müssen rekursive „zusätzliche Operationen“ definiert werden.

Die Invarianten können mit der in dieser Arbeit beschriebenen Abbildung auf SQL abgebildet werden.

8.3 Überprüfung und Illustration der Abbildung mit Hilfe eines Beispiels

Die vorhergehend beschriebene Abbildung wurde mit Hilfe eines Beispiels überprüft. Die Abbildung erwies sich für dieses Beispiel in Bezug auf die Syntax als fehlerfrei. Korrekte Beispieldaten konnten eingefügt werden. Fehlerhafte Daten, welche die Strukturen des spezifizierten Objektmodells verletzt hätten, wurden als solche erkannt. Im Anhang B findet der interessierte Leser eine detaillierte Beschreibung des Tests, welcher durch ein Rose-Script unterstützt wurde.

8.4 Andere Abbildungen als Sonderfälle der beschriebenen Abbildung

Dieser Abschnitt zeigt, daß man die Ergebnisse der beschriebenen Abbildung zum Zwecke erhöhter Datenbank-Performance modifizieren kann und damit Ergebnisse, ähnlich den Abbildungen von Datenmodellen nach SQL-DDL aus verschiedener Literatur (z.B. [BlahPrem98]), erhält.
Die in diesem Abschnitt dargelegten Überlegungen dienen jedoch nur dazu, die Qualität der beschriebenen Abbildung als sehr grundlegend und aufwandsminimiert darzustellen und dienen *nicht* als Grundlage für die weitere Arbeit (insbesondere *nicht* für die Abbildung von OCL-Ausdrücken). Da diese Abbildung nicht der zentrale Gegenstand dieser Diplomarbeit ist und die Beschreibung in [BlahPrem98] ebenfalls informal ist, sollen die Performance-verbessernden Umformungen im folgenden nur kurz verbal skizziert werden.

Die beschriebene Abbildung nimmt als allgemeinen Fall Multiplizitäten mit einer Multiplizitätsobergrenze größer 1 an, wodurch die „Allgemeinheit“ an der Struktur gemessen wird. In [BlahPrem98] werden vor allem bei Attributen die Multiplizitäten 0..1 und 1..1 als allgemeiner Fall und die anderen Multiplizitäten als Spezialfall beschrieben. Das Maß der „Allgemeinheit“ ist hier die praktische Häufigkeit des Vorkommens.

Bei der in [BlahPrem98] beschriebenen Abbildung werden für Klassen jeweils Relationen generiert, welche die Attribute und Rollen mit einer Multiplizitätsobergrenze von 1 aufnehmen. Bei der beschriebenen Abbildung wird dies erreicht, indem alle Attribut- und Assoziationsrelationen einer Klasse, welche eine MUG von 1 besitzen zu einer Relation zusammengefaßt werden und die überflüssigen (da mehrfachen) Fremdschlüsselspalten gestrichen werden. Faßt man alle Attribute und Rollen einer Klasse zusammen, so entspricht dies dem vertikalen Verfahren. Schließt man die Attribute und Rollen aller Elternklassen mit ein, so entspricht dies dem horizontalen Verfahren. Werden für einen Vererbungsbaum alle Attribute und Rollen zusammengefaßt, so erhält man eine Universalrelation.

Der Effekt dieser Zusammenfassung ist die Verringerung der Anzahl der Indexe und eine gewisse Vereinfachung des Zusammenbaus vollständiger Objekte aus den Daten im RDBMS (theoretisch dürfte damit die Ordnung der Raum- und Zeitkomplexitäten nicht verringert werden, praktisch sind die Einsparungen jedoch möglicherweise schwerwiegend da reale RDBMS für praktische Einsatzfälle optimiert sind).

Für das abstrakte Beispielmodell wurden für die beiden Attribute der Klasse Klasse_A zwei separate Tabellen generiert. Auch für die Assoziation Assoz_2 wurde eine separate Tabelle generiert. Beide Attribute und die Rolle klasse_D haben eine MOG von 1.

```
CREATE TABLE Klasse_A_Attribut_A1
(
  oid INTEGER NOT NULL REFERENCES PersistenteObjekte,
  Attribut_A1 REAL NOT NULL
)
go

CREATE TABLE Klasse_A_Attribut_A2
(
  oid INTEGER NOT NULL REFERENCES PersistenteObjekte,
  Attribut_A2 INTEGER NOT NULL
)
go

CREATE TABLE Assoz_2
(
  klasse_A INTEGER NOT NULL REFERENCES PersistenteObjekte,
  klasse_E INTEGER NOT NULL REFERENCES PersistenteObjekte,
  UNIQUE(klasse_A, klasse_E)
)
go
```

Die Inhalte der drei Tabellen können demzufolge zusammengefügt werden, wobei die neue Tabelle den Namen der Klasse erhält.

```
CREATE TABLE Klasse_A
(
  oid INTEGER NOT NULL REFERENCES PersistenteObjekte, -- oid
  Attribut_A1 REAL NOT NULL,
  oid INTEGER NOT NULL REFERENCES PersistenteObjekte, -- oid
  Attribut_A2 INTEGER NOT NULL,
  klasse_A INTEGER NOT NULL REFERENCES PersistenteObjekte, -- oid
  klasse_E INTEGER NOT NULL REFERENCES PersistenteObjekte,
  UNIQUE(klasse_A, klasse_E)
)
go
```

Der Fremdschlüssel-Verweis auf eine Instanz der Klasse Klasse_A ist jetzt dreimal vorhanden, die entsprechenden Zeilen sind mit dem Kommentar "-- oid" versehen. Bis auf eine Zeile sind die anderen zu streichen, wobei eine Änderung des UNIQUE-Constraints in der letzten Zeile erforderlich wird. Das Attribut Attribut_A2 hat eine MUG von 0, in dieser Implementierung müssen somit Null-Werte zugelassen werden.

```
CREATE TABLE Klasse_A
(
  oid INTEGER NOT NULL REFERENCES PersistenteObjekte(oid), -- oid
  Attribut_A1 REAL NOT NULL,
  Attribut_A2 INTEGER NULL,
  klasse_E INTEGER NOT NULL REFERENCES PersistenteObjekte(oid),
  UNIQUE(klasse_E)
)
go
```

Somit werden zwei Indexe eingespart. Die Integritätsprüfungen durch Datenbankprozeduren für die MOGs 1 können entfallen, alle anderen Datenbankprozeduren müssen umformuliert werden. Letzteres würde auch auf SQL-Code für OCL-Constraints zutreffen.

In Bezug auf die Relation „PersistenteObjekte“ nimmt die beschriebene Abbildung den strukturell allgemeineren Fall der Mehrfachvererbung als allgemeinen Fall an. Die in [BlahPrem98] beschriebene Abbildung beschreibt analog den in der Praxis häufigeren Fall der Einfachvererbung und nennt die Möglichkeit der Mehrfachvererbung nur. Im Falle von Einfachvererbung kann die Tabelle „PersistenteObjekte“ auf die Wurzelklassen der Vererbungsbäume aufgeteilt werden, wobei dort der Primärschlüssel von „PersistenteObjekte“ jeweils den entsprechenden Fremdschlüssel überflüssig macht.

Die bereits zusammengefaßte Tabelle für die Klasse Klasse_A wird als Beispiel um die Spalten der Tabelle PersistenteObjekte erweitert, da sie eine Wurzelklasse im Vererbungsgraph ist. Das CHECK-Constraint wurde bereits an die Tatsache angepaßt, daß eine mengenmäßige Aufspaltung der Tabelle PersistenteObjekte vorgenommen wird, und die entsprechende Teilmenge nur die Instanzen der Klasse Klasse_A umfaßt.

```
CREATE TABLE Klasse_A
(
  oid INTEGER NOT NULL PRIMARY KEY,
  klasse VARCHAR(64) NOT NULL,
  name VARCHAR(64) NULL,

  oid INTEGER NOT NULL REFERENCES PersistenteObjekte(oid), -- oid
  Attribut_A1 REAL NOT NULL,
  Attribut_A2 INTEGER NULL,
  klasse_E INTEGER NOT NULL REFERENCES PersistenteObjekte(oid),
  UNIQUE(klasse_E)
)
go
```

Die mit dem Kommentar "-- oid" gekennzeichnete Zeile ist durch die Auflösung der Tabelle PersistenteObjekte nun falsch und überflüssig. Die Zeile für das Attribut "klasse" ist im konkreten Fall ebenfalls überflüssig, da die Klasse Klasse_A keine Kindklassen besitzt. Die so vereinfachte Tabelle entspricht genau der Code-Erzeugung, welche bei einer Durchführung von Hand am naheliegendsten ist:

```
CREATE TABLE Klasse_A
(
  oid INTEGER NOT NULL PRIMARY KEY,
  name VARCHAR(64) NULL,
  Attribut_A1 REAL NOT NULL,
  Attribut_A2 INTEGER NULL,
  klasse_E INTEGER NOT NULL REFERENCES PersistenteObjekte(oid),
  UNIQUE(klasse_E)
)
go
```

Wie das Beispiel andeutet, kann die Umformung vom Ergebnis der beschriebenen Abbildung zum Ergebnis einer beliebigen anderen Abbildung durch die aus dem Bereich des Entwurfs relationaler Datenbasisschemata definierten Regeln zur Relationensynthese und Normalisierung erfolgen (im Beispiel intuitiv benutzt). Diese Aussage ist gültig, sofern die anderen Abbildungen genau die Ausgangsinformationen aus dem Modell nutzen, wie die beschriebene Abbildung. Für Abbildungen, welche in Modelle in bestimmten Situationen eine besondere Semantik interpretieren oder zusätzliche Modellinformationen benutzen, funktioniert dies nicht (z.B. alle Aussagen zur Verwendung natürlicher Schlüssel (Wert-basierte Identität) in [BlahPrem98]).

8.5 Abbildung der Operatoren mit Zugriff auf Daten und Metadaten des Anwendungsmodells

8.5.1 Abbildung der Navigationsoperatoren

Navigation auf Attribute	
OCL-NAVATTR	Any → BasicSet

Abbildungsmuster

❶.<Attributname>
```
SELECT elem = <Attributname>
FROM <Klassenname>_<Attributname>
WHERE oid = ❶
```

Navigation auf Assoziationsenden	
OCL-NAVASSE	Any → AnySetSeq

Abbildungsmuster für Any → AnySet

❶.<Rollenname2>
```
SELECT elem = <Rollenname2>
FROM <Assoziationsname>
WHERE <Rollenname1> = ❶
```

Abbildungsmuster für Any → Seq

❶.<Rollenname2>
```
SELECT elem = <Rollenname2>, snr = <Rollenname2>_snr
FROM <Assoziationsname>
WHERE <Rollenname1> = ❶
```

Navigation auf Assoziationsobjekte	
OCL-NAVASOB	Any → Set

Abbildungsmuster

❶.<assoziationsklassenname>
```
SELECT elem = Assoziationsklasse
FROM <Assoziationsname>
WHERE <Rollenname1> = ❶
```

Navigation von Assoziationsobjekten auf Assoziationsenden	
OCL-NAVAOAE	Any → Any

Abbildungsmuster

❶.<Rollenname2>
```
SELECT elem = <Rollenname2>
FROM <Assoziationsname>
WHERE Assoziationsklasse = ❶
```

Navigation über Qualifikation ohne Qualifikatorangabe	
OCL-NAVUNQU	Any → Set

Abbildungsmuster

❶.<Rollenname2> ➠
```
SELECT elem = <Rollenname2>
FROM <Assoziationsname>
WHERE <Rollenname1> = ❶
```

Navigation über Qualifikation mit Qualifikatorangabe	
OCL-NAVQUAL	Any × Basic {× Basic}^{n} → AnySetSeq

Abbildungsmuster für Any × Basic {× Basic}^{n} → AnySet

❶.<Rollenname2>[❷, ...] ➠
```
SELECT elem = <Rollenname2>
FROM <Assoziationsname>
WHERE <Rollenname1> = ❶
  AND <Rollenname2>_<Qualifikatorname1> = ❷
  AND ...
```

Abbildungsmuster für Any × Basic {× Basic}^{n} → Seq

❶.<Rollenname2> [❷, ...] ➠
```
SELECT elem = <Rollenname2>, snr = <Rollenname2>_snr
FROM <Assoziationsname>
WHERE <Rollenname1> = ❶
  AND <Rollenname2>_<Qualifikatorname1> = ❷
  AND ...
```

8.5.2 Abbildung der Operatoren über Typen

Extension	
OCL-EXT	Type → Set

Abbildungsmuster

❶.allInstances ➠
```
SELECT elem=oid
FROM PersistenteObjekte
WHERE klasse IN (SELECT Kind
                 FROM MetaGeneralisierungen
                 WHERE Elter = ❶)
```

Name des Typs	
OCL-TYPENAME	Type → Str

Abbildungsmuster

❶.name ➠
```
❶
```

Menge der Namen der Attribute	
OCL-ATTRIBS	Type → Set

Abbildungsmuster

❶.attributes
```
SELECT name
FROM MetaAttribute
WHERE klasse = ❶
```

Menge der Namen der Rollen	
OCL-ASSOENDS	Type12 → Set1

Abbildungsmuster

❶.associationEnds
```
SELECT name
FROM MetaAssoziationsenden
WHERE klasse = ❶
```

Menge der Namen der Operationen	
OCL-OPERATNS	Type → Set

Abbildungsmuster

❶.operations
```
SELECT name
FROM MetaOperationen
WHERE klasse = ❶
```

Menge der Supertypen	
OCL-SUPERTPS	Type → Set

Abbildungsmuster

❶.supertypes
```
SELECT Elter
FROM MetaGeneralisierungen
WHERE Kind = ❶ AND Schritte = 1
```

Menge aller Supertypen - transitiv	
OCL-ALLSUPTS	Type → Set

Abbildungsmuster

❶.allSupertypes
```
SELECT Elter
FROM MetaGeneralisierungen
WHERE Kind = ❶ AND Schritte > 0
```

8.5.3 Abbildung der Operatoren über Instanzen

Vergleich zweier Instanzen (Identität)	
OCL-OBJEQU	Any × Any → Bool

Abbildungsmuster

❶ = ❷ ⇛
❶ = ❷

Prüfung zweier Instanzen auf Ungleichheit	
OCL-OBJNEQ	Any × Any → Bool

Abbildungsmuster

❶ <> ❷ ⇛
❶ <> ❷

Klasse zu Instanz ermitteln	
OCL_OCLTYPE	Any → Type

Abbildungsmuster

❶.oclType ⇛

```
SELECT klasse
FROM PersistenteObjekte
WHERE oid = ❶
```

wahr wenn Objekt den selben Typ hat wie der übergebene	
OCL_ISTYPEOF	Any × Type → Bool

Abbildungsmuster

❶.oclIsTypeOf(❷) ⇛

```
EXISTS (SELECT oid
        FROM PersistenteObjekte
        WHERE oid = ❶ AND klasse = ❷)
```

wahr wenn der Typ des Objekts Subtyp (transitiv) des übergebenen Typs ist	
OCL_ISKINDOF	Any × Type → Bool

Abbildungsmuster

❶.oclIsKindOf(❷) ⇛

```
EXISTS (SELECT p.oid
        FROM PersistenteObjekte p, MetaGeneralisierungen g
        WHERE p.oid = ❶ AND p.klasse = g.Kind AND g.Elter = ❷)
```

"Cast"	
OCL_ASTYPE	Any × Type → Any

Dieser Operator nimmt keine Transformation vor. Die übergebene oid wird unverändert zurückgegeben. Nur für den Fall, daß der erste Parameter nicht den als zweiten Parameter übergebenen Typ hat, wird ein Null-Wert zurückgegeben.

Abbildungsmuster

❶.oclAsType(❷)

```
SELECT p.oid
FROM PersistenteObjekte p, MetaGeneralisierungen g
WHERE p.oid = ❶ AND p.klasse = g.Kind AND g.Elter = ❷
```

9 Abbildung „weiterer Aspekte der OCL“ auf SQL

In diesem Kapitel wird die Abbildung der Aspekte der OCL, welche nicht formal nach dem Schema Datentypen/Operatoren beschrieben werden konnten, auf SQL dargestellt.

9.1 Invarianten

Eine Invariante der OCL

```
<Klassenname>
  [--OCL-Ausdruck(self)--]
```

kann auf Ebene der OCL in einen Ausdruck, welcher genau einen Wahrheitswert zum Ergebnis hat, umgeformt werden:

```
<Klassenname>.allInstances->forAll([--OCL-Ausdruck(self)--])
```

Das Ergebnis dieser Abbildung muß noch im UML-Modell verankert werden. Durch die Abbildung auf SQL wird diese „Verankerung“ auf die Datenbank übertragen.

Am Ende jeder betroffenen Transaktion muß demzufolge eine Stored Procedure aufgerufen werden, welche bei einem Ergebnis von FALSE die Transaktion zurücksetzt bzw. eine Fehlermeldung ausgibt:

```
CREATE PROCEDURE <generierterName> AS
BEGIN
  IF ( NOT
   SQL-CODE_FUER(<Klassenname>].allInstances->forAll([--OCL-Ausdruck--])) )
  BEGIN
    ROLLBACK TRANSACTION
    RAISERROR <generierteNummer>,
                  'Verletzung des Constraints <generierterName>'
  END
END
```

Die OCL-Ausdrücke `allInstances` und `forAll` können auch direkt umgesetzt werden, wenn `[--OCL-Ausdruck(oid)--]` nicht prozedural ist:

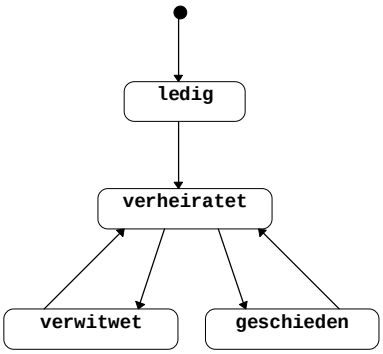
```
CREATE PROCEDURE <generierterName> AS
BEGIN
  IF (EXISTS
          (SELECT oid
             FROM PersistenteObjekte p, MetaGeneralisierungen m
             WHERE p.klasse = m.Kind AND m.Elter = '<Klassenname>'
               AND NOT [--OCL-Ausdruck(oid)--]))
  BEGIN
    ROLLBACK TRANSACTION
    RAISERROR <generierteNummer>,
                  'Verletzung des Constraints <generierterName>'
  END
END
```

Für SQL-92 wäre auch eine Abbildung auf ASSERTIONs möglich:

```
CREATE ASSERTION <generierterName>
  CHECK (NOT EXISTS
          (SELECT oid
           FROM PersistenteObjekte p, MetaGeneralisierungen m
           WHERE p.klasse = m.Kind AND m.Elter = '<Klassenname>'
             AND NOT [--OCL-Ausdruck(oid)--]))
```

9.2 Wächter- und Änderungsbedingungen

Um Wächterbedingungen in OCL auf SQL abbilden zu können, müßten auch die Zustandsübergangsdiagramme, welche diese Bedingungen enthalten, abgebildet werden. Dies wäre auch möglich. So könnte das folgende Zustandsübergangsdiagramm



auf

```
CREATE ASSERTION  -- für RDBMS Informix
 FOR Person
  AS
   IF familienstand.OLD = ledig THEN
     familienstand.NEW IN (ledig, verheiratet)
   ...
```

abgebildet werden.

Da der Aufwand für die Abbildung von STDs auf SQL sehr hoch ist und vergleichsweise wenig Nutzen bringt (viele Modelle für Datenbanken verzichten laut [BlahPrem98] ganz auf dynamische Modellierung), soll in dieser Arbeit nicht weiter darauf eingegangen werden. Ein weiterer Grund, die dynamische Modellierung nicht automatisch auf SQL abzubilden, ist der folgende:
Ein Teil des dynamischen Modells könnte sinnvoll auf SQL abgebildet werden, indem die korrekten Zustandsübergänge durch Integritätsbedingungen überprüft werden oder für Ereignisse Stored Procdures generiert werden, welche die spezifizierten Zustandsänderungen vornehmen. Der Teil der Dynamik, welcher nicht Gegenstand des Transaktionsschutzes sein soll, wie z.B. der Zustand von Benutzungsschnittstellenobjekten (Realisierung z.B. in C++), soll nicht auf SQL abgebildet werden. Es kann jedoch nicht automatisch zwischen diesen beiden Arten von dynamischen Modellkomponenten unterschieden werden, welche durchaus innerhalb eines Diagramms verflochten sein können.

9.3 Vor- und Nachbedingungen für Operationen und Methoden

Wenn Operationen und Methoden mit Stored Procedures realisiert werden sollen, so wäre eine Abbildung der Vor- und Nachbedingungen auf SQL möglich. Die Ausdrücke könnten wie beschrieben auf SQL abgebildet werden, bei Rückgabe von FALSE wäre ein Zurücksetzen der aktuellen Transaktion mit Fehlermeldung eine angemessene Reaktion. Es besteht jedoch die weitaus günstigere Möglichkeit, mit Hilfe der Vor- und Nachbedingungen die Korrektheit des implementierenden Codes (automatisch) zu überprüfen. Dies würde den Rahmen dieser Diplomarbeit jedoch sprengen.

9.4 Abbildung von Operationen und Operationsaufrufen

An dieser Stelle soll nur die Abbildung von sog. „Zusätzlichen Operationen“ der OCL behandelt werden, welche vor allem zur Formulierung rekursiver OCL-Ausdrücke notwendig sind. Die Darstellung beschreibt keine exakten Abbildungsmuster sondern nur das Prinzip.

„Zusätzliche Operationen“ können direkt auf Stored Procedures abgebildet werden, wobei das Objekt, auf dem die Operation ausgeführt wird, als Parameter übergeben wird:

```
Klasse
  operation([--formale Parameterliste--]): [--Ergebnistyp--];
  operation = [--Operationskörper--]
```

➠

```
CREATE PROCEDURE operation objektId INTEGER, [--formale Parameterliste--],
[--Ergebnistyp--] OUTPUT AS
BEGIN
  [--Operationskörper--]
END
```

Der Aufruf
`objekt->operation([--aktuelle Parameterliste--])`

wird auf

```
EXECUTE operation objektId, [--aktuelle Parameterliste--]
```

abgebildet.

Bei rekursiven Aufrufen wird die Aufruftiefe von Sybase™ auf maximal 16 beschränkt. Dies ist eine schwerwiegende Einschränkung, welche nicht umgangen werden kann.

Die Operationskörper werden wie normale OCL-Ausdrücke auf SQL abgebildet.

Zu klären ist nun noch die Abbildung der Parameterlisten und deren Benutzung (füllen und auslesen) durch die SQL-Generate für die OCL-Ausdrücke.

Für die Elementardatentypen ist die Abbildung der Parameter problemlos:

```
[--name--]: [--typ--] = [--defaultwert--],
```

wird abgebildet auf:

```
@[--name--] [--typ--] = [--defaultwert--],
```

Tabellen, Tabellennamen, Mengen o.ä. können über die Parameter nicht übergeben werden.

Für alle Parameter von Kollektionstypen müssen deshalb (innerhalb einer Datenbank) gemeinsame Tabellen für Parameter angelegt werden:

```
CREATE TABLE SetBagOfInteger (tsel INTEGER, elem INTEGER)

CREATE TABLE SetBagOfReal (tsel INTEGER, elem REAL)

CREATE TABLE SetBagOfBoolean (tsel INTEGER, elem BIT)

CREATE TABLE SetBagOfString (tsel INTEGER, elem VARCHAR(200))

CREATE TABLE SetBagOfEnumeration (tsel INTEGER, elem VARCHAR(200))
```

```
CREATE TABLE SetBagOfOid (tsel INTEGER, elem INTEGER)


CREATE TABLE SeqOfInteger (tsel INTEGER, elem INTEGER, snr INTEGER)

CREATE TABLE SeqOfReal (tsel INTEGER, elem REAL, snr INTEGER)

CREATE TABLE SeqOfBoolean (tsel INTEGER, elem BIT, snr INTEGER)

CREATE TABLE SeqOfString (tsel INTEGER, elem VARCHAR(200), snr INTEGER)

CREATE TABLE SeqOfEnumeration
  (tsel INTEGER, elem VARCHAR(200), snr INTEGER)

CREATE TABLE SeqOfOid (tsel INTEGER, elem INTEGER, snr INTEGER)
```

Die Tabellen enthalten für alle Operationsaufrufe gemeinsam die aktuellen Parameter des zugehörigen Kollektionstyps. Durch den Tabellen-Selektor (tsel) werden sie in Teiltabellen aufgesplittet.

Ein neuer Tabellen-Selektor kann durch die Operation

```
SELECT MAX(tsel)+1
FROM ParameterTabelleX
```

bestimmt werden.

Das Einfügen in die Parametertabellen erfolgt unter Angabe des Tabellen-Selektors:

```
INSERT INTO ParameterTabelleY VALUES (TabellenSelektor, ...)
```

bzw.

```
INSERT INTO ParameterTabelleY
  SELECT TabellenSelektor, ...
  FROM ...
```

Die tatsächliche Parameterkollektion kann aus der Gesamtkollektion durch einfache Selektion über den Tabellenselektor ermittelt werden:

```
SELECT elem, ...
FROM ParameterTabelleZ
WHERE tsel = TabellenSelektor
```

Über die Parameterlisten der Stored Procedures können nun die Tabellen-Selektoren anstelle von Tabellen (was ja nicht möglich ist) übergeben werden.

Nach der Benutzung sollten nicht benötigte Teilkollektion gelöscht werden:

```
DELETE FROM ParameterTabelleD
WHERE tsel = TabellenSelektor
```

Die Stored Procdures von Sybase™ sind für die Umsetzung von (rekursiven) OCL-Operationen nur bedingt geeignet. Der größte Nachteil ist die auf 16 begrenzte Iterationstiefe. Da als Parameter keine Relationen übergeben werden können und die Rückgabe des Ergebnisses umständlich ist, muß für die Parameterübergabe eine spezielle Abbildung verwendet werden. Operationen sind prozedurale Konstrukte und keine Ausdrücke wie in der OCL, welche sich dort einfach in andere Ausdrücke einbauen lassen.
Für Stored Procedures (bzw. Persistent Stored Modules) müßten also beliebige Iterationstiefe (bei Oracle8™ vorhanden), beliebige Ergebnis- und Parametertypen (vor allem Tabellen) sowie deren Verwendung wie Subquerys gefordert werden.

Das nichtprozedurale Konzept der OCL-Operationen würde durch rekursive, parameterisierbare Views noch besser umgesetzt werden können (als Parameter Relationen). Ein Beispiel für Views mit Parametern und rekursivem Selbstaufruf könnte wie folgt aussehen:

```
-- rekursive Bestimmung aller Nachfahren einer Klasse
-- in einem Vererbungsgraphen
CREATE VIEW AlleKindklassen (TABLE p1 (elem INTEGER))
AS
  SELECT elem = Klassen.oid
    FROM p1, Klassen
    WHERE p1.elem = Klassen.Elternklasse
  UNION
  SELECT elem
    FROM AlleKindklassen(p1)
```

9.5 Typübereinstimmung und Vorrangregeln

Diese Aspekte haben für die Abbildung keine Bedeutung (müssen aber vor Abbildung geprüft bzw. beachtet werden).

9.6 Kommentare

OCL-Kommentare können direkt von SQL übernommen werden. Eine Berücksichtigung für Fehlermeldungen des generierten SQL-Codes wäre auch denkbar.

9.7 Abbildung der undefinierten Werte auf SQL

Welche Bedeutung ein undefinierter Wert hat, wenn er bei der Auswertung einer Invariante als Ergebnis auftritt, wird in [OCL97] nicht beschrieben.
In SQL-92 werden undefinierte Werte in CHECK-CONSTRAINTS wie true behandelt. Dies ist intuitiv nicht unbedingt leicht nachvollziehbar, da in diesem Fall die Einhaltung der Integritätsbedingung ungewiß ist und Transaktionen die Integrität eigentlich sicherstellen sollten, anstelle nach dem Prinzip „Im Zweifelsfalle für den Angeklagten“ zu verfahren.
Da bei der Behandlung von undefinierten Werten in OCL-Ausdrücken und dem daraus resultierenden SQL-Code keine Widersprüche festgestellt werden konnten, kann nur nach o.g. Prinzip angenommen werden, daß die Abbildung korrekt ist. Für detailliertere Untersuchungen sind die Informationen in [OCL97] nicht ausreichend (siehe auch [SchaKort98] Abschnitt 4.3).

9.8 Bennennungskonventionen

Die Abbildungsmuster berücksichtigen nicht, daß beim Vorhandensein expliziter Rollennamen auch der Klassenname mit kleinem Anfangsbuchstaben verwendet werden kann. In praktischen Realisierungen muß dies jedoch berücksichtigt werden.

9.9 Implizite Mengeneigenschaft

Das Problem der impliziten Mengeneigenschaft wurde zum Zwecke der kompakten, leichtverständlichen Darstellung der Abbildungsmuster nicht berücksichtigt. Für praktische Realisierungen müßten hier ebenfalls noch Lösungen gefunden werden.

9.10 Kombination der Operatoren und Auswertungsrichtung; Pfadnamen

Die Auswertungsrichtung und Pfadnamen sind nur beim Parsen von Bedeutung. Auf die Probleme mit der Kombination der Operatoren wurde bereits in den vorhergehenden Kapiteln eingegangen.

9.11 Mengensysteme

Mengensysteme wurden bereits in Kapitel 2 ausgeschlossen und sind somit nicht mehr zu berücksichtigen.

9.12 Abkürzung von OCL-COLLECT und erweiterte Variante von OCL-FORALL

Diese „Makros“ müssen während oder vor dem Parsen von OCL-Ausdrücken aufgelöst werden.

10 Probleme bei der Abbildung und Lösungsansätze

In diesem Kapitel sollen Probleme bei der Abbildung von OCL nach SQL identifiziert und Wege zur Verbesserung gesucht werden. Aus Zeit- und Umfangsgründen konnten diese Wege nicht vollständig untersucht und beschrieben werden. Es geht nur darum, dieses wichtige Thema anzureißen und möglicherweise Denkanstöße für weiterführende Arbeiten zu vermitteln.

10.1 Erweiterung der OCL um spezielle Operatoren

An einem Beispiel soll gezeigt werden, wie die durch die bisherige Abbildung noch nicht verwendeten Elemente von SQL dazu verwendet werden können, die Qualität der Abbildung zu verbessern.

Grundlage des Beispiels ist der folgende Ausschnitt aus einem Klassendiagramm:

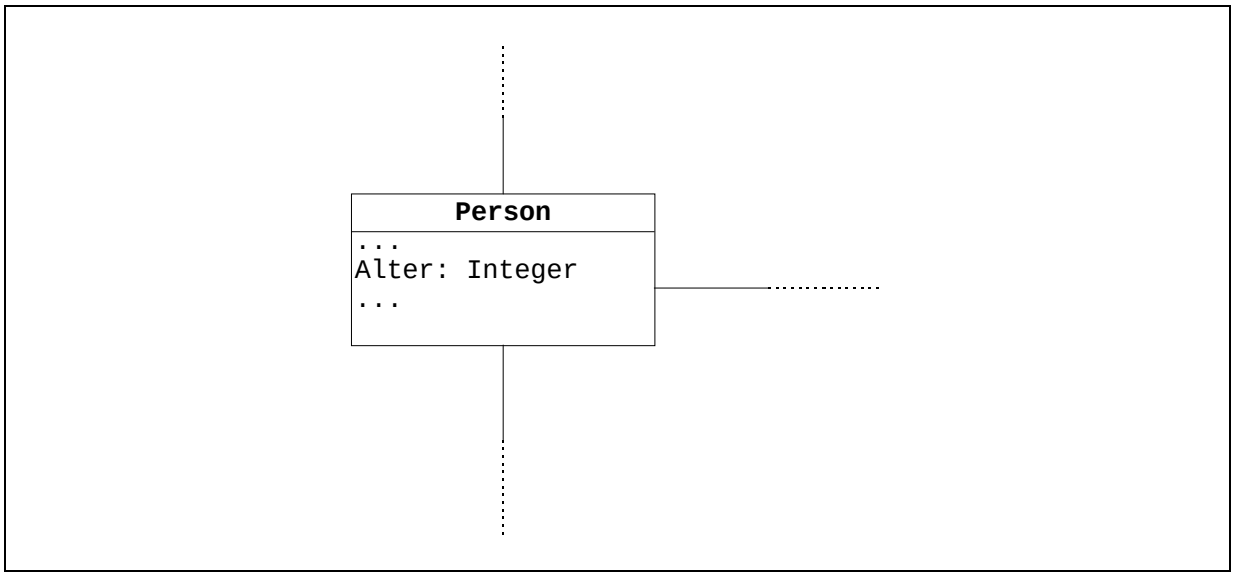


Abbildung 29: Beispiel für Maximum

Innerhalb einer Invariante soll das Maximum das Alters aller Instanzen von Person benötigt werden. Dieses kann durch folgenden Teilausdruck ermittelt werden:

```
Person.allInstances.Alter->iterate(elem; acc: Integer = 0 | max(elem, acc))
```

Zu berücksichtigen ist, daß der Punkt vor Alter die Korzform von collect ist. Der Ausdruck sieht nach der Expansion dieser Kurzform wie folgt aus:

```
Person.allInstances->collect(elem |
              elem.Alter)->iterate(elem; acc: Integer = 0 | max(elem, acc))
```

10.1.1 Abbildung auf SQL

Für den Ausschnitt aus dem Klassendiagramm wird u.a. folgender SQL-Code generiert:

```
CREATE TABLE PersistenteObjekte
(
  oid INTEGER NOT NULL PRIMARY KEY,
  klasse VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  ...
)
```

```
CREATE TABLE Person_Alter
(
  oid INTEGER NOT NULL REFERENCES PersistenteObjekte,
  Alter INTEGER NOT NULL
)

...
```

Die Abbildung des Teilausdrucks erfolgt so:

Für `Person`:
`'Person'`

Für `Person.allInstances`:
```
SELECT elem=oid
FROM PersistenteObjekte
WHERE klasse IN (SELECT Kind
                 FROM MetaGeneralisierungen
                 WHERE Elter = 'Person')
```

Für `Person.allInstances->collect(elem | elem.Alter)`:
```
SELECT elem = (SELECT elem = Alter
               FROM Person_Alter
               WHERE oid = elem)
FROM (SELECT elem = oid
      FROM PersistenteObjekte
      WHERE klasse IN (SELECT Kind
                       FROM MetaGeneralisierungen
                       WHERE Elter = 'Person'))
```

Eine Subquery ist bei Sybase™ in der FROM-Klausel nicht zulässig und für `OCL-ITERATE` existiert kein nichtprozedurales Abbildungsmuster. Die weitere Abbildung muß somit prozedural erfolgen.

Vollständig für Sybase:
```
-- Deklarationsteil --
CREATE TABLE Zwischenergebnis1 (elem INTEGER)
CREATE TABLE Zwischenergebnis2 (elem INTEGER)

DECLARE @actElem INTEGER
DECLARE @actAcc INTEGER

DECLARE c1 CURSOR FOR
  SELECT elem
  FROM Zwischenergebnis2
  FOR READ ONLY

DECLARE @Zwischenergebnis3 INTEGER

DECLARE @Zwischenergebnis4 INTEGER

-- Anweisungsteil --
TRUNCATE TABLE Zwischenergebnis

INSERT INTO Zwischenergebnis1
  SELECT elem=oid
  FROM PersistenteObjekte
  WHERE klasse IN (SELECT Kind
                   FROM MetaGeneralisierungen
                   WHERE Elter = 'Person')

INSERT INTO Zwischenergebnis2
  SELECT elem = (SELECT elem = Alter
                 FROM Person_Alter
```

```
                  WHERE oid = elem)
  FROM Zwischenergebnis1

SELECT @Zwischenergebnis3 = 0

SELECT @actAcc = @Zwischenergebnis3

OPEN c1

FETCH c1 INTO @actElem

WHILE (@@sqlstatus = 0)
BEGIN
  -- Anweisungen für [--3(@actAcc, @actElem)--]
  IF (@actAcc>@actElem)
    SELECT @Zwischenergebnis4 = @actAcc
  ELSE
    SELECT @Zwischenergebnis4 = @actElem

  SELECT @actAcc = @Zwischenergebnis4

  FETCH c1 INTO @actElem
END

CLOSE c1

-- Ergebnis liegt in @actAcc
```

Diese Form der Bestimmung des Maximums des Alters ist sehr aufwendig. Eine Verringerung des Ausführungsaufwandes durch das Datenbanksystem nach den Regeln der Relationenalgebra ist nicht möglich.

10.1.2 Einführung eines neuen Operators

Das geforderte Maximum des Attributs Alter aller Instanzen der Klasse Person kann in SQL leicht durch die Aggregatfunktion max bestimmt werden. Intuitiv kann eine entsprechende Anfrage leicht erstellt werden. Aber wie kann diese einfachere Abbildung auch durch ein formales Verfahren erreicht werden?

Dazu muß die OCL zunächst um einen Operator `OCLX-MAXCOLL` als Aggregatfunktion über Kollektionen erweitert werden (das `X` in `OCLX-MAXCOLL` steht für eXtended).

`OCLX-MAXCOLL`		`_->max : Coll → IntReal`

Dieser Operator bestimmt das größte Element einer Kollektion von ganzen oder reellen Zahlen.

Weiterhin ist eine neue Äquivalenzbedingung einzuführen:

`OCLX-MAXCOLL ↔ OCL-ITERATE, OCL-CONINT, OCL-ADD`

`❶->max  ≡  ❶->iterate(`*`elem`*`; `*`acc`*` : `***`Type`*` = 0 | max(`*`elem`*`, `*`acc`*`))`

(Für ***Type*** ist fallabhängig `Integer` oder `Real` einzusetzen.)

Für den neuen Operator `OCLX-MAXCOLL` muß nun noch ein Abbildungsmuster von OCL nach SQL definiert werden:

Größtes Element einer Kollektion	
`OCLX-MAXCOLL`	`Coll → IntReal`

Abbildungsmuster

❶->max

```
SELECT MAX(elem)
FROM ❶
```

10.1.3 Durchführung der verbesserten Abbildung

Vor der Abbildung muß nun eine Umformung des Ausdrucks vorgenommen werden, so daß der neue Operator `OCLX-MAXCOLL` auch zur Anwendung kommt.

```
Person.allInstances->collect(elem |
             elem.Alter)->iterate(elem; acc: Integer = 0 | max(elem, acc))
```

Die Äquivalenzbedingung zu `OCLX-MAXCOLL` kann mit folgendem Ergebnis angewendet werden:

```
Person.allInstances->collect(elem | elem.Alter)->max
```

Die Abbildung von `Person.allInstances->collect(elem | elem.Alter)` wurde bereits wie folgt ermittelt:

```
SELECT elem = (SELECT elem = Alter
               FROM Person_Alter
               WHERE oid = elem)
FROM (SELECT elem = oid
      FROM PersistenteObjekte
      WHERE klasse IN (SELECT Kind
                       FROM MetaGeneralisierungen
                       WHERE Elter = 'Person'))
```

Nun ist noch das Abbildungsmuster von `OCLX-MAXCOLL` anzuwenden:

```
SELECT MAX(elem)
FROM (SELECT elem = (SELECT elem = Alter
                     FROM Person_Alter
                     WHERE oid = elem)
      FROM (SELECT elem = oid
            FROM PersistenteObjekte
            WHERE klasse IN (SELECT Kind
                             FROM MetaGeneralisierungen
                             WHERE Elter = 'Person'))
```

Diese SQL-Konstruktion kann von Sybase™ 11 nicht verarbeitet werden. Mit Oracle8™ kann jedoch eine Auswertung erfolgen.
Es wäre auch möglich das Abbildungsergebnis nach den Regeln der Relationenalgebra zu vereinfachen:

```
SELECT MAX(a.Alter)
FROM Person_Alter a, PersistenteObjekte p, MetaGeneralisierungen g
WHERE a.oid = p.oid AND p.klasse = g.Kind AND g.Elter = ‘Person’
```

Wenn bekannt ist, daß Person keine weiteren Kindklassen besitzt und keine Instanzen anderer Klassen somit ein Attribut vom Typ Alter besitzen können, kann in diesem konkreten Fall eine noch stärkere Vereinfachung erfolgen:

```
SELECT MAX(Alter)
FROM Person_Alter
```

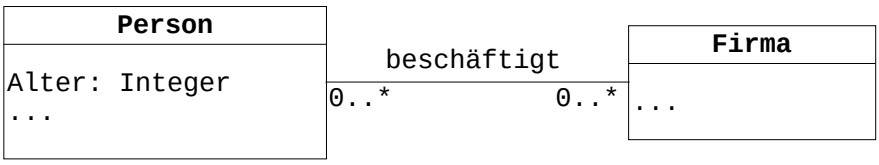
10.1.4 Zusammenfassung

Dieser Abschnitt hat gezeigt, wie Elemente von SQL, welche keine direkte Entsprechung in OCL besitzen, bei der Abbildung trotzdem berücksichtigt werden können. Dazu wurde die OCL exemplarisch um eine direkte Entsprechung einer SQL-Funktion erweitert. Die Qualität der Abbildung konnte dadurch drastisch verbessert werden, da ein prozedurales Abbildungsergebnis von 56 Zeilen durch ein deskriptives (nichtprozedurales) Abbildungsergebnis von 9 Zeilen ersetzt wurde.
Für eine generelle Lösung müßten für alle Features von SQL, welche in der OCL keine direkte Entsprechung haben und somit durch die Abbildung von OCL nach SQL nicht erfaßt wurden, derartige Abbildungsregeln und Äquivalenzbedingungen definiert werden. Für diese Diplomarbeit ist die Durchführung dessen aus Zeit- und Umfangsgründen nicht möglich.

10.2 Vermeidung von Ausdrücken mit OCL-ITERATE bei Navigation

Die OCL definiert die Navigationsoperatoren nur von einzelnen Objekten ausgehend, und nicht wie bei ONN und NE auch von Mengen bzw. von Kollektionen aus. Ein Äquivalent zur Navigation von Kollektionen aus wird möglich, da für die OCL der Operator `OCL-COLLECT` und eine Kurzform (Punkt-Symbol) eingeführt wurde. Da die dem Operator `OCL-COLLECT` übergebenen Funktionen aber nur Elementardatentypen als Ergebnis besitzen dürfen, bleiben Einschränkungen bestehen. Navigationen von Kollektionen auf Kollektionen müssen mit `OCL-ITERATE` nachgebildet werden, wie das nachfolgende Beispiel illustriert.

Folgender Ausschnitt aus einem Klassendiagramm soll zugrunde gelegt werden:



Ein OCL-Teilausdruck soll alle Firmen ermitteln, welche Mitarbeiter über 50 Jahre beschäftigen:

```
(Person.allInstances->select(Alter > 50)).firma
```

Die Kurzform von `OCL-COLLECT` (Punkt-Operator) muß in die explizite Form übertragen werden:

```
Person.allInstances->select(Alter > 50)->collect(p | p.firma)
```

Dieser Ausdruck ist nach der OCL-Definition nicht gültig, da die an `OCL-COLLECT` übergebene Funktion eine Menge von Firmen als Ergebnis besitzt. Der gewünschte OCL-Teilausdruck muß also mit Hilfe von `OCL-ITERATE` formuliert werden:

```
Person.allInstances->select(Alter > 50)->iterate(elem; acc: Bag = Bag{} |
                                                 acc->union(elem.firma))
```

Es wäre nun möglich, den Operator `OCL-COLLECT` zu erweitern. Die daraus resultierenden Abbildungsmuster wären jedoch alle prozeduraler Gestalt. Dies hängt damit zusammen, daß die Subquery in der SELECT-Klausel (siehe Abbildungsmuster zu `OCL-COLLECT`) in diesem Fall eine mehrelementige statt einer einelementigen Relation zurückgeben würde. Es konnten auch keine anderen nichtprozeduralen Abbildungsmuster gefunden werden.

Es wäre aber vergleichsweise einfach, die OCL um die Navigation von Kollektionen aus zu erweitern und dann nichtprozedurale Abbildungsmuster auf SQL zu formulieren. Dies soll an dieser Stelle nur exemplarisch für die Navigation auf Attribute ausgeführt werden (alle anderen Navigationsoperatoren können analog erweitert werden):

Erweiterte Navigation auf Attribute	
OCLX-NAVATTR	Coll → BagSeq

Abbildungsmuster für Coll → Bag

❶.<Attributname> ➠
```
SELECT elem = <Attributname>
FROM <Klassenname>_<Attributname>
WHERE oid IN ❶
```

Abbildungmuster für Seq → Seq **(nur wenn einwertiges Attribut)**

❶.<Attributname> ➠
```
SELECT elem = a.<Attributname>, snr = b.snr
FROM <Klassenname>_<Attributname> a, ❶ b
WHERE a.oid = b.oid
-- wenn Multiplizität 0..1, dann anschließend Sequenznummern verdichten
```

Wenn, wie in diesem Abschnitt angedeutet, anstelle der „Kurzform für Collect“ eine Definition der Navigation von einzelnen Objekten auf Kollektionen von Objekten ausgedehnt werden würde, könnte eine Verbesserung der Abbildung OCL nach SQL erreicht werden.
Darüber hinaus würden auch in der OCL Ausdrücke mit OCL-ITERATE kürzer dargestellt werden können. Der Umfang der Defininition würde sich nur geringfügig erhöhen. Die in [SchaKort98] in Abschnitt 4.5 beschriebene Mehrdeutigkeit würde damit auch umgangen, da nun eine explizite Formulierung mit OCL-COLLECT für diesen Fall nötig wäre (die Kurzform macht hier unter dem Gesichtspunkt der Anwendung gar keinen Sinn).

10.3 Verwendung von CHECK-CONSTRAINTs anstelle von ASSERTIONs

Da die ASSERTIONs von SQL-92 in kommerziellen Realisierungen wie Sybase™ 11 und Oracle8™ nicht vorhanden sind, können Invarianten praktisch nur auf Stored Procedures abgebildet werden.
Einige OCL-Invarianten, welche auf ASSERTIONs abgebildet werden können, könnten jedoch auf die in Sybase™ und Oracle™ vorhanden CHECK-CONSTRAINTs abgebildet werden. Auf Grund der Einschränkungen der CHECK-CONSTRAINTs trifft dies aber nur auf einen Teil der Invarianten zu. CHECK-CONSTRAINTs können sich nur auf die Elemente eines Tupels beziehen und nur SQL-Operatoren verwenden, welche den OCL-Operatoren der Gruppen ❶ und ❷ entsprechen (plus OCL-CONSET und OCL-ELEM). Insbesondere sind keine Subqueries möglich.
Somit könnte für Invarianten, welche nur Navigation auf einwertige Attribute sowie OCL-CONSET, OCL-ELEM und die Operatoren der Gruppen ❶ und ❷ enthalten, eine spezielle Abbildung auf CHECK-CONSTRAINTS beschrieben werden, welche sich in einigen wenigen Details von den vorhergehend beschriebenen Abbildungen unterscheidet.

Bsp.:
```
Person
  Alter >= 0 and Alter <= 969
```

➠

```
CREATE TABLE Person
(
  ...
  CHECK Alter >= 0 AND Alter <= 969;
  ...
)
```

10.4 Verwendung von TRIGGERn anstelle von ASSERTIONs

Alle Invarianten, für die durch elementare SQL-Datenmanipulationsoperationen immer direkt von einem konsistenten Zustand in den nächsten übergegangen werden kann (gemessen an der OCL-Invariante), kann auf jeden Fall eine Realisierung mit Triggern erfolgen, auch wenn keine Realisierung durch CHECK-CONSTRAINTs möglich ist.

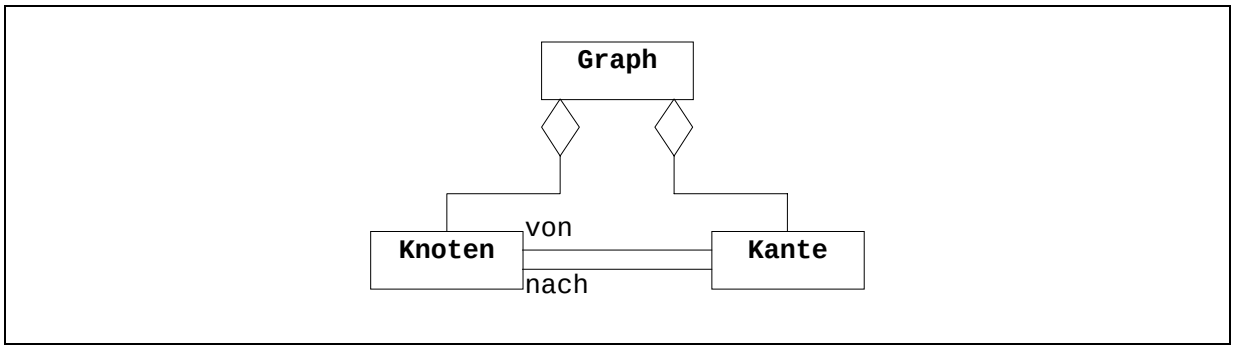


Abbildung 30: Datenmodell für Graphenstrukturen

Zum Beispiel kann in einem Datenmodell zur Speichung von Graphenstrukturen (siehe Abb.) gefordert werden, daß die Knoten der verschiedenen, in der Datenbank abgelegten Graphen nicht graphenübergreifend durch Kanten verbunden sein können:

```
-- Eine Kante kann nur zwei Knoten miteinander verbinden,
-- welche zu dem selben Graphen gehören wie sie selbst
Kante
  self.von.graph = self.graph and self.nach.graph = self.graph
```

Für dieses Datenmodell könnte der folgende SQL-Code generiert werden:

```
CREATE TABLE Graph
(
   oid INTEGER NOT NULL PRIMARY KEY,
  ...
)

CREATE TABLE Knoten
(
  oid INTEGER NOT NULL PRIMARY KEY,
  graph INTEGER NOT NULL REFERENCES Graph,
  ...
)

CREATE TABLE Kante
(
  oid INTEGER NOT NULL PRIMARY KEY,
  graph INTEGER NOT NULL REFERENCES Graph,
  von INTEGER NOT NULL REFERENCES Knoten,
  nach INTEGER NOT NULL REFERENCES Knoten,
  ...
)
```

In zugehörige Invariante könnte durch folgende TRIGGER realisiert werden:

```
CREATE TRIGGER ON Kante
FOR INSERT, UPDATE
  AS
    IF (EXISTS
          (SELECT oid
           FROM INSERTED, Knoten
           WHERE INSERTED.von = Knoten.oid
             AND NOT INSERTED.graph = Knoten.graph)
        OR
        EXISTS
          (SELECT oid
           FROM INSERTED, Knoten
           WHERE INSERTED.nach = Knoten.oid
             AND NOT INSERTED.graph = Knoten.graph))
    BEGIN
      ROLLBACK TRANSACTION
      ...
    END

CREATE TRIGGER ON Knoten
FOR INSERT, UPDATE
  AS
    IF (EXISTS
          (SELECT oid
           FROM INSERTED, Kante
           WHERE Kante.von = INSERTED.oid
             AND NOT INSERTED.graph = Kante.graph)
        OR
          (SELECT oid
           FROM INSERTED, Kante
           WHERE Kante.nach = INSERTED.oid
             AND NOT INSERTED.graph = Kante.graph))
    BEGIN
      ROLLBACK TRANSACTION
      ...
    END
```

Ein Verfahren zur Abbildung von OCL-Invarianten auf SQL-Trigger wäre aufwendiger und umfangreicher als die allgemeine Abbildung von OCL auf SQL und wird deshalb in dieser Arbeit nicht vorgestellt.
Bei der vorhergehend betrachteten Realisierung der Invariante fällt auf, daß nicht nur ein Trigger über die für die Klasse Kante generierte Tabelle definiert werden muß, sondern über alle Tabellen, bei denen Änderungen zu einer Verletzung der Invariante führen können.
Eine mögliche Grundlage, diese Abbildung formal beherrschen zu können, wäre die Darstellung der Funktionalität von TRIGGERn mit Mitteln der OCL.

11 Zusammenfassung und Ausblick

Im Kapitel 2 dieser Arbeit wurde die OCL vorgestellt. Dabei wurde eine Darstellungstechnik erarbeitet, welche sich für den Vergleich mit anderen Sprachen und die Abbildung der OCL auf SQL besser eignet, als die in [OCL97] gewählte Darstellungsform. Wie auch in [SchaKort98] dargelegt wird, ist die objektorientierte Zuordnung der Operatoren der OCL zu bestimmten Datentypen nicht sinnvoll, da die Instanzen der Datentypen keinen Zustandsänderungen unterworfen sind und die Operatoren häufig eine Zwischenstellung zwischen den Datentypen einnehmen. Aus diesem Grund wurden die Operatoren von den Datentypen losgelöst und die vielen Formen des Polymorphismus (siehe [SchaKort98]) aus der Darstellung entfernt. Nun war es möglich, die Operatoren wirklich grundlegend zu klassifizieren. In Kapitel 2 wurde die im folgenden nochmals dargestellte, grundlegende Illustration zur Einteilung der OCL-Operatoren in Gruppen eingeführt:

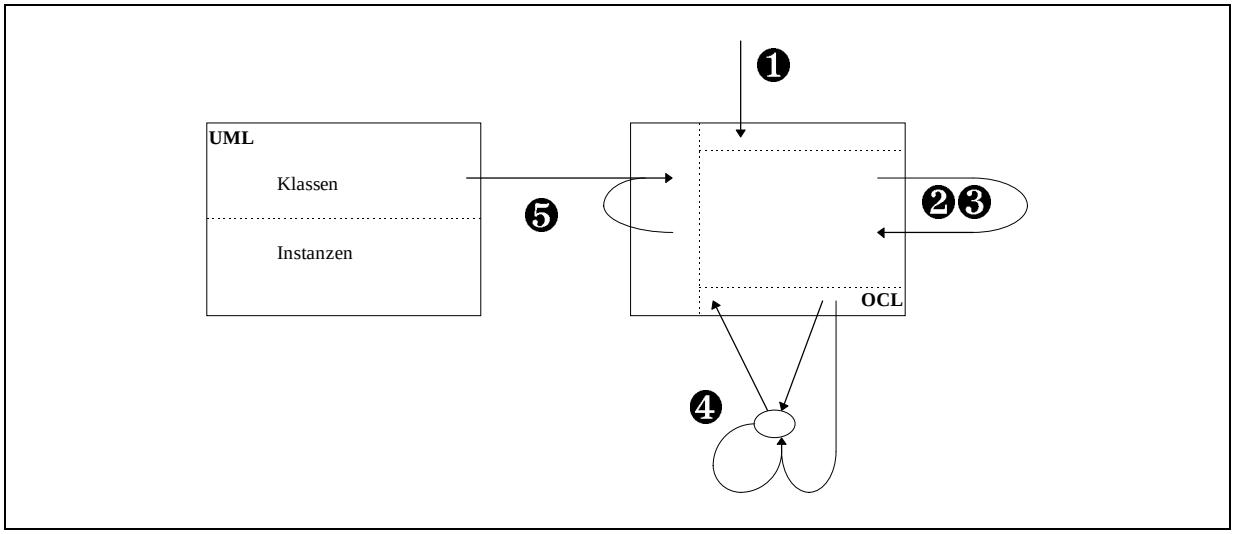


Abbildung 31: Gruppen von OCL-Operatoren

Die Gruppe ❶ umfaßt die Konstanten. Gruppe ❷ enthält die Operatoren, welche nur zwischen Elementardatentypen operieren. In Gruppe ❸ wurden alle Operatoren zusammengefaßt, welche mit den Kollektionstypen in Zusammenhang stehen, mit Ausnahme der Operatoren vom Typ einer Funktion höherer Ordnung, welche in Gruppe ❹ abgesondert wurden. Die Operatoren, welche nicht nur über OCL-Zwischenergebnissen operieren, sondern Daten aus dem Anwendungsmodell bzw. dessen Ausprägung entnehmen, bilden die relativ kleine Gruppe ❺.

Diese in der Natur der OCL-Operatoren begründete Einteilung, spaltet auch die Abbildung von OCL nach SQL in sinnvolle Abschnitte:

Elementare OCL-Operatoren (Gruppen ❶ und ❷)	Die Operatoren der Gruppen ❶ und ❷ lassen sich vollkommen problemlos auf SQL-92 abbilden. Oft ergibt sich sogar syntaktische Identität.
Kollektionsoperatoren (Gruppen ❸ und ❹)	Die Operatoren der Gruppen ❸ und ❹ bringen die wesentlichen Probleme bei der Abbildung mit sich.
„weitere Aspekte der OCL“	Probleme treten ebenfalls bei den „weiteren Aspekten der OCL“ auf. Eine formale Darstellung der Abbildungsmuster wird schon durch die informale Beschreibung in Kapitel 2 erschwert.
Datenbankschema-abhängige Operatoren (Gruppe ❺)	Die Operatoren der Gruppe ❺, welche in Abhängigkeit von der gewählten Abbildung von UML-Klassendiagrammen auf SQL immer unterschiedlich abgebildet werden müssen, riefen mit der gewählten Abbildung keine Probleme hervor.

Es ist damit gelungen, den für praktische Anwendungen anzupassenden Teil der Arbeit (Gruppe ❺) von den nicht anzupassenden Teilen (Gruppen ❶ bis ❹) und den wichtigsten Problemen bei der Abbildung (Gruppen ❸ und ❹) zu isolieren.

11.1 Eignung der Notationen und Techniken für den Datenbank-Entwurf

Der Vergleich der Notationen von OCL, ONN und NE zeigte, daß für die Formulierung von Navigationsausdrücken in etwa gleichmächtige Konzepte angeboten werden. Der Artikel [HHK] stellt dabei jedoch keine eigenständige Notation vor, sondern hat vor allem die Präzisierung der Semantik von Navigationsausdrücken im Allgemeinen zum Ziel.

Im weiteren Verlauf der Arbeit wurde gezeigt, daß UML-Klassendiagramme mit in OCL formulierten Invarianten dazu geeignet sind, vollständige relationale Datenbankschemata mit deskriptiven Integritätsbedingungen abstrakt zu beschreiben. Dabei übersteigt die Ausdrucksmächtigkeit der OCL die Möglichkeiten von SQL-92 und vor allem der SQL-Implementierungen.

Um eine möglichst hohe Performance zu erreichen, werden in der Praxis jedoch deskriptive Integritätsbedingungen nur teilweise angewendet (bzw. von SQL-Implementierungen nur teilweise realisiert). Eine Abbildung von deskriptiven OCL-Invarianten auf die in der Praxis häufig eingesetzten prozeduralen Techniken (Trigger-Aktionen, Integritätssicherung über Stored Procedures) wäre nötig, um den Anforderungen der Praxis zu entsprechen. Diese könnten mit einer Technik, ähnlich dem Funktionalen Modell der OMT (ONN; [BlahPrem98]), auch direkt beschrieben werden (unter Verlust von Abstraktion). Dazu wäre jedoch eine formale Fundierung und Präzisierung der Technik zu Beschreibung des Funktionalen Modells nötig, welches im Gegensatz zur OCL nicht nur Mängel in der formalen Darstellung besitzt, sondern vollkommen informal beschrieben wird. Aus diesem Grunde würde eine Erweiterung der OCL um Konstrukte zur Datenmanipulation (was die Natur der Sprache vollkommen verändern würde) unter Umständen mit weniger Aufwand zum selben Ziel führen.
Mit einer solchen Erweiterung wäre es möglich, die damit beschriebenen Methoden auf Datenbankprozeduren abzubilden oder eine Überprüfung der Methoden auf Einhaltung von Invarianten und Nachbedingungen durchzuführen.

Eine Abbildung von in OCL formulierten Bedingungen in UML-Diagrammen über die Invarianten hinaus (Vor- und Nachbedingungen, Wächter- und Änderungsbedingungen) würde einen deutlich erhöhten Aufwand bzw. Erweiterungen der OCL erfordern.

11.2 Prototypische Implementierungen

Die Abbildung von UML-Klassendiagrammen, wie in Kapitel 8 beschrieben, konnte mit geringem Aufwand und in zufriedenstellender Qualität mit der Scriptsprache von Rational Rose automatisiert werden (siehe Anhang C). Diese Implementierung konnte auch zur Überprüfung dieser Abbildung sinnvoll eingesetzt werden.

Die Abbildung von OCL auf SQL wurde aus mehreren Gründen nicht prototypisch implementiert. Zum einen ist der OCL-Parser von IBM, welcher zugrundegelegt werden sollte, noch nicht ausgereift. Aussagen zur Unvollständigkeit dieses Parsers finden sich in seiner Dokumentation ([OCL-PARSER]). Diese Unvollständigkeit hat ihre Ursachen auch in der unvollständigen Definition der OCL, welche in [SchaKort98] kritisiert wird. Da die Typüberprüfung vom OCL-Parser nicht vollständig realisiert werden konnte, wurde davon ausgegangen, daß die genannten Probleme zu großen Schwierigkeiten bei einer Implementierung der Abbildung von OCL nach SQL geführt hätten.
Aus diesem Grund wurde der Schwerpunkt auf eine Verbesserung der beschriebenen Abbildung und der Darstellungstechniken gelegt. Ein großer Teil der einzelnen Abbildungsmuster wurde an Beispielen überprüft. Auf der beiliegenden CD befindet sich die Datei `"Zwischenversion\AbbildungOclSqlMitBsp.063.doc"` (118 Seiten) mit einer detaillierteren Abbildung und vielen Beispielen. Diese zu unübersichtliche Darstellung wurde in der nun vorliegenden Fassung der Diplomarbeit überarbeitet, so daß sich eine abstraktere Darstellung ohne Beispiele ergab.
Wie die Beispiele in Kapitel 10 zeigen, ist eine direkte automatische Anwendung der Abbildungsmuster nicht besonders zweckmäßig. Die damit erzielte Qualität der Abbildung wäre wesentlich schlechter, als eine von einer natürlichen Intelligenz durchgeführte Abbildung, welche unter Einbeziehung von Nebeninformationen durchgeführt wird (Kenntnis über Alternativen, den semantischen Hintergrund des Anwendungsmodells und der Invarianten sowie über Qualitätsmaßstäbe für die Abbildungsergebnisse). Um qualitativ hochwertige Ergebnisse zu erzielen, müßte eine Reihe weiterer alternativer Abbildungsmuster eingeführt werden (Wie in Kapitel 10 angedeutet). Ein Werkzeug zur Abbildung müßte in der Lage sein, Umformungen gemäß den

Äquivalenzbedingungen der OCL (siehe Kapitel 2) und der Relationenalgebra vorzunehmen und die beste Alternative mit einer Bewertungsfunktion auszuwählen. Nur so könnten Abbildungsergebnisse von brauchbarer Qualität erreicht werden.

11.3 Grundlegende Erkenntnisse

Im diesem Abschnitt soll versucht werden, die wesentlichen Erkenntnisse aus dieser Arbeit und Anregungen für folgende Arbeiten in Form von kurzen Textpassagen zusammenzufassen.

Die Beschreibung der OCL in [OCL97] ist nicht ausreichend.

Die Tatsache, daß die OCL in [OCL97] auf nur 32 Seiten beschrieben wird, könnte die Vermutung nahelegen, daß es sich bei der OCL um eine kleine und wenig mächtige Sprache handelt. Der geringe Umfang von [OCL97] ergibt sich jedoch vor allem daraus, daß wichtige Aspekte nur verbal und unvollständig beschrieben werden, wie auch in [SchaKort98] dargelegt wurde.

Die OCL ist eine mächtige Sprache.

Die Betrachtung der OCL im Kapitel 2 in dieser Arbeit zeigt jedoch, daß die OCL eine mächtige Sprache ist. Die OCL ist eine reine Ausdruckssprache und somit nicht für Datenmanipulation und Beschreibung von Algorithmen vorgesehen. Es ist aber möglich, mit der OCL sehr komplexe Berechnungen zu formulieren. Neben Boolescher Algebra, den Algebren der ganzen und reellen Zahlen sowie der Mengenalgebra (Kollektionen) und Zeichenkettenoperationen definiert die OCL vor allem auch den bedingten Ausdruck (`OCL-COND`), die freie Iteration (`OCL-ITERATE`) und die Konstruktion von Intervallkollektionen aus den Intervallgrenzen (`OCLH-INTRVAL`). Mit dem bedingten Ausdruck ist es möglich, viele Berechnungen durchzuführen, welche in prozeduralen Sprachen durch die Verzweigung realisiert werden. Die freie Iteration deckt einen großen Teil der Möglichkeiten von while-Schleifen ab. In Verbindung mit der Konstruktion von Intervallkollektionen können durch die freie Iteration auch for-Schleifen nachgebildet werden. Die freie Iteration selbst basiert auf dem Konzept der Funktionen höherer Ordnung.

Die Mächtigkeit der OCL macht Abstraktionsmechanismen erforderlich.

Die im Mittelpunkt dieser Arbeit stehende Abbildung von OCL nach SQL ließ sich nur in genügender Qualität beschreiben, indem die OCL in Module zerlegt wurde. Durch die Einteilung der Operatoren in 5 Gruppen wurde dies erreicht. Es konnte somit gezeigt werden, daß die Probleme bei der Abbildung von OCL auf SQL nicht durch das Herunterbrechen der objektorientierten Strukturen auf Tabellenstrukturen verursacht werden, sondern in der gegenüber der OCL geringeren Berechnungskraft von SQL, welche für die Formulierung von Integritätsbedingungen benötigt wird.

Ein Modellierungszwischenschritt zwischen UML/OCL und SQL wäre nützlich gewesen.

Daß die Nachbildung komplexer objektorientierter Strukturen wie Generalisierung und bidirektionale Assoziationen durch Relationen kein Problem darstellt, hätte auch gezeigt werden können, wenn zunächst eine Abbildung von UML-Klassendiagrammen und OCL auf Relationen und eine geeignete (mathematische) Notation für die Formulierung von Integritätsbedingungen (Obermenge der Relationenalgebra) vorgenommen worden wäre. Die Probleme wären dann erst bei der Abbildung des Zwischenmodells auf SQL aufgetreten und hätten dann noch eingehender analysiert werden können.

Die Abbildung der OCL auf SQL-92 ist weitestgehend möglich.

Da die OCL eine reine Ausdruckssprache ist, war es weitestgehend möglich, die Operatoren der OCL auf die deskriptiven Elemente von SQL-92 abzubilden. Nur die freie Iteration (`OCL-ITERATE`) und die Konstruktion von Intervallkollektionen (`OCLH-INTRVAL`) überfordern die Möglichkeiten von SQL-92. Diese können mit den prozeduralen Erweiterungen von SQL3 (PSMs) oder kommerziellen RDBMS nachgebildet werden. Die Formulierung von deskriptiven Integritätsbedingungen ist damit jedoch nicht möglich. Der Nutzer bzw. die Anwendungsprogramme müssen die auf Datenbankprozeduren abgebildeten Integriätsprüfungen dann selbst aufrufen.

Die Abbildung auf Sybase™ ist problematisch.

Da das Datenbanksystem Sybase™ 11 ebenfalls über prozedurale Erweiterungen verfügt, kann natürlich die OCL vollständig auf den SQL-Dialekt von Sybase™ (Transact SQL) abgebildet werden. Da Sybase™ jedoch einige deskriptive Elemente von SQL-92 nicht implementiert (z.B. CASE-WHEN-THEN-ELSE-END und INTERSECT) und die Kombinationsmöglichkeiten der Teilanfragen stark beschränkt sind (vor allem keine Subquery in der FROM-Klausel), muß die Abbildung weitestgehend auf die prozeduralen Erweiterungen vorgenommen werden (*Fast alle* Abbildungsmuster der Gruppen ❻ und ❹ setzten den ersten Parameter in die FROM-Klausel ein!).

Es existieren Probleme mit prozeduralen Abbildungsmustern.

Es ist in dieser Arbeit nicht gelungen, eine übersichtliche und vollständig formale Darstellung für die prozeduralen Abbildungsmuster zu finden. Selbst wenn eine solche Darstellungstechnik gefunden worden wäre, bleiben die grundsätzlichen Probleme mit den Ergebnissen prozeduraler Abbildungsmuster bestehen. Neben der bereits beschriebenen Eigenschaft, daß eine Verwendung für deskriptive Integritätsbedingungen nicht möglich ist, zeichnen sich diese Ergebnisse dadurch aus, daß der SQL-Code schwer zu überblicken ist und vom Datenbanksystem nicht optimiert werden kann.

11.4 Weiterführende Untersuchungen

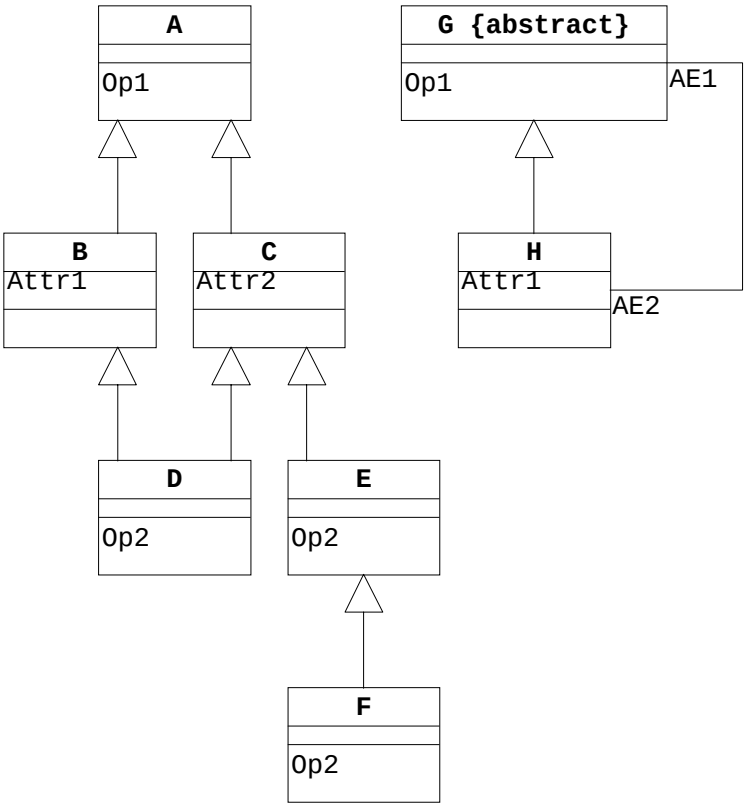
Ein interessanter Ansatz wäre eine Erweiterung der OCL um Datenmanipulationsmöglichkeiten bzw. eine exaktere formale Darstellung der funktionalen Modells der OMT (ONN; [BlahPrem98]). Eine Abbildung der damit beschriebenen Methoden auf Datenbankprozeduren oder eine Überprüfung der Methoden auf Einhaltung von Invarianten und Nachbedingungen wären damit realisierbar.

Bei objektorientierten CASE-Werkzeugen wie Rational Rose™ oder Select Enterprise™ könnten mit einer solchen erweiterten Sprache im letzten Entwurfsschritt die Methodenkörper beschrieben werden. Damit würde die Möglichkeit des Round-Trip-Engineerings für Anwendungen auf der Basis von Hostsprachen wie C++ und relationalen Datenbanksystemen mit prozeduralen Erweiterungen eröffnet werden. Das relationale Paradigma könnte gegenüber dem Entwickler vollständig versteckt werden. Die Automatisierung der Vermittlung zwischen den Paradigmen wäre also möglich.

Anhang A

Dieser Anhang zeigt an einem Beispiel, wie für ein UML-Anwendungsmodell die Meta-Daten für die SQL-Datenbank generiert werden.

Das folgende Beispiel-Anwendungsmodell wird zu Grunde gelegt:



Für jedes Anwendungsmodell werden die Tabellen für die Metainformationen in gleicher Weise verwendet:

```
CREATE TABLE MetaAnwendungsmodellKlassen
(
  name VARCHAR(64) NOT NULL PRIMARY KEY,
  abstrakt BIT NOT NULL
)
go

CREATE TABLE MetaGeneralisierungen
(
  Elter VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  Kind VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  Schritte INTEGER NOT NULL
)
go

CREATE TABLE MetaAttribute
(
  klasse VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  name VARCHAR(64) NOT NULL
)
go

CREATE TABLE MetaAssoziationsenden
(
  klasse VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
```

```
  name VARCHAR(64) NOT NULL
)
go

CREATE TABLE MetaOperationen
(
  klasse VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  name VARCHAR(64) NOT NULL
)
go
```

Für das Beispielmodell müssen jetzt die Metadaten in die vorbereiteten Tabellen eingetragen werden:

```
INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘D’, 0)
go

INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘E’, 0)
go

INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘F’, 0)
go

INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘G’, 1)
go

INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘H’, 0)
go

INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘A’, 0)
go

INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘B’, 0)
go

INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘C’, 0)
go


INSERT INTO MetaGeneralisierungen VALUES ('D', 'D',  0 )
go

INSERT INTO MetaGeneralisierungen VALUES ('B', 'D',  1 )
go

INSERT INTO MetaGeneralisierungen VALUES ('A', 'B',  2 )
go

INSERT INTO MetaGeneralisierungen VALUES ('C', 'D',  1 )
go

INSERT INTO MetaGeneralisierungen VALUES ('A', 'C',  2 )
go

INSERT INTO MetaGeneralisierungen VALUES ('E', 'E',  0 )
go

INSERT INTO MetaGeneralisierungen VALUES ('C', 'E',  1 )
go

INSERT INTO MetaGeneralisierungen VALUES ('A', 'C',  2 )
go

INSERT INTO MetaGeneralisierungen VALUES ('F', 'F',  0 )
go
```

```
INSERT INTO MetaGeneralisierungen VALUES ('E', 'F',  1 )
go

INSERT INTO MetaGeneralisierungen VALUES ('C', 'E',  2 )
go

INSERT INTO MetaGeneralisierungen VALUES ('A', 'C',  3 )
go

INSERT INTO MetaGeneralisierungen VALUES ('G', 'G',  0 )
go

INSERT INTO MetaGeneralisierungen VALUES ('H', 'H',  0 )
go

INSERT INTO MetaGeneralisierungen VALUES ('G', 'H',  1 )
go

INSERT INTO MetaGeneralisierungen VALUES ('A', 'A',  0 )
go

INSERT INTO MetaGeneralisierungen VALUES ('B', 'B',  0 )
go

INSERT INTO MetaGeneralisierungen VALUES ('A', 'B',  1 )
go

INSERT INTO MetaGeneralisierungen VALUES ('C', 'C',  0 )
go

INSERT INTO MetaGeneralisierungen VALUES ('A', 'C',  1 )
go

INSERT INTO MetaAttribute VALUES ('H', 'Attr1')
go

INSERT INTO MetaAttribute VALUES ('B', 'Attr1')
go

INSERT INTO MetaAttribute VALUES ('C', 'Attr2')
go

INSERT INTO MetaAssoziationsenden VALUES ('H', 'AE1')
go

INSERT INTO MetaAssoziationsenden VALUES ('G', 'AE2')
go

INSERT INTO MetaOperationen VALUES ('D', 'Op2')
go

INSERT INTO MetaOperationen VALUES ('E', 'Op2')
go

INSERT INTO MetaOperationen VALUES ('F', 'Op2')
go

INSERT INTO MetaOperationen VALUES ('G', 'Op1')
go

INSERT INTO MetaOperationen VALUES ('A', 'Op1')
go
```

Der SQL-Code wurde mit Rational Rose und dem Rose-Script, welches im Anhang C näher beschrieben wird, erzeugt.

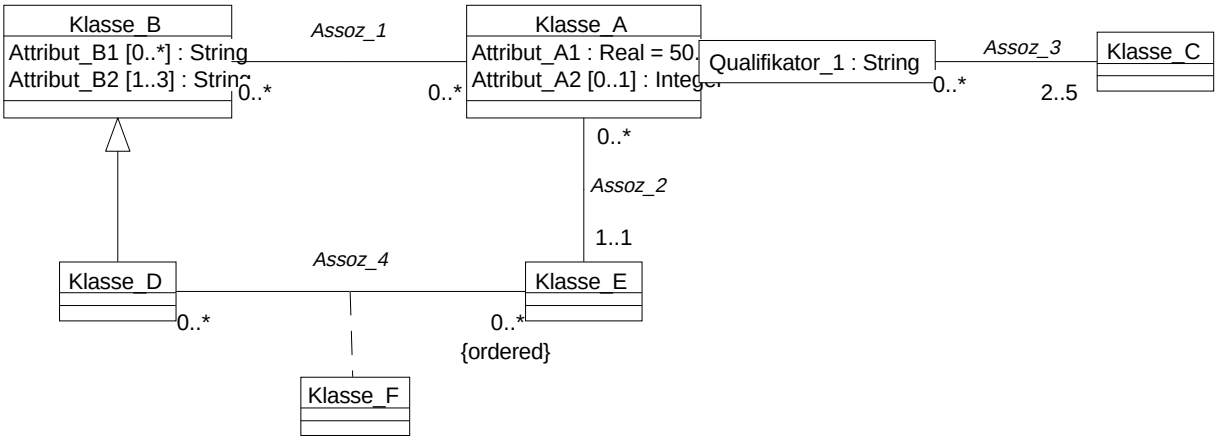
Anhang B

Die in Abschnitt 8.2 beschriebene Abbildung soll an einem Beispiel demonstriert und in den problematischsten Bereichen geprüft werden. Dazu wird zunächst ein abstrakter Diskursbereich beschrieben. Anschließend wird die Abbildung vorgenommen. Nun können korrekte Beispielinstanzen eingefügt werden. An fehlerhaften Beispieldaten wird abschließend demonstriert, daß deren Ablage in der Datenbank als fehlerhaft erkannt wird und eine Realisierung mit ASSERTIONs nach dem SQL-Standard eine Ablage solcher Daten in der Datenbank somit von Server-Seite aus unterbinden würde.

B.1 Abstrakter Diskursbereich als Beispiel

Als Beispiel wurde ein abstrakter Diskursbereich gewählt. Der entscheidende Grund dafür ist, daß kein realer Diskursbereich gefunden werden konnte, welcher alle wesentlichen Konstrukte genau einmal enthält. Zum anderen könnten abweichende persönlichen Standpunkte des Lesers zu einem realen Diskursbereich von der eigentlichen Zielstellung, der Überprüfung der Korrektheit der Abbildung, ablenken.

Der abstrakte Diskursbereich ist in folgendem Klassendiagramm dargestellt:



Aus Umfangsgründen wird auf die Überprüfung der n-ären Assoziation verzichtet.

B.2 Abbildung des abstrakten Diskursbereichs auf SQL-DDL

Im folgenden sind die Ergebnisse der Abbildung des Beispielmodells auf SQL-DDL nach dem beschriebenen Verfahren aufgeführt. Die Abbildung wurde mit Rational Rose und einem Rose-Script, welches in Anhang C näher beschrieben wird, vorgenommen.

```
CREATE TABLE MetaAnwendungsmodellKlassen
(
  name VARCHAR(64) NOT NULL PRIMARY KEY,
  abstrakt BIT NOT NULL
)
go

CREATE TABLE MetaGeneralisierungen
(
  Elter VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  Kind VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  Schritte INTEGER NOT NULL
)
go

CREATE TABLE MetaAttribute
(
  klasse VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  name VARCHAR(64) NOT NULL
)
go

CREATE TABLE MetaAssoziationsenden
(
  klasse VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  name VARCHAR(64) NOT NULL
)
go

CREATE TABLE MetaOperationen
(
  klasse VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  name VARCHAR(64) NOT NULL
)
go

INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘Klasse_F’, 0)
go

INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘Klasse_A’, 0)
go

INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘Klasse_B’, 0)
go

INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘Klasse_C’, 0)
go

INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘Klasse_D’, 0)
go

INSERT INTO MetaAnwendungsmodellKlassen VALUES (‘Klasse_E’, 0)
go


INSERT INTO MetaGeneralisierungen VALUES ('Klasse_F', 'Klasse_F',  0 )
go

INSERT INTO MetaGeneralisierungen VALUES ('Klasse_A', 'Klasse_A',  0 )
go

INSERT INTO MetaGeneralisierungen VALUES ('Klasse_B', 'Klasse_B',  0 )
go

INSERT INTO MetaGeneralisierungen VALUES ('Klasse_C', 'Klasse_C',  0 )
```

```
go

INSERT INTO MetaGeneralisierungen VALUES ('Klasse_D', 'Klasse_D',  0 )
go

INSERT INTO MetaGeneralisierungen VALUES ('Klasse_B', 'Klasse_D',  1 )
go

INSERT INTO MetaGeneralisierungen VALUES ('Klasse_E', 'Klasse_E',  0 )
go

INSERT INTO MetaAttribute VALUES ('Klasse_A', 'Attribut_A1')
go

INSERT INTO MetaAttribute VALUES ('Klasse_A', 'Attribut_A2')
go

INSERT INTO MetaAttribute VALUES ('Klasse_B', 'Attribut_B1')
go

INSERT INTO MetaAttribute VALUES ('Klasse_B', 'Attribut_B2')
go

INSERT INTO MetaAssoziationsenden VALUES ('Klasse_A', 'klasse_B')
go

INSERT INTO MetaAssoziationsenden VALUES ('Klasse_B', 'klasse_A')
go

INSERT INTO MetaAssoziationsenden VALUES ('Klasse_E', 'klasse_A')
go

INSERT INTO MetaAssoziationsenden VALUES ('Klasse_A', 'klasse_E')
go

INSERT INTO MetaAssoziationsenden VALUES ('Klasse_A', 'klasse_C')
go

INSERT INTO MetaAssoziationsenden VALUES ('Klasse_C', 'klasse_A')
go

INSERT INTO MetaAssoziationsenden VALUES ('Klasse_E', 'klasse_D')
go

INSERT INTO MetaAssoziationsenden VALUES ('Klasse_D', 'klasse_E')
go

CREATE TABLE PersistenteObjekte
(
  oid INTEGER NOT NULL PRIMARY KEY,  -- Objektidentifikator
  klasse VARCHAR(64) NOT NULL REFERENCES MetaAnwendungsmodellKlassen,
  name VARCHAR(64) NULL              -- optionaler Objekt-Name
)
go

CREATE PROCEDURE ErmittleFreieOID (@neueOid INTEGER OUTPUT)
  AS
    IF ((SELECT COUNT(*) FROM PersistenteObjekte) < 1)
    BEGIN
      SELECT @neueOid = 1
    END
    ELSE
    BEGIN
      SELECT @neueOid = MAX(oid)+1
      FROM PersistenteObjekte
    END
go

CREATE TRIGGER KonstanteOIDuKlasse
  ON PersistenteObjekte
  FOR UPDATE AS
  IF UPDATE(oid)
  BEGIN
    ROLLBACK TRANSACTION
```

```
    RAISERROR 17011
        'Fehler: Aenderung des Objektidentifikators (OID) ist unzulaessig!'
  END
  IF UPDATE(klasse)
  BEGIN
    ROLLBACK TRANSACTION
    RAISERROR 17012
        'Fehler: Aenderung der Klassenzugehoerigkeit ist unzulaessig!'
  END
go

CREATE TABLE Klasse_A_Attribut_A1
(
  oid INTEGER NOT NULL REFERENCES PersistenteObjekte(oid),
  Attribut_A1 REAL NOT NULL
)
go

CREATE TRIGGER KonstanteOID_Klasse_A_Attribut_A1
  ON Klasse_A_Attribut_A1
  FOR UPDATE AS
  IF UPDATE(oid)
  BEGIN
    ROLLBACK TRANSACTION
    RAISERROR 17021 'Fehler: Aenderung der Objektzugehoerigkeit eines Attributs
ist unzulaessig!'
  END
go

CREATE PROCEDURE Pruefe_Klasse_A_Attribut_A1
  AS
  BEGIN
    -- Attributwerte duerfen nur Objekten zugeordnet werden, welche dieses
    -- Attribut entsprechend ihrer Klassenzugehoerigkeit auch besitzen
    -- koennen.
    IF EXISTS
      (SELECT p.oid
       FROM Klasse_A_Attribut_A1 a, PersistenteObjekte p
       WHERE a.oid=p.oid AND klasse NOT IN (SELECT Kind
                                            FROM MetaGeneralisierungen
                                            WHERE Elter = 'Klasse_A'))
    BEGIN
      RAISERROR 17022 'Attribute zu falschen Klassen zugeordnet!'
    END

    -- Keinem Objekt duerfen weniger Attributwerte zugeordnet werden, als
    -- die MUG spezifiziert.
    IF EXISTS
      (SELECT p.oid
       FROM PersistenteObjekte p, Klasse_A_Attribut_A1 a
       WHERE p.klasse IN (SELECT Kind
                          FROM MetaGeneralisierungen
                          WHERE Elter = 'Klasse_A')
         AND p.oid *= a.oid
       GROUP BY p.oid
       HAVING COUNT(a.oid) < 1)
    BEGIN
      RAISERROR 17023 'Attribute verletzen MUG!'
    END

    -- Keinem Objekt duerfen mehr Attributwerte zugeordnet werden, als
    -- die MOG spezifiziert.
    IF EXISTS
      (SELECT oid
       FROM Klasse_A_Attribut_A1
       GROUP BY oid
       HAVING COUNT(*) > 1)
    BEGIN
      RAISERROR 17024 'Attribute verletzen MOG!'
    END
END
go

CREATE TABLE Klasse_A_Attribut_A2
```

```
(
  oid INTEGER NOT NULL REFERENCES PersistenteObjekte(oid),
  Attribut_A2 INTEGER NOT NULL
)
go

CREATE TRIGGER KonstanteOID_Klasse_A_Attribut_A2
  ON Klasse_A_Attribut_A2
  FOR UPDATE AS
  IF UPDATE(oid)
  BEGIN
    ROLLBACK TRANSACTION
    RAISERROR 17021 'Fehler: Aenderung der Objektzugehoerigkeit eines Attributs
ist unzulaessig!'
  END
go

CREATE PROCEDURE Pruefe_Klasse_A_Attribut_A2
  AS
  BEGIN
    -- Attributwerte duerfen nur Objekten zugeordnet werden, welche dieses
    -- Attribut entsprechend ihrer Klassenzugehoerigkeit auch besitzen
    -- koennen.
    IF EXISTS
      (SELECT p.oid
       FROM Klasse_A_Attribut_A2 a, PersistenteObjekte p
       WHERE a.oid=p.oid AND klasse NOT IN (SELECT Kind
                                           FROM MetaGeneralisierungen
                                           WHERE Elter = 'Klasse_A'))
    BEGIN
      RAISERROR 17022 'Attribute zu falschen Klassen zugeordnet!'
    END

    -- Keinem Objekt duerfen mehr Attributwerte zugeordnet werden, als
    -- die MOG spezifiziert.
    IF EXISTS
      (SELECT oid
       FROM Klasse_A_Attribut_A2
       GROUP BY oid
       HAVING COUNT(*) > 1)
    BEGIN
      RAISERROR 17024 'Attribute verletzen MOG!'
    END
END
go

CREATE TABLE Klasse_B_Attribut_B1
(
  oid INTEGER NOT NULL REFERENCES PersistenteObjekte(oid),
  Attribut_B1 VARCHAR(200) NOT NULL
)
go

CREATE TRIGGER KonstanteOID_Klasse_B_Attribut_B1
  ON Klasse_B_Attribut_B1
  FOR UPDATE AS
  IF UPDATE(oid)
  BEGIN
    ROLLBACK TRANSACTION
    RAISERROR 17021 'Fehler: Aenderung der Objektzugehoerigkeit eines Attributs
ist unzulaessig!'
  END
go

CREATE PROCEDURE Pruefe_Klasse_B_Attribut_B1
  AS
  BEGIN
    -- Attributwerte duerfen nur Objekten zugeordnet werden, welche dieses
    -- Attribut entsprechend ihrer Klassenzugehoerigkeit auch besitzen
    -- koennen.
    IF EXISTS
      (SELECT p.oid
       FROM Klasse_B_Attribut_B1 a, PersistenteObjekte p
       WHERE a.oid=p.oid AND klasse NOT IN (SELECT Kind
```

```
                                        FROM MetaGeneralisierungen
                                        WHERE Elter = 'Klasse_B'))
    BEGIN
      RAISERROR 17022 'Attribute zu falschen Klassen zugeordnet!'
    END
END
go

CREATE TABLE Klasse_B_Attribut_B2
(
  oid INTEGER NOT NULL REFERENCES PersistenteObjekte(oid),
  Attribut_B2 VARCHAR(200) NOT NULL
)
go

CREATE TRIGGER KonstanteOID_Klasse_B_Attribut_B2
  ON Klasse_B_Attribut_B2
  FOR UPDATE AS
  IF UPDATE(oid)
  BEGIN
    ROLLBACK TRANSACTION
    RAISERROR 17021 'Fehler: Aenderung der Objektzugehoerigkeit eines Attributs
ist unzulaessig!'
  END
go

CREATE PROCEDURE Pruefe_Klasse_B_Attribut_B2
  AS
  BEGIN
    -- Attributwerte duerfen nur Objekten zugeordnet werden, welche dieses
    -- Attribut entsprechend ihrer Klassenzugehoerigkeit auch besitzen
    -- koennen.
    IF EXISTS
      (SELECT p.oid
       FROM Klasse_B_Attribut_B2 a, PersistenteObjekte p
       WHERE a.oid=p.oid AND klasse NOT IN (SELECT Kind
                                        FROM MetaGeneralisierungen
                                        WHERE Elter = 'Klasse_B'))
    BEGIN
      RAISERROR 17022 'Attribute zu falschen Klassen zugeordnet!'
    END

    -- Keinem Objekt duerfen weniger Attributwerte zugeordnet werden, als
    -- die MUG spezifiziert.
    IF EXISTS
      (SELECT p.oid
       FROM PersistenteObjekte p, Klasse_B_Attribut_B2 a
       WHERE p.klasse IN (SELECT Kind
                          FROM MetaGeneralisierungen
                          WHERE Elter = 'Klasse_B')
         AND p.oid *= a.oid
       GROUP BY p.oid
       HAVING COUNT(a.oid) < 1)
    BEGIN
      RAISERROR 17023 'Attribute verletzen MUG!'
    END

    -- Keinem Objekt duerfen mehr Attributwerte zugeordnet werden, als
    -- die MOG spezifiziert.
    IF EXISTS
      (SELECT oid
       FROM Klasse_B_Attribut_B2
       GROUP BY oid
       HAVING COUNT(*) > 3)
    BEGIN
      RAISERROR 17024 'Attribute verletzen MOG!'
    END
END
go

CREATE TRIGGER AbstrakteKlassenUndDefaults
  ON PersistenteObjekte
  FOR INSERT AS
  IF EXISTS (SELECT oid
```

```
            FROM INSERTED i, MetaAnwendungsmodellKlassen m
            WHERE i.klasse = m.name AND m.abstrakt = 1)
  BEGIN
    ROLLBACK TRANSACTION
    RAISERROR 17014
        'Fehler: Instanzen für abstrakte Klassen unzulaessig!'
  END
    INSERT INTO Klasse_A_Attribut_A1
    SELECT oid, 50.1
    FROM inserted
    WHERE klasse IN (SELECT Kind
                     FROM MetaGeneralisierungen
                     WHERE Elter = 'Klasse_A')
go

CREATE TABLE Assoz_1
(
  klasse_A INTEGER NOT NULL REFERENCES PersistenteObjekte,
  klasse_B INTEGER NOT NULL REFERENCES PersistenteObjekte,
  UNIQUE(klasse_A, klasse_B)
)
go

CREATE PROCEDURE Pruefe_Assoz_1_klasse_A
  AS
  BEGIN
    -- Die in der Assoziationstabelle referenzierten Objekte muessen der
    -- Klasse, fuer welche die Rolle definiert ist oder einer ihrer
    -- Kindklassen angehoeren.
    IF EXISTS
      (SELECT oid
       FROM Assoz_1 a, PersistenteObjekte p
       WHERE a.klasse_A=p.oid
         AND klasse NOT IN (SELECT Kind
                            FROM MetaGeneralisierungen
                            WHERE Elter = 'Klasse_A'))
    BEGIN
      RAISERROR 17003 'Rollen zu falschen Klassen zugeordnet!'
    END
END
go

CREATE PROCEDURE Pruefe_Assoz_1_klasse_B
  AS
  BEGIN
    -- Die in der Assoziationstabelle referenzierten Objekte muessen der
    -- Klasse, fuer welche die Rolle definiert ist oder einer ihrer
    -- Kindklassen angehoeren.
    IF EXISTS
      (SELECT oid
       FROM Assoz_1 a, PersistenteObjekte p
       WHERE a.klasse_B=p.oid
         AND klasse NOT IN (SELECT Kind
                            FROM MetaGeneralisierungen
                            WHERE Elter = 'Klasse_B'))
    BEGIN
      RAISERROR 17003 'Rollen zu falschen Klassen zugeordnet!'
    END
END
go

CREATE PROCEDURE Pruefe_Assoz_1
  AS
  BEGIN
    EXECUTE Pruefe_Assoz_1_klasse_A
    EXECUTE Pruefe_Assoz_1_klasse_B
  END
go

CREATE TABLE Assoz_2
(
  klasse_E INTEGER NOT NULL REFERENCES PersistenteObjekte,
  klasse_A INTEGER NOT NULL REFERENCES PersistenteObjekte,
  UNIQUE(klasse_E, klasse_A)
```

```
)
go

CREATE PROCEDURE Pruefe_Assoz_2_klasse_E
  AS
  BEGIN
    -- Die in der Assoziationstabelle referenzierten Objekte muessen der
    -- Klasse, fuer welche die Rolle definiert ist oder einer ihrer
    -- Kindklassen angehoeren.
    IF EXISTS
      (SELECT oid
       FROM Assoz_2 a, PersistenteObjekte p
       WHERE a.klasse_E=p.oid
         AND klasse NOT IN (SELECT Kind
                            FROM MetaGeneralisierungen
                            WHERE Elter = 'Klasse_E'))
    BEGIN
      RAISERROR 17003 'Rollen zu falschen Klassen zugeordnet!'
    END

    -- Keinem Objekt duerfen weniger Objekte der Zielrolle zugeordnet
    -- werden, als die MUG spezifiziert.
    IF EXISTS
      (SELECT p.oid
       FROM PersistenteObjekte p, Assoz_2 a
       WHERE p.klasse IN (SELECT Kind
                          FROM MetaGeneralisierungen
                          WHERE Elter = 'Klasse_A')
         AND p.oid *= a.klasse_A
       GROUP BY p.oid
       HAVING COUNT(a.klasse_E) < 1)
    BEGIN
      RAISERROR 17004 'Rollen verletzen MUG!'
    END

    -- Keinem Objekt duerfen mehr Objekte der Zielrolle zugeordnet
    -- werden, als die MOG spezifiziert.
    IF EXISTS
      (SELECT klasse_A
       FROM Assoz_2
       GROUP BY klasse_A
       HAVING COUNT(*) > 1)
    BEGIN
      RAISERROR 17005 'Rollen verletzen MOG!'
    END
END
go

CREATE PROCEDURE Pruefe_Assoz_2_klasse_A
  AS
  BEGIN
    -- Die in der Assoziationstabelle referenzierten Objekte muessen der
    -- Klasse, fuer welche die Rolle definiert ist oder einer ihrer
    -- Kindklassen angehoeren.
    IF EXISTS
      (SELECT oid
       FROM Assoz_2 a, PersistenteObjekte p
       WHERE a.klasse_A=p.oid
         AND klasse NOT IN (SELECT Kind
                            FROM MetaGeneralisierungen
                            WHERE Elter = 'Klasse_A'))
    BEGIN
      RAISERROR 17003 'Rollen zu falschen Klassen zugeordnet!'
    END
END
go

CREATE PROCEDURE Pruefe_Assoz_2
  AS
  BEGIN
    EXECUTE Pruefe_Assoz_2_klasse_E
    EXECUTE Pruefe_Assoz_2_klasse_A
  END
go
```

```
CREATE TABLE Assoz_3
(
  klasse_A INTEGER NOT NULL REFERENCES PersistenteObjekte,
  klasse_C INTEGER NOT NULL REFERENCES PersistenteObjekte,
  klasse_C_Qualifikator_1 VARCHAR(200) NOT NULL,
  UNIQUE(klasse_A, klasse_C)
)
go

CREATE PROCEDURE Pruefe_Assoz_3_klasse_A
  AS
  BEGIN
    -- Die in der Assoziationstabelle referenzierten Objekte muessen der
    -- Klasse, fuer welche die Rolle definiert ist oder einer ihrer
    -- Kindklassen angehoeren.
    IF EXISTS
      (SELECT oid
       FROM Assoz_3 a, PersistenteObjekte p
       WHERE a.klasse_A=p.oid
         AND klasse NOT IN (SELECT Kind
                            FROM MetaGeneralisierungen
                            WHERE Elter = 'Klasse_A'))
    BEGIN
      RAISERROR 17003 'Rollen zu falschen Klassen zugeordnet!'
    END
END
go

CREATE PROCEDURE Pruefe_Assoz_3_klasse_C
  AS
  BEGIN
    -- Die in der Assoziationstabelle referenzierten Objekte muessen der
    -- Klasse, fuer welche die Rolle definiert ist oder einer ihrer
    -- Kindklassen angehoeren.
    IF EXISTS
      (SELECT oid
       FROM Assoz_3 a, PersistenteObjekte p
       WHERE a.klasse_C=p.oid
         AND klasse NOT IN (SELECT Kind
                            FROM MetaGeneralisierungen
                            WHERE Elter = 'Klasse_C'))
    BEGIN
      RAISERROR 17003 'Rollen zu falschen Klassen zugeordnet!'
    END

    -- Keinem Objekt duerfen weniger Objekte der Zielrolle zugeordnet
    -- werden, als die MUG spezifiziert.
    IF EXISTS
      (SELECT p.oid
       FROM PersistenteObjekte p, Assoz_3 a
       WHERE p.klasse IN (SELECT Kind
                          FROM MetaGeneralisierungen
                          WHERE Elter = 'Klasse_A')
         AND p.oid *= a.klasse_A
       GROUP BY p.oid, klasse_C_Qualifikator_1
       HAVING COUNT(a.klasse_C) < 2)
    BEGIN
      RAISERROR 17004 'Rollen verletzen MUG!'
    END

    -- Keinem Objekt duerfen mehr Objekte der Zielrolle zugeordnet
    -- werden, als die MOG spezifiziert.
    IF EXISTS
      (SELECT klasse_A
       FROM Assoz_3
       GROUP BY klasse_A, klasse_C_Qualifikator_1
       HAVING COUNT(*) > 5)
    BEGIN
      RAISERROR 17005 'Rollen verletzen MOG!'
    END
END
go
```

```
CREATE PROCEDURE Pruefe_Assoz_3
  AS
  BEGIN
    EXECUTE Pruefe_Assoz_3_klasse_A
    EXECUTE Pruefe_Assoz_3_klasse_C
  END
go

CREATE TABLE Assoz_4
(
Assoziationsklasse INTEGER NOT NULL REFERENCES PersistenteObjekte,
  klasse_E INTEGER NOT NULL REFERENCES PersistenteObjekte,
  klasse_E_snr INTEGER NOT NULL,
  UNIQUE(klasse_D, klasse_E_snr),
  klasse_D INTEGER NOT NULL REFERENCES PersistenteObjekte,
  UNIQUE(klasse_E, klasse_D)
)
go

CREATE PROCEDURE Pruefe_Assoz_4_klasse_E
  AS
  BEGIN
    -- Die in der Assoziationstabelle referenzierten Objekte muessen der
    -- Klasse, fuer welche die Rolle definiert ist oder einer ihrer
    -- Kindklassen angehoeren.
    IF EXISTS
      (SELECT oid
       FROM Assoz_4 a, PersistenteObjekte p
       WHERE a.klasse_E=p.oid
         AND klasse NOT IN (SELECT Kind
                            FROM MetaGeneralisierungen
                            WHERE Elter = 'Klasse_E'))
    BEGIN
      RAISERROR 17003 'Rollen zu falschen Klassen zugeordnet!'
    END

    -- Die Sequenz-Nummern duerfen nur Zahlen von 0..COUNT-1 enthalten.
    IF EXISTS
      (SELECT klasse_D
       FROM Assoz_4
       GROUP BY klasse_D
       HAVING MAX(klasse_E_snr) >= COUNT(klasse_E))
    BEGIN
      RAISERROR 17006 'Listensortierung fehlerhaft!'
    END
END
go

CREATE PROCEDURE Pruefe_Assoz_4_klasse_D
  AS
  BEGIN
    -- Die in der Assoziationstabelle referenzierten Objekte muessen der
    -- Klasse, fuer welche die Rolle definiert ist oder einer ihrer
    -- Kindklassen angehoeren.
    IF EXISTS
      (SELECT oid
       FROM Assoz_4 a, PersistenteObjekte p
       WHERE a.klasse_D=p.oid
         AND klasse NOT IN (SELECT Kind
                            FROM MetaGeneralisierungen
                            WHERE Elter = 'Klasse_D'))
    BEGIN
      RAISERROR 17003 'Rollen zu falschen Klassen zugeordnet!'
    END
END
go

CREATE PROCEDURE Pruefe_Assoz_4
  AS
  BEGIN
    EXECUTE Pruefe_Assoz_4_klasse_E
    EXECUTE Pruefe_Assoz_4_klasse_D

    -- Die in der Assoziationstabelle als Assoziationsobjekt referenzierten
```

```
    -- Objekte muessen der Klasse, welche als Assoziationsklasse definiert
    -- ist oder einer ihrer Kindklassen angehoeren.
    IF EXISTS
      (SELECT oid
       FROM Assoz_4 a, PersistenteObjekte p
       WHERE a.Assoziationsklasse=p.oid
         AND klasse NOT IN (SELECT Kind
                            FROM MetaGeneralisierungen
                            WHERE Elter = 'Klasse_F'))
    BEGIN
      RAISERROR 17003 'Assoziationen zu falschen Assoziationsklassen zugeordnet!'
    END

    -- Links und Linkobjekte muessen eineindeutig zugeordnet sein.
    IF EXISTS
      (SELECT p.oid
       FROM PersistenteObjekte p, Assoz_4 a
       WHERE p.klasse IN (SELECT Kind
                          FROM MetaGeneralisierungen
                          WHERE Elter = 'Klasse_F')
         AND p.oid *= a.Assoziationsklasse
       GROUP BY p.oid
       HAVING COUNT(a.Assoziationsklasse) <> 1)
    BEGIN
      RAISERROR 17004 'Zuordnung Assoziation-Klasse falsch!'
    END
  END
go
```

Die abgebildeten SQL-DDL-Anweisungen wurden im Test erfolgreich ausgeführt.

B.3 Ablage korrekter Daten (Objekte) in der Datenbank

Die hier dargestellten Einfüge-Operationen dienen neben der Überprüfung der beschriebenen Abbildung hauptsächlich der Nachvollziehbarkeit der Überprüfung der beschriebenen Abbildung. Der Leser kann daran Detail-Probleme näher beleuchten, auf eine detaillierte Beschreibung wurde jedoch auf Umfangsgründen verzichtet. Transaktionen fassen Operationen auf Objekten zusammen, eine Anwendungssemantik wird diesen Transaktionen nicht unterstellt. Innerhalb der Transaktionen müssen die beschriebenen Konsistenzbedingungen nicht erfüllt werden, sondern erst an deren Ende.

```
-- Zunächst werden Instanzen der Klassen eingefügt, welche nicht
-- existenziell von anderen Klassen abhängen, d.h. für alle Klassen außer
-- Klasse_A und Klasse_F.

BEGIN TRANSACTION
INSERT INTO PersistenteObjekte
  VALUES (1, 'Klasse_B', 'Instanz_B1')
INSERT INTO Klasse_B_Attribut_B1
  VALUES (1, 'String1')
INSERT INTO Klasse_B_Attribut_B1
  VALUES (1, 'String2')
INSERT INTO Klasse_B_Attribut_B2
  VALUES (1, 'String1')
EXECUTE Pruefe_Klasse_B_Attribut_B1
EXECUTE Pruefe_Klasse_B_Attribut_B2
EXECUTE Pruefe_Assoz_1
COMMIT TRANSACTION
go

 BEGIN TRANSACTION
INSERT INTO PersistenteObjekte
  VALUES (2, 'Klasse_B', 'Instanz_B2')
INSERT INTO Klasse_B_Attribut_B2
  VALUES (2, 'String1')
INSERT INTO Klasse_B_Attribut_B2
```

```
  VALUES (2, 'String2')
EXECUTE Pruefe_Klasse_B_Attribut_B1
EXECUTE Pruefe_Klasse_B_Attribut_B2
EXECUTE Pruefe_Assoz_1
COMMIT TRANSACTION
go

INSERT INTO PersistenteObjekte
  VALUES (3, 'Klasse_C', 'Instanz_C1')
go

INSERT INTO PersistenteObjekte
  VALUES (4, 'Klasse_C', 'Instanz_C2')
go

INSERT INTO PersistenteObjekte
  VALUES (5, 'Klasse_C', 'Instanz_C3')
go

INSERT INTO PersistenteObjekte
  VALUES (6, 'Klasse_C', 'Instanz_C4')
go

INSERT INTO PersistenteObjekte
  VALUES (7, 'Klasse_C', 'Instanz_C5')
go

INSERT INTO PersistenteObjekte
  VALUES (8, 'Klasse_C', 'Instanz_C6')
go

INSERT INTO PersistenteObjekte
  VALUES (9, 'Klasse_C', 'Instanz_C7')
go

INSERT INTO PersistenteObjekte
  VALUES (10, 'Klasse_C', 'Instanz_C8')
go

BEGIN TRANSACTION
INSERT INTO PersistenteObjekte
  VALUES (11, 'Klasse_D', 'Instanz_D1')
INSERT INTO Klasse_B_Attribut_B2
  VALUES (11, 'String1')
EXECUTE Pruefe_Klasse_B_Attribut_B1
EXECUTE Pruefe_Klasse_B_Attribut_B2
EXECUTE Pruefe_Assoz_1
EXECUTE Pruefe_Assoz_4
COMMIT TRANSACTION
go

INSERT INTO PersistenteObjekte
  VALUES (12, 'Klasse_E', 'Instanz_E1')
go

 INSERT INTO PersistenteObjekte
  VALUES (13, 'Klasse_E', 'Instanz_E2')
go

INSERT INTO PersistenteObjekte
  VALUES (14, 'Klasse_E', 'Instanz_E3')
go

INSERT INTO PersistenteObjekte
  VALUES (15, 'Klasse_E', 'Instanz_E4')
go
```

```
INSERT INTO PersistenteObjekte
  VALUES (16, 'Klasse_E', 'Instanz_E5')
go

INSERT INTO PersistenteObjekte
  VALUES (17, 'Klasse_E', 'Instanz_E6')
go

-- Nun wird exemplarisch eine Instanz der Klasse Klasse_A eingefügt und für
-- die notwendigen Verknüpfungen gesorgt.

BEGIN TRANSACTION
INSERT INTO PersistenteObjekte
  VALUES (18, 'Klasse_A', 'Instanz_A1')
INSERT INTO Assoz_3
  VALUES (18, 3, 'Qual1')
INSERT INTO Assoz_3
  VALUES (18, 4, 'Qual1')
INSERT INTO Assoz_3
  VALUES (18, 5, 'Qual1')
INSERT INTO Assoz_3
  VALUES (18, 6, 'Qual2')
INSERT INTO Assoz_3
  VALUES (18, 7, 'Qual2')
INSERT INTO Assoz_2
  VALUES (12, 18)
EXECUTE Pruefe_Klasse_A_Attribut_A1
EXECUTE Pruefe_Klasse_A_Attribut_A2
EXECUTE Pruefe_Assoz_1
EXECUTE Pruefe_Assoz_2
EXECUTE Pruefe_Assoz_3
COMMIT TRANSACTION
go

-- Ebenso wird eine Instanz der Klasse Klasse_F eingefügt.

BEGIN TRANSACTION
INSERT INTO PersistenteObjekte
  VALUES (19, 'Klasse_F', 'Instanz_F1')
INSERT INTO Assoz_4
  VALUES (19, 12, 0, 11)
EXECUTE Pruefe_Assoz_4
COMMIT TRANSACTION
go

-- Instanzen der Klasse Klasse_A können über die Assoziation Assoz_1 auf
-- Grund der Generalisierung zwischen Klasse_B und Klasse_D mit Instanzen
-- beider Klassen verküpft sein, wie im folgenden ausgeführt wird.

BEGIN TRANSACTION
INSERT INTO Assoz_1
  VALUES (18, 1)
EXECUTE Pruefe_Assoz_1
COMMIT TRANSACTION
go

BEGIN TRANSACTION
INSERT INTO Assoz_1
  VALUES (18, 11)
EXECUTE Pruefe_Assoz_1
COMMIT TRANSACTION
go

-- Um sicherzustellen, daß keine Prüf-Prozedur vergessen wurde, werden
-- nochmals alle Prüf-Prozeduren aufgerufen.
```

```
EXECUTE Pruefe_Klasse_A_Attribut_A1
EXECUTE Pruefe_Klasse_A_Attribut_A2
EXECUTE Pruefe_Klasse_B_Attribut_B1
EXECUTE Pruefe_Klasse_B_Attribut_B2
EXECUTE Pruefe_Assoz_1
EXECUTE Pruefe_Assoz_2
EXECUTE Pruefe_Assoz_3
EXECUTE Pruefe_Assoz_4
go
```

Die fehlerfreie Ausführung dieser Anweisungen beweist, daß es fehlerfreie Daten (Objekte) gibt, welche in der Datenbank gespeichert werden können. Dies beweist vor allem die syntaktische Korrektheit. Daß wirklich alle korrekten Daten (unendlich viele Möglichkeiten) gespeichert werden können, kann mit diesem Beispiel natürlich nicht bewiesen werden.

B.4 Auslösen von Fehlermeldungen durch inkonsistente Dateneingabe

In diesem Abschnitt soll die Wirksamkeit der Konsistenzprüfungen demonstriert werden. Durch die geringen Wechselwirkungen der Abbildungskomponenten kann mit großer Wahrscheinlichkeit angenommen werden, daß alle derartigen Fehler gefunden werden. Die Art des Fehlers wird bei jedem Beispiel zunächst benannt. Primär- und Fremdschlüsselbedingungen wurden nicht in die Aufstellung mit aufgenommen.

```
-- Benutzung eines nicht definierten Klassennamens

INSERT INTO PersistenteObjekte
  VALUES (20, 'Klasse_G', 'KlasseGNichtModelliert')
go

-- Aenderung eines Objektidentifikators

UPDATE PersistenteObjekte
  SET oid = 20
  WHERE oid = 17
go

-- Aenderung eines Klassennamens

UPDATE PersistenteObjekte
  SET klasse = 'Klasse_F'
  WHERE oid = 1
go

-- Aenderung der Objektzugehoerigkeit eines Attributs

BEGIN TRANSACTION
UPDATE Klasse_B_Attribut_B2
  SET oid=1
  WHERE oid=2 AND Attribut_B2='String2'
EXECUTE Pruefe_Klasse_B_Attribut_B2
COMMIT TRANSACTION
go

-- Zuweisung eines Attributs zu einem Objekt einer Klasse,
-- fuer die das Attribut nicht spezifiziert ist

BEGIN TRANSACTION
INSERT INTO Klasse_A_Attribut_A2
  VALUES (19, 72)
EXECUTE Pruefe_Klasse_A_Attribut_A2
COMMIT TRANSACTION
go
```

```
-- Kompensation, da nur Fehlermeldung und fehlerhafte Daten
-- in der Datenbank

BEGIN TRANSACTION
DELETE FROM Klasse_A_Attribut_A2
  WHERE oid=19
EXECUTE Pruefe_Klasse_A_Attribut_A2
COMMIT TRANSACTION
go

-- Unterschreitung der MUG fuer Attribute

BEGIN TRANSACTION
INSERT INTO PersistenteObjekte
  VALUES (20, 'Klasse_B', 'Instanz_Klasse_B_ohne_Attribut_B2')
EXECUTE Pruefe_Klasse_B_Attribut_B2
COMMIT TRANSACTION
go

-- Kompensation, da nur Fehlermeldung und fehlerhafte Daten
-- in der Datenbank

BEGIN TRANSACTION
DELETE FROM PersistenteObjekte
  WHERE oid=20
EXECUTE Pruefe_Klasse_B_Attribut_B2
COMMIT TRANSACTION
go

-- Ueberschreitung der MOG fuer Attribute

BEGIN TRANSACTION
INSERT INTO Klasse_A_Attribut_A2
  VALUES (18, 72)
INSERT INTO Klasse_A_Attribut_A2
  VALUES (18, 54)
EXECUTE Pruefe_Klasse_A_Attribut_A2
COMMIT TRANSACTION
go

-- Kompensation, da nur Fehlermeldung und fehlerhafte Daten
-- in der Datenbank

BEGIN TRANSACTION
DELETE FROM Klasse_A_Attribut_A2
  WHERE oid=18
EXECUTE Pruefe_Klasse_A_Attribut_A2
COMMIT TRANSACTION
go

-- Verknuepfung mit Objekt von falscher Klassenzugehoerigkeit

BEGIN TRANSACTION
INSERT INTO Assoz_1
  VALUES (3, 1)
EXECUTE Pruefe_Assoz_1
COMMIT TRANSACTION
go

-- Kompensation, da nur Fehlermeldung und fehlerhafte Daten
-- in der Datenbank

BEGIN TRANSACTION
DELETE FROM Assoz_1
  WHERE klasse_B=1 AND klasse_A=3
EXECUTE Pruefe_Assoz_1
```

```
COMMIT TRANSACTION
go

-- Unterschreitung der MUG fuer Rollen
-- nicht festgestellt!
BEGIN TRANSACTION
DELETE FROM Assoz_2
  WHERE klasse_A=18 AND klasse_E=12
EXECUTE Pruefe_Assoz_2
COMMIT TRANSACTION
go

-- qualifiziert

BEGIN TRANSACTION
DELETE FROM Assoz_3
  WHERE klasse_A=18 AND klasse_C=6
EXECUTE Pruefe_Assoz_3
COMMIT TRANSACTION
go

BEGIN TRANSACTION -- Dann ist der Qualifikatorwert weg
                  -- kein Fehler mehr !
                  -- unverwendete Qualifikatorwerte
                  -- irgendwo speichern
                  -- deshalb Prof. Hussmann:
                  --   andere Abbildung in OMT-91
DELETE FROM Assoz_3
  WHERE klasse_A=18 AND klasse_C=7
EXECUTE Pruefe_Assoz_3
COMMIT TRANSACTION
go

-- Kompensation, da nur Fehlermeldung und fehlerhafte Daten
-- in der Datenbank

BEGIN TRANSACTION
INSERT INTO Assoz_2
  VALUES (12, 18)
EXECUTE Pruefe_Assoz_2
COMMIT TRANSACTION
go

BEGIN TRANSACTION
INSERT INTO Assoz_3
  VALUES (18, 6, 'Qual2')
EXECUTE Pruefe_Assoz_3
COMMIT TRANSACTION
go

BEGIN TRANSACTION
INSERT INTO Assoz_3
  VALUES (18, 7, 'Qual2')
EXECUTE Pruefe_Assoz_3
COMMIT TRANSACTION
go

-- Ueberschreitung der MOG fuer Rollen

BEGIN TRANSACTION
INSERT INTO Assoz_2
  VALUES (14, 18)
EXECUTE Pruefe_Assoz_2
COMMIT TRANSACTION
go
```

```
 -- qualifiziert

BEGIN TRANSACTION
INSERT INTO Assoz_3
  VALUES (18, 8, 'Qual1')
INSERT INTO Assoz_3
  VALUES (18, 9, 'Qual1')
INSERT INTO Assoz_3
  VALUES (18, 10, 'Qual1')
EXECUTE Pruefe_Assoz_3
COMMIT TRANSACTION
go

-- Kompensation, da nur Fehlermeldung und fehlerhafte Daten
-- in der Datenbank

BEGIN TRANSACTION
DELETE FROM Assoz_2
  WHERE klasse_A=18 AND klasse_E=14
EXECUTE Pruefe_Assoz_2
COMMIT TRANSACTION
go

BEGIN TRANSACTION
DELETE FROM Assoz_3
  WHERE klasse_A=18 AND klasse_C=10
EXECUTE Pruefe_Assoz_3
COMMIT TRANSACTION
go

-- Objekt falscher Klasse als Assoziationsobjekt

BEGIN TRANSACTION
UPDATE Assoz_4
  SET Assoziationsklasse=10
  WHERE klasse_D=11 AND klasse_E=12
EXECUTE Pruefe_Assoz_4
COMMIT TRANSACTION
go

-- Kompensation, da nur Fehlermeldung und fehlerhafte Daten
-- in der Datenbank

BEGIN TRANSACTION
UPDATE Assoz_4
  SET Assoziationsklasse=19
  WHERE klasse_D=11 AND klasse_E=12
EXECUTE Pruefe_Assoz_4
COMMIT TRANSACTION
go

-- Objekt ohne Link

BEGIN TRANSACTION
INSERT INTO PersistenteObjekte
  VALUES (20, 'Klasse_F', 'Instanz_F2')
EXECUTE Pruefe_Assoz_4
COMMIT TRANSACTION
go

-- Kompensation, da nur Fehlermeldung und fehlerhafte Daten
-- in der Datenbank

BEGIN TRANSACTION
DELETE FROM PersistenteObjekte
  WHERE oid=20
```

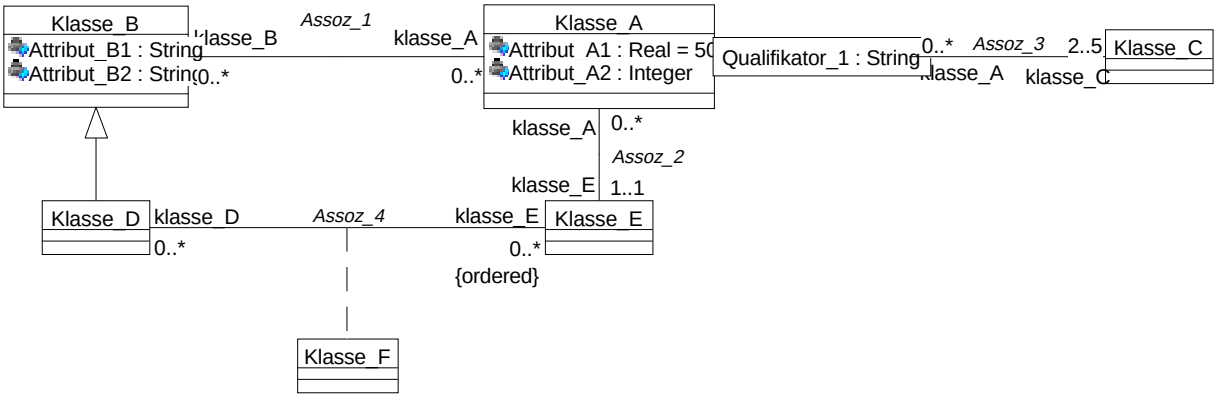
```
EXECUTE Pruefe_Assoz_4
COMMIT TRANSACTION
go

-- Objekt mit mehr als einem Link

BEGIN TRANSACTION
INSERT INTO Assoz_4
  VALUES (19, 15, 1, 11)
EXECUTE Pruefe_Assoz_4
COMMIT TRANSACTION
go

-- Kompensation, da nur Fehlermeldung und fehlerhafte Daten
-- in der Datenbank

BEGIN TRANSACTION
INSERT INTO PersistenteObjekte
  VALUES (20, 'Klasse_F', 'Instanz_F2')
UPDATE Assoz_4
  SET Assoziationsklasse=20
  WHERE klasse_D=11 AND klasse_E=15
EXECUTE Pruefe_Assoz_4
COMMIT TRANSACTION
go

-- Sequenz-Nummer nicht eindeutig

BEGIN TRANSACTION
UPDATE Assoz_4
  SET klasse_E_snr=0
  WHERE klasse_D=11 AND klasse_E=15
EXECUTE Pruefe_Assoz_4
COMMIT TRANSACTION
go

-- Sequenz-Nummern nicht dicht

BEGIN TRANSACTION
UPDATE Assoz_4
  SET klasse_E_snr=5
  WHERE klasse_D=11 AND klasse_E=15
EXECUTE Pruefe_Assoz_4
EXECUTE Pruefe_Assoz_4_klasse_E -- Fehlermeldung nur damit
                                -- Fehler von SQL Anywhere ???
COMMIT TRANSACTION
go

-- Kompensation, da nur Fehlermeldung und fehlerhafte Daten
-- in der Datenbank

BEGIN TRANSACTION
UPDATE Assoz_4
  SET klasse_E_snr=1
  WHERE klasse_D=11 AND klasse_E=15
EXECUTE Pruefe_Assoz_4
EXECUTE Pruefe_Assoz_4_klasse_E
COMMIT TRANSACTION
go
```

Alle Fehler wurden beim Test erkannt.

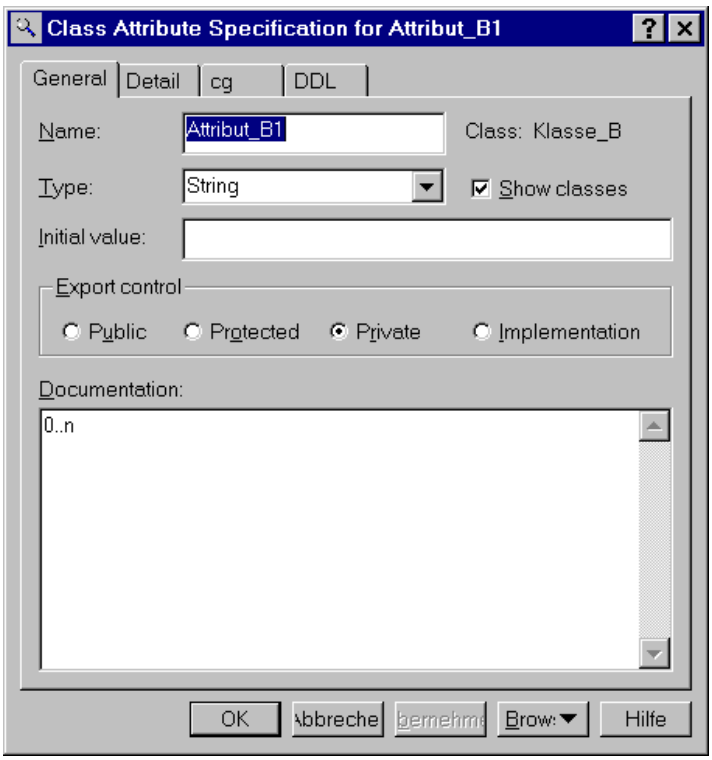
Anhang C

In diesem Anhang wird beschrieben, wie Rational Rose in Verbindung mit dem Rose-Script „GenerateSQL08.ebs“ genutzt werden kann, um die in Kapitel 8 beschriebene Abbildung von UML-Klassendiagrammen auf SQL zu automatisieren.

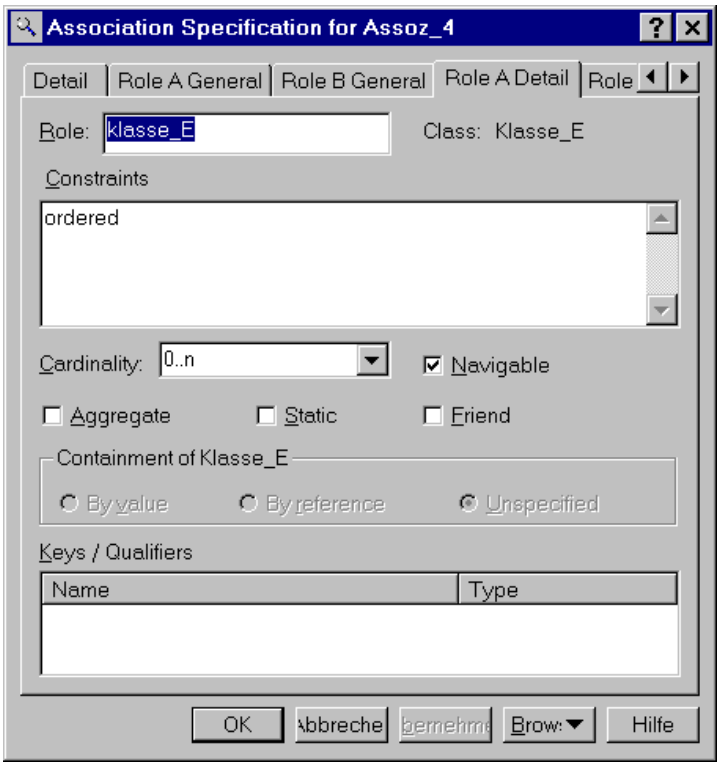
Das Klassendiagramm aus Anhang B wird zunächst mit Rational Rose erstellt:



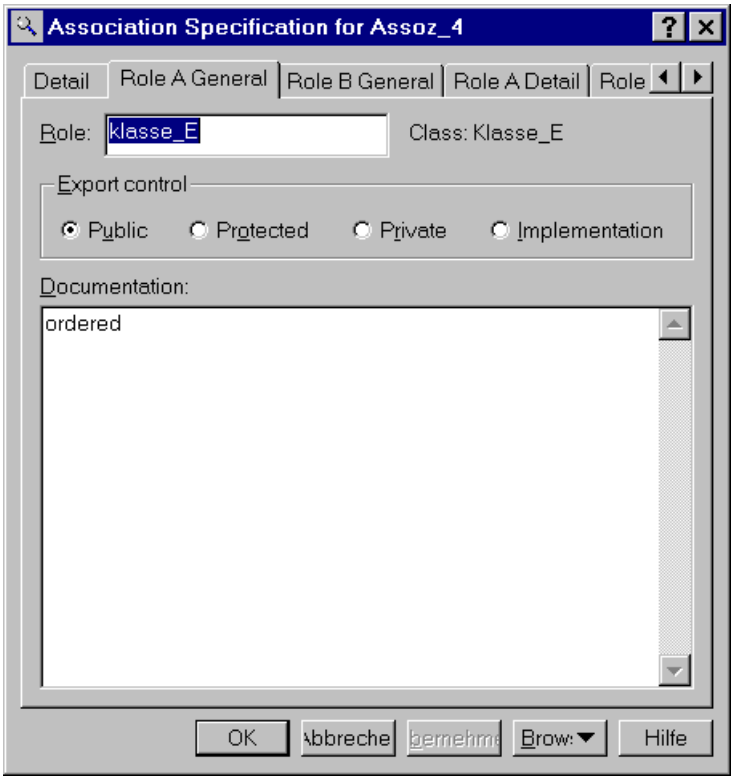
Dabei fällt auf, daß mit Rational Rose den Attributen keine Multiplizitäten zugewiesen werden können. Diese müssen im Feld Dokumentation eingetragen werden. Das Feld Dokumentation darf dann nur die Multiplizität enthalten, wobei anstelle des Symbols „*“ wie auch bei den Rollenmultiplizitäten das Symbol „n“ zu verwenden ist. Ist das Feld leer, wird eine Multiplizität von 1..1 angenommen.



Wie im folgenden illustriert, wird das Constraint "ordered" bei den Rollen-Details im Feld „Constraints“ eingetragen und dann im Diagramm dargestellt.

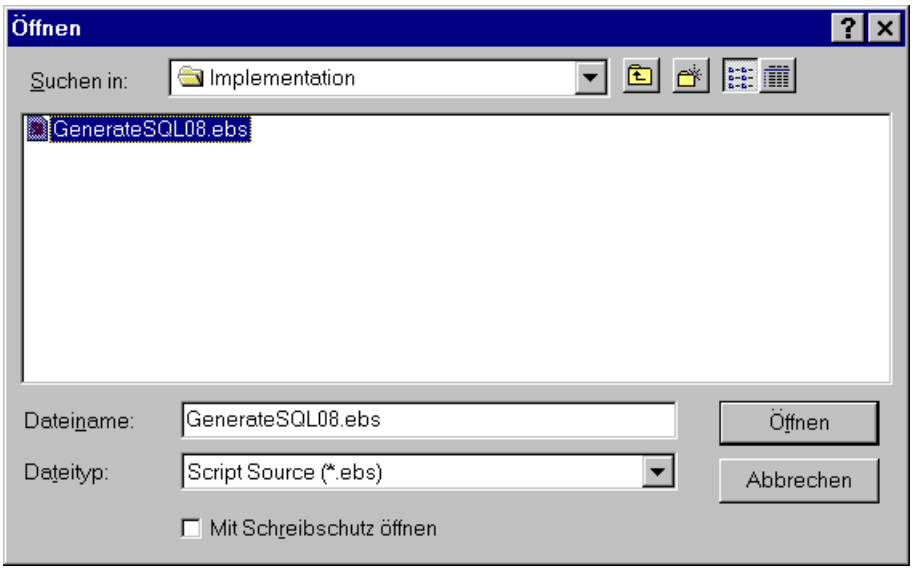


Da das Rose-Script dieses Feld nicht auslesen kann, muß die Zeichenkette "ordered" im Dokumentationsfeld zur Rolle nochmals angegeben werden.

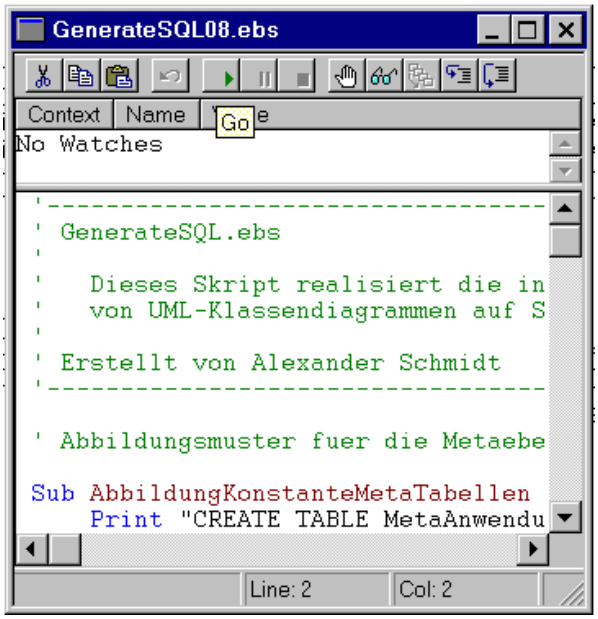


Da n-äre Assoziationen mit Rational Rose nicht modelliert werden können, können diese bei der Generierung von SQL-Code natürlich auch nicht berücksichtigt werden.

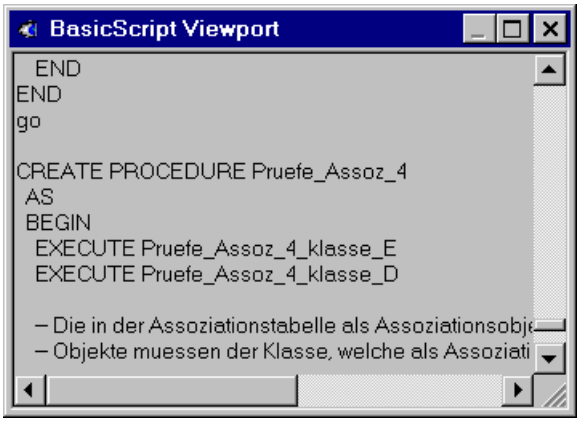
Nachdem das Modell erstellt wurde, kann das Script ausgeführt werden. Dazu ist im Menü von Rational Rose „Tools/Open Script...“ zu wählen. Das Script „GenerateSQL08.ebs“ muß im Verzeichnis „Implementation“ der CD zur Diplomarbeit geöffnet werden:



Nun kann das Script über den Button „Go“ () gestartet werden:



Die Ausgabe des SQL-Code erfolgt in den „BasicScript Viewport“.



Von dort kann der Code durch Markieren und Nutzung des Pop-Up-Menüs in die Zwischenablage kopiert und einer beliebigen Verwendung zugeführt werden.

Abbildungsverzeichnis

Abkürzungsverzeichnis

OCL	Object Constraint Language
SQL	Structured Query Language
UML	Unified Modelling Language
RDBMS	Relational Database Management System
OMT	Object Modelling Technique
ODMG	Object Data Management Group
MUG	Multiplizitäts-Untergrenze
MOG	Multiplizitäts-Obergrenze
STD	State Transition Diagram
PSM	Persistent Stored Module

Literaturverzeichnis

[BlahPrem98] M. Blaha, W. Premerlani: Object-oriented modelling and design for database applications. Prentice-Hall, Inc. 1998

[SchaKort98] M. Schader, A. Korthaus: The Unified Modelling Language, Technical Aspects ans Applications. Physica-Verlag 1998

[Melton93] J. Melton: Understanding the new SQL: a complete guide. Morgan Kaufmann 1993

[Reuter87] P.C. Lockemann, J.W. Schmidt: Datenbank-Handbuch. Springer 1987
(Kapitel 4 von Andreas Reuter)

[Hußmann97] H. Hußmann: Formal foundations for software engineering methods. Springer 1997

[ODMG-2.0] R.G.G. Cattell, D. K. Barry: The object database standard: ODMG 2.0.
Morgan Kaufmann Publishers, Inc.

[BronSem91] I. N. Bronstein, K. A. Semendjajew: Taschenbuch der Mathematik. B. G. Teubner Verlagsgesellschaft 1991

[Vossen94] G. Vossen: Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme. Addison-Wesley 1994

[DbFokus] Vergleich der SQL-Implementierungen: Datenbank Fokus 06/98. S. 67-74

[Fischer97] M. Fischer: Persistente Objekte und verteilte Transaktionen. DA TU Dresden 1997

Verzeichnis weiterer Quellen

Quellen aus dem WWW

Bis auf 2 Quellen sind alle Quellen aus dem WWW auf der beiliegenden CD gespiegelt, da Quellen im WWW u.U. nach gewisser Zeit nicht mehr (im Originalzustand) verfügbar sein können.

[OCL97] (CD)	Object Constraint Language Specification (version 1.1); Rational Software Corp. u.a. hhtp://www.rational.com/uml
[HHK] (CD)	A. Hamie, J. Howse, S. Kent; Navigation Expressions in Object-Oriented Modelling; University of Brighton, UK http://www.biro.brighton.ac.uk/index.html
[HH97] (CD)	A. Hamie, J. Howse; Interpreting Syntropy in Larch. Technical Report ITCM97/C1; University of Brighton, 1997 http://www.biro.brighton.ac.uk/index.html
[UML-SEMA] (CD)	UML Semantics (version 1.1); Rational Software Corp. u.a. hhtp://www.rational.com/uml
[OCL-PARSER] (CD)	OCL-Parser http://www.software.ibm.com/ad/ocl
[SQL92SYN]	Syntax-Referenz zu SQL-92 im WWW http://cuiwww.unige.ch/db-research/Enseignement/analyseinfo/SQL92
[DWB] (CD)	Prof. Meyer-Wegener; Daten- und Wissensbanken, Skript zu Vorlesung http://wwwdb.inf.tu-dresden.de/~www/LV/DWB/
[VDH]	Prof. Meyer-Wegener; Verteilte Datenhaltung, Skript zur Vorlesung http://wwwdb.inf.tu-dresden.de/
[LFI] (CD)	Prof. Horst Reichel; Logik für Informatiker, Vorlesungsskript http://wwwtcs.inf.tu-dresden.de/scripte/
[DSDB] (CD)	Prof. Meyer-Wegener; Datenstrukturen und Datenbanken, Skript zu Vorlesung http://wwwdb.inf.tu-dresden.de/~www/LV/DSDB/

Software-Verzeichnis

Neben der Software wurden vor allem die Handbüchern und Online-Dokumentation genutzt.

Sybase™	Sybooks und SQL Anywhere
Oracle™	Oracle8 Documentation
MS Visual C++™	Handbücher
Rational Rose™	Rational Rose mit Online-Dokumentation
Borland Turbo Pascal™	Turbo Pascal 6.0; Programmierhandbuch; Borland GmbH 1990

Danksagung

Mein Dank gilt allen, die mich während der Erstellung meiner Diplomarbeit unterstützt haben. Besonders dankbar bin ich meinen Eltern, welche mir mein Studium ermöglichten.
Sehr dankbar bin ich meiner Betreuerin, Frau Dr. Demuth, für die vielen motivierenden Hinweise zu den Zwischenergebnissen der Arbeit und auf geeignete Literaturstellen sowie für kompetente Antworten zu technischen Fragen.
Herzlich bedanken möchte ich mich auch bei Herrn Prof. Hußmann, welcher mir mit Frau Dr. Demuth zusammen vor allem beim Zeitmanagement für die Arbeit geholfen hat und wichtige Hinweise zur OCL-Spezifikation gab.

Alexander Schmidt Dresden, den 18. September 1998

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, daß ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher werde in gleicher noch in ähnlicher From anderen Prüfungskommisionen vorgelegt und auch nicht veröffentlicht.

Dresden, den 18. September 1998