

OCL IN DER PRAXIS

Mit der „Object Constraint Language“ (OCL) bietet die OMG eine Möglichkeit, Validierungsvorschriften für Objekte – mit anderen Worten Geschäftsregeln – auf formale Art und Weise zu spezifizieren. Der wahre Vorteil der OCL zeigt sich jedoch erst dann, wenn die im Modell definierten Regeln mit Hilfe von MDA-Ansätzen auch in der laufenden Anwendung geprüft werden. Der Artikel zeigt die Umsetzung der OCL im Rahmen des Frameworks „pleXX“ und konkrete Beispiele ihrer Verwendung im Projekt „Connect“ der DaimlerChrysler AG.

„Ein Artikel kann von mehreren Lieferanten bezogen werden, von denen maximal einer Hauptlieferant sein kann.“ Dies ist eine typische Anforderung, wie sie von vielen Kunden für ihr neu zu entwickelndes Softwaresystem gestellt wird. Zusammen mit einigen anderen Anforderungen kann daraus das in **Abbildung 1** dargestellte Klassendiagramm abgeleitet werden. Aus diesem Modell lässt sich jedoch die Bedingung „maximal ein Lieferant ist Hauptlieferant“ nicht mehr ablesen. Wie eine solche Einschränkung der gültigen Objektzustände von der Anforderungsanalyse über das Design bis in den Code gelangt, bleibt bis heute häufig ein nicht nachvollziehbares Mysterium. Werkzeuge für Requirements-Engineering und der konsequente Einsatz regressionsfähiger Unit-Tests helfen zwar dabei, das Vergessen oder Verletzen der Anforderungen zu erkennen, bevor es der Anwender tut. Dennoch bekämpfen diese Maßnahmen nur die Auswirkungen der zuvor aufgetretenen Mängel. Ziel muss es sein,

- die Anforderungen des Auftraggebers im Modell festzuhalten und damit formal zu dokumentieren und
- exakt diese Anforderungen später im laufenden Programm auch zu prüfen.

Die OCL

Das erste Problem löst die Object Management Group (OMG) mit der Einführung der *Object Constraint Language* (OCL) im Rahmen der UML-Spezifikation (vgl. [UML]). Die OCL ist eine formale Sprache, mit der sich Ausdrücke in UML-Modellen formulieren lassen. Solche Ausdrücke dienen unter anderem der Spezifikation von:

- invarianten Bedingungen für Klassen oder Typen,

- Vor- und Nachbedingungen für Operationen,
- Abfrageausdrücken,
- Ableitungsregeln für berechenbare Attribute.

Die primäre Anwendung der OCL liegt sicherlich in der Formulierung von Invarianten im Sinne von Einschränkungen (*Constraints*) der Gültigkeit von Objektmodellen. Dieser Artikel befasst sich ausschließlich mit diesem Bereich der OCL.

Am einfachsten erschließen sich die Möglichkeiten der Sprache durch einige Beispiele. Der folgende OCL-Ausdruck definiert, dass die untere Preisgrenze eines Artikels immer kleiner oder gleich der oberen Preisgrenze ist:

```
context Artikel inv:
    minPreis <= maxPreis
```

Und folgender Ausdruck stellt sicher, dass ein Ersatzartikel immer zur gleichen Warengruppe gehört wie der Artikel selbst:

```
context Artikel inv:
    self.ersatzartikel.warengruppe = self.warengruppe
```

Die eingangs erwähnte Einschränkung „maximal ein Hauptlieferant“ resultiert in folgendem Ausdruck:

```
context Artikel inv EinHauptlieferant:
    lieferanten->select(l | l.hauptlieferant = true)->size() <= 1
```

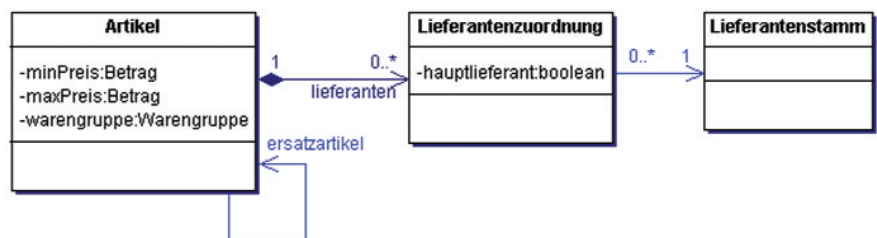


Abb. 1: Beispielmodell ohne Constraints

die autoren



Achim Demelt (E-Mail: a.demelt@excellent.de) ist Projektleiter bei der eXXcellent solutions GmbH in Ulm mit den Schwerpunkten Effizienzsteigerung bei der Softwareentwicklung durch modellgetriebene und generative Ansätze.



Dieter Mitrik (E-Mail: d.mitrik@gmx.de) hat Informatik an der FH Ulm studiert und das Thema OCL im Rahmen seiner Diplomarbeit behandelt.

Dieser ist in etwa so zu lesen: Die Größe der Untermenge der Lieferantenzuordnungen mit gesetztem Kennzeichen hauptlieferant muss kleiner oder gleich eins sein.

Aus diesen wenigen Beispielen lassen sich bereits einige Grundzüge und Bestandteile der OCL ablesen:

- Ein OCL-Ausdruck wird immer im Rahmen eines Kontexts definiert. Ist der Kontext klar (z. B. durch die

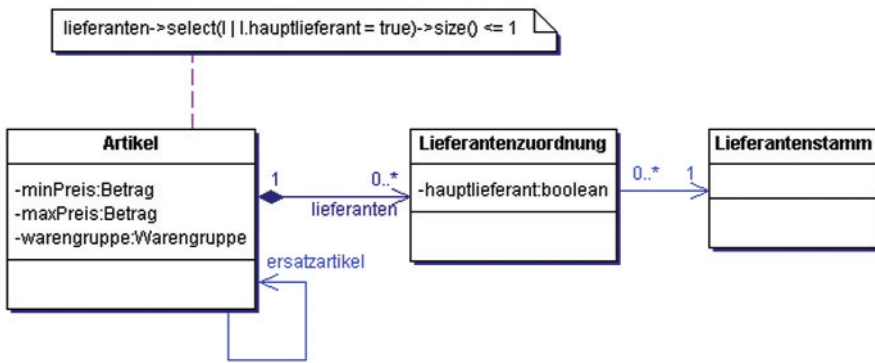


Abb. 2: Beispielmmodell mit Constraints

Zuordnung eines Ausdrucks im Modell wie in **Abbildung 2**), so kann die explizite Angabe durch das Schlüsselwort context auch entfallen.

- Die Art des Ausdrucks wird durch das dem Kontext folgende Schlüsselwort bestimmt. Bei den obigen Beispielen handelt es sich immer um Invarianten (Schlüsselwort inv). Ebenfalls möglich ist beispielsweise die Angabe der Schlüsselwörter pre und post für Vor- und Nachbedingungen. Ein Name für den Ausdruck (im Beispiel oben bei EinHauptlieferant) ist optional.
- Die OCL kennt einfache Vergleichsoperatoren. Die Syntax mit einfachem „=“ für Gleichheit und „<“ für Ungleichheit orientiert sich hier eher an SQL als an C/C++ oder Java.
- Navigationen über Beziehungen sind mit dem Punkt-Operator („.“) möglich.
- Auf Sammlungstypen (Collections) sind mehrere so genannte Iterator-Operationen definiert (im Beispiel ist select() zu sehen). Andere Varianten sind reject(), any(), forAll() oder das allgemeinere und sehr mächtige iterate(). Ihre Syntax ist stark an mathematische Mengenausdrücke angelehnt.
- Es existieren weitere Operationen für Sammlungstypen, wie z. B. size(), isEmpty() oder includes(object). Diese sind mit dem Pfeiloperator „->“ gekennzeichnet. Bei Nicht-Collections erfolgt eine automatische/implizite Typenwandlung.

Die gesamte Mächtigkeit der OCL aufzuzeigen würde bei Weitem den Umfang des Artikels sprengen – hier sei auf die OCL-Spezifikation verwiesen. Einige wichtige Eigenschaften der OCL-Ausdrücke sollen jedoch nicht verschwiegen werden:

- Die Auswertung eines OCL-Ausdrucks ist frei von Seiteneffekten. Das heißt, ein OCL-Ausdruck kann niemals den Zustand von Objekten während seiner Auswertung verändern.
- Die OCL ist keine Programmiersprache. Sie stellt daher keinerlei Konstrukte für Kontrollflusssteuerung zur Verfügung.
- Die OCL ist eine typsichere Sprache. Alle Ausdrücke müssen also typkonform sein. Ein Integer kann beispielsweise nicht mit einem String verglichen werden. Selbstverständlich kann aber die OCL mit Vererbung und Polymorphie umgehen.
- Die Darstellung von Constraints in Klassendiagrammen ist derzeit noch ungenügend. Die OMG schlägt in der UML-Spezifikation eine Notation als Kommentar vor (siehe **Abb. 2**). In größeren Modellen mit mehreren Constraints werden die Diagramme dabei allerdings sehr unübersichtlich.

Rahmenbedingungen

Das Ziel der formalen Dokumentation von Geschäftsregeln ist mit der OCL damit sicherlich erreicht. Allerdings stellt sich nun immer noch die große Aufgabe, diese Regeln im Sinne der *Model Driven Architecture (MDA)* (vgl. [MDA]) in lauffähigen Code umzuwandeln, um sie später in der laufenden Anwendung auch prüfen zu können.

Die wichtigste Erkenntnis hierbei ist die Tatsache, dass ein OCL-Ausdruck niemals für sich alleine stehen kann, sondern sich immer auf Klassen mit deren Attributen und Relationen bezieht. Die Quellcode-Generierung von OCL-Constraints kann also nur dann sinnvoll betrieben werden, wenn auch alle anderen Elemente des

UML-Modells, auf die sich die *Constraints* beziehen, ebenfalls generiert werden.

Bei der hier vorgestellten OCL-Realisierung ist diese Voraussetzung durch die Einbettung in das MDA-Framework „pleXX“ (vgl. [eXX], [Dem03a]) gegeben. Damit ist eine große Hürde bereits genommen: Der im Framework integrierte Generator analysiert das UML-Modell, bietet über ein Metamodell Zugriff auf die Modellelemente und ermöglicht die Generierung von Java-Klassen. Ferner besitzt das Framework bereits die Möglichkeit, *Constraints* direkt in Form von Java-Klassen zu formulieren und zur Laufzeit geregelt prüfen zu lassen (vgl. [Dem03b]).

Die Herausforderung besteht nun also „nur“ noch darin, die OCL-Syntax zu parsen und in Java-Klassen zu transformieren, die vom Framework auf gewohntem Wege verarbeitet werden können. Die Verwendung von bereits am Markt und in der Open-Source-Gemeinde vorhandenen OCL-Parsern (z. B. [Fin00] und [OCLE]) erwies sich dabei allerdings als wenig fruchtbar. Gründe hierfür sind im Wesentlichen die Notwendigkeit der Integration in das bestehende Framework sowie die Begrenzung bzw. Erweiterung des OCL-Sprachschatzes:

Begrenzung

Die gesamte Mächtigkeit der OCL ist zum einen nicht in vollem Umfang auf die Programmiersprache Java anwendbar und hätte zum anderen den Rahmen einer geplanten Realisierung gesprengt.

Erweiterung

Die OCL beschreibt lediglich die Bedingungen, die zur Erfüllung einer bestimmten Anforderung notwendig sind. Eine Nichterfüllung wird mit einem Fehler quittiert. Die OCL macht dabei keinerlei Angaben, wie sich ein Fehler darstellen soll. Im praktischen Einsatz ist es jedoch ebenso wichtig, im Fehlerfall eine festgelegte Aktion durchzuführen oder dem Anwender geeignete Informationen über die Fehlerursache anzugeben. Zu diesem Zweck erweitert die hier vorgestellte Realisierung den OCL-Sprachumfang um das zusätzliche Schlüsselwort otherwise, das bei jedem logischen Ausdruck (boolescher Ergebniswert) angewandt werden kann. Ist das Ergebnis des Ausdrucks logisch falsch (*false*), so wird die als Alternative angegebene Aktion ausge-



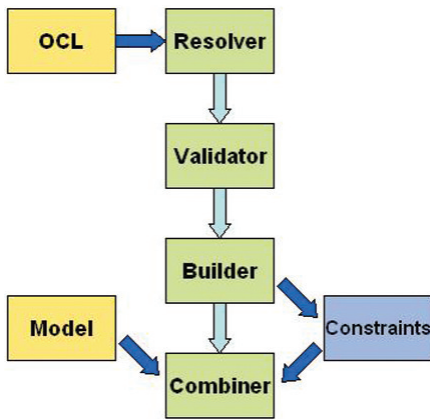


Abb. 3: Ablauf des OCL-Generators

führt. Den Inhalt der alternativen Aktion bestimmt der Entwickler direkt im OCL-Ausdruck als Java-Code. Das *pleXX*-Framework erwartet an dieser Stelle die Rückgabe eines *ConstraintViolation*-Objekts, das der Anwendung als Fehlerindikation weiter gegeben wird.

Realisierung

Die Transformation der OCL-Ausdrücke in äquivalente Java-Klassen erfolgt im so genannten „OCL Compiler“. Dieser interpretiert die modellierten OCL-Ausdrücke, prüft sie gegen das bestehende Objektmodell und erzeugt schließlich die *Constraint*-Klassen, die vom Framework ausgewertet werden.

Das Einlesen der OCL-Syntax erfolgt durch einen eigenen Parser, der mittels des Compiler-Compilers „SableCC“ (vgl. [Gag98]) erzeugt wurde. Der Parser beruht auf einer an den Anwendungszweck angepassten Grammatik der OCL 2.0, die – wie oben beschrieben – begrenzt und erweitert wurde.

Ergebnis des Parser-Durchlaufs ist der abstrakte Syntaxbaum *AST* (*Abstract Syntax Tree*) des eingegebenen OCL-Ausdrucks. Eine *Resolver* genannte Komponente im OCL-Compiler prüft nun die Typkonformität der Teilausdrücke. Das heißt, es wird geprüft, ob die Typen der Baumknoten im AST gemäß den Regeln der OCL zueinander konform sind. Dazu werden alle beteiligten Typen (dazu gehören Benutzerklassen, primitive Typen und Sammlungstypen) in ihre Entsprechung im OCL-Typsystem überführt. Gleichzeitig löst der *Resolver* alle impliziten Bezeichner auf. Dazu gehören nicht vollständig qualifizierte Bezeichner für Klassen (z.B. *Person* anstatt *de::firma::projekt::objekte::Person*)

oder Laufvariablen bei Iterator-Ausdrücken (z. B. *l* in *lieferanten->exists(l | l.hauptlieferant)*). Ferner erweitert der *Resolver* implizite Ausdrücke, wie die Konvertierung zu Sammlungstypen beim Pfeiloperator oder der so genannten *collect*-Kurznotation (d. h. *freunde.name* entspricht *freunde->collect(f | f.name)*).

In einem nächsten Schritt prüft die Komponente *Validator* den typisierten OCL-Ausdruck auf semantische Korrektheit. Das bedeutet, dass beispielsweise ein *Constraint*, der für ein Attribut *name* definiert ist, nicht ein Attribut *alter* prüft. Die weitaus wichtigere Aufgabe der Komponente besteht jedoch darin, diejenigen Stellen im Objektmodell zu ermitteln, an denen die *Constraint*-Prüfung durchgeführt werden muss. Das Framework *pleXX* bezeichnet diese Stellen als *Trigger* (Auslöser). Die Änderung eines als Auslöser deklarierten Attributs hat die Prüfung aller abhängigen *Constraints* zur Folge. So ist im eingangs erwähnten *Constraint* *EinHauptlieferant* das Kennzeichen *hauptlieferant* ein *Trigger* für die *Constraint*-Prüfung im Kontext *Artikel*. Hierin liegt im Übrigen auch ein versteckter, aber sehr wichtiger Vorteil der Verwendung von OCL-*Constraints* gegenüber den herkömmlichen Java-*Constraints* des *pleXX*-Frameworks: Die Ermittlung der *Trigger* kann bei letzterer Variante nicht automatisch erfolgen, sondern sie müssen vom Entwickler manuell definiert werden. Diese potenzielle Fehlerquelle durch Vergessen eines *Triggers* wird vom OCL-Compiler eliminiert.

Nach diesen Vorbereitungen kann eine *Builder* genannte Komponente nun mit der Generierung der eigentlichen *Constraint*-Klassen beginnen. Für jeden im Modell definierten *Constraint* erzeugt die Komponente eine eigene Java-Klasse, die alle vom *pleXX*-Framework geforderten Schnittstellen implementiert. Für die Implementierung der *Constraint*-Prüfung traversiert die Komponente über die Teilausdrücke des OCL-Ausdrucks. In Abhängigkeit vom gerade bearbeiteten Ausdruck erzeugt sie ein Java-Code-Fragment. Ein Fragment besteht in der Regel aus der Deklaration von Zwischenvariablen, aus Wertprüfungen (z. B. Attributwerte bei *Triggern*), aus der Prüfung von Abbruchbedingungen (z. B. *otherwise*), aus der Expansion von Iterator-Ausdrücken oder aus der Konvertierung von Typen (z. B. implizite Umwandlung in

Sammlungstypen). Die Verkettung aller generierten Codefragmente ergibt dann die fertige *Constraint*-Prüfung.

Zuletzt müssen die soeben erzeugten *Constraint*-Klassen durch die Komponente *Combiner* mit den Geschäftsobjekten im Modell verknüpft werden. Dies geschieht bei OCL-*Constraints* über dieselben Mechanismen wie bei herkömmlichen *pleXX-Constraints*. Damit findet der Kern-generator des Frameworks, der im Anschluss gestartet wird, ein gewohntes Bild seiner „herkömmlichen“ *Constraints* – also mit dem Modell verknüpfte Java-Klassen – vor. Der gesamte Ablauf ist schematisch in **Abbildung 3** dargestellt.

Beispiele

Die Praxistauglichkeit der OCL im Allgemeinen und der oben beschriebenen Implementierung im Speziellen lässt sich an Hand einiger Anwendungsfälle aus einem konkreten Projekt recht gut belegen. Es handelt sich dabei um Auszüge aus dem Projekt „Connect“ zur Verwaltung von Bauteilen und Komponenten für Leitungssätze im Automobilbau, das für die DaimlerChrysler AG entwickelt wurde. Zunächst wurde die Einführung der neuen Technologie recht vorsichtig mit einfachen *Constraints* wie

```
inv: minLeitungsdurchmesser <= maxLeitungsdurchmesser
```

oder

```
inv: stecker->isUnique(stk | stk.name)
```

erprobt. Recht schnell kann man sich dann allerdings auch an etwas komplexere Ausdrücke heranwagen, wie beispielsweise an den folgenden *Constraint*, der aufgrund des Zustandes einer Aufzählungsvariable (*typ*) entweder das Feld *eeKomponente* oder die beiden Relationen zu *buchsenAv* und *steckerAv* als Pflichtfelder erwartet. Aus pragmatischen Gründen wird die Regel in die zwei Ausdrücke

```
inv EeKomponenteConstraint:
  typ = VerwendungszweckTyp::EE_KOMPONENTE implies
  eeKomponente->notEmpty()
```

und

```
inv TrennstelleConstraint:
  typ = VerwendungszweckTyp::TRENNSTELLE implies
  (buchsenAv->notEmpty() and steckerAv->notEmpty())
```

aufgespalten. Auch komplizierte Iterator-Ausdrücke wie der Folgende stellen kein Problem dar:



```

inv: lieferantenzuordnungen->select
(l | l.hauptlieferant = true)->size() <= 1 otherwise {
return new de.excellent.connect.util.constraints.ConnectConstraintViolation(
this, contextObject, contextAttribute,
de.excellent.connect.client.resources.business.generated.BusinessResources.CONSTRAINT_EINHAUPTLIEFERANT,
false, contextAttribute.getPropertyValue(contextObject), contextValue, null);
}
    
```

Listing 1: Ein Constraint mit Fehlerindikation im „otherwise“-Block

```

inv: avStift->notEmpty() implies
eeStecker.avKontaktierung.avStifte->exists(einStift |
einStift = self.avStift)
    
```

Hier wird geprüft, ob das Objekt, das über die Relation self.avStift referenziert wird, auch in einer Liste gleichartiger Objekte enthalten ist, die über einen komplexen Navigationsweg abgefragt werden.

Fazit

Nach den ersten Kinderkrankheiten, wie beispielsweise der Generierung doppelter

Variablenamen für Zwischenergebnisse der OCL-Auswertung, hat sich die Realisierung als recht robust erwiesen. Einziger Wermutstropfen in der täglichen Anwendung sind die derzeit noch etwas unhandlichen otherwise-Blöcke zur Reaktion auf den Fehlerfall. Das Beispiel in Listing 1 macht das deutlich.

Der offensichtliche Nachteil liegt in der notwendigen Package-Angabe für alle verwendeten Klassen und Konstanten, da die OCL keine import- oder include-Konstrukte kennt. Hier wäre eine zusätzliche (proprie-

täre) Spracherweiterung hilfreich. Glücklicherweise ist die Reaktion auf Constraint-Verletzungen oft sehr ähnlich, sodass die Produktivität durch in diesem Fall vertretbaren Einsatz von Copy&Paste weiterhin hoch gehalten werden kann. Der Lohn der Mühe ist dann ersichtlich, wenn das Framework die so erstellten Fehlermeldungen ohne weiteres Zutun in der Benutzungsoberfläche präsentiert (siehe Abb. 4).

Abschließend lässt sich feststellen, dass sich der Einsatz der OCL im Projekt in jedem Fall gelohnt hat. Das eingangs gesetzte Ziel ist erreicht: Anforderungen werden korrekter spezifiziert und ohne Interpretations- und Reibungsverluste in Quellcode-Implementierungen überführt. Der modellgetriebene und generative Ansatz trägt einmal mehr zur Steigerung der Softwarequalität bei. ■



Abb. 4: Ein geprüfter OCL-Constraint in der laufenden Anwendung

Literatur & Links

[Dem03a] A. Demelt, Ein Framework für Business-Objekte, in: JavaSPEKTRUM 01/03
 [Dem03b] A. Demelt, Objektvalidierung mit Constraints, in: JavaSPEKTRUM 04/03
 [Fin00] F. Finger, Design and Implementation of a Modular OCL Compiler, TU Dresden, Diplomarbeit, März 2000 (siehe www-st.inf.tu-dresden.de/ocl/ff3/diplom.pdf)
 [Gag98] E.M. Gagnon, SableCC, an Object-Oriented Compiler Framework, McGill University, Montréal, Diplomarbeit, März 1998 (siehe: www.sablecc.org/thesis.pdf)
 [MDA] MDA-Homepage der OMG: www.omg.org/mda
 [OCLE] Object Constraint Language Environment, Laboratorul de Cercetare în Informatic?, Babes-Bolyai Universität in Cluj-Napoca (siehe: lci.cs.ubbcluj.ro/ocle/)
 [eXX] eXXcellent solutions GmbH, pleXX-Homepage, siehe: www.excellent.de/plexx
 [UML] UML-Homepage der OMG: www.uml.org

