Die Object Constraint Language der UML

Udo Kelter

31.05.2009

Zusammenfassung dieses Lehrmoduls

Die Object Constraint Language (OCL) ist eine Sprache, in der man zu einem UML-Modell zusätzliche Einschränkungen und Integritätsbedingungen angeben kann. Dieses Lehrmodul führt die wichtigsten Anwendungensfälle und Details der OCL ein.

Vorausgesetzte Lehrmodule:

obligatorisch:
- Die Unified Modelling Language (UML) Version 2
- Objektorientierte Modellierung
- Zustandsautomaten

Stoffumfang in Vorlesungsdoppelstunden: 1.0

Inhaltsverzeichnis

1 Motivation **3**

2 Die Object Constraint Language (OCL) – Anwendungen und Sprachumfang **4**

2.1 Der Kontext von OCL-Ausdrücken 6
2.2 OCL-Ausdrücke 7

3 Arten von OCL-Bedingungen **7**

3.1 Invarianten 7
3.2 Vor- und Nachbedingungen 8
3.3 Initiale und abgeleitete Werte 9
3.4 Spezifikation von Abfragen 9

4 Organisation von OCL-Bedingungen **9**

4.1 Bezeichner von OCL-Bedingungen 10
4.2 Pakete 10

5 Navigation und Kollektionen **10**

5.1 Elementare Funktionen mit Kollektionen 11
5.2 Die Funktion `select` 11
5.3 Die Funktion `reject` 12
5.4 Rollen mit Kardinalität 0..1 12
5.5 Die Funktion `collect` 13
5.6 Die Funktion `forAll` 14
5.7 Die Funktion `exists` 14
5.8 Die Funktion `iterate` 14

6 Gemeinsame Teilausdrücke **15**

6.1 let-Anweisung 15
6.2 Definitions-Ausdrücke 16

Literatur 16

©2009 Udo Kelter **Stand: 31.05.2009**
Dieser Text darf für nichtkommerzielle Nutzungen als Ganzes und unverändert in elektronischer oder gedruckter Form beliebig weitergegeben werden und in WWW-Seiten, CDs und Datenbanken aufgenommen werden. Jede andere Nutzung, insb. die Veränderung und Überführung in andere Formate, bedarf der expliziten Genehmigung. Die jeweils aktuellste Version ist über http://kltr.de erreichbar.

1 Motivation

Modelle sind vereinfachte Darstellungen eines Systems, d.h. viele Details werden weggelassen. Um die Sprachen, in denen Modelle formuliert werden, einfach zu halten, enthalten diese Sprachen oft gar keine Sprachkonstrukte, mit denen man bestimmte Details ausdrücken kann.

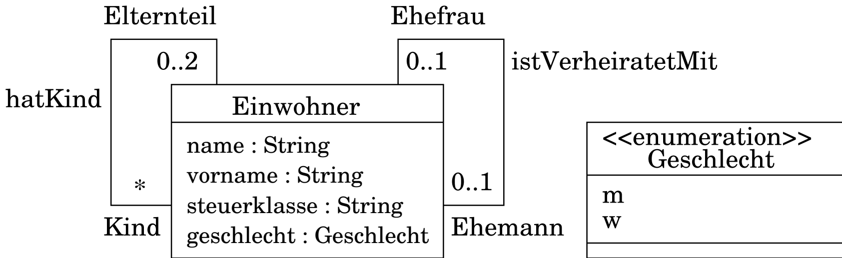

Abbildung 1: Daten eines Einwohnermeldeamts

Hierzu ein Beispiele: Das (Analyse-) Klassendiagramm in Bild 1 beschreibt auszugsweise den Datenbestand eines Einwohnermeldeamts.

Ein Klassendiagramm beschreibt vor allem die Typen der auftretenden Daten. Die konkreten Nutzdaten bilden, wenn man von Details der Implementierungstechnologie abstrahiert, ein Netzwerk aus Objekten und Beziehungen, die Attribute haben können. Die Klassen und Beziehungstypen eines Klassendiagramms geben an, welche Typen überhaupt in den Nutzdaten auftreten dürfen. Im vorliegenden Beispiel müssen alle Objekte vom Typ `Einwohner` sein, die Beziehungen müssen den Typ `hatKind` oder `istVerheiratetMit` haben.

Kardinalitäten ermöglichen eine weitere Eingrenzung der zulässigen Objektnetzwerke: eingegrenzt wird die Anzahl der Beziehungen, in denen ein Objekt stehen darf. Im vorliegenden Beispiel dürfen zu jedem Einwohner maximal zwei Einwohner[1] als Eltern registriert sein, und jeder Einwohner kann nur mit einem anderen verheiratet sein.

Man erkennt unschwer, daß diese bisherigen Eingrenzungen unsin-

[1]Einwohner ist hier als geschlechtsneutrale Bezeichnung zu verstehen!

©2009 Udo Kelter

nige Datenbestände zulassen, weil weitergehende Integritätsbedingungen nicht ausgedrückt werden:

- Wenn für einen Einwohner zwei andere Einwohner als Eltern registriert sind, müssen diese unterschiedliche Werte im Attribut `geschlecht` haben.
- Ein Einwohner darf nicht Elternteil von sich selber sein.
- Der Einwohner, der in einer `istVerheiratetMit`-Beziehung die Rolle `Ehefrau` spielt, muß im Attribut `geschlecht` den Wert “w” für weiblich stehen. Entsprechendes gilt für die Rolle `Ehemann`.

Die vorstehenden Integritätsbedingungen können mit den graphischen Ausdrucksmitteln in Klassendiagrammen *überhaupt nicht ausgedrückt* werden. Sofern man also derartige Details spezifizieren möchte, benötigt man zusätzliche Ausdrucksmittel oder anders gesagt eine ergänzende Sprache.

2 Die Object Constraint Language (OCL) – Anwendungen und Sprachumfang

Die **Object Constraint Language** (**OCL**) ist eine Sprache, mit der man in diversen UML-Modelltypen detailliertere Bedingungen zu den auftretenden Daten spezifizieren kann. Beispiele hierfür sind:

- in Datenmodellen (insb. für die persistenten Datenbestände einer Applikation): die Angabe weitergehender Integritätsbedingungen wie in den obigen Beispielen. Diese werden **Invarianten** genannt, da sie zu jedem Zeitpunkt gelten müssen.
- in der funktionalen Spezifikation von Operationen einer Klasse die Angabe von:

  (a) **Vorbedingungen**: diese geben Merkmale von Eingabeparametern einer Operation oder globalen Variablen an, die vor der Operationsausführung erfüllt sein müssen, andernfalls scheitert die Operationsausführung.

(b) **Nachbedingungen**: diese geben Merkmale von Ausgabeparametern einer Operation oder globalen Variablen an, die nach der Operationsausführung gelten müssen.

– in Transitionen in Zustandsmaschinen: die Angabe von **Guards**, also Bedingungen, die erfüllt sein müssen, damit bestimmte Transitionen schalten können.

In allen vorstehenden Anwendungsbeispielen müssen die Bedingungen letztlich in Form von **Booleschen Ausdrücken** angegeben werden, die sich auf den aktuell vorliegenden Datenbestand beziehen. Solche Booleschen Ausdrücke sind daher ein zentraler Bestandteil der OCL.

Die Datenbestände, auf die man sich in diesen Ausdrücken bezieht, können beliebige Teilgraphen des aktuell vorliegenden Objektnetzwerks sein. Um die interessierenden Datenelemente zu lokalisieren, benötigt man eine **Abfragesprache**. Diese Abfragesprache muß insb. die Navigation in Objektnetzwerken unterstützen, um ausgehend von einem Objekt, dessen Eigenschaften spezifiziert werden sollen, über Beziehungen andere Objekten zu erreichen, an denen relevante Daten stehen.

Da in den Abfragen Selektionsbedingungen auftreten, müssen auch Ausdrücke über alle Basisdatentypen formulierbar sein, namentlich arithmetische Ausdrücke und Ausdrücke mit Texten.

Wir können bereits hier festhalten, daß eine Sprache zur Formulierung von Bedingungen in Objektnetzwerken relativ umfangreich sein muß und viele Konzepte einer Programmiersprache (insb. Ausdrücke und darin vorkommende Operatoren, Funktionen usw.) enthalten wird. Dementsprechend umfangreich und komplex ist die Spezifikation der OCL, s. [OCL06], und aus diesem Grund kann in diesem Lehrmodul auch nur eine informelle Einführung der wichtigsten Teile der OCL gegeben werden.

Die ohnehin benötigte Abfragesprache legt es nahe, diese für weitere Zwecke zu benutzen:

– bei reinen Abfrageoperationen einer Klasse: zur vollständigen Spezifikation des Ergebnisses der Operationsausführung, sofern mit der

Abfragesprache formulierbar[2]

- bei abgeleiteten Attributen einer Klasse: zur Spezifikation, wie das Attribut abzuleiten ist; je nach dem Wertebereich des Attributs ist ein arithmetischer, textueller oder anderer Ausdruck erforderlich
- bei beliebigen Attributen: zur Spezifikation von initialen Werten
- bei Beziehungen mit Kardinalität *: analog zur Spezifikation von Initialwerten und abgeleiteten Attributen können vorhandene Beziehungen zu anderen Objekten spezifiziert werden

2.1 Der Kontext von OCL-Ausdrücken

Jede OCL-Bedingung ist einem Modellelement zugeordnet - sind also auf Typebene definiert - und gilt für alle Instanzen dieses Modellelements. Im obigen Beispiel unseres Einwohnermeldeamts müssen die Bedingungen bei jedem einzelnen Einwohner-Objekt erfüllt sein. Ein solches anonymes Einwohner-Objekt bildet daher den Ausgangspunkt zur Formulierung der Bedingungen.

Das Modellelement, auf das sich eine OCL-Bedingung bezieht, wird als dessen **Kontext** bezeichnet. Meistens ist der Kontext eine Klasse, in selteneren Fällen eine Operation oder ein (Entwurfs-) Objekt.

Die anonyme Instanz dieses Modellelements kann in den Bedingungen mit dem Schlüsselwort `self` bezeichnet werden.

Notation von OCL-Bedingungen. Die OCL ist keine selbständige Sprache, sondern Ausdrücke in der OCL beziehen sich immer auf Datenelemente, die i.d.R. in einem Klassendiagramm definiert sind.

Häufig werden die OCL-Bedingungen direkt in das Klassendiagramm eingetragen, das das Kontext-Element enthält. Syntaktisch wird der OCL-Ausdruck in einen Kommentar eingetragen, der mit einer gestrichelten Linie mit dem Kontext-Element verbunden ist. Das Kontext-Element wird dann nicht mehr eigens im Kommentar benannt.

[2] Man kann dies auch als Sonderfall einer Nachbedingung ansehen, die das Ergebnis der Operationsausführung nicht nur partiell und deskriptiv, sondern vollständig und operational angibt.

Alternativ ist eine rein textuelle Notation von OCL-Bedingungen verfügbar. Die textuelle Notation ist vorteilhafter, wenn viele und/oder komplexe Bedingungen notiert werden müssen.

2.2 OCL-Ausdrücke

Wie schon erwähnt benötigt man in der OCL Ausdrücke für diverse primitive Datentypen (Integer, Real, String, Boolean u.a.). Die Sprachspezifikation definiert daher entsprechende Datentypen, zug. Operationen und syntaktische Regeln (s. Kapitel 11 in [OCL06]). Die ähneln weitgehend den gewohnten Programmiersprachen. Wir stellen sie hier nur auszugsweise kurz vor.

Einfache Vergleiche können mit den üblichen Vergleichsoperatoren (`>`, `>=` usw.) gebildet werden, die Gleichheit wird mit `=` (nicht mit `==`!) geprüft. Boolesche OCL-Ausdrücke können mit den Operatoren `and`, `or`, `xor` und `implies` zu einem neuen Ausdruck mit der offensichtlichen Bedeutung kombiniert werden. Es gibt keinen Operator `not`, stattdessen eine Boolesche Funktion `not(...)`. Darüber hinaus gibt in der OCL für viele Basistypen, darunter Boolean, bedingte Ausdrücke der Form

`if` *boolexpr* `then` *expr1* `else` *expr2* `endif`

worin *boolexpr* ein Boolescher Ausdruck ist und *expr1* und *expr2* zwei Ausdrücke mit gleichem (Basis-) Typ sind.

`implies` hat die geringste Präzedenz, `and`, `or`, `xor` die nächsthöhere, `if-then-else-endif` eine noch höhere. Mit runden Klammern kann man explizit gewünschte Bindungen erzwingen.

3 Arten von OCL-Bedingungen

3.1 Invarianten

Wie schon erwähnt sind Invarianten OCL-Bedingungen, die i.d.R. Klassen (i.a. Classifiern) zugeordnet sind. Die textuelle Notation hat folgende Form:

context *Kontext*

```
inv:     Boolescher OCL-Ausdruck
```

Darin sind context und inv Schlüsselworte. Die folgende Invariante drückt z.B. aus, daß die Steuerklasse keinen leeren Text enthalten darf:

```
context  Einwohner
inv:     self.steuerklasse > ''
```

Die anonyme Instanz dieses Modellelements kann nicht nur mit dem Schlüsselwort `self` bezeichnet werden, sondern auch mit einem Namen, der mit Doppelpunkt getrennt vor dem Kontextelement steht. Beispiel:

```
context  Hans : Einwohner
inv:     Hans.steuerklasse > ''
```

3.2 Vor- und Nachbedingungen

Bei Vor- und Nachbedingungen ist der Kontext eine Operation eines Datentyps. In der textuellen Notation wird die Operation gemäß folgendem Schema angegeben:

```
context  Typename::operationName(param1 : Type1, ...  ):
         ReturnType
pre:     ....
post:    result = ....
```

Hinter den Schlüsselworten pre und post steht die jeweilige Vor- und Nachbedingung.

In der Vor- und Nachbedingung kann wieder das Schlüsselwort `self` benutzt werden; gemeint ist hier aber eine Instanz des Typs, zu dem die Operation gehört, denn eine Operation als solche kann nicht instanziiert werden. `self` steht also für das Objekt, auf dem die Operation aufgerufen worden ist.

In der Vor- und Nachbedingung können die Namen der Parameter als Datenwerte benutzt werden.

In der Nachbedingung kann das Schlüsselwort `result` benutzt werden, um den Rückgabewert der Operation zu benennen, sofern ein Rückgabewert vorhanden ist.

©2009 Udo Kelter

Als (unvollständiges) Beispiel betrachten wir die Operation heiratet, die nur auf einem männlichen Einwohner aufgerufen werden darf und als Argument einen weiblichen Einwohner haben muß:

```
context  Einwohner::heiratet( e : Einwohner )
pre:     self.geschlecht=Geschlecht::m
pre:     self.ehefrau->size()=0
pre:     e.geschlecht=Geschlecht::w
post:    self.ehefrau->includes(e) and self.ehefrau->size()=1
```

3.3 Initiale und abgeleitete Werte

Für initiale und abgeleitete Werte bilden Attribute den Kontext. Hier benutzte Schlüsselworte: init und derive. Beispiel:

```
context  Einwohner::steuerklasse
init:    ''
context  Einwohner::kinderzahl
derive:  self.Kind->size()
```

3.4 Spezifikation von Abfragen

Eine Operation, die eine reine Abfrage ohne Seiteneffekte ist, kann, wenn die Ausdrucksfähigkeit der OCL ausreicht, auch als OCL-Ausdruck mit passendem Ergebnistyp spezifiziert werden. Kontext ist hier eine Operation. Beispiel:

```
context  Einwohner::zahltoechter (): Integer
derive:  self.Kind->select(geschlecht = Geschlecht::w)->size()
```

4 Organisation von OCL-Bedingungen

Zu einem UML-Modell kann es viele OCL-Bedingungen geben. Ob diese in einer oder mehreren Dateien verwaltet werden, ist im Prinzip unerheblich, ebenso die Reihenfolge der Aufschreibung in den Dateien oder ggf. der graphischen Darstellung in Diagrammen.

Alle Bedingungen, die sich auf den gleichen Kontext beziehen, können hinter eine einzige Kontext-Klausel geschrieben werden.

©2009 Udo Kelter

4.1 Bezeichner von OCL-Bedingungen

OCL-Bedingungen können optional einen Bezeichner erhalten, mit dem man Bezug auf sie nehmen kann. Der Bezeichner wird zwischem dem Schlüsselwort inv, pre, post usw. und dem Doppelpunkt notiert. Beispiel für eine Invariante mit Bezeichner:

```
context  Hans : Einwohner
inv keineLeereSteuerklasse: Hans.steuerklasse > ''
```

4.2 Pakete

Um die OCL-Bedingungen den richtigen Paketen zuzuorden, kann zusätzlich mit den Schlüsselworten package, gefolgt von einer Paketbezeichnung, und endpackage ein Bezug auf das angegebene Paket angegeben werden. Zwischen diesen beiden Eingrenzungen können beliebig viele OCL-Bedingungen stehen. Beispiel:

```
package  Package::SubPackage
context  X
inv:     ......
context  X::operationName(..)
pre:     ......
endpackage
```

5 Navigation und Kollektionen

Wir hatten bereits in den vorigen Beispielen mehrfach Beispiele benutzt, in denen auf Attribute von Objekten zugegriffen wurde; notiert wurde dies in der üblichen Punktnotation (z.B. `self.steuerklasse`).

Etwas komplizierter ist die Navigation über Beziehungen. Zunächst einmal reicht der Name des Beziehungstyps i.a. nicht aus, da Beziehungstypen reflexiv sein können, z.B. der Beziehungstyp `hatKind` im Beispiel in Bild 1, und daher die Navigationsrichtung unklar wäre. Daher muß i.a. der *Rollenname* angegeben werden, wenn man über Beziehungen navigieren möchte. Beispiel: mit dem Ausdruck

```
self.Kind
```

navigiert man über die Beziehungen des Typs `self.hatKind`, und zwar die Beziehungen, in denen `self` die Rolle `Elternteil` spielt.

5.1 Elementare Funktionen mit Kollektionen

Da ein Objekt in mehreren Beziehungen eine Rolle spielen kann, ergibt die Navigation über eine bestimmte Richtung einer Beziehung i.a. eine Kollektion von Zielobjekten. Wenn die Beziehungen geordnet sind, ist die resultierende Kollektion ebenfalls geordnet.

Der OCL-Standard definiert mehrere Arten von Kollektionen (geordnete/ungeordnete, mit/ohne Dubletten), s. Kap. 11 in [OCL06]. Für solche Kollektionen sind diverse Funktionen vordefiniert, u.a. `isEmpty()`, `notEmpty()` und `size()`. Angewandt werden diese Funktionen auf die Kollektion in der Pfeil-Schreibweise; so ergibt der Ausdruck

```
self.Kind->size()
```

die Anzahl der Kinder der gerade betrachteten Person. Mit diesen Konzepten können wir nun ausdrücken, daß für eine Person höchstens zwei Eltern existieren dürfen:

```
context  Einwohner
inv:     self.Elternteil->size() <= 2
```

Generell kann man mit Hilfe der Funktionen `size()` alle üblichen Kardinalitäten auch als OCL-Bedingungen ausdrücken.

5.2 Die Funktion `select`

Mit der vorstehenden Invarianten haben wir noch nicht ausgedrückt, daß jede Person höchstens einen weiblichen und einen männlichen Elternteil haben kann. Hierzu benötigen wir eine Funktion, die aus einer Kollektion bestimmte Elemente selektiert. Dies leistet die Funktion `select()`.

Das Hauptargument, die zu selektierende Kollektion, steht vor dem Pfeil. In der einfachsten Syntaxvariante ist das "Argument" von

©2009 Udo Kelter

`select` ein Boolescher Ausdruck[3], der für jedes einzelne Objekt der Kollektion ausgewertet wird und dieses implizit als Referenz benutzt. Die registierten Mütter eines Einwohners erhalten wir daher mit folgendem Ausdruck:

```
self.Elternteil->select(geschlecht = Geschlecht::w)
```

Die Restriktion, daß pro Einwohner nur eine Mutter vorhanden sein darf, kann wie folgt ausgedrückt werden:

```
context Einwohner
inv:    self.Elternteil->select( geschlecht = Geschlecht::w
        )->size() <= 1
```

In einer zweiten Syntaxvariante von `select()` wird explizit eine Laufvariable angegeben, die durch einen senkrechten Strich von dem Selektionskriterium getrennt wird. Beispiel:

```
select( ew | ew.geschlecht = Geschlecht::w )
```

In der dritten Syntaxvariante wird zusätzlich der Typ der Laufvariablen angegeben. Beispiel:

```
select( ew : Einwohner | ew.geschlecht = Geschlecht::w )
```

5.3 Die Funktion `reject`

Die Funktion `reject` hat die gleichen Syntaxvarianten wie `select()`, selektiert werden aber nur die Kollektionselemente, bei denen das Selektionskriterium nicht zutrifft. Für eine Bedingung B ist also `reject(`B`)` äquivalent zu `select(not(`B`))`.

5.4 Rollen mit Kardinalität 0..1

Sofern eine Rolle in einem Beziehungstyp die Kardinalität 0..1 hat, kann beim Navigieren über diese Beziehungen in Richtung dieser Rolle immer höchstens ein Objekt gefunden werden. Der Ausdruck

[3] `select(...)` sieht zwar aus wie ein üblicher Funktionsaufruf, der Inhalt zwischen den Klammern ist aber *keine übliche Parameterliste*, die eine kommagetrennte Liste von Werten oder Objektreferenzen enthalten würde, sondern ein Stück "Quelltext" mit einer speziellen Syntax.

```
self.Ehefrau
```

liefert eine solche Kollektion mit maximal einem Objekt. Wenn wir wissen, daß diese Kollektion nicht leer ist, also genau ein Objekt enthält, können wir dieses mittels einer Erweiterung der Syntax auch direkt als Objekt ansehen[4] und z.B. mit dem Ausdruck

```
self.Ehefrau.name
```

direkt auf den Namen der Ehefrau zugreifen. Mithilfe dieses Konstrukts können wir z.B. sehr einfach festlegen, daß Ehefrauen stets weiblich sind:

```
context  Einwohner
inv:     self.Ehefrau.geschlecht = Geschlecht::w
```

5.5 Die Funktion `collect`

Mit den Funktionen `select` und `reject` kann man eine vorhandene Kollektion verkleinern, aber keine anderen Elemente hinzufügen. Ferner kann man ausgehend von `self` durch die bisherigen Navigationsverfahren immer nur Kollektionen von Objekten bilden, nicht hingegen Kollektionen von Attributwerten. Letzteres ist durch die Funktion `collect` möglich. Die allgemeine Syntax von `collect` ist

```
collect( v : Type | expression-with-v )
```

expression-with-v ist ein Ausdruck, der `v` als ungebundene Laufvariable enthält und der einen beliebigen Datenwert liefert. Beispiel:

```
self.Elternteil->collect( ew : String | ew.name )
```

Auch hier gibt es vereinfachte Syntaxvarianten wie bei `select()`, bei denen die Typangabe bzw. die Laufvariable und die Typangabe weggelassen wird.

Ergebnis von `collect` ist die Kollektion der Einzelergebnisse, die sich bei Auswertung der *expression-with-v* für alle Werte der Laufvariablen ergeben. Hierbei können Duplikate entstehen, d.h. zwei verschiedenen Werten der Laufvariablen werden gleiche Einzelergebnisse

[4] Ansonsten müßte man umständlich mit einem Iterator über die einelementige Menge iterieren.

©2009 Udo Kelter

zugeordnet. Wenn `collect` also auf eine Menge angewandt wird, dann ist das Ergebnis i.a. keine Menge mehr, sondern ein *bag*, auch Multimenge genannt.

Wenn man trotzdem eine duplikatfreie Menge benötigt, kann die Multimenge mit der vordefinierten Konversionsfunktion `asSet()` in eine Menge konvertiert werden.

Sofern die Ausgangsmenge geordnet war (z.B. beim Navigieren über geordnete Beziehungen), ist auch das Ergebnis geordnet.

5.6 Die Funktion `forAll`

Die Funktion `forAll` wird angewandt auf eine Kollektion und prüft bei jedem Element, ob eine gegebene Bedingung erfüllt ist. Falls dies bei allen Elementen zutrifft, liefert sie *True* zurück, sonst *False*. Die allgemeine Syntax von `forAll` ist

```
forAll( v : Type | boolean-expression-with-v )
```

Auch hier gibt es vereinfachte Syntaxvarianten wie bei `select()`, bei denen die Typangabe bzw. die Laufvariable und die Typangabe weggelassen wird. Beispiel: Der Ausdruck

```
self.Elternteil->forAll(steuerklasse = 'IV')
```

ist wahr, wenn beide Eltern von `self` die Steuerklasse IV haben.

5.7 Die Funktion `exists`

Die Funktion `exists` hat die gleiche Syntax wie `forAll`. Sie liefert *True*, wenn die angegebene Bedingung bei wenigstens einem Element der Kollektion erfüllt ist.

5.8 Die Funktion `iterate`

`iterate` ist eine Verallgemeinerung der bisher vorgestellten Funktionen mit Kollektionen. Die Kernidee besteht darin, über die Elemente einer Kollektion zu iterieren, bei jedem Element einen Ausdruck auszuwerten und die Ergebnisse dieser Auswertungen in einem "Akkumulator" aufzusammeln. Die allgemeine Syntax von `iterate` ist:

```
iterate( elem : Type; acc : Type = <expression> |
                          expression-with-elem-and-acc )
```

`elem` ist die Laufvariable, `acc` der Akkumulator. Der Akkumulator muß einen Intialwert erhalten, z.B. `Bag{}` für eine leere Multimenge. Bei jedem Iterationsschritt wird der Laufvariablen das gerade betrachtete Element zugewiesen, der Akkumulator behält den Inhalt vom vorigen Iterationsschritt bzw. beim ersten Iterationsschritt den Intialwert. Endergebnis der Ausführung von `iterate` ist der letzte Inhalt des Akkumulators.

Die Funktionen `collect`, `forAll` u.a. können mit Hilfe von `iterate` implementiert werden. Beispielsweise ist der Ausdruck

```
collection->collect( x : T | x.property )
```

äquivalent zu

```
collection->iterate( x : T;  acc : T2 = Bag{}   |
                          acc->including(x.property) )
```

`including(object : T) : Bag(T)` ist eine der vordefinierten Funktionen auf Multimengen (s. Abschnitt 11.7.4 in [OCL06]). Sie fügt das Argument `object` in die Multimenge ein und liefert die vergrößerte Multimenge zurück.

6 Gemeinsame Teilausdrücke

6.1 let-Anweisung

Sofern *innerhalb* einer OCL-Bedingung mehrfach der gleiche Teilausdruck auftritt, kann man ihn auch durch eine let-Anweisung herausziehen, einem Bezeichner zuweisen und diesen dann mehrfach verwenden. Die let-Anweisung wird zwischen der inv- o.ä. Klausel und der eigentlichen Bedingung angeordnet; sie besteht in dieser Reihenfolge aus

- dem Schlüsselwort let, dem Bezeichner für den Teilausdruck und einem ':',
- einer Typangabe, z.B. `integer` oder `Collection(`*t*`)` für eine Kollektion von Objekten des Typs *t*, gefolgt von '='

©2009 Udo Kelter

– einem passenden Ausdruck
– dem Schlüsselwort in

Beispiel:

```
context  Einwohner
inv:
let      eltern : Collection(Einwohner) = self.Elternteil
in       eltern->select(geschlecht = Geschlecht::m) <= 1 and
         eltern->select(geschlecht = Geschlecht::w) <= 1
```

6.2 Definitions-Ausdrücke

Eine let-Anweisung ist nur lokal innerhalb einer OCL-Bedingung gültig. Sofern man im gleichen Kontext mehrere OCL-Bedingungen formuliert, ist es praktisch, Hilfsvariablen oder Funktionen zu definieren, die in allen OCL-Bedingungen benutzt werden können. Dies ist mit einem Definitions-Ausdruck möglich.

Ein Definitions-Ausdruck ist durch das Schlüsselwort def markiert und steht nach einer Kontext-Klausel und vor den folgenden inv- u.a. Klauseln.

Ein Definitions-Ausdruck kann eine Variable in der gleichen Syntax wie eine let-Anweisung definieren, darüber hinaus auch Funktionen. Beispiele:

```
context  Einwohner
def:     eltern : Collection(Einwohner) = self.Elternteil
def:     hatKind (g:Geschlecht): Boolean
         = self.Kind->select(g = geschlecht)->isEmpty()
inv:     ...
```

Literatur

[OCL06] Object Constraint Language, Version 2.0; OMG document formal/2006-05-01, Object Management Group, Inc.; 2006; http://www.omg.org/spec/OCL/2.0/

©2009 Udo Kelter