UNIVERSITÄT
BAYREUTH

MASTERARBEIT

Constraint-System für eine mehrschichtige Metamodellierungsumgebung

Michael Zeising

Masterarbeit im Fach Angewandte Informatik zur Erlangungen des akademischen Grades *Master of Science*

Vorgelegt von:	Michael Zeising, B. Sc. Geboren am 5. März 1984 in Leipzig
Zeitraum:	1. November 2010 bis 15. März 2011
Prüfer:	Prof. Dr.-Ing. Stefan Jablonski Prof. Dr. Bernhard Westfechtel
Betreuer:	Bernhard Volz, M. Sc.

ERKLÄRUNG

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und wurde nie als Teil einer Prüfungsleistung angenommen. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ort, Datum

Unterschrift

KURZFASSUNG

In vielen Bereichen wird die Bewältigung von komplexen Problemstellungen durch Modelle unterstützt. Modelle beschreiben Software-Systeme, geschäftliche Abläufe, Kommunikationsbeziehungen zwischen Menschen und vieles mehr. Sogenannte Metamodelle beschreiben dabei die Struktur und Bedeutung von Modellen, dienen also als „Sprache“ für deren Formulierung. Die meisten Modellierungswerkzeuge sind eng an ein bestimmtes Metamodell gekoppelt, können also nur zur Entwicklung einer bestimmten „Art“ von Modellen dienen. Ein Ansatz flexiblere Werkzeuge zu erhalten besteht darin, zwischen der Repräsentation und der Bedeutung von Modellen zu trennen. Ein flexibles Werkzeug basiert dann auf einem Metamodell, dass lediglich die Repräsentation von Modellen beschreibt und kann damit zur Entwicklung von Metamodellen selbst dienen.

Zu Beginn der Entwicklung eines Modells darf das Werkzeug so wenige Einschränkungen wie möglich vorgeben. Für bestimmte Anwendungsfälle sind hingegen strikte Regeln für die Form eines Modells sinnvoll. Das Werkzeug muss es daher ermöglichen einem Modell je nach Bedarf Regeln bezüglich seiner Struktur aufzuerlegen.

Für viele inhaltliche Zusammenhänge wären sehr komplexe Modelle notwendig um alle Randbedingungen präzise zu erfassen und manches lässt sich unter Umständen mit den Mitteln der Modellierungssprache überhaupt erst gar nicht ausdrücken. Auch inhaltlich muss es daher möglich sein, dass Modell durch beliebige Randbedingungen zu verfeinern.

In dieser Arbeit wird eine Sprache zur Formulierung solcher Randbedingungen (engl. *constraints*) entwickelt. Diese dienen einerseits dazu, die Modellierungssprache selbst einzuschränken, ermöglichen also den oben erwähnten Wechsel zwischen freien und strikten Modellierungsparadigmen. Andererseits ermöglicht sie eine inhaltliche Verfeinerung von Modellen über die Modellierungssprache hinaus.

ABSTRACT

Models support the accomplishment of complex tasks in many domains. They describe software systems, business procedures, contact relationships among people and a lot more. In this context, meta models declare the structure and meaning of models and thereby form a “language” for expressing models. Most of the modeling tools are strongly coupled to some meta model. They are only capable of supporting the development of one single “kind” of model. An approach for obtaining more flexible tools lies in separating representation and meaning of models. A flexible tool is then based on a meta model that only describes the representation of models and may then be used to develop meta models in turn.

During the initial stages of developing a model the tool must only impose minimal restrictions. In some cases however, strict rules confining the structure of models are reasonable. Tools must therefore be capable of imposing restrictions on the structure of models on demand.

Some instances require very complex models for capturing all their details precisely and some may even not be expressed by the modeling language at all. So it should be possible to refine models by restrictions regarding their content.

Within this thesis a language for expressing such constraints is developed. On the one hand they restrict the modeling language itself and therefore enable the above mentioned switch from free to strict modeling paradigms. On the other hand they allow for refining a model's content beyond the capabilities of the modeling language.

ABBILDUNGEN

TABELLEN

ABKÜRZUNGEN

UML	Unified Modeling Language
OMME	Open Meta Modeling Environment
LMM	Linguistic Meta Model
OCL	Object Constraint Language
OMG	Object Management Group
MOF	Meta Object Facility
QVT	Query View Transformation
PRR	Production Rule Representation
EMOF	Essential Meta Object Facility
EVL	Epsilon Validation Language
EOL	Epsilon Object Language
EMC	Epsilon Model Connectivity
EMF	Eclipse Modeling Framework
LMMC	Linguistic Meta Model Constraints
ANTLR	Another Tool for Language Recognition
EBNF	Erweiterte Backus-Naur-Form

INHALT

1 Motivation

Modelle dienen zur Kommunikation von Ideen und helfen die Komplexität von Problemen zu beherrschen. In der Software-Entwicklung werden Modelle zunehmend gegenüber Programmcode bevorzugt und als „Artefakte erster Klasse“ behandelt. Sie können unabhängig von einer Plattform oder einer Technologie entworfen werden und ermöglichen daher eine Betrachtung auf Problem- statt auf Implementierungsebene [3]. Modelle werden jedoch nicht nur für die Entwicklung von Software verwendet. Ein Ansatz der *Boeing Company* ermöglicht es z. B., das Foto eines Flugzeugs mit Gummimotor zunächst durch ein informelles Modell zu beschreiben, das dann soweit formalisiert werden kann, dass das Flugzeug zu einem simulierten Flug in der Lage ist [4].

Modellierungsumgebungen unterstützen den Entwurf von Modellen und den Umgang mit ihnen. Den klassischen Vertretern wie dem *IBM Rational Software Architect* [5] liegt wiederum ein bestimmtes Model zugrunde, das beschreibt, welche Form die entworfenen Modelle haben können, also welche Modelle sich mit der Software überhaupt entwerfen lassen. Dieses Modell dient somit wiederum zur Formulierung von Modellen und wird daher als „Metamodell“ bezeichnet. Das Metamodell des *Software Architect* ist die *Unified Modeling Language* (UML) [6]. Ein wichtiges Unterscheidungsmerkmal solcher Modellierungsumgebungen ist ihre Anpassungsfähigkeit. Änderungen am Metamodell, also z. B. an der UML, sind (wenn überhaupt) nur in einem Rahmen erlaubt, in dem sie die UML vorsieht. Ein solcher Rahmen sind die UML Profile, mit denen sich das Metamodell an eine Domäne anpassen lässt. Diese Profile erlauben aber keine beliebigen Anpassungen [6, 7]. Änderungen, die über den Rahmen der Profile hinausgehen erfordern eine Änderung der Modellierungsumgebung [8].

Plattformen zur Entwicklung von domänenspezifischen Sprachen ermöglichen den Entwurf einer eigenen Modellierungssprache in einer textuellen und/oder einer grafischen Syntax. Allerdings erlauben Frameworks wie *Xtext* [9] nur zwei Metaebenen: in der entwickelten Sprache wird ein Modell formuliert, das sich selbst nicht wieder als Sprache verwenden lässt, d. h. von dem keine weiteren Instanzen gebildet werden können. Außerdem erfordert eine Änderung der Sprache die erneute Generierung der Entwicklungsumgebung und der mit ihr verbundenen Werkzeuge z. B. zur Code-Generierung. Eine Änderung an der Sprache resultiert also in einem neuen „Produkt“ [8].

Das eigentliche Problem besteht darin, dass im Metamodell der Modellierungsumgebung stets zwei Aspekte vermischt werden. Sowohl die Struktur der Modelle, also deren Form, als auch deren inhaltliche Bedeutung werden in einem Metamodell zusammengefasst. Eine flexible Modellierungsumgebung erfordert allerdings die Trennung dieser Aspekte. Eine Möglichkeit hierfür bietet die *orthogonale Klassifikation* [10] wie sie in Abbildung 1 dargestellt ist. Auf der linken Seite (**L1**) befindet sich ein Metamodell, das ausschließlich die Darstellung von Metamodellhierarchien beschreibt. Da es nur den syntaktischen Aspekt der Modelle spezifiziert, wird es als *Linguistisches Metamodell* bezeichnet. Auf der rechten Seite (**L0**) befindet sich das inhaltliche Metamodell, also eine konkrete Metamodellhierarchie.

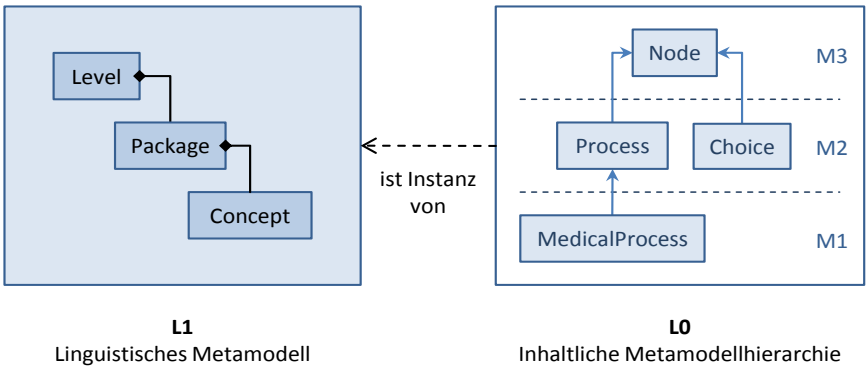


Abbildung 1: Orthogonale Klassifikation: das Linguistische Metamodell beschreibt die Darstellung einer konkreten Hierarchie aus Metamodellen

Die *Open Meta Modeling Environment* (OMME) ist eine Modellierungsplattform auf der Grundlage eines solchen Linguistischen Metamodells [8]. Es handelt sich also nicht um ein Werkzeug zur Erstellung von Modellen zu *einem* Metamodell sondern um ein Werkzeug für die Erstellung von Metamodellhierarchien, also eine sogenannte „Metamodellierungsumgebung“. Die Ausprägung des Linguistischen Metamodells wie sie in der OMME zum Einsatz kommt, wird im Folgenden als *das* LMM bezeichnet.

1.1 Problemstellung und Ziele

Das LMM dient zur Darstellung von Hierarchien aus Metamodellen. In manchen Situationen sind strenge Regeln für die Form einer solchen Hierarchie sinnvoll. Dazu gehören Vorgaben wie eine sich selbst beschreibende oberste Ebene oder Einschränkungen für die Beziehungen zwischen den Metaebenen. Die Einhaltung solcher Regeln wird als strikte Modellierung bezeichnet [11]. Informelle Skizzen, die als erster Entwurf entstehen dürfen hingegen so wenige Restriktionen wie möglich erfahren und erfordern daher eine nicht-strikte Modellierung [8].

Anforderung 1: Constraints auf der Modellierungssprache Um nun sowohl eine strikte als auch eine nicht-strikte Modellierungsweise unterstützen zu können, muss sich das Modellierungsparadigma des LMM jederzeit austauschen lassen und darf nicht fest in die Plattform integriert werden. Grundsätzlich darf die Hierarchie zunächst keinen Einschränkungen unterliegen und erlaubt so zunächst stets eine nicht-strikte Modellierung. Ein striktes Paradigma wird den Metamodellen dann in Form von Randbedingungen (engl. *constraints*) auferlegt. Ein Ziel der Arbeit ist es, eine Sprache zu integrieren, mit der sich solche Einschränkungen der Modellierungssprache formulieren lassen. Da sich diese Constraints auf das Linguistische Metamodell beziehen (**L1** in Abbildung 1), werden sie im Folgenden als linguistische oder sprachliche Constraints bezeichnet.

Auf inhaltlicher Seite (**L0** in Abbildung 1) dient eine Metaebene als „Sprache“ für die darunterliegende. Sie beschreibt deren Aufbau durch Typen, Eigenschaften und viele weitere Aspekte. Eine Ebene könnte z. B. das Konzept „Person“ einführen. Eine Person hat einen Namen und kann eine Beziehung „parents“ mit anderen Personen eingehen. Dieser Zusammenhang lässt sich wie in Abbildung 2 gezeigt als UML-Klassendiagramm darstellen.

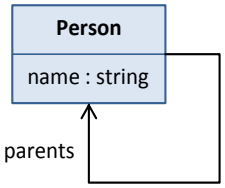


Abbildung 2: Inhaltliches Beispiel in seiner Darstellung als UML-Klassendiagramm

Auf einer darunterliegenden Ebene ließen sich nun Instanzen des Konzepts definieren und miteinander in Beziehung setzen. Bestimmte Aspekte des Modells lassen sich allerdings nur durch sehr komplexe Metamodelle oder gar nicht beschreiben. Die Einschränkung z. B., dass eine Person nicht ihr eigener Elter oder ihr eigener Vorfahre sein darf, lässt sich mit den Mitteln des LMM nicht ausdrücken. Metamodelle lassen sich also nicht präzise genug formulieren.

Anforderung 2: Constraints auf dem Inhalt einer Metamodellhierarchie Um Metamodellhierarchien präzisieren zu können, muss die Modellierungssprache entsprechend erweitert werden. Eine Möglichkeit der Erweiterung ist auch hier die nachträgliche Belegung der Metamodelle mit Einschränkungen, also Constraints. Da sich diese Constraints aber nun nicht auf das LMM selbst, sondern auf eine konkrete Metamodellhierarchie beziehen, werden sie im Folgenden als inhaltliche Constraints bezeichnet. Ein weiteres Ziel der Arbeit ist also, dass sich mit der oben genannten Constraint-Sprache neben sprachlichen auch inhaltliche Constraints entwerfen und auf Metamodelle anwenden lassen. Hierbei ist wichtig, dass inhaltliche Constraints auf inhaltliche Elemente bezogen werden können. Das bedeutet z. B., dass sich ein Constraint auf alle „Personen“ statt auf alle „Konzepte mit dem Namen Person“ beziehen. Diese Constraints müssen sich also „in den Worten“ der Domäne formulieren lassen.

Anforderung 3: Rückmeldung und Unterstützung Wird ein Constraint verletzt, dann muss klar ersichtlich sein, welcher Teil des Modells aus welchem Grund gegen welchen Constraint verstößt und wie schwer diese Abweichung wiegt. Es muss also möglich sein, die entsprechenden Informationen zu ermitteln bzw. einen Constraint mit den notwendigen Informationen anreichern zu können. Nur so kann gewährleistet werden, dass das Constraint-System im Rahmen der Modellierungsumgebung Rückmeldungen an den Benutzer liefert und unterstützend wirkt.

Anforderung 4: Modellierungs- statt Implementierungssprache Wie oben erwähnt, werden Modelle eingesetzt, weil sie eine weitgehend technologieunabhängige Beschreibung auf Problemebene ermöglichen. Sowohl sprachliche als auch inhaltliche Constraints bilden dabei einen Teil des Modells. Die Constraint-Sprache muss also eine Formulierung auf derselben Abstraktionsebene erlauben, was sie von einer universellen Implementierungssprache unterscheidet.

Anforderung 5: Auswertung ohne Generierungsprozess In Frameworks wie Xtext oder dem EMF muss Code generiert werden, bevor Instanzen des spezifizierten Modells erzeugt werden können [9, 12]. Eine wichtige Eigenschaft der OMME hingegen ist, dass eine Metaebene ohne einen solchen Generierungsprozess wiederum als Sprache für die darunterliegende Ebene verwendet werden kann. Die Plattform erfordert an keinem Punkt die Generierung von Code oder ähnlichem. Daher darf auch zwischen dem Entwurf und der Auswertung der Constraints keine Generierung stehen. Constraints müssen sich direkt auf das jeweilige Modell anwenden lassen.

1.2 Aufbau der Arbeit

Zunächst werden in Abschnitt 2 bestehende Sprachen zur Validierung von Modellen betrachtet. Dabei muss vor allem bewertet werden, inwiefern sich die Sprachen zur Lösung der genannten Probleme und für eine Integration in die OMME eignen.

Anschließend wird in Abschnitt 3 das Konzept einer Constraint-Sprache erarbeitet, die die genannten Anforderungen erfüllen kann. Dabei werden in erster Linie die zentralen Mittel und Eigenschaften der Sprache zusammengestellt.

In Abschnitt 4 wird schließlich eine prototypische Implementierung der Sprache vorgestellt. Dabei stehen das Modell und die Syntax der Sprache und deren Integration in die Modellierungsumgebung im Vordergrund.

Abschließend wird die erarbeitete Lösung in Abschnitt 6 bewertet. Außerdem werden Erweiterungen und Verbesserungen vorgeschlagen, die sich in Zukunft auf der Grundlage des erarbeiteten Konzepts realisieren ließen.

2 Verwandte Arbeiten

Im Folgenden sollen bestehende formale Sprachen zur Validierung von Modellen betrachtet werden. Dabei soll zum einen die Sprache an sich untersucht werden; zum anderen aber auch inwiefern sie sich für eine Integration in die Metamodellierungsumgebung OMME eignet.

2.1 Object Constraint Language (OCL)

Die *Object Constraint Language* (OCL) war ursprünglich eine Erweiterung der UML. Sie ersetzte die bis dato verwendeten informellen Anmerkungen an UML-Modellen durch maschineninterpretierbare Constraints. Mittlerweile bildet die OCL einen eigenständigen Standard der *Object Management Group* (OMG) und ist seit Version 2.0 auf beliebige MOF-basierte Metamodelle anwendbar [13]. Neben ihrer Verwendung als Validierungssprache bildet die OCL außerdem den Kern weiterer OMG-Ansätze wie die Modelltransformationssprache *Query View Transformation* (QVT) [14] und die Regelsprache *Production Rule Representation* (PRR) [15].

Die OCL erlaubt im Wesentlichen die Formulierung folgender Konstrukte:

- Invarianten auf den Instanzen bestimmter Typen:

```
context Family::Person
   inv NotOwnParent: not self.parents->includes(self)
```

- Vor- und Nachbedingen für Operationen:

```
context Meeting::duration() : Integer
   pre: self.end > self.start
   post: result = self.end - self.start
```

- Anfrageausdrücke:

```
Family::Person.allInstances()->select(Person person | person.isMale())
```

- Ableitungsregeln für Attribute:

```
context Family::Person::numberOfSiblings : Integer
   derive: self.siblings->size()
```

2.1.1 Integration von Metamodellen

Die OCL-Implementierung des *Eclipse MDT*-Projekts ist prinzipiell unabhängig von der verwendeten Modellierungssprache und arbeitet auf einer Reihe von Schnittstellen, die vom Metamodell abstrahieren. Das Projekt bietet bereits zwei Bindungen für die UML und auch für das Ecore-Metamodell. Da das LMM in Ecore formuliert ist, ließen sich also mit der OCL-Ecore-Bindung bereits sprachliche Constraints auf LMM formulieren und Anforderung 1 ist bereits erfüllt.

Für inhaltliche Constraints (Anforderung 2) ist eine Bindung für LMM notwendig, also eine Implementierung der Schnittstellen für den Zugriff auf LMM-basierte Modelle. Dies wird allerdings durch folgende Umstände erschwert:

- Die Abstraktion basiert auf der Annahme, dass die Modellierungssprache auf der UML bzw. auf EMOF basiert [16]. Das heißt, dass für Konzepte wie z. B. *Classifier*, *Element* oder *Operation* Analoga im Metamodell der einzubindenden Sprache gefunden werden

müssen. Das LMM unterscheidet sich allerdings in vielen Punkten von EMOF. Manche der Konzepte wie z. B. *State* können durch Platzhalter gefüllt werden, manch andere aber nicht.
- Die Schnittstelle ist aufgrund dieser Annahme durchwegs typsicher gestaltet. Dies erfordert, dass neben dem reflektiven Zugriff auf das Metamodell und dem Zugriff auf seine Instanzen auch die gesamte abstrakte Syntax, das interne Typsystem und die Standardbibliothek von OCL implementiert werden müssen [16].

2.1.2 Integration in die Modellierungsumgebung

Invarianten in OCL lassen sich lediglich mit einem Namen versehen. Weitere „Metainformationen" sind nicht vorgesehen. Für eine Integration in die Modellierungsumgebung ist es allerdings notwendig, dass sich Invarianten in mindestens zwei Kategorien unterteilen lassen: in Regeln, die bei Verletzung zu einem Fehler führen und Konventionen, die bei Verletzung lediglich eine Warnung erzeugen [17]. Dieses Problem ließe sich durch eine externe Abbildung von Namen auf Kategorien zumindest kompensieren.

Eine weitere Metainformation, die in OCL nicht unterstützt wird ist eine kontextbezogene Meldung. Im Rahmen einer Entwicklungsumgebung ist es notwendig, dass eine Invariante bei ihrer Verletzung eine aussagekräftige Meldung erzeugt [17]. Solche Meldungen müssen selbst OCL-Ausdrücke sein um Hinweise der Form „*erwartete x aber bekam y*" zu ermöglichen. Dieses Problem lässt sich damit nicht durch einen zusätzlichen Mechanismus beheben und macht OCL für eine benutzerfreundliche Modellierungsumgebung unbrauchbar. Anforderung 3 kann durch eine Integration der OCL also nicht erfüllt werden.

2.1.3 OCL als Modellierungssprache

Die OCL hat den klaren Anspruch keine Implementierungssprache für technisch versierte Anwender sondern eine einfach verständliche Modellierungssprache auf hohem Abstraktionsniveau zu sein [13]. Dennoch beinhaltet die Sprache einige syntaktische Elemente, die sie für Domänenexperten schwer zugänglich macht:

- Für den Aufruf von Eigenschaften eines Skalars wird der Operator `.` verwendet, wohingegen beim Zugriff auf Sammlungen (Sequenzen und Mengen) der Operator `->` verwendet werden muss. Für die Trennung von Namensräumen bei qualifizierten Namen wird wiederum der Operator `::` verwendet. Diese Unzulänglichkeit sorgt für unnötige Komplexität [18].
- Viele eingebaute Typen und Operationen beinhalten das Präfix `Ocl` bzw. `ocl`, wodurch die Ausdrücke insgesamt schlecht lesbar werden.

Diese syntaktischen Schwächen sind hinnehmbar. Schwerwiegender ist allerdings eine konzeptuelle Schwäche: das Typsystem der OCL ist das einer objektorientierten Implementierungssprache [18]. So sind neben den üblichen primitiven Typen (*Integer*, *Real* usw.) auch Sammlungen (*Collection*, *Tuple* usw.) definiert. Um Typsicherheit zu gewährleisten sind diese ähnlich zu Java parametrisiert. Dieses Vorgehen befindet sich aus Sicht der Modellierung auf einem zu niedrigen Abstraktionsniveau, nämlich auf Implementierungsebene. Nimmt man z. B. an, man hätte im Rahmen eines organisatorischen Modells die Beziehung zwischen Mitarbeiter und Abteilung wie in Abbildung 3 modelliert.

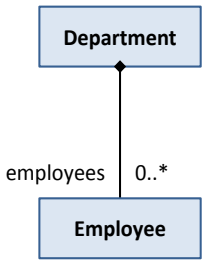


Abbildung 3: Klassendiagramm, das durch die OCL präzisiert werden soll

Wenn man nun in OCL den Wert der Beziehung z. B. in einer Variablen speichern möchte, muss man diese entsprechend typisieren:

```
let someEmployees : Set(Employee) = someDepartment.employees
```

An dieser Stelle findet offensichtlich ein Bruch zwischen der Modellierungs- und der Constraint-Sprache statt, denn diese Formulierung lässt vermuten, dass die Beziehung nicht wie in Abbildung 3 sondern eher wie in Abbildung 4 modelliert wurde.

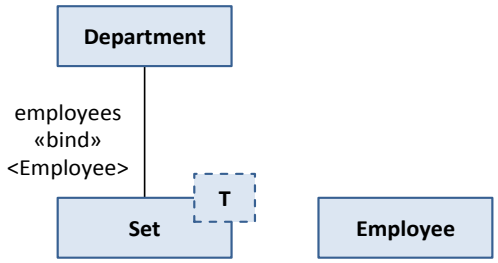


Abbildung 4: Klassendiagramm, das das Typsystem der OCL vermuten lässt

Diese Abbildung von Modellierungs- auf Implementierungsebene muss z. B. im Rahmen von Code-Generierung geschehen aber nicht für Modell-Constraints. Anforderung 4 kann aufgrund des Typsystems der OCL also nicht erfüllt werden.

Die gängigen OCL-Implementierungen bieten die Möglichkeit, die Constraints ohne einen Generierungsprozess zu interpretieren [16, 19]. Anforderung 5 wird daher erfüllt.

2.2 Epsilon Validation Language (EVL)

Das *Epsilon*-Projekt umfasst eine Reihe von Sprachen für den Umgang mit Modellen und die zugehörigen Werkzeuge. Kern des Projekts ist die *Epsilon Object Language* (EOL) für die Navigation innerhalb eines Modells. Alle weiteren Sprachen – darunter auch die *Epsilon Validation Language* (EVL) – basieren auf der EOL. Gegenüber der OCL weist die EVL vor allem die folgenden Veränderungen auf [17]:

- Die Syntax der Sprache wurde vereinfacht. So entfällt z. B. der Operator ->.
- Imperative Konstrukte wie Anweisungen und Zuweisungen wurden in die Sprache aufgenommen.
- Kontextbezogene Meldungen werden unterstützt.
- Invarianten lassen sich in *constraint* (erzeugt einen Fehler) und *critique* (erzeugt eine Warnung) unterscheiden.
- Es lassen sich Abhängigkeiten zwischen Constraints herstellen.

- Für eine Invariante lassen sich Reparaturen (*fixes*) angeben, die das Modell so ändern, dass die Invariante nicht mehr verletzt wird.

Folgender EVL-Constraint stellt z. B. sicher, dass der Name einer *Class* mit einem Großbuchstaben beginnt. Mit dem Constraint werden außerdem Reparaturanweisungen und eine Meldung verbunden, die sich auf die Instanz bezieht:

```
context UML::Class {
  constraint ClassNameIsUpperCase {
    check : self.name.substring(0, 1)
        = self.name.substring(0, 1).toUpperCase()
    message : 'Name ' + self.name
        + ' does not start with an upper case letter'
    fix {
      title : 'Change name to upper case'
      do {
        self.name := self.name.substring(0, 1).toUpperCase()
            + self.name.substring(1);
      }
    }
  }
}
```

Die EVL wird auch außerhalb des Epsilon-Projekts z. B. in der Modellierungsumgebung *MetaDepth* als Validierungssprache verwendet [20].

2.2.1 Einbindung von Metamodellen

Ähnlich zur MDT-OCL-Implementierung sind auch die Epsilon-Sprachen prinzipiell unabhängig von der verwendeten Modellierungssprache. Sie werden über eine Reihe von Schnittstellen namens *Epsilon Model Connectivity* (EMC) vom jeweiligen Metamodell entkoppelt. Der Unterschied gegenüber MDT-OCL ist, dass die Schnittstellen der EMC nicht typsicher sind (vgl. Abbildung 5).

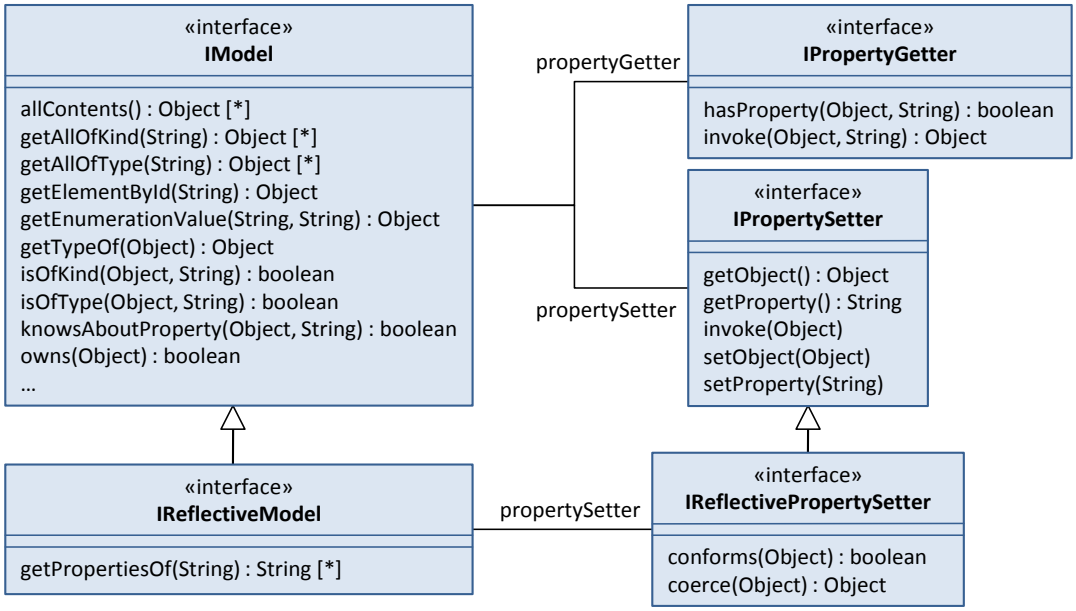


Abbildung 5: Ausschnitt aus der *Epsilon Model Connectivity*-Schicht [1]

Die Bindung an das Metamodell ist dadurch zwar weniger robust, die Schnittstelle kann aber wesentlich einfacher gehalten werden [1]. Die Integration eines eigenen Metamodells wie das LMM ist daher mit deutlich weniger Aufwand verbunden und Anforderung 2 lässt sich erfüllen. Eine Implementierung der EMC für das Ecore-Metamodell ist bereits Teil des Projekts, wodurch Anforderung 1 bereits erfüllt ist.

2.2.2 Eignung für Metamodellierungsplattform

Durch die Unterstützung von kontextbezogenen Meldungen und die Klassifizierung von Constraints in verschiedene „Schweregrade“ bietet die EVL weitreichende Möglichkeiten zur Unterstützung des Benutzers bei der Modellierung und erfüllt damit Anforderung 3.

Das Typsystem der OCL wurde unverändert in die EVL übernommen. Die Sprache befindet sich also in dieser Hinsicht ebenso auf Implementierungsebene wie OCL. Zusätzlich wurden imperative Konstrukte wie Anweisungen, Blöcke und Zuweisungen aufgenommen, sodass die Abstraktion vom Maschinenmodell verloren geht und es sich bei der EVL nicht mehr um eine deklarative Sprache handelt. Anforderung 4 wird von der EVL daher nicht erfüllt.

Die entsprechende Literatur enthält keine näheren Informationen zur Typisierung der EVL. An der Grammatik wird aber ersichtlich, dass Typangaben durchwegs optional sind[1]. Die Sprache ist damit de facto dynamisch typisiert. Das erweist sich als Problem, da sie verwendet werden soll um statisch typisierte Modelle zu präzisieren (siehe Abschnitt 3.5.1).

Die EVL-Constraints werden von einem Interpreter über die EMC-Schicht direkt auf das Modell angewendet, wodurch Anforderung 5 erfüllt ist.

2.3 Check Language

Die Sprache *Check* wurde ursprünglich von *openArchitectureWare* entwickelt und ist heute Teil des Eclipse-Projekts *M2T Xpand*. Die Sprache ist ebenfalls stark an die OCL angelehnt und bietet ihr gegenüber ähnliche Verbesserungen wie die EVL [21].

Die Check-Constraints müssen zur Entwurfszeit formuliert werden. Sie können sich daher lediglich auf das Metamodell der domänenspezifischen Sprache [9], also das LMM beziehen und nicht auf dessen Instanzen. Es ist somit zum einen nicht möglich zur Laufzeit der Modellierungsumgebung ohne Generierungsprozess Invarianten auf der Modellierungssprache zu formulieren. Anforderung 5 kann daher durch Check nicht erfüllt werden. Zum anderen ließen sich damit auch keine inhaltlichen Constraints formulieren, da sich diese auf Instanzen des LMM beziehen. Anforderung 2 kann durch eine Integration von Check also ebenfalls nicht erfüllt werden.

2.4 Z-Notation und Alloy

Die *Z-Notation*, eine auf Mengen und Relationen basierende Modellierungssprache, ist ein früher Vorläufer der OCL. Bereits hier wurde ein System durch Typen beschrieben und durch Invarianten bezüglich dieser Typen weiter eingeschränkt [2].

---

[1] Grammatik der Basissprache EOL in `http://dev.eclipse.org/svnroot/modeling/org.eclipse.gmt.epsilon/trunk/plugins/org.eclipse.epsilon.eol.engine/src/org/eclipse/epsilon/eol/parse/EolParserRules.g` Revision 1221 vom 21.10.2010

[2] In Prolog beginnen Konstanten mit einem kleinen und Variablen mit einem großen Buchstaben.

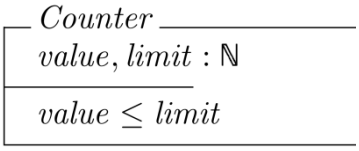


Abbildung 6: Die Definition eines Typs *Counter* mit zwei Eigenschaften *value* und *limit* und einer Invariante in der Z-Notation [2]

Alloy ist eine formale Sprache, die direkt auf der Z-Notation aufbaut. Die Sprache verfügt über einen sehr kleinen Kern mit klar definierter Semantik. Im Unterschied zur UML werden Typen und Beziehungen durchwegs als Mengen und Relationen behandelt [22]. Strukturelle Aspekte wie sie von der UML abgedeckt werden und Randbedingungen wie sie in OCL formuliert werden können sind in Alloy in einer Sprache vereint. So lassen sich manche Invarianten bereits im strukturellen Teil des Modells beschreiben. Folgender Ausdruck deklariert z. B. eine Beziehung `parent` zwischen den Typen `File` und `Directory`, die sich mit der Beziehung `children` nicht überschneiden darf:

```
parent (~ children) : File -> Directory
```

Alloy erfüllt Anforderung 4 von allen vorgestellten Ansätzen am besten, da die Sprache von jeglichen Implementierungsdetails abstrahiert. Alloy scheidet für eine Integration in die Modellierungsumgebung aber aus, da die Constraint-Sprache ausdrücklich Teil der Modellierungssprache ist und sich nicht auf andere Sprachen anwenden lässt [18].

2.5 Prolog-basierte Ansätze

Prolog ist eine deklarative logische Programmiersprache, die es erlaubt regelbasierte Anfragen an Modelle zu stellen. In Prolog bildet eine sogenannte *Wissensbasis* das eigentliche Programm. Im folgenden Beispiel enthält die Wissensbasis zwei *Fakten* und eine *Regel*[2]:

```
parent(alice, bob).
brother(bob, charlie).
uncle(X, Y) :-
   parent(X, Z),
   brother(Z, Y).
```

Das Programm wird ausgeführt, indem die Wissensbasis geladen und Anfragen an den Interpreter gestellt werden:

```
?- uncle(alice, Uncle).
Uncle = charlie.
```

Die Lösungen der Anfrage werden nach einem festen Suchalgorithmus ermittelt. Durch die Verwendung von freien Variablen erlaubt die Sprache eine sehr prägnante Formulierung komplexer Constraints.

Das *Prolog Intelligence Plugin* für das *Generic Eclipse Modeling System* (GEMS) ermöglicht es, Prolog-Constraints auf EMF-Modelle anzuwenden. Es fügt das Modell einer Wissensbasis hinzu und ruft anschließend den SWI-Prolog-Interpreter über eine JPL-Schnittstelle auf [23]. Das *Model Manipulation Toolkit* (MoMaT) integriert u. a. UML-Modelle und *Ereignisgesteuerte Prozessketten* (EPK) indem es sie in eine Wissensbasis importiert und per Prolog-Anfragen auf sie zugreift [24].

[2] In Prolog beginnen Konstanten mit einem kleinen und Variablen mit einem großen Buchstaben.

Die Beispiele für Prolog-basierte Plattformen zeigen ein grundlegendes Problem auf: wird das Modell nicht bereits in Form von Prolog-Fakten dargestellt muss der Auswertung der Constraints stets ein aufwändiger Transformationsschritt vorgeschalten werden. Die Modellelemente müssen in atomare Aussagen überführt und der Wissensbasis hinzugefügt werden. Anforderung 5 kann von Prolog-basierten Ansätzen daher nicht erfüllt werden.

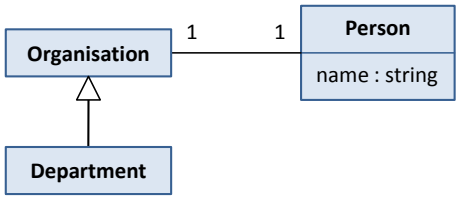


Abbildung 7: UML-Modell, das auf Prolog-Fakten abgebildet werden soll

Das in Abbildung 7 gezeigte Klassendiagramm würde im MoMaT-Projekt z. B. auf folgende Aussagen abgebildet werden [24]:

```
me(class-0,           [name-'Person', attributes-ids(1, 14)]).
me(feature-1,         [name-name, type-string]).
me(class-10,          [name-'Organisation', attribute-ids(18)]).
me(class-11,          [name-'Department']).
me(feature-13,        [multiplicity-1, type-id(0)]).
me(feature-14,        [multiplicity-1, type-id(10)]).
me(generalization-15, [from-id(11), to-id(10)]).
me(association-17,    [ends-ids([13, 14])]).
```

Constraints und Transformationen müssen bzgl. dieser Darstellung formuliert werden, was ein weiteres Problem aufzeigt: vom Modell zur Anfrage ist ein erheblicher Denkschritt notwendig. Die Abbildung auf Prolog ist nicht eindeutig und der Benutzer muss sich zunächst mit der jeweiligen Abbildung auseinandersetzen bevor er in der Lage ist, Anfragen zu formulieren. Anforderung 4 wird also nicht erfüllt.

Dazu kommt, dass es sich bei Prolog um eine dynamisch bzw. gar nicht typisierte Sprache handelt. Dies steht im Widerspruch zu typisierten Modellen (vgl. Abschnitt 3.5.1).

2.6 Zusammenfassung

Der OCL fehlen die Mittel für eine Rückmeldung an den Benutzer und ihr Typsystem verlangt einen Wechsel von der Modellierung zur Implementierung. Weiterentwicklungen wie die EVL bieten kleine Verbesserungen, übernehmen aber das Typsystem. Interessante Ansätze wie der Constraint-Teil von Alloy lassen sich nicht aus der entsprechenden Modellierungssprache herauslösen und Prolog-basierte Lösungen erfordern aufwändige Transformationen und ein inhaltliches Umdenken. Die vorgestellten Ansätze eignen sich also aus verschiedenen Gründen nicht für eine Integration in die Metamodellierungsplattform.

3 Konzept einer Constraint-Sprache für eine mehrschichtige Metamodellierungsumgebung

Im Folgenden soll die konzeptionelle Grundlage der Sprache *Linguistic Meta Model Constraints* (LMMC) zur Validierung von Metamodellhierarchien entworfen werden. Sie soll es erlauben Constraints so weit wie möglich auf Problemebene zu formulieren und dabei so wenige Implementierungsdetails wie möglich preisgeben. Daneben soll sie sich nahtlos in die OMME-Plattform integrieren wozu auch eine unterstützende Entwicklungsumgebung für die Sprache notwendig ist.

Zunächst soll der Entwurf und die Auswertung von Constraints in den Entwurfsprozess von Modellen eingeordnet werden. Anschließend soll an dieser Stelle der Zusammenhang zwischen der Modellierungs- und der Constraint-Sprache festgelegt werden. Außerdem werden die Grundlagen der Sprache wie das Paradigma und die Typisierung entwickelt. Daneben werden Mittel konzipiert, die besonders prägnante Formulierungen im Zusammenhang mit Modellen als Graphen erlauben. Zuletzt werden schließlich Aspekte beleuchtet, die für die Integration der Constraint-Sprache in die Modellierungssprache von Bedeutung sind.

3.1 Linguistische Ebenen

LMMC soll als Constraint-Sprache in die Modellierungsplattform OMME integriert werden. Grundlage dieser Plattform bildet die *orthogonale Klassifikation* [10], also die Trennung von Semantik und Darstellung der Modelle. Zur Darstellung wird das *Linguistische Metamodell* (LMM) [25] verwendet; es dient also zur Repräsentation von Metamodellhierarchien.

3.1.1 Das Linguistische Metamodell (LMM)

Abbildung 8 zeigt die wesentlichen und für das Konzept relevanten Bausteine des LMM. Einstiegspunkt bildet das Modell (*MModel*), das sich in Metaebenen (*MLevel*) untergliedert. Modelle und Ebenen können jeweils miteinander in Beziehung gesetzt werden (*MModelReference* und *MLevelReference*) und das Modell kann eine Semantik in Form einer Constraint-Sammlung referenzieren (*MSemanticsReference*). Der Inhalt einer Metaebene lässt sich in Paketen (*MPackage*) strukturieren, die schließlich sogenannte Referenztypen (*MReferenceType*) enthalten. Die beiden Referenztypen sind das Konzept (*MConcept*) und die Aufzählung (*MEnumeration*). Ein Konzept kann gleichzeitig Attribute (*MAttribute*) deklarieren und Attributen Werte zuweisen (*MAssignment*). Es verfügt also sowohl über eine Typ- als auch über eine Instanzfacette und bildet damit ein sogenanntes *Clabject* [26]. Statt diesem technischen Begriff wird der neutrale Begriff „Konzept“ verwendet. Diese Vereinigung aus Klasse und Objekt lässt sich mit der UML bzw. der MOF z. B. nicht realisieren. Durch dieses und weitere Modellierungsmuster ermöglicht das LMM eine prägnante Formulierung flexibler Modelle ohne die formale Grundlage zu verlieren [8].

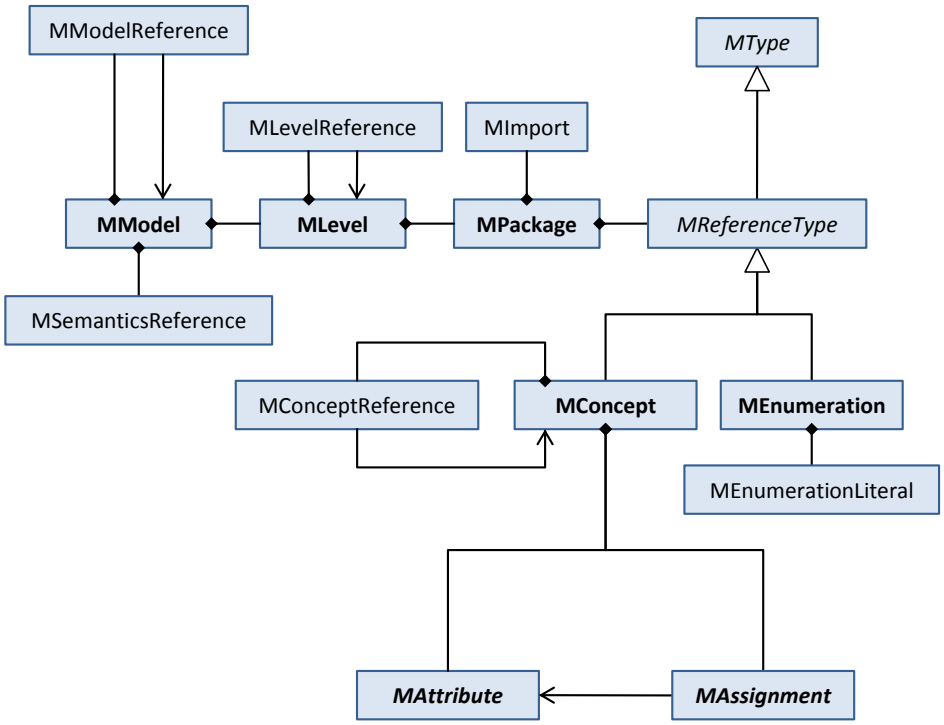


Abbildung 8: Struktur des Linguistischen Metamodells

3.1.2 Sprachliche Constraints

Sprachliche Constraints schränken die Modellierungssprache selbst, also das LMM ein. Sie beziehen sich daher auf Elemente der Sprache. Eine Sammlung an sprachlichen Constraints entscheidet also über die zulässigen Modellierungskonstrukte und kann daher als „Modellierungsparadigma“ bezeichnet werden. Da das LMM in Ecore formuliert ist, bildet Ecore das Metamodell dieser Constraints. Bezieht sich also eine Invariante auf einen Typ *MConcept*, so müssen alle *EObject*s der *EClass* mit dem Namen „MConcept“ überprüft werden (vgl. Abbildung 9).

3.1.3 Inhaltliche Constraints

Innerhalb einer in LMM dargestellten Metamodellhierarchie dient eine Ebene als „Sprache“ für die darunterliegende. Inhaltliche Constraints beziehen sich auf Konzepte, die im Kontext des LMM als Typen auftreten. Sie decken also den Teil ab, der in einem UML-Modell von OCL erfüllt wird und bilden somit die „Geschäftsregeln“ (engl. *business rules*). Das Metamodell dieser Constraints bildet das LMM. Wenn sich eine Invariante also auf einen Typ *Process* bezieht, müssen innerhalb des Modells alle Konzepte überprüft werden, die eine Instanziierungsbeziehung mit dem *MConcept* namens „Process“ eingehen (vgl. Abbildung 9).

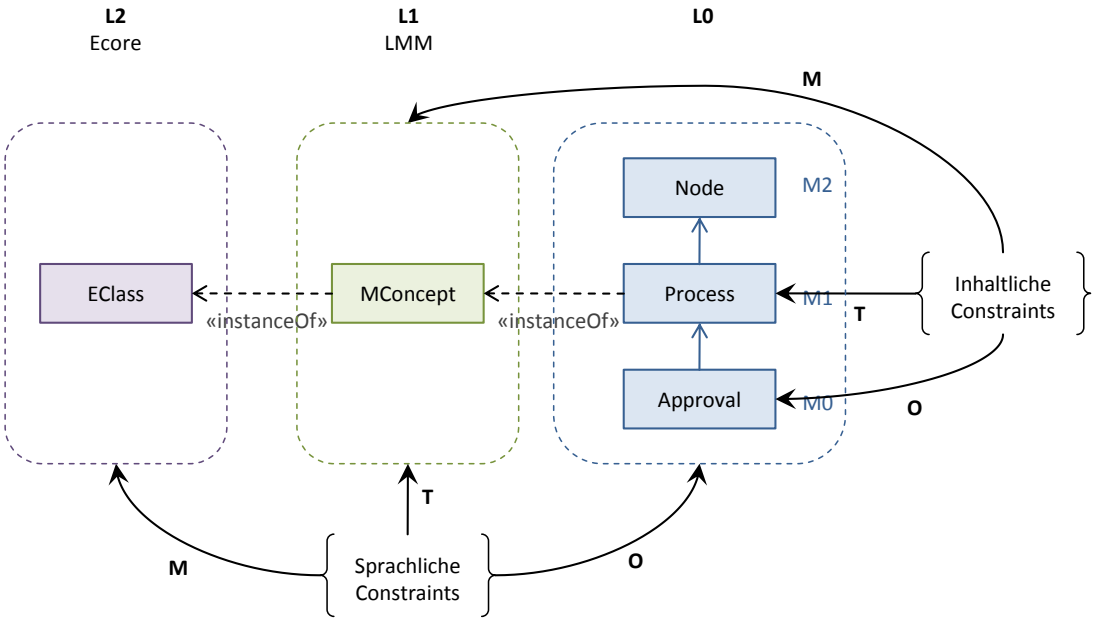


Abbildung 9: Beziehungen zwischen den beiden Constraint-Arten und den sprachlichen Ebenen. Der Pfeil *O* gibt an, auf welcher Ebene die validierten Objekte liegen; *T* gibt an, auf welcher Ebene deren Typen liegen, also auf welche Ebene sich die Constraints beziehen; *M* gibt an, auf welcher Ebene wiederum deren Typen liegen, also welche Ebene das Metamodell der Constraints bildet.

3.2 Struktur von Constraints

Das Prinzip der Trennung von strukturellem Modell und Constraints wie es die OCL umsetzt soll für LMMC grundsätzlich beibehalten werden. Die grundlegende Deklaration der Konzepte, ihrer Eigenschaften und der Beziehungen zwischen ihnen werden auf inhaltlicher Ebene vom LMM und auf sprachlicher Ebene von Ecore abgedeckt. In LMMC wird anschließend auf die Typen des jeweiligen Metamodells Bezug genommen. In Form von Invarianten bezüglich dieser Typen wird das Modell eingeschränkt. Es werden also auf Typebene Zusammenhänge beschrieben, die für alle Instanzen des Typs gelten müssen.

Durch die Klassifikation von Constraints in Regeln und Konventionen und die Unterstützung von Meldungen, wie sie die EVL bietet (vgl. Abschnitt 2.2), wird eine Integration der Constraint-Sprache in eine Modellierungsumgebung überhaupt erst sinnvoll. Daher sollen diese Konzepte auch in LMMC zur Verfügung stehen.

Unabhängig davon ob das jeweils zugrunde liegende Metamodell (LMM oder Ecore) Operationen kennt oder nicht, lassen sich die Typen durch Methoden erweitern. Diese können nur im Rahmen von LMMC aufgerufen werden; der jeweilige Typ im Modell wird also nicht verändert. Die Methoden ermöglichen die Wiederverwendung von Constraint-Teilen und tragen somit zur Übersichtlichkeit der Constraint-Sammlung bei. LMMC ermöglicht also die Definition von Invarianten und Methoden für Typen.

3.3 Einordnung in den Modellierungsprozess

Abbildung 10 zeigt, wie sich der Umgang mit Constraints in den Modellierungsprozess einfügt. Das Modell und die Sammlung von Constraints werden unabhängig voneinander angelegt. Invarianten werden bezüglich eines bestimmten Metamodells formuliert, das zunächst referenziert werden muss. Im Fall von sprachlichen Constraints ist das Metamodell das LMM, was

Teil der Plattform und damit immer bereits verfügbar ist. Für inhaltliche Constraints muss die LMM-Instanz referenziert werden, die als Metamodell dienen soll. Wurde ein Metamodell referenziert, können Constraints entworfen werden. Um nun ein bestimmtes Modell mit den Constraints einzuschränken, müssen diese innerhalb des Modells referenziert werden.

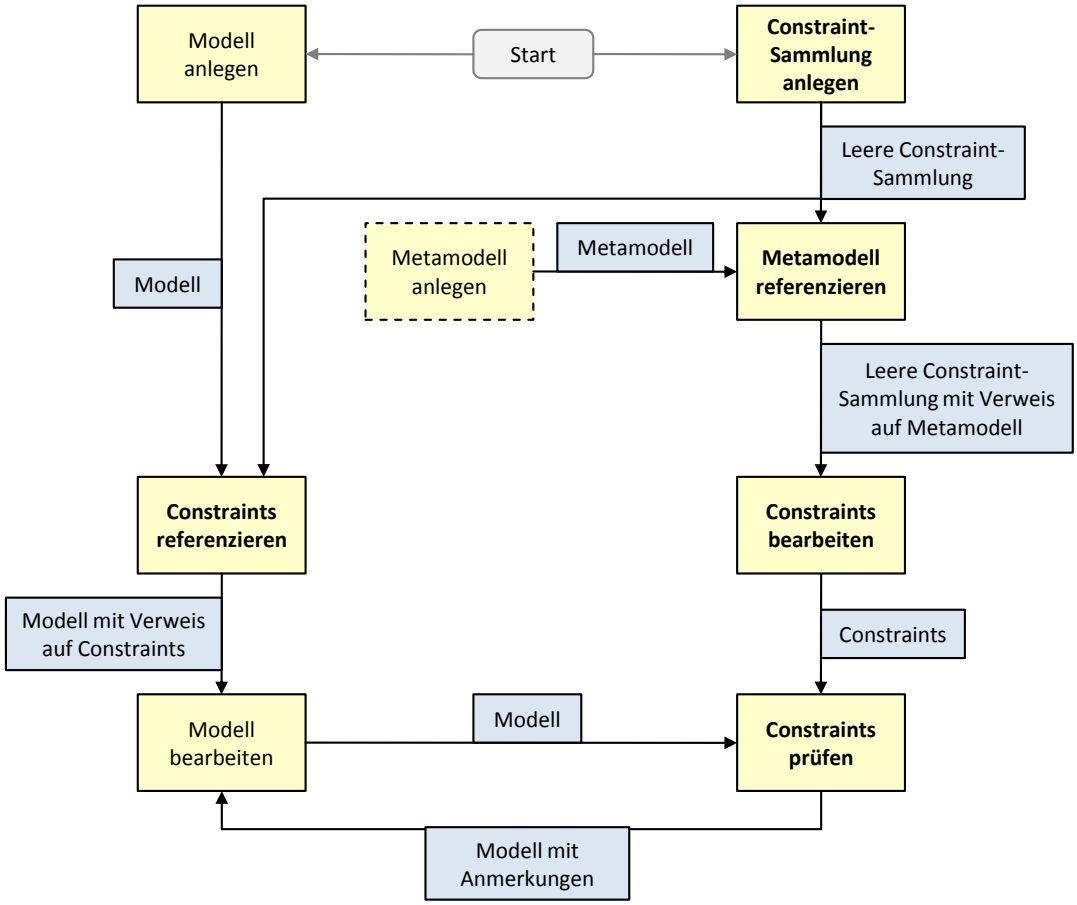


Abbildung 10: Zusammenhang zwischen dem Umgang mit Constraints und dem Bearbeitungsprozess von Modellen im Rahmen der OMME-Plattform. Die gelben Rechtecke stellen die Schritte des Prozesses dar. Die blauen symbolisieren die Daten, die von einem Schritt angeboten und einem anderen benötigt werden.

Nun kann die Prüfung der Constraints an beliebigen Punkten des Bearbeitungsprozesses des Modells eingehängt werden. Für die weiteren Entwurfsentscheidungen sind zwei Aspekte von besonderer Wichtigkeit:

- Die Bearbeitung des zu überprüfenden Modells und die Prüfung der Constraints schließen sich zeitlich aus. Während der Auswertung muss also sichergestellt sein, dass sich das Modell nicht verändert. Diese Annahme wird auch im Rahmen anderer Sprachen zur Validierung von Modellen getroffen [13].
- Die Auswertungsreihenfolge von (unabhängigen) Constraints ist nicht festgelegt. Referenziert ein Modell außerdem zugleich sprachliche und inhaltliche Constraints ist auch hier nicht festgelegt, in welcher Reihenfolge die beiden „Arten“ von Constraints ausgewertet werden.

3.4 Funktionales Paradigma

Eine Constraint-Sprache für die Modellierung befindet sich an der Grenze zwischen domänenspezifischer Sprache auf der einen Seite und universeller Programmiersprache auf der anderen. Einerseits ist sie als domänenspezifische Sprache für einen besonderen Zweck konzipiert und verfügt daher über einen begrenzten Sprachumfang und begrenzte Ausdrucksfähigkeit [27]. Andererseits muss sie in der Lage sein, auch komplexeste Constraints darzustellen und benötigt dafür die Vollständigkeit einer universellen Sprache. Der Rumpf eines Constraints, also die Bedingung, die eingehalten werden soll, verlangt eine Sprache mit sehr hoher Ausdrucksfähigkeit. Es stellt sich die Frage, welches Paradigma diese umsetzen soll.

In imperativen Sprachen werden Algorithmen in Form von Anweisungen dargestellt:

```
factorial(n) {
    b := 1;
    while n > 0 {
        b := n * b;
        n := n - 1;
    }
    return b;
}
```

Diese imperative Darstellung des Algorithmus besteht aus einer Reihe von Zuweisungen, Sprüngen und Speichermanipulationen und spiegelt somit das Maschinenmodell, also die Von-Neumann-Architektur wider [28].

Funktionale Sprachen (und deklarative im Allgemeinen) hingegen kennen keinen internen Speicherzustand und keine Anweisungen. Programme setzen sich aus Ausdrücken und Funktionen zusammen. Anstelle von Schleifen und Sprüngen enthalten sie Rekursion. Funktionale Sprachen erreichen dadurch ein höheres Abstraktionsniveau als imperative, da sie vom Maschinenmodell abstrahieren:

```
factorial(n) =
    if n > 1 then n * factorial(n - 1)
    else 1
```

Daneben verfügen sie typischerweise über eine Reihe von speziellen Konstrukten, die sehr kompakte Formulierungen und weitere Abstraktionen ermöglichen (anonyme Funktionen, Funktionen höherer Ordnung, Mustererkennung usw.). Reine funktionale Sprachen erzeugen keine Seiteneffekte und werden daher als *referentiell transparent* bezeichnet. Aufgrund dieser Eigenschaft lassen sich funktionale Programme leicht verifizieren – formal und auch beim Schreiben und Debugging [28].

Auch wenn einige wenige Ansätze wie die Sprache *Erlang* in der Telekommunikationsbranche erfolgreich sind [29], gelten funktionale Sprachen im Allgemeinen als akademisch und spielen in der Software-Industrie insgesamt eine untergeordnete Rolle [30]. Dennoch basieren viele bei Endanwendern erfolgreiche Sprachen oder Systeme auf funktionalen Prinzipien. *Microsoft Excel* ist im Grunde eine funktionale Programmiersprache [31]. SQL enthält viele der typischen Elemente [32] und auch die Transformationssprache XSLT wurde als funktionale Sprache identifiziert [33]. Nicht zuletzt halten auch immer mehr funktionale Konzepte Einzug in die bedeutenden ursprünglich imperativen Sprachen wie C# und Java [34, 35].

Die Constraint-Sprache soll es ermöglichen auf hohem Abstraktionsniveau Randbedingungen für Modelle zu formulieren. Sie soll auf Problemebene angesiedelt, also möglichst

implementierungsunabhängig sein. Darüber hinaus dürfen sich mit der Sprache keine Änderungen am Modell bewirken lassen. Zuweisungen sollten daher idealerweise nicht zur Verfügung stehen. Ein Paradigma, dass all diese Anforderungen erfüllt ist das funktionale. Die Sprache LMMC soll also im Kern eine funktionale Sprache sein und die typischen Konstrukte soweit sie für die Modellierung relevant sind unterstützen.

3.5 Typisierung

3.5.1 Statische Typisierung

Bei der Typisierung von formalen Sprachen unterscheidet man generell zwischen dynamischer und statischer Typisierung. In dynamisch typisierten Sprachen werden die Typen erst zur Laufzeit überprüft. Daher sind Variablen hier oft nicht typisiert, da sie beliebige Werte aufnehmen können. Diese Sprachen gelten als besonders gut geeignet für Prototypen mit sich oft ändernden oder gar unbekannten Anforderungen [36]. Bekannte Vertreter dieser Kategorie sind *Smalltalk* [37] und *Python* [38]. In einer statisch typisierten Sprache hingegen wird der Typ einer Entität bereits zur Entwurfszeit festgelegt. Statisch typisierten Sprachen wird zugesprochen, dass sie eine frühe Erkennung von Programmfehlern ermöglichen, zur Dokumentation durch Signaturen beitragen und mächtigere Entwicklungsumgebungen ermöglichen (z. B. durch *Refactoring* und Code-Vervollständigung) [36]. Diese Argumentation bezieht sich allerdings auf Implementierungssprachen und lässt sich sicher nur zum Teil auf die Modellierung übertragen.

Eine dynamische Typisierung hätte praktische Nachteile beim Entwurf von Constraints. Sie erfordert mehr Achtsamkeit und das Aufspüren von Fehlern aufgrund falscher Objekte ist mühsam und lässt sich nur bedingt durch eine Entwicklungsumgebung unterstützen. Darüber hinaus dient das LMM zur Formulierung von statisch typisierten Modellen [39]. Eine ebenso typisierte Constraint-Sprache erleichtert dem Benutzer den Übergang zwischen Modellierung und Constraints und ermöglicht außerdem die oben genannte Unterstützung durch eine Entwicklungsumgebung. LMMC verfügt also über eine streng statische Typisierung, die vollständig zur Entwurfszeit angegeben werden muss.

3.5.2 Modellierungsorientiertes Typsystem

Wie in Abschnitt 2.1.3 gezeigt gleicht das Typsystem der OCL dem einer objektorientierten Implementierungssprache wie z. B. Java. Obwohl im entsprechenden UML-Modell eine Beziehung z. B. mit `0..*` `T` typisiert wurde, muss sie in OCL auf `Collection(T)` oder eine Spezialisierung abgebildet werden [13]. Dies entspricht einer Abbildung von der Modellierungs- auf die Implementierungsebene, wie sie z. B. bei der Generierung von Java-Code aus Ecore-Modellen geschieht. Da sich aber sowohl die Modellierungs- als auch die Constraint-Sprache auf der Modellierungsebene befinden sollen, darf diese Abbildung nicht notwendig sein.

In LMMC besteht eine Typangabe aus einer Referenz auf den eigentlichen Typ – im Folgenden „Kerntyp“ genannt – und einer Kardinalität. Für die Angabe dieser Kardinalität sollen die Multiplizitäten der UML [6] in vereinfachter Form übernommen werden. Eine Multiplizität ist ein einschließendes Intervall aus nicht-negativen Ganzzahlen, das aus einer unteren und einer oberen Grenze besteht. Die obere Grenze muss dabei nicht festgelegt sein und kann auf unendlich gesetzt werden. Eine Typangabe T in LMMC ist demnach ein Tupel aus dem Kerntyp t und einem Intervall $\mathrm{m_t}$ von der Untergrenze l bis zur Obergrenze u:

$$\mathrm{T} = (\mathrm{t}, \mathrm{m_t}) \qquad \text{mit } \mathrm{m_t} = [\mathrm{l}, \mathrm{u}]$$

Die Typisierung $(\mathrm{Process}, [1, *])$ verlangt also z. B. mindestens eine Instanz des Typs $\mathrm{Process}$, während $(\mathrm{Agent}, [0, 1])$ eine oder keine Instanz von Agent erfordert.

Beim Umgang mit einem Typsystem ist entscheidend, wie sich Typen zueinander verhalten, d. h. ob ein Typ mit einem anderen kompatibel ist oder nicht. Dazu habe eine Funktion f folgende Signatur:

$$f(X) : Y$$

Sie nimmt also einen formalen Parameter vom Typ $X = (x, m_x)$ entgegen und liefert einen Wert vom Typ $Y = (y, m_y)$. Soll die Funktion nun mit einem Aktualparameter $P = (p, m_p)$ aufgerufen werden, muss bestimmt werden ob der Typ P mit dem Typ X kompatibel ist. In der Objektorientierung gilt im Wesentlichen, dass ein Typ A mit einem Typ B kompatibel ist, wenn A eine Spezialisierung von B oder A mit B identisch ist [40]. Dies lässt sich ohne Einschränkung auf die Kerntypen p und x der Typangaben P und X übertragen. Für die Multiplizitäten lässt sich der Zusammenhang als Teilmengenrelation verstehen. Die Multiplizität m_p ist demnach kompatibel mit m_x, wenn das Intervall m_p eine Teilmenge von m_x beschreibt.

Insgesamt gilt also, dass ein Typ P mit einem Typ X kompatibel ist, wenn sein Kerntyp p den Kerntyp x spezialisiert oder mit ihm identisch ist und die Multiplizität m_p eine Teilmenge von m_x ist:

$$\textbf{compatible}(P, X) \Leftrightarrow (\textbf{extends}(p, x) \vee p \equiv x) \wedge m_p \subset m_x$$

Nimmt man z. B. an, dass der Typ $Special$ eine Spezialisierung von $General$ ist, dann sind folgende Angaben kompatibel:

- $(General, [1, 1])$ und $(General, [1, 1])$
- $(Special, [1, 1])$ und $(General, [1, 1])$
- $(General, [0, 1])$ und $(General, [0, *])$
- $(Special, [1, 1])$ und $(General, [0, *])$

Nicht kompatibel sind degegen z. B. folgende Paare:

- $(General, [1, 1])$ und $(Special, [1, 1])$
- $(Special, [0, 1])$ und $(Special, [1, *])$

Dieses Typsystem erhält den Charakter des Modells, indem die für Modelle üblichen Kardinalitäten auch in der Constraint-Sprache verwendet werden können. Dies macht Implementierungskonzepte wie Sammlungstypen und Typparameter für die Abbildung von mehrwertigen Beziehungen überflüssig und vermeidet den Bruch zwischen Modellierung und Constraints.

3.5.3 Sichere Typverengung

In vielen Fällen ist eine explizite Typverengung notwendig und muss innerhalb der Constraint-Sprache zur Verfügung stehen. Abbildung 11 zeigt ein einfaches Beispiel, bei dem eine Typverengung notwendig ist.

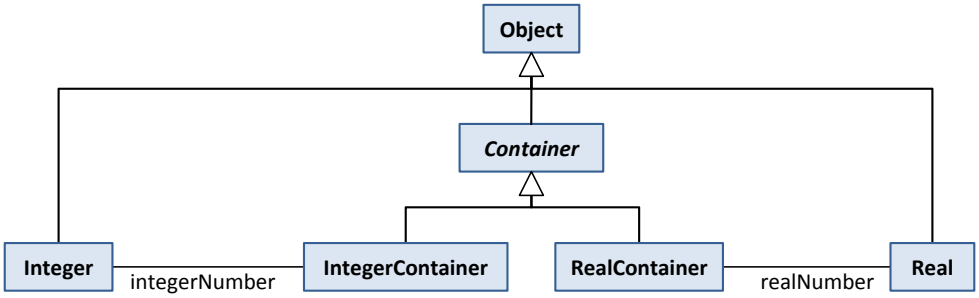


Abbildung 11: Klassendiagram zur Erläuterung der sicheren Typverengung

Im folgenden Code-Ausschnitt wird ein Objekt vom Typ `Container` erhalten und die enthaltene Zahl soll je nach Spezialisierung ermittelt werden. In Implementierungssprachen dient dazu das sogenannte *Casting*, mit dem sich der Typ eines Ausdrucks explizit ändern lässt:

```
Container c;
IntegerContainer i = (IntegerContainer) c;
```

Falls zwischen mehreren Spezialisierungen unterschieden werden muss, wird üblicherweise eine Typprüfung vorangestellt:

```
Container c;
Object n;
if (c instanceof IntegerContainer)
    n = ((IntegerContainer) c).integerNumber;
else if (c instanceof RealContainer)
    n = ((RealContainer) c).realNumber;
```

Dabei ist es generell möglich, dass das Casting nicht zulässig ist und einen Laufzeitfehler verursacht. Manche Sprachen vereinigen die Typprüfung mit dem Casting, wie z. B. beim *as*-Ausdruck in C# [35]:

```
Integer n = (c as IntegerContainer).integerNumber;
```

Ist hier `IntegerContainer` mit dem Typ in `c` vereinbar, hat der Ausdruck in Klammern den Typ `IntegerContainer` und der Zugriff wird ausgeführt. Ist das implizite Casting nicht zulässig, hat der Ausdruck den Wert `null` und verursacht somit in diesem Beispiel ebenfalls einen Laufzeitfehler. Die Formulierung ist zwar kompakter, kann aber offensichtlich dennoch zu Fehlern führen. Darüber hinaus ist die Semantik des Konstrukts nicht ohne Dokumentation erkennbar.

In LMMC soll ausschließlich eine Form möglich sein, die im Folgenden als *sichere Typverengung* bezeichnet wird. Dazu dient die Bedingung (*if*) in Verbindung mit einer Typprüfung (*is*):

```
if E is T then F
```

Ist der Typ des Ausdrucks *E* mit dem Typ *T* vereinbar, so wird der *then*-Zweig ausgewertet und der Typ des Ausdrucks *E* nur innerhalb des *then*-Zweigs auf den Typ *T* verengt. Dies erlaubt folgende Formulierung ohne explizites Casting:

```
if c is IntegerContainer then
    c.integerNumber
else if c is RealContainer then
    c.realNumber
```

Diese Lösung bietet zwei Vorteile. Zum einen können keine Fehler durch unzulässiges Casting erzeugt werden, da der entsprechende Zweig nicht interpretiert wird, wenn die Typen nicht vereinbar sind. Zum anderen ist die Semantik klar erkennbar, da sowohl für ein positives als auch für ein negatives Ergebnis der Verlauf der Auswertung intuitiv vorhergesagt werden kann.

3.6 Funktionen als Objekte erster Klasse

Einen wesentlichen Bestandteil funktionaler Sprachen bilden anonyme Funktionen. Dabei handelt es sich um Funktionen, die zur Laufzeit erzeugt werden und anschließend als Objekte erster Klasse behandelt werden können. Sie können also einer anderen Funktion (bzw. Methode) als Aktualparameter übergeben oder als Ergebnis einer Funktion zurückgegeben werden. Die Idee geht auf das λ-Kalkül zurück. Daher wird der Ausdruck, der eine anonyme Funktion erzeugt auch als λ-Ausdruck bezeichnet [28]. Er besteht aus einer Liste der Formalparameter x_1, x_2 … x_i und einem Rumpf *e*. Die übliche Form eines Lambda-Ausdrucks ist also

$$x_1, x_2 \ldots x_i \longmapsto e$$

In Sprachen wie Java 6, die das Konzept nicht unterstützen, muss es mit Hilfe von sogenannten SAM-Typen (*single abstract method*) nachgeahmt werden. Soll z. B. eine Sammlung sortiert werden, muss eine anonyme Instanz des SAM-Typs *Comparator<T>* erzeugt werden:

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

Das Sortieren ließe sich nun mit Hilfe anonymer Funktionen wesentlich prägnanter formulieren [41]:

```
Collections.sort(people,
    #{ Person x, Person y -> x.getLastName().compareTo(y.getLastName())
});
```

Die Funktion in geschweiften Klammern bildet ein Prädikat, das dann von *sort* für den Vergleich zweier Objekte verwendet wird. Die Methode *sort* wird in diesem Fall als „Funktion höherer Ordnung“ bezeichnet, da sie wiederum eine Funktion als Parameter entgegennimmt.

Als *Closures* bezeichnet man anonyme Funktionen, die sogenannte „freie Variablen“ referenzieren. Dabei handelt es sich um Variablen, die nicht zu den Formalparametern der Funktion gehören. Beim Aufruf einer Closure muss daher der lexikalische Kontext (die Werte der referenzierten freien Variablen zum Zeitpunkt der Konstruktion) für die Auswertung wiederhergestellt werden.

Der Nutzen anonymer Funktionen bzw. Closures wurde auch im Rahmen nicht-funktionaler Sprachen erkannt. So werden in der Sprache C# seit Version 2.0 anonyme Funktionen und seit Version 3.0 auch Lambdaausdrücke unterstützt [35]. Für Java sind diese Konzepte erst für die JDK-Version 8 geplant [34]. Auch OCL enthält das Konzept in eingeschränkter Form und erlaubt so die Übergabe von Iteratorausdrücken an spezielle Methoden [13].

Anonyme Funktionen in Verbindung mit Funktionen höherer Ordnung bilden ein zentrales Konzept funktionaler Sprachen, das maßgeblich zur deren Abstraktion vom Maschinenmodell und von Listenverarbeitungen beiträgt. Die entsprechenden Sprachmittel sollen also auch in LMMC uneingeschränkt zur Verfügung stehen.

3.7 Graphbasierte Navigation

Modelle liegen in Form von gerichteten Graphen vor. Zur effizienten Navigation innerhalb des Modells soll die Sprache daher entsprechend spezialisierte Konstrukte bereitstellen. Solche Konstrukte ließen sich zwar mit den Mitteln der Sprache selbst implementieren; durch den festen Einbau ergeben sich allerdings zwei Vorteile. Zum einen kann sichergestellt werden, dass die Implementierung korrekt ist. Typische Fehler z. B. durch Endlosschleifen in zyklischen Beziehungen lassen sich ausschließen. Zum anderen kann das Konstrukt durch Benutzer schnell wiedererkannt werden und verbessert damit die Lesbarkeit der Sprache.

3.7.1 Transitive Hülle über Beziehungen

Eine oft benötigte Operation ist die Berechnung der transitiven Hülle. So ist die Spezialisierungsbeziehung *parent* im Metamodell der UML zunächst nur eine binäre Relation, die angibt ob *eine* Klasse *eine* andere spezialisiert [6]. Sollen nun alle Spezialisierungen gesammelt werden um z. B. die gesamte Vererbungshierarchie und damit den effektiven Typ zu ermitteln, muss die transitive Hülle $parent^+$ über der Beziehung berechnet werden (vgl. Abbildung 12).

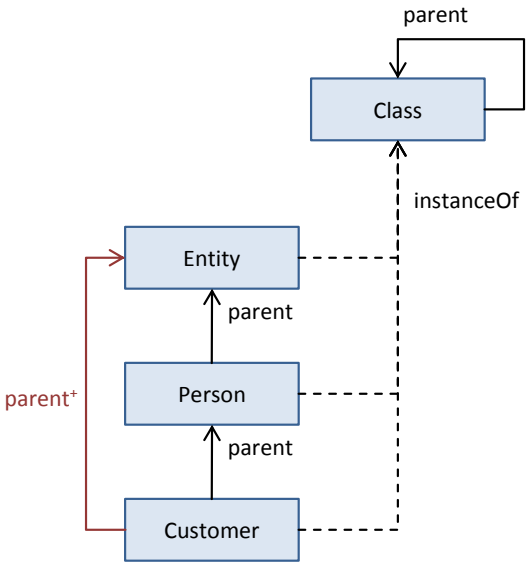


Abbildung 12: Ein UML-Klassendiagramm und seine Verbindung zum UML-Metamodell. Die Spezialisierungsbeziehung *parent* besteht nur zwischen jeweils zwei Klassen. Die Beziehung $parent^+$ bildet von *Customer* aus die Hülle über *parent*.

Die Operation ist nur dann zulässig, wenn für die Beziehung *R* gilt, dass der Zieltyp *B* der Beziehung ihren Quelltyp *A* (der Typ, der die Beziehung deklariert) spezialisiert oder mit ihr identisch ist:

$$R \subseteq A \times B \text{ mit } A \times B \coloneqq \{\, (a,b) \mid (a \in A) \wedge (b \in B) \wedge (B \text{ spezialisiert } A \vee A \equiv B) \,\}$$

Abbildung 13 zeigt drei Beispiele entsprechender Beziehungen. Im Fall **a** ist der Quelltyp der Beziehung identisch mit ihrem Zieltyp. Damit ist sichergestellt, dass die Ziele der Beziehung ebenfalls über die Beziehung *R* verfügen. Im Fall **b** erbt *B* alle Eigenschaften von *A* und damit ist sichergestellt, dass die Ziele der Beziehung selbst wieder über die Beziehung verfügen. Im Fall **c** hingegen kann nicht sichergestellt werden, dass ein Ziel der Beziehung wiederum über *R* verfügt. In diesem Fall lässt sich die transitive Hülle über *R* nicht berechnen.

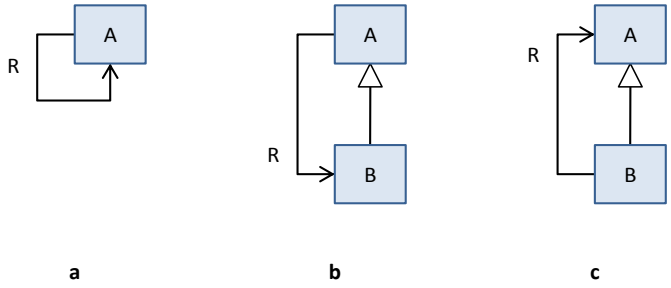


Abbildung 13: Drei mögliche Zusammenhänge zwischen deklarierendem Typ *A* und Zieltyp *B* einer Beziehung

Diese Einschränkung bei der transitiven Navigation von Beziehungen kann und muss bereits zur Entwurfszeit überprüft werden, da eine Verletzung dieser Regel eventuell zu einem Zugriff auf nicht vorhandene Eigenschaften und damit zu einem Laufzeitfehler führt.

Da zur Berechnung der transitiven Hülle keine Parameter benötigt werden, lässt sich diese Form der Navigation als Operator $\mathbf{op}_T$ in die Sprache integrieren, so dass folgender Ausdruck gilt:

```
Customer opT parents = { Person, Entity }
```

In dieser Form steht die transitive Navigation auch in der Sprache *Alloy* (vgl. Abschnitt 2.4) zur Verfügung [18]. Ein gängiger Algorithmus zur Ermittlung der transitiven Hülle traversiert den Graphen in einer Tiefensuche. Wurde ein Ast bereits besucht, wird er nicht weiter verfolgt. Übertragen auf die Navigation von Beziehungen ergibt sich folgender Algorithmus in Pseudo-Code:

```
procedure CallFeatureTransitively(instance, property)
    result := {}
    CallFeatureTransitivelyHelper(instance, property, result)
    return result

procedure CallFeatureTransitivelyHelper(instance, property, result)
    values := CallFeature(instance, property)
    foreach value in values do
          if value ∉ result then
                result := result ∪ {value}
                CallFeatureTransitivelyHelper(value, property, property)
```

Demnach wird `CallFeatureTransitively` aufgerufen um die Eigenschaft `property` transitiv von der Instanz `instance` aus zu navigieren. Dazu wird die Ergebnismenge `result` initialisiert und die rekursive Prozedur `CallFeatureTransitivelyHelper` aufgerufen. Die Prozedur `CallFeature` realisiert die einstufige Navigation der Beziehung und liefert die entsprechenden Ziele zurück. Jedes dieser Ziele wird, falls es nicht schon im Ergebnis enthalten ist, aufgenommen. Anschließend wird die Prozedur mit jedem Ziel als Instanz erneut aufgerufen.

3.7.2 Prüfung auf Zyklen

In vielen Anwendungsfällen sollen zyklische Beziehungen ausgeschlossen werden; sei es in Vererbungshierarchien, in organisatorischen Modellen oder in Komponentendiagrammen.

Eine Prüfung auf Ringschluss lässt sich mit Hilfe der transitiven Navigation auf kompakte Weise realisieren. Der oben gezeigte Algorithmus zur Ermittlung der Hülle bricht ab, sobald ein besuchtes Element bereits im Ergebnis enthalten ist. Um also auf einen Ringschluss zu prüfen,

muss lediglich überprüft werden ob die Quelle der Eigenschaft in der transitiven Hülle enthalten ist:

$x \in (x\ \mathbf{op}_T\ \texttt{parents})$

3.7.3 Transitive Navigation über Indirektionen

In vielen Fällen ist eine transitive Navigation nicht nur über eine Eigenschaft sondern über komplexe Strukturen hinweg notwendig. Im LMM z. B. werden Referenzen zwischen Konzepten mit Hilfe eines zusätzlichen Typs, der *MConceptReference* abgebildet und können dadurch mit Eigenschaften wie der Art der Referenz versehen werden (vgl. Abbildung 14).

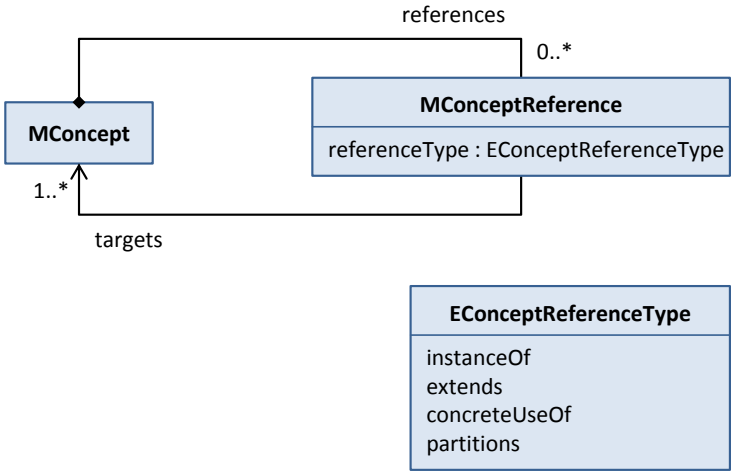


Abbildung 14: Ausschnitt aus dem LMM zur Darstellung von Konzeptreferenzen

Eine transitive Navigation ist notwendig, wenn z. B. alle Generalisierungen eines Konzepts ermittelt werden sollen. Dazu müssen zum jeweiligen Ursprungskonzept die Ziele aller *extends*-Referenzen betrachtet werden. Die Navigation auf einer Stufe lässt sich problemlos als Funktion folgender Form implementieren:

$$\text{extendedConcepts}(c) \coloneqq t \mid (\text{targets}(r,t) \mid \text{references}(c,r) \wedge \text{referenceType}(r, \text{extends}))$$

Die Ziele („targets“) aller Referenzen („references“) vom Typ „extends“

Diese Funktion lässt sich dann wiederum als Methode transitiv aufrufen und damit alle erweiterten Konzepte ermitteln:

$c\ \mathbf{op}_T\ \texttt{extendedConcepts()}$

Die Einschränkungen die bei der einfachen transitiven Navigation für den Typ der Eigenschaft gelten, müssen hier auf den Typ der Methode übertragen werden.

Im Beispiel in Abbildung 15 referenziert das Konzept *A* die Konzepte *B* und *C*, *B* wiederum *D* und *D* referenziert *A* wodurch ein Zyklus zwischen *A*, *B* und *D* entsteht. Dabei wird angenommen, dass alle drei Referenzen vom Typ *extends* sind. Ein Aufruf von `extendedConcepts` an *A* liefert alle Konzepte, die *A* direkt erweitert, also *B* und *C*. Wird die Methode transitiv aufgerufen, liefert sie alle erweiterten Konzepte, also *B*, *C*, *D* und *A*. Das Konzept *A* wird also mit in das Ergebnis aufgenommen. Um zu überprüfen, ob *A* Teil eines Zyklus ist, muss also nur überprüft werden ob es Teil der transitiven Hülle ist.

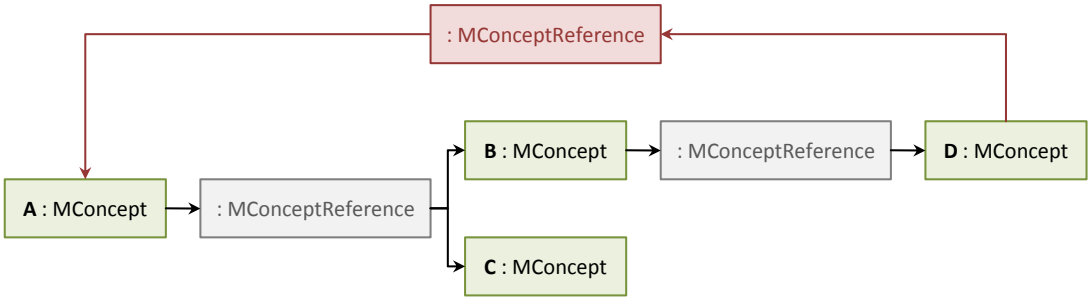


Abbildung 15: Beispielmodell zur transitiven Navigation mit Indirektion

3.8 Entkopplung des Modellzugriffs

Der Zugriff auf das validierte Modell bzw. auf sein Metamodell geschieht nicht unmittelbar sondern wird über eine Abstraktionsschicht – die „Modellfassade“ – entkoppelt (vgl. „Bindung“ in Abschnitt 2.1.1 und „Epsilon Model Connectivity“ in 2.2.1). Dadurch ist es möglich, die Sprache auf beliebige Metamodelle anzuwenden. Einzige Voraussetzung ist, dass für das Metamodell eine entsprechende Implementierung der Fassade existiert. Eine Fassade für ein Metamodell gliedert sich in zwei Bestandteile: einen Entwurfs- und einen Interpretationsteil.

Zur Entwurfszeit kann nicht auf Instanzen des Metamodells zugegriffen werden, da das zu validierende Modell noch nicht bekannt ist. Der Entwurfsteil ermöglicht also lediglich einen reflektiven Zugriff auf das Metamodell, der für die statische Typprüfung verwendet werden kann:

- Welche Elemente des Metamodells gelten als Typen?
- Welche Eigenschaften deklariert ein Typ?
- Welchen Typ hat eine Eigenschaft?

Zur Interpretationszeit hingegen ist kein Zugriff auf das Metamodell mehr notwendig. Der Interpretationsteil der Fassade regelt also nur den Zugriff auf dessen Instanz, also das zu validierende Modell:

- Welche Elemente sind Instanzen eines bestimmten Typs?
- Welchen Wert hat eine Eigenschaft innerhalb einer bestimmten Instanz?

Je nach linguistischer Ebene ist eine andere Sicht auf das Modell notwendig. Um sprachliche Constraints zu ermöglichen ist eine Implementierung mit Ecore als Metamodell notwendig, das entsprechend *EClass*es wie *MConcept* und *MAttribute* als Typen sieht und die *EObject*s in der LMM-Instanz als deren Instanzen. Für inhaltliche Constraints ist eine Implementierung notwendig, die LMM als Metamodell verwendet und die Typfacette von Konzepten als Typen und deren Instanzfacette als Instanzen versteht (vgl. Abbildung 9).

Da das Ecore-Metamodell einer unveränderlichen Semantik unterliegt, können die beiden Ecore-Fassaden fest implementiert werden. Dies gilt nicht für die LMM-Fassaden für inhaltliche Constraints. Nach Anforderung 1 soll das Modellierungsparadigma des LMM und damit auch seine Semantik austauschbar sein. Welche Attribute ein Konzept deklariert oder welche Elemente als Instanz eines bestimmten Konzepts angesehen werden können, hängt vom jeweils referenzierten Paradigma ab. Die LMM-Fassaden müssen daher das gewählte Paradigma einbeziehen und können somit nur zum Teil fest implementiert werden.

4 Implementierung von Interpreter und Entwicklungsumgebung

Um die genannten Konzepte validieren zu können, soll eine prototypische Implementierung erstellt werden. Dazu müssen zunächst das Metamodell und die konkrete Syntax der Sprache definiert werden. Für einen komfortablen Entwurf von Constraints muss eine entsprechende Entwicklungsumgebung bereitgestellt werden. Zur Anwendung der Sprache auf Modelle muss schließlich ein Interpreter entworfen und in die Entwicklungsumgebung für LMM-Modelle integriert werden. Im Folgenden werden die einzelnen Schritte der Implementierung im Detail beschrieben.

4.1 Semantisches Modell

Das semantische Modell der Sprache ist eine Repräsentation des Gegenstands, den sie beschreibt. Es sollte unabhängig von der Sprache selbst existieren können und sich zu Testzwecken auch ohne sie instanziieren lassen. Dies stellt sicher, dass das Modell die Semantik des Gegenstands vollständig erfasst [27].

4.1.1 Eclipse Modeling Framework (EMF)

Das *Eclipse Modeling Framework* (EMF) ist ein Rahmenwerk zur Erstellung von Modellen und zur Generierung von Code aus diesen Modellen. Ziel des Projekts ist es die Lücke zwischen dem Modellieren und dem Programmieren in Java zu schließen. Es bildet mittlerweile die Basis für viele auf Eclipse basierende Werkzeuge und Rahmenwerke im Umfeld von Java, XML und UML [42].

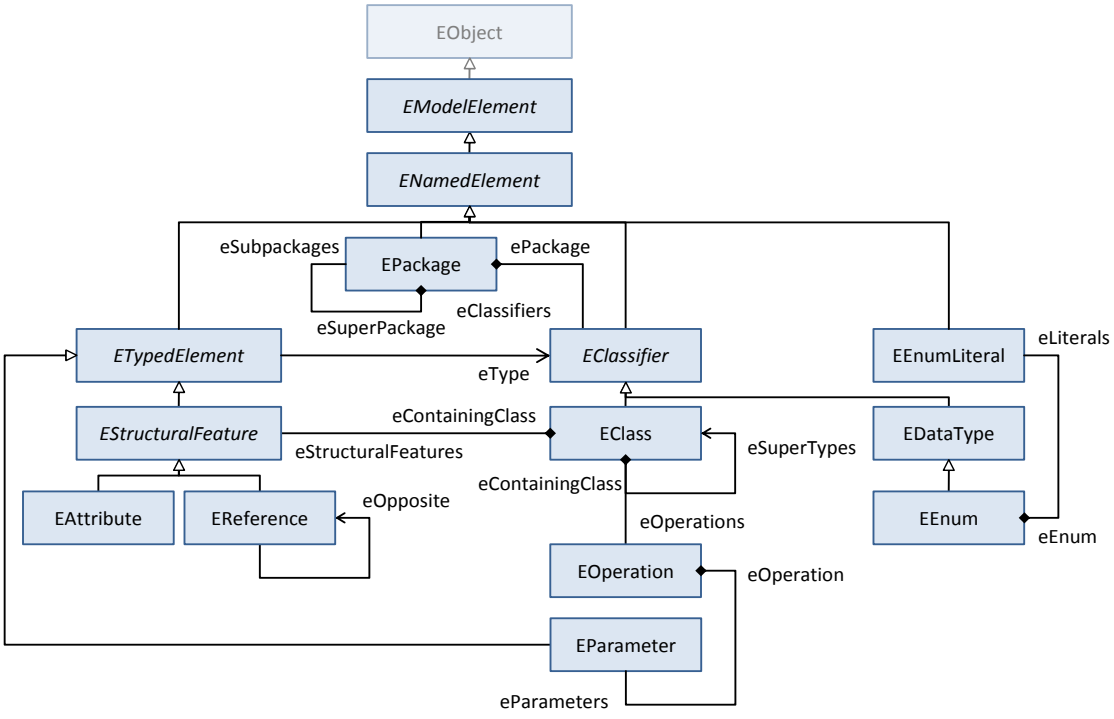


Abbildung 16: Grundlegende Struktur des Ecore Metamodells

Modelle werden in EMF durch das *Ecore* Metamodell beschrieben. Ecore ist seinerseits ein einfacher Dialekt der *Essential Meta Object Facility* (EMOF), einer Untermenge des Metamodells der UML [43]. Abbildung 16 zeigt die wesentliche Struktur des Ecore-Metamodells [12].

Das semantische Modell von LMMC wird in Ecore modelliert und ist damit für andere EMF-basierte Projekte zugänglich. Das Modell wird zur Modularisierung in drei Pakete unterteilt: Module (*lmmc*), Ausdrücke (*expressions*) und Typen (*types*).

4.1.2 Module

Invarianten werden in Modulen (*CModule*) zusammengefasst. Dabei enthält eine LMMC-Ressource (also eine Datei) genau ein Modul. Es muss mindestens ein Metamodell importiert werden (*CModelImport*), wodurch dessen Namen im Modul verfügbar werden. Die Typen des Metamodells werden durch Kontexte (*CContext*) selektiert. Innerhalb eines Kontextes – also für den selektierten Typ – lassen sich dann Methoden (*CMethod*) und Invarianten (*CInvariant*) definieren. Eine Methode hat einen eindeutigen Namen, einen Typ, Formalparameter und einen Ausdruck als Rumpf. Bei einer Invariante handelt es sich entweder um eine Regel, deren Verletzung zu einem Fehler führt oder um eine Konvention, deren Verletzung lediglich einen Hinweis auslöst. Sie hat ebenfalls einen eindeutigen Namen und einen Ausdruck als Rumpf.

Eine Invariante kann von anderen abhängig sein. In diesem Fall wird sie erst interpretiert, wenn alle abhängigen Invarianten als gültig ausgewertet wurden. Die Regel, dass ein Name mit einem Großbuchstaben beginnen muss kann z. B. abhängig sein von einer anderen Regel, die fordert, dass der Name nicht leer ist.

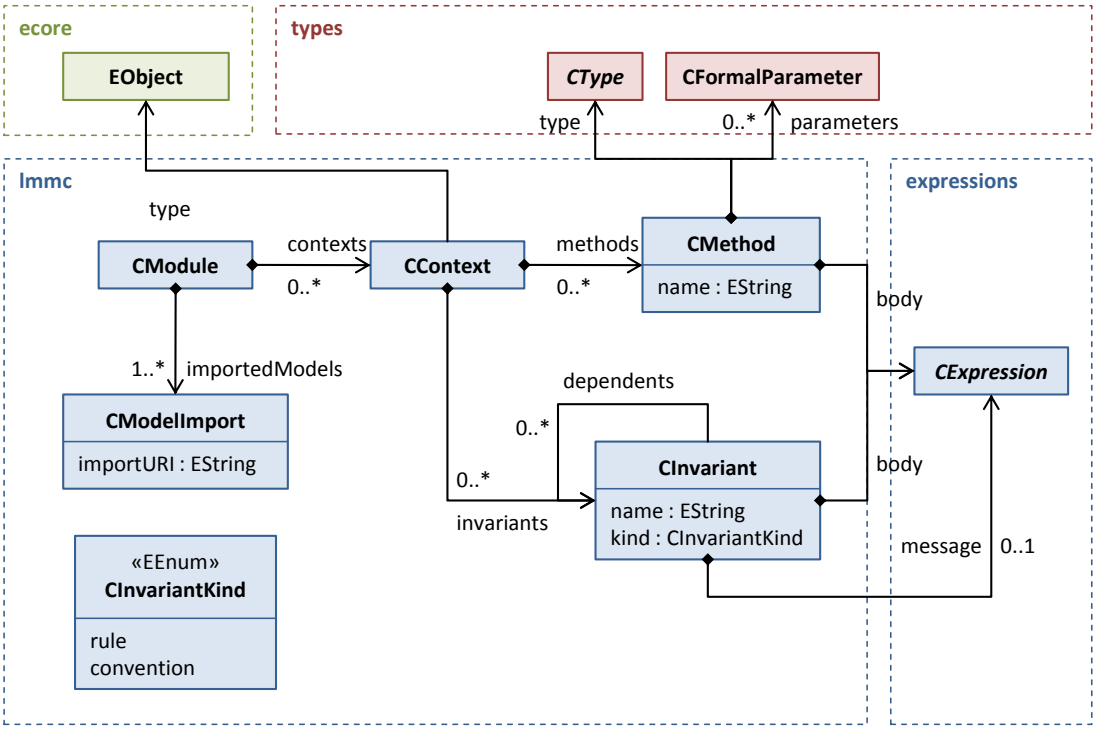


Abbildung 17: Semantisches Modell von Modulen in LMMC

4.1.3 Ausdrücke

Die Ausdrücke bilden die wesentliche Konstrukte der Sprache. Jeder Ausdruck erweitert die abstrakte Klasse *CExpression*.

Der *CCall* modelliert den Aufruf einer Methode oder das Referenzieren einer Eigenschaft *feature* an einem Objekt. Während der *CSelfCall* einen Aufruf am umgebenden impliziten Objekt (meistens *self* oder *this* genannt) darstellt, handelt es sich beim *CMemberCall* um den Aufruf an einem explizit referenzierten Objekt. Ein Aufruf kann außerdem transitiv sein (vgl. Abschnitt 3.7).

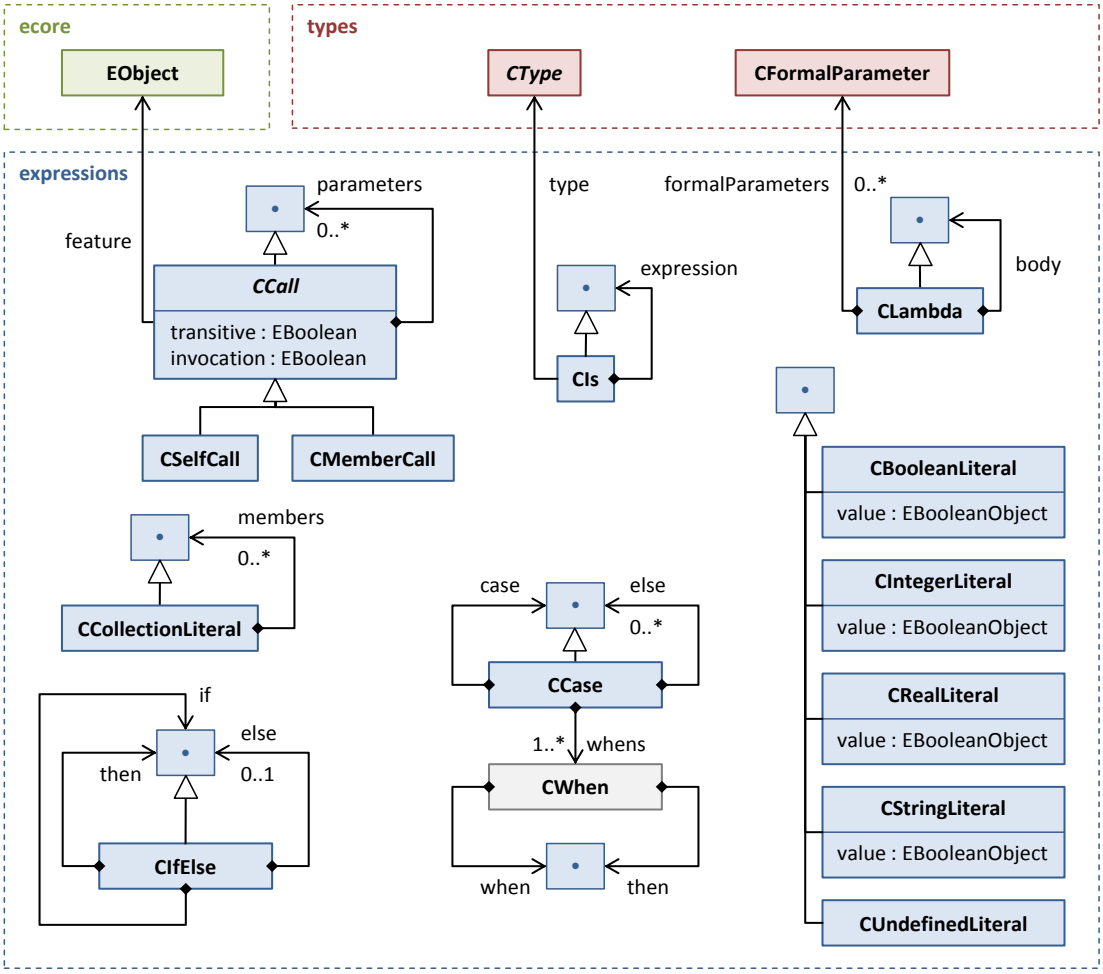


Abbildung 18: Semantisches Modell für Ausdrücke in LMMC. Die Klasse „·“ steht für *CExpression*.

Der Ausdruck, auf den sich der Aufruf bezieht wird bei einem *CMemberCall* als erster Parameter referenziert (vgl. Abbildung 19). Auch binäre Infix-Operationen wie Vergleiche oder arithmetische Operationen können als *CMemberCall* dargestellt werden. Das referenzierte Element (*feature*) muss das *EObject* aus Ecore spezialisieren. Damit lässt sich die Sprache auf alle Modelle beziehen, die ihrerseits in Ecore modelliert wurden.

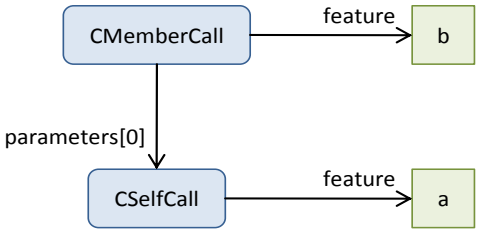


Abbildung 19: Instanz des semantischen Modells für den Ausdruck „a.b“

Das *CIs*-Konstrukt vergleicht einen Ausdruck mit einem Typ. *CLambda* modelliert einen Lambda-Ausdruck und konstruiert eine anonyme Funktion mit ihren Formalparametern und einem Ausdruck im Rumpf. *CIfElse* und *CCase* modellieren die typischen Verzweigungskonstrukte wie sie aus Programmiersprachen bekannt sind. Als funktionale Sprache kennt LMMC allerdings keine Sprünge, also keine Konstrukte wie *break* oder *continue*. Im Unterschied zu imperativen C-ähnlichen Sprachen besitzt das *CCase*-Konstrukt daher keine sogenannte *fall-through*-Semantik [40]. Der *then*-Ausdruck des ersten passenden *CWhen* wird daher ausgewertet und bildet den Wert des gesamten *CCase*-Ausdrucks. Es wird also immer nur eine Konsequenz ausgewertet.

Mit den Literalen lassen sich schließlich konstante Ganzzahlen, reelle Zahlen, Wahrheitswerte und Zeichenketten erzeugen. Außerdem lässt sich der undefinierte Wert mit *CUndefinedLiteral* erzeugen, der einer Nullreferenz in Implementierungssprachen entspricht. Mit *CCollectionLiteral* lassen sich unveränderliche Sammlungen erzeugen. Die Ausdrücke, die die Elemente der Sammlung bilden werden bei der Konstruktion der Sammlung zu Konstanten ausgewertet. Eine Verschachtelung von Sammlungen wird bei der Typprüfung ausgeschlossen, so dass eine Sammlung also immer aus einwertigen Konstanten besteht.

Die Typisierung der Ausdrücke ließe sich nur für die atomaren Literale (*CBoolean*-, *CInteger*-, *CReal*-, *CString*- und *CUndefinedLiteral*) fest im Metamodell spezifizieren. Für alle anderen Ausdrücke lässt sich kein fester Typ angeben, weshalb die Typisierung aller Ausdrücke gesondert berechnet wird.

4.1.4 Typen

Das Paket *types* enthält die eingebauten Datentypen der Sprache und die Konzepte, die zur Quantifizierung von Typen dienen. Die Klassen zur Quantifizierung werden als Container verwendet und zur Typbestimmung instanziiert. Die Datentypen dienen hingegen lediglich als Marker und werden nie instanziiert.

Elemente in LMMC sind stets mit dem abstrakten *CType* typisiert. Dabei kann das *CUndefinedLiteral* aus dem Paket *expressions* als einziges Element innerhalb Sprache den Typ *CUndefined* haben. Anonyme Funktionen werden mit *CFunction* typisiert. Dieser Typ beinhaltet den Rückgabetyp (*returnType*) der Funktion und die Typen ihrer Parameter (*parameterTypes*).

Wie in Abschnitt 3.5.2 beschrieben werden gewöhnliche Typen stets als Paar aus Multiplizität und Kerntyp angegeben. *CQuantified* bildet den Container für eine solche Typangabe. Anstelle von beliebigen Intervallen bietet die Sprache die vier grundlegenden Kardinalitäten [1, 1] („genau ein“), [0, 1] („ein oder kein“), [1, *] („ein oder viele“) und [0, *] („kein oder viele“). *COne* quantifiziert genau eine Instanz. *CMany* umfasst alle mehrwertigen Multiplizitäten und *CMaybe* alle mit Null als Untergrenze. Der Kerntyp muss das *EObject* aus Ecore erweitern. Damit lassen sich alle Metamodelle referenzieren, die ihrerseits auf Ecore basieren.

Als interne Datentypen bietet LMMC die Zeichenkette (*CString*), die Ganzzahl (*CInteger*), die reelle Zahl (*CReal*) und den Wahrheitswert (*CBoolean*).

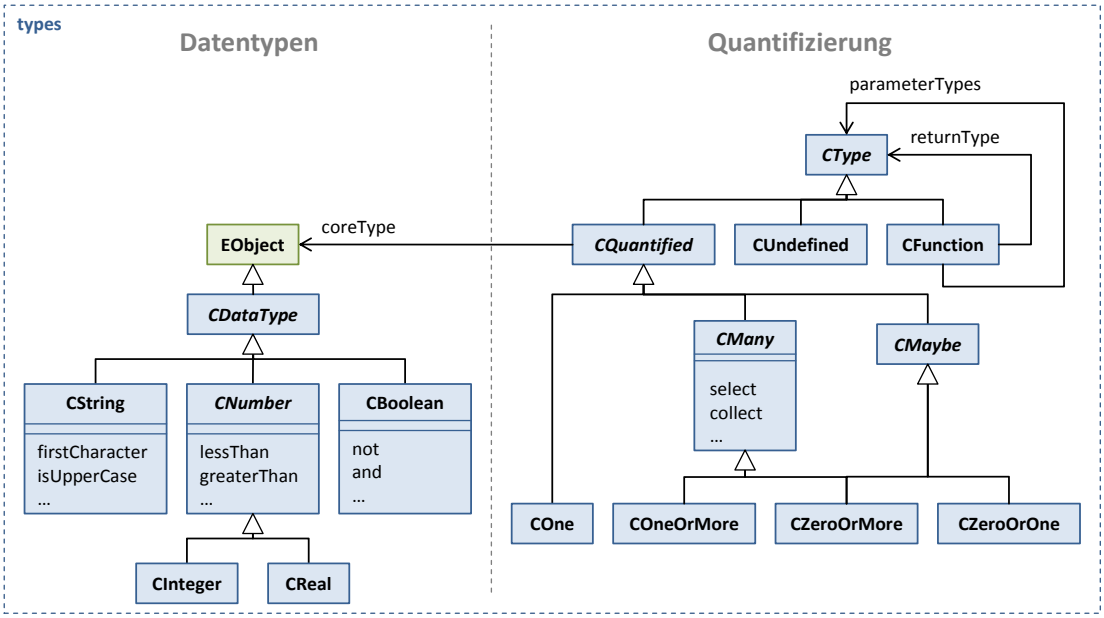


Abbildung 20: Semantisches Modell der Typen in LMMC

4.2 Typisierung von Ausdrücken

Im Folgenden soll die Typisierung von Ausdrücken in LMMC betrachtet werden. Dazu dienen die Hilfsfunktionen $\tau(x)$, $\mu(x)$, $T(X)$ und $M(X)$. Die Funktion $\tau(x)$ liefert den Kerntyp eines Elements x, die Funktion $\mu(x)$ dessen Multiplizität. Fasst man die Multiplizitäten in LMMC als Intervalle auf, gilt die in Abbildung 21 gezeigte Ordnung auf Basis der Teilmengenrelation. Die Funktion $M(X)$ liefert die „gemeinsame Multiplizität“ aus einer Menge X von Multiplizitäten, also deren Vereinigungsmenge.

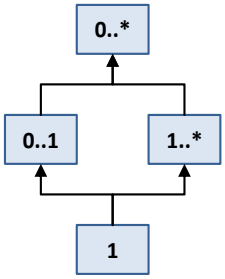


Abbildung 21: Ordnung der Multiplizitäten auf Basis der Teilmengenrelation

Die Funktion $T(X)$ liefert den gemeinsamen Supertyp einer Menge X von Kerntypen, also deren „speziellste gemeinsame Generalisierung“ (SGG) innerhalb einer Vererbungshierarchie (vgl. Abbildung 22). Existiert kein solcher Typ liefert die Funktion kein Ergebnis.

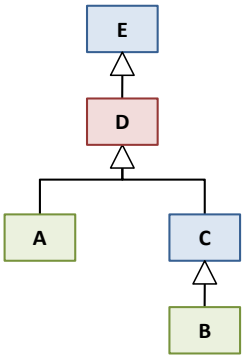


Abbildung 22: Beispiel für den gemeinsamen Supertyp: die speziellste gemeinsame Generalisierung von *A* und *B* ist der Typ *D*

Ausdruck	Typ
CUndefinedLiteral	CUndefined
CIntegerLiteral	COne(CInteger) *Genau eine Ganzzahl*
CRealLiteral	COne(CReal)
CBooleanLiteral	COne(CBoolean)
CStringLiteral	COne(CString)
CCollectionLiteral	COneOrMore(T({$t \in$ { τ(m) | $m \in$ members } })) *Kerntyp bildet die SGG aus den Elementen der Sammlung.*
CIfElse	M({ μ(then), μ(else) }) (T({ τ(then), τ(else) })) *Die gemeinsame Multiplizität aus dem then- und dem else-Zweig und als Kerntyp die SGG aus diesen Zweigen.*
CCase	M({ μ(t) | $t \in$ { w.then | $w \in$ whens } } ∪ μ(else)) (T({ τ(t) | $t \in$ { w.then | $w \in$ whens } } ∪ τ(else))) *Die gemeinsame Multiplizität aus den then-Zweigen der when-Zeige und dem else-Zweig und als Kerntyp die SGG aus diesen Zweigen.*
CLambda	CFunction(returnType = τ(body), parameterTypes = { τ(p) | $p \in$ parameters })
CCall	μ(feature)(τ(feature)) *Multiplizität und Kerntyp entsprechen denen der referenzierten Eigenschaft.*
CIs	COne(CBoolean)

Tabelle 1: Typisierung der Ausdrücke in LMMC

4.3 Konkrete Syntax

Die konkrete Syntax bildet die textuelle Darstellung der Sprache. Da die Sprache als problembezogene Modellierungssprache verwendet wird, soll „syntaktisches Rauschen“ weitgehend vermieden werden. Das bedeutet, dass das Verhältnis zwischen dem eigentlichen Problem und der Syntax, die zu seiner Darstellung dient möglichst klein gehalten werden soll [44].

4.3.1 Lexikalische Bausteine

Die lexikalischen Grundelemente der Sprache bestimmen, wie aus einem Zeichenstrom eine Folge von atomaren Symbolen verschiedenen Typs zu erzeugen ist [45]. Im Folgenden werden diese Bausteine zusammen mit einer regulären Grammatik zu ihrer Erzeugung aufgezählt. Die Syntax dieser Grammatik entspricht dabei der ANTLR-Metasprache, einer Erweiterung der *Erweiterten Backus-Naur-Form* (EBNF) [46].

Namen

Namen dienen zur eindeutigen Kennzeichnung von Methoden, Invarianten und Modellelementen wie Typen und Eigenschaften. Sie beginnen mit einem Buchstaben oder einem Unterstrich und enthalten Buchstaben, Ziffern oder Unterstriche. Um Kollisionen mit Schlüsselwörtern zu vermeiden, kann ein Zirkumflex-Akzent (^) vorangestellt werden:

```
NAME        : '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
```

Beispiele für Namen sind:

- `Alice`
- `bob`
- `_charly`
- `^else`

Zeichenketten

Literale zur Angabe von Zeichenketten werden durch doppelte Anführungszeichen (") begrenzt. Innerhalb der Zeichenkette müssen Steuerzeichen (neue Zeile etc.) und das Anführungszeichen durch einen umgekehrten Schrägstrich (\) markiert werden:

```
STRING      : '"' ( '\\' ( 'b'|'t'|'n'|'f'|'r'|'"'|'"'|'\\' ) | !( '\\'|'"' ) )* '"'
```

Beispiele für Zeichenketten sind:

- `"Hello World"`
- `"Escaped \"Word\""`

Ganzzahlen

Literale zur Angabe von dezimalen Ganzzahlen bestehen entweder nur aus der Ziffer 0 oder sie beginnen mit einer Ziffer von 1 bis 9 und bestehen weiter aus Ziffern von 0 bis 9:

```
INTEGER     : '0' | ( '1'..'9' ( '0'..'9' )* )
```

Reelle Zahlen

Literale zur Angabe von reellen Zahlen bestehen aus einem optionalen ganzzahligen Teil, einem Dezimaltrennzeichen und einem gebrochenen Teil:

```
REAL        : INTEGER? '.' INTEGER
```

Kommentare

Für die Angabe von Kommentaren werden die üblichen Markierungen für einzeilige und mehrzeilige Kommentare verwendet:

COMMENT : ('/*' -> '*/') | ('//' !('\n'|'\r')* ('\r'? '\n')?)

Leerraum Als Leerraum sind das Leerzeichen, der Tabulator, die neue Zeile und der Zeilenvorschub erlaubt:

WHITESPACE : (' '|'\t'|'\r'|'\n')+

Schlüsselwörter Folgende Schlüsselwörter haben eine besondere Bedeutung und können nur als Namen verwendet werden, wenn ein Zirkumflex-Akzent vorangestellt wird (vgl. „Namen“):

- **and**
- **case**
- **else**
- **false**
- **if**
- **implies**
- **in**
- **not**
- **or**
- **then**
- **true**
- **undefined**
- **when**

4.3.2 Typangaben

Elementreferenzen Um ein Element im Metamodell zu referenzieren, wird sein qualifizierter Name angegeben. Die Namensräume werden dabei durch Punkte getrennt:

QUALIFIED : **NAME** ('.' **NAME**)*

Typbeschreibungen Soll ein Formalparameter oder eine Methode typisiert werden, so muss ihr Typ in Form einer vollständigen Typbeschreibung angegeben werden. Diese beinhaltet die Multiplizität und den eigentlichen Kerntyp (vgl. Abschnitt 3.5.2):

Type : *Quantified* | *Function*

Quantified : ('1..*' | '0..1' | '0..*')? **QUALIFIED**

Function : '(' (*Type* (',' *Type*)*)? ')' '=>' *Type*

Beispiele für eine Typbeschreibung sind:

- `Model.M1.ER.Entity`
- **`0..*`** `Model.M1.ER.Relationship`
- `(`**`integer, integer`**`) => `**`boolean`**

4.3.3 Ausdrücke

Als funktionale Sprache kennt LMMC keine Blöcke aus Anweisungen. Der Rumpf einer Methode oder einer Invariante ist stets ein einziger Ausdruck. Im Gegensatz zu einer Anweisung hat jeder Ausdruck einen Wert.

Literale Aus den Terminalen für Zeichenketten und Zahlen werden entsprechende Ausdrücke abgeleitet. Dazu kommen Literale für die Erzeugung von Wahrheitswerten, Sammlungen und Nullreferenzen:

IntegerLiteral : **INTEGER**
RealLiteral : **REAL**
StringLiteral : **STRING**
BooleanLiteral : 'true' | 'false'
UndefinedLiteral : 'undefined'
CollectionLiteral : '[' (*Expression* (',' *Expression*)*)? ']'

Lokale Aufrufe

Aufrufe von Eigenschaften oder Methoden am umgebenden Objekt werden als lokaler Aufruf bezeichnet. Er besteht aus dem Namen der Eigenschaft oder Methode gefolgt von einer kommagetrennten Liste der Aktualparameter in Klammern:

SelfCall : **NAME** ('(' *Expression* (',' *Expression*)*)? ')')?

Primäre Ausdrücke

Die Literale, der lokale Aufruf und der geklammerte Ausdruck werden als primäre Ausdrücke zusammengefasst:

Primary : *BooleanLiteral* |
IntegerLiteral |
RealLiteral |
StringLiteral |
UndefinedLiteral |
CollectionLiteral |
SelfCall |
'(' *Expression* ')'

Element-Aufrufe

Der Aufruf einer Eigenschaft oder Methode an einem bestimmten Objekt hat eine ähnliche Form. Als Objekt kommen nur primäre Ausdrücke in Frage:

MemberCall : *Primary* ('.' '+'? **NAME**
('(' *Expression* (',' *Expression*)*)? ')')?
)*

Um den Aufruf als transitiv zu markieren, wird dem Punkt ein Plus angehängt.

Operatoren

Operatoren werden als Methodenaufrufe dargestellt. Sie können entweder über ihren Methodennamen, also z. B. `1.add(2)` oder über ihren Alias in Infix-Form referenziert werden, also `1 + 2`. Für benannte Operatoren wie **`implies`** und **`or`** existiert kein Alias. Diese können über ihren Namen sowohl in einem Aufruf als auch in Infix-Form verwendet werden.

Lambda-Ausdrücke Lambda-Ausdrücke erzeugen eine anonyme Funktion. Sie bestehen aus den Formalparametern und dem Rumpf der Funktion:

Lambda : (*FormalParameter* (',' *FormalParameter*)*)?
'|' *Expression*

FormalParameter : *Type* **NAME**

Bedingungen Im Gegensatz zu imperativen Sprachen ist das wenn-dann-Konstrukt in LMMC ein Ausdruck und hat demnach einen Wert:

IfThenElse : 'if' *Expression* 'then' *Expression* ('else' *Expression*)?

Um mehrere Fälle kompakt abprüfen zu können dient das case-when-Konstrukt:

Case : 'case' *Expression*
('when' *Expression* 'then' *Expression*)+
('else' *Expression*)?

Die Schlüsselwörter wurden nicht *switch* und *case* genannt um die abweichende Semantik zu verdeutlichen (vgl. Abschnitt 4.1.3).

4.3.4 Module

Invarianten und Methoden werden über Kontexte Typen zugeordnet. Kontexte werden wiederum in Modulen zusammengefasst. Ein Modul referenziert Metamodelle und enthält Kontexte:

Module : *ModelImport* Context**
ModelImport : 'model' **STRING**

Ein Kontext selektiert einen Typ und enthält Methoden und Invarianten:

Context : 'on' **QUALIFIED** '{' *Method* Invariant** '}'

Eine Methode hat einen Rückgabetyp, einen eindeutigen Namen und kann Formalparameter deklarieren:

Method : *Type* **NAME**
'(' (*FormalParameter* (',' *FormalParameter*)*)? ')'
'=' *Expression*

Eine Invariante ist entweder eine Regel oder eine Konvention. Sie kann einen Namen haben, kann von anderen Invarianten abhängig sein und kann eine Meldung produzieren:

Invariant : ('rule' | 'convention') **NAME**?
('requires' **NAME** (',' **NAME**)*)?
'ensures' *Expression*
('causes' *Expression*)?

4.3.5 Operatoren

Ein Teil der Operationen, die die Klassen der internen Typen in Abschnitt 4.1.4 deklarieren wird auf Operatoren abgebildet. Bei einer Infix-Notation ohne Klammern muss die Auswertungsreihenfolge geklärt werden. Dies lässt sich erreichen, indem für Operatoren eine Ordnung in Form von Prioritäten vorgegeben wird. Tabelle 2 zeigt die Operationen zusammen mit

ihren Operator-Aliasen und deren Prioritäten. Die höchste Priorität ist die Eins; der Punkt-Operator bindet also am stärksten.

Typ	Operation	Operator-Alias	Priorität
		.	1
		`.+`	1
CBoolean	not	**`not`**	2
CNumber	negate	- (unär)	2
CNumber	multiply	*	3
CNumber	divide	/	3
CNumber	add	+	4
CNumber	subtract	-	4
CNumber	greaterThanOrEquals	>=	5
CNumber	lessThanOrEquals	<=	5
CNumber	greaterThan	>	5
CNumber	lessThan	<	5
CQuantified	in	**`in`**	5
CQuantified	equals	=	6
CQuantified	notEquals	!=	6
CBoolean	and	**`and`**	7
CBoolean	or	**`or`**	8
CBoolean	implies	**`implies`**	9

Tabelle 2: Operatoren und deren Prioritäten in LMMC

4.3.6 Aliase für interne Typen

Die Namen der Typklassen werden nicht direkt sondern über Aliasnamen angeboten. Tabelle 3 zeigt die Aliase der internen Datentypen in LMMC.

Typklasse	Alias
CUndefined	*keine*[3]
CInteger	**`integer`**
CReal	**`real`**
CBoolean	**`boolean`**
CString	**`string`**

Tabelle 3: Aliase für interne Typen in LMMC

4.3.7 Eingebaute Methoden mehrwertiger Elemente

Mehrwertige Elemente, also solche, die mit einer Instanz von *CMany* (vgl. Abschnitt 4.1.4) typisiert sind, verfügen implizit über bestimmte eingebaute Methoden. Dabei handelt es sich um Methoden, die für den Umgang mit Sammlungen benötigt werden, also Mengenoperationen und Funktionen höherer Ordnung. Tabelle 4 beschreibt diese Methoden, die ähnlicher Form auch in der OCL zur Verfügung stehen [13]. Die Funktion *F*, die für `select`, `collect`, `forAll` und `exists` als Prädikat verwendet wird, muss dabei mindestens einen Formalparameter deklarieren. Für genau einen Parameter iteriert die Funktion über den Elementen der Sammlung. Für *n* Parameter iteriert sie über allen Kombinationen der Länge *n*. Daher ist z. B. auch folgender Ausdruck möglich:

[3] Nur der `undefined`-Ausdruck hat den Typ *CUndefined*. Da der Typ also nur diese eine Instanz hat, gibt es keine Notwendigkeit andere Elemente damit zu typisieren und keinen Alias.

```
elements.exists(Element left, Element right | left.property =
right.property)
```

Methode	Beschreibung
`R = C.select(F)`	*R* enthält alle Elemente aus *C*, für die die Funktion *F* `true` liefert. Diese Methode entspricht der Funktion höherer Ordnung *filter* [47].
`R = C.collect(F)`	Die Funktion *F* wird auf alle Elemente in *C* angewandt und die Ergebnisse werden als *R* geliefert. Diese Methode entspricht der Funktion höherer Ordnung *map* [48].
`R = C.forAll(F)`	*R* ist `true`, falls die Funktion *F* für alle Elemente in *C* `true` liefert.
`R = C.exists(F)`	*R* ist `true`, falls die Funktion *F* für mindestens ein Element in *C* `true` liefert.
`R = C.union(D)`	*R* enthält die Vereinigung aus *C* und *D*, wobei Duplikate eliminiert werden.
`R = C.includes(X)`	*R* ist `true`, falls *X* in *C* enthalten ist oder alle Elemente aus *X* in *C* enthalten sind.
`R = C.any()`	*R* ist ein beliebiges Element aus *C*.
`R = C.count()`	*R* ist die Anzahl der Elemente in *C*.

Tabelle 4: Methoden mehrwertiger Elemente in LMMC

4.4 Modellfassaden

Wie in Abschnitt 3.8 beschrieben teilt sich der Zugriff auf das Modell in einen Entwurfs- und einen Interpretationsteil. Den Entwurfsteil der Schnittstelle bildet die `EditorModelFacade`, den Interpretationsteil die `InterpreterModelFacade`.

Für den Entwurf und die Interpretation von sprachlichen Constraints wird jeweils eine Implementierung für Ecore als Metamodell benötigt. Instanzen von *EClass*, also z. B. *MConcept* und *MAttribute* werden als Typen behandelt. Die Instanzen dieser Klassen im Sinne von Ecore – also die Bestandteile einer LMM-Instanz – bilden die zu validierenden Objekte.

Um inhaltliche Constraints entwerfen und auswerten zu können wird jeweils eine Implementierung für LMM als Metamodell benötigt. Die Typfacetten von Konzepten innerhalb eines LMM-Modells werden als Typen behandelt. Die entsprechenden Instanzfacetten werden als Instanzen behandelt. Im Unterschied zu Ecore ist die Semantik von LMM veränderlich und wird wiederum durch sprachliche Constraints bestimmt. Die LMM-Implementierung der Modellfassaden lässt sich also nicht vollständig vorgeben. Welche Attribute ein Konzept deklariert oder welche Konzepte mit einem anderen vereinbar sind hängt von der jeweils verwendeten Semantik ab. Die Implementierung muss sich also in dieser Hinsicht dynamisch verhalten. Sie muss ihrerseits die sprachlichen Constraints, die für das validierte Modell gelten interpretieren, also einen LMMC-Interpreter aufrufen.

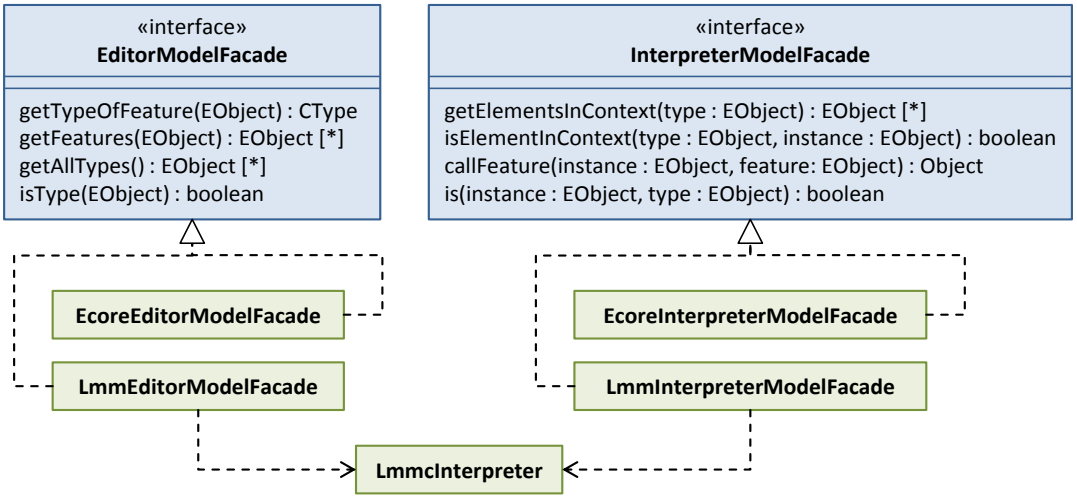


Abbildung 23: Implementierung der beiden Modellfassaden für Ecore und LMM

Auch die Implementierungen der Modellfassaden können als transparent aufgefasst werden. Da sich das Modell während seiner Validierung nicht ändert (vgl. Abschnitt 3.3), liefert der Aufruf einer Methode an einer Fassade für gleiche Argumente stets das gleiche Ergebnis. Der Zugriff auf das Modell stellt den teuersten Teil der Auswertung von LMMC-Constraints dar. Aus diesem Grund werden die Modellfassaden nach dem „Decorator"-Muster [49] mit einem Cache umgeben.

4.4.1 Registrierung von Modellfassaden

Die Modellfassaden müssen beim Editor und dem Interpreter registriert werden. Die Komponenten bieten hierzu auf der Eclipse-Plattform einen Erweiterungspunkt (engl. *extension points*) [50] an, mit dem sich die Fassaden dann verknüpfen lassen. Zur Definition solcher Erweiterungspunkte dienen XML-Schemata. Abbildung 24 zeigt das Schema, das den Erweiterungspunkt zur Registrierung von Modellfassaden definiert.

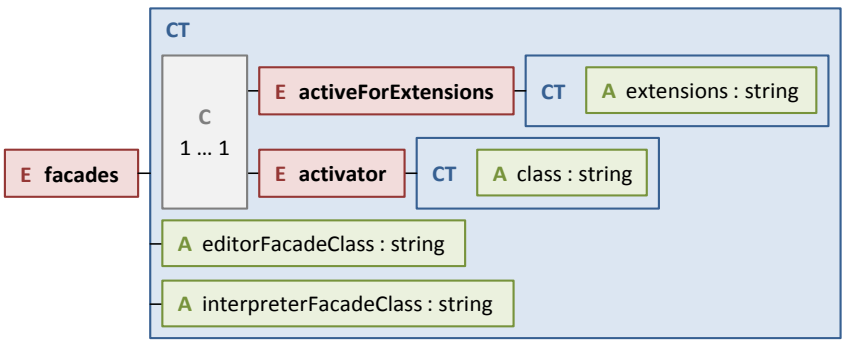


Abbildung 24: XML-Schema des Erweiterungspunkts zur Registrierung von Modellfassaden. Das Schema besteht aus Elementen (*E*), Attributen (*A*), komplexen Typen (*CT*) und Wahlmöglichkeiten (*C*).

Ein Paar aus Fassaden (für Editor und Interpreter) lässt sich demnach durch ein Element *facades* registrieren. Es referenziert die beiden Klassen für Editor- und Interpreter-Fassade über *editor-* und *interpreterFacadeClass*. Außerdem muss ein Kriterium zur Aktivierung der Fassade angegeben werden. Dazu kann eine Liste von Dateiendungen über ein *activeForExtensions*-Element angegeben werden. Lässt sich nicht anhand der Endung erkennen ob die Fassaden aktiviert werden sollen, dann kann über ein *activator*-Element auch eine Klasse referenziert

werden, die die Schnittstelle *ModelFacadeActivator* implementiert. Diese Schnittstelle definiert nur die Methode *activate(resource : org.eclipse.emf.ecore.resource.Resource) : boolean*, die für eine gegebene EMF-Ressource zurückgibt ob die Fassaden aktiviert werden sollen. Die Auswahl (engl. *choice*) im Schema bewirkt, dass immer nur eine Aktivierungsstrategie ausgewählt werden kann.

4.5 Parser und Entwicklungsumgebung

4.5.1 Xtext

Xtext ist eine Plattform zur Entwicklung von externen domänenspezifischen Sprachen. Mit Hilfe einer Metasprache und einer Reihe von Schnittstellen lassen sich die verschiedenen Aspekte der zu entwickelnden Sprache spezifizieren. Auf der Basis dieser Informationen lässt sich dann ein Großteil der benötigten Infrastruktur generieren. Dazu zählen unter anderem ein Parser und Teile einer Entwicklungsumgebung auf der Basis des *Eclipse*-Frameworks [9].

4.5.2 Parser

Zur Definition der Syntax dient in Xtext eine spezielle Grammatiksprache, die es gleichzeitig erlaubt syntaktische auf semantische Elemente abzubilden. Das bedeutet, dass innerhalb der Grammatik auch der Aufbau des semantischen Modells vorgegeben wird. Folgender Ausschnitt z. B. gibt die Grammatik einer LMMC-Invariante in einer EBNF-artigen Form an und spezifiziert gleichzeitig den Aufbau des entsprechenden semantischen Modells, indem syntaktische Regeln den Modellelementen zugewiesen werden:

```
CInvariant:
    kind=CInvariantKind
    (name=NAME)?
    ('requires' dependents+=[CInvariant] (','
dependents+=[CInvariant])*)?
    'ensures' expression=CExpression
    ('causes' message=CExpression)?;
```

Aus dieser Xtext-Grammatik wird dann mit Hilfe des ANTLR-Parser-Generators ein LL(*)-Parser generiert. Die Syntax von LMMC macht es notwendig, dass der Parser das sogenannte *Backtracking* betreibt. Das bedeutet, dass er in der Lage ist, aus einer fälschlicherweise gewählten Regel zurückzukehren und mit einer anderen Regel fortzufahren [44]. Syntaktische Fehler werden vom Parser erkannt und mit der LMMC-Ressource verknüpft, so dass sie in der Entwicklungsumgebung angezeigt werden können.

4.5.3 Namensauflösung

Enthält das Modell der Sprache Verweise, also z. B. von der Verwendung einer Variablen zu ihrer Deklaration dann werden vom Parser zunächst Stellvertreterobjekte (engl. *proxies*) als deren Ziele eingesetzt. Durch das *Linking* werden diese Stellvertreter anschließend durch echte Referenzen ersetzt und damit logische Namen aufgelöst. In diesem Schritt werden also bereits alle Referenzen auf das Metamodell, auf Variablen, Methoden und auf eingebaute Operationen vollständig aufgelöst. Dadurch entfallen klassische Mechanismen wie Symboltabellen zur Interpretationszeit [45]. Fehler wie unbekannte oder nicht sichtbare Namen werden also in der Linking-Phase erkannt und ähnlich wie Syntaxfehler mit der Ressource verknüpft.

Als Grundlage für das Linking muss neben der Grammatik auch das sogenannte *Scoping* implementiert werden. Hierbei wird festgelegt, welche Objekte als Ziel einer Referenz im Modell überhaupt in Frage kommen.

4.5.4 *Validierung und statische Typprüfung*

Im Anschluss an das Linking folgt die Validierung des semantischen Modells. Den Hauptteil der Validierung bildet die statische Typprüfung. Hierfür wird die *Xtext/TS*-Erweiterung verwendet. Sie ermöglicht es Berechnungsvorschriften und Regeln für die Typen einer Xtext-basierten Sprache anzugeben [51]. Bei der Typprüfung werden u. a. folgende Regeln überprüft:

- Der Rumpf einer Invariante hat den Typ *boolean* (Invarianten sind Boole'sche Ausdrücke)
- Der Rumpf einer Methode hat einen Typ, der mit dem Typ der Methode vereinbar ist
- Der Typ eines Aktualparameters ist mit dem Formalparameter vereinbar

Neben der Typisierung lassen sich in der Validierungsphase außerdem Regeln zur Konsistenz der Constraints prüfen. Dazu zählt z. B., dass Constraint-Abhängigkeiten keine Zyklen bilden dürfen. Abbildung 25 veranschaulicht die beschriebenen Schritte vom Quelltext der Constraints zu einem aufgelösten und validierten Modell.

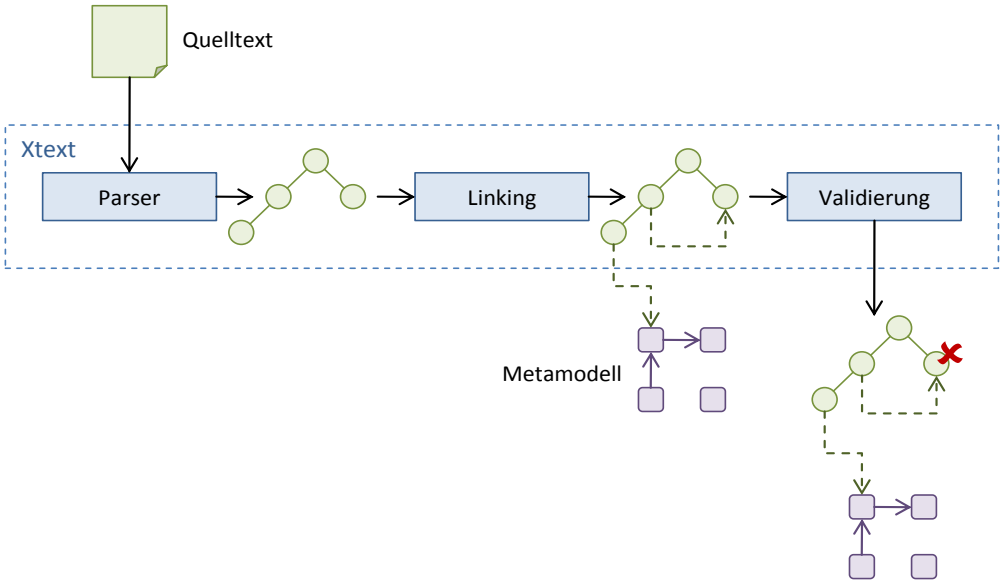
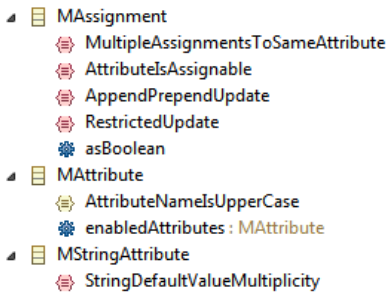


Abbildung 25: Die drei Phasen bei der Erzeugung eines aufgelösten und validierten Modells aus einer Quelldatei

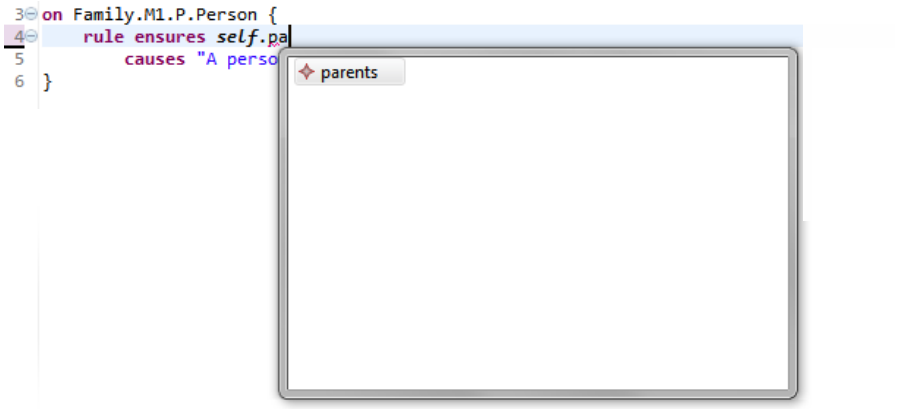
4.5.5 *Unterstützende Elemente der Entwicklungsumgebung*

Auf der Basis der beschriebenen Infrastruktur werden von Xtext weitere Elemente bzw. deren Vorlagen generiert, die in der Summe eine moderne Entwicklungsumgebung für die Sprache bilden. Viele dieser Elemente basieren auf bekannten Funktionen aus den *Java Development Tools* (JDT):

- Inhaltsübersicht der editierten Quelldatei



- Syntaxhervorhebung
- Code-Vervollständigung



- *Wizards* zur Erstellung neuer Projekte und Dateien
- Automatische Formatierung von Quelltext

4.6 Interpreter

Die Instanz des semantischen Modells wird vom Interpreter ausgewertet und dabei auf das zu validierende Modell angewendet. Über eine Schnittstelle entsprechend dem „Beobachter"-Muster [49] liefert der Interpreter dabei Constraint-Verletzungen an den Aufrufer. Die Interpretation geschieht direkt und ohne vorherige Analyse- oder Optimierungsphasen.

4.6.1 Auswertung von Ausdrücken

Die Hauptaufgabe des Interpreters besteht in der Auswertung von Ausdrücken. Ein Ausdruck stellt eine hierarchische Objektstruktur dar, die während der Auswertung nicht verändert werden muss. Zur Realisierung bietet sich daher das „Besucher"-Muster an [49]. Ein externer baumbasierter Interpreter bietet darüber hinaus den mit Abstand schnellsten Weg zur Implementierung einer domänenspezifischen Sprache [44].

Zusätzlich zum üblichen Besucher-Muster ist der Typ eines Ausdrucks generisch und es lässt sich ein Kontextobjekt an die Besuchermethoden übergeben. Daher sind die Klassen und Schnittstellen entsprechend parametrisiert (vgl. Abbildung 26). Die von EMF generierte *Switch*-Klasse (`LmmcExpressionsSwitch<T>`) ließe sich auch als Besucher verwenden, bietet allerdings keine Möglichkeit einen Kontext zu übergeben oder Ausnahmen zu erzeugen.

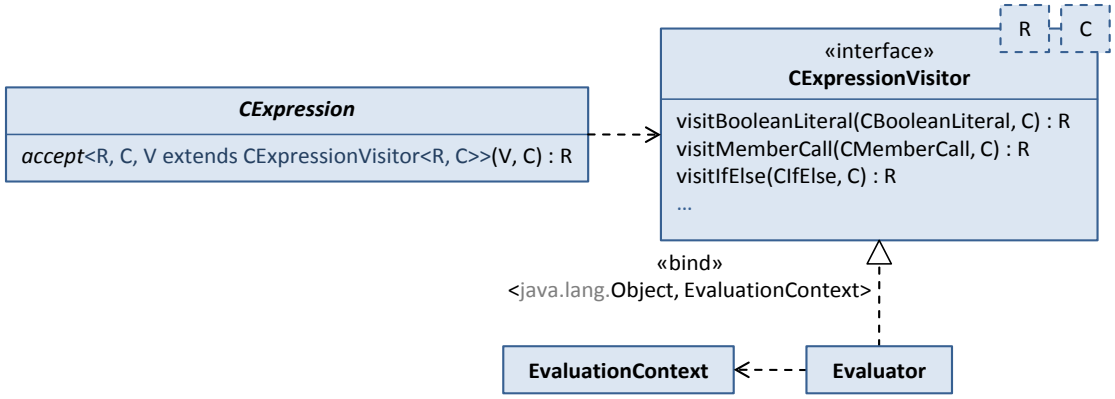


Abbildung 26: Der *Evaluator* zur Auswertung von LMMC-Ausdrücken

4.6.2 Realisierung von Closures

Falls eine anonyme Funktion Variablen referenziert, die nicht Teil ihrer Formalparameter sind, handelt es sich um eine *Closure*. Die Werte solcher freien Variablen müssen beim Aufruf der Funktion rekonstruiert werden. Eine Möglichkeit der Implementierung ist, das Funktionsobjekt zusammen mit den referenzierten Variablen und deren Werte zum Zeitpunkt der Konstruktion zu „konservieren“. Dazu wird die Funktion in einer Datenstruktur `AnonymousFunction` verpackt, die tatsächlich alle entsprechenden Werte enthält und erst mit der Funktion selbst wieder für die *Garbage Collection* freigegeben wird. Bei der Konstruktion eines `AnonymousFunction`-Objekts wird der Ausdruck im Rumpf der Funktion untersucht und die Werte freier Variablen bei Bedarf gesichert.

4.6.3 Memoisierung

Die Memoisierung ist ein einfacher aber effektiver Caching-Mechanismus der die wiederholte Berechnung von Funktionsaufrufen erspart. Dabei wird für die entsprechende Funktion ein Cache angelegt, der für einen Vektor von Aktualparametern den jeweiligen Rückgabewert speichert. Bei folgender Implementierung der Fibonacci-Folge wird die Funktion sehr oft mit denselben Aktualparametern aufgerufen:

```
fibonacci(integer n) =
    if n <= 2 then
        1
    else
        fibonacci(n - 1) + fibonacci(n – 2)
```

Eine Memoisierung führt bei dieser Funktion zu einer wesentlich effizienteren Ausführung und einem erheblichen Geschwindigkeitsgewinn, da das jeweils letzte und vorletzte Element der Folge nicht neu berechnet werden muss [52].

Eine solche Optimierung ist auch für LMMC interessant, da sie als rein funktionale Sprache nicht die Möglichkeit bietet, Rückgabewerte von Methoden zwischen zu speichern.

Voraussetzung ist, dass es sich um eine referenziell transparente Funktion handelt. Das bedeutet, dass der Rückgabewert der Funktion ausschließlich von den Aktualparametern abhängt [53]. Für Methoden in imperativen objektorientierten Sprachen ist das Verfahren daher generell nicht anwendbar, da diese auf den Objektzustand zugreifen können und sich dieser zwischen zwei Aufrufen verändern kann.

Für LMMC hingegen ist das Verfahren auf alle Methoden anwendbar, da folgende Annahmen gemacht werden können:

- Der Zustand des Modells ändert sich während eines Validierungszyklus (einer Interpretation) nicht (vgl. Abschnitt 3.3).
- Methoden lassen sich nur im Kontext von Modellelementen – also an unveränderlichen Objekten – definieren.
- Mit der Sprache lassen sich keine Seiteneffekte hervorrufen.

Diese Annahmen gewährleisten, dass Methoden in LMMC referenziell transparent sind und damit lässt sich die Memoisierung für jede Methode transparent anwenden.

4.6.4 Constraint-Abhängigkeiten

In LMMC ist es möglich mit `requires` Abhängigkeiten zwischen Constraints herzustellen. Ein Constraint wird erst überprüft (ausgewertet), wenn alle Constraints von denen er abhängig ist, erfüllt sind oder er von keinem Constraint abhängig ist. Die EVL unterscheidet mit `satisfiesOne` und `satisfiesAll` ob nur ein abhängiger Constraint erfüllt sein muss oder alle [17]. Diese

Konstrukte sind in LMMC nicht notwendig, da sie sich durch entsprechende Bool'sche Ausdrücke innerhalb der Constraints nachbilden lassen.

Die Abhängigkeiten entscheiden also über die Auswertungsreihenfolge der Constraints. Während der Interpretation lässt sich zu jedem Zeitpunkt mit einem Abhängigkeitsgraphen ermitteln, welche Invarianten als nächstes überprüft werden können und welche nicht mehr ausgewertet werden müssen. Abbildung 27 zeigt den zeitlichen Verlauf der Auswertung von sieben Invarianten unter denen Abhängigkeiten bestehen.

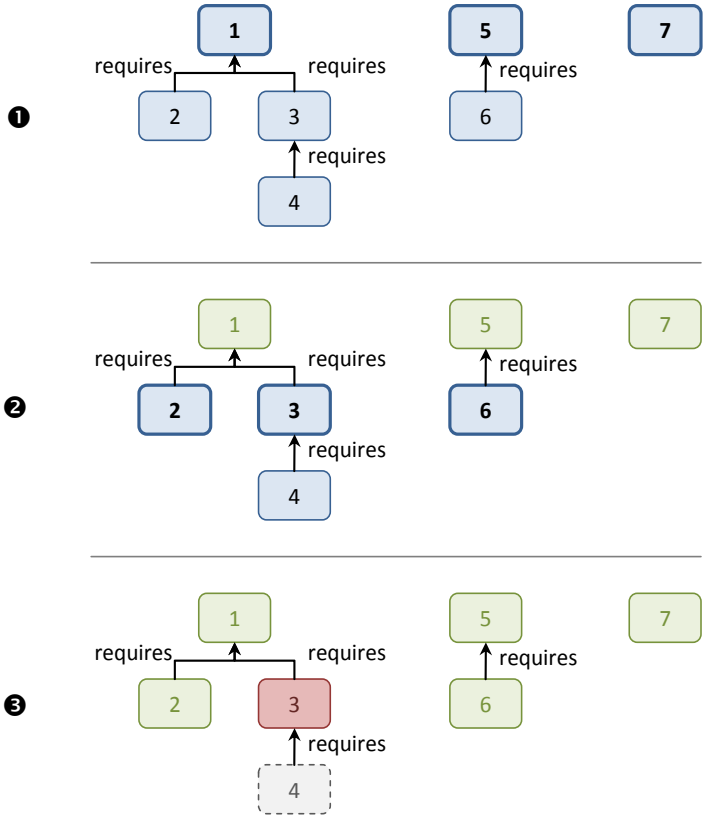


Abbildung 27: Verlauf der Auswertung von abhängigen Invarianten

Zum Zeitpunkt ❶ ist noch keine Invariante ausgewertet. Da nur **1**, **5** und **7** unabhängig sind, können sie nun parallel überprüft werden. Zum Zeitpunkt ❷ steht fest, dass alle drei Invarianten gelten. Damit können nun **2**, **3** und **6** parallel ausgewertet werden. In ❸ wurden die Invarianten ausgewertet und da Invariante **3** verletzt ist, müssen keine der von ihr abhängigen verbleiben Invarianten ausgewertet werden.

Die Auswertung lässt sich per *Multi-Threading* parallelisieren. Es stehen nun aber potenziell sehr viele Invarianten zur gleichen Zeit zur Auswertung bereit und das Starten von Threads ist mit gewissen Kosten verbunden. Daher stellt der LMMC-Interpreter eine feste Zahl an Threads – einen sogenannten *Thread-Pool* – zur Auswertung von Constraints zur Verfügung. Die Größe dieses Pools wird für ein *N*-Prozessor-System im Allgemeinen mit *N* + 1 Threads bemessen [54].

4.7 Integration

Auch die Entwicklungsumgebung für LMM-Modelle basiert auf dem Xtext-Framework [8]. Die LMMC-Constraints sollen überprüft werden, während das LMM-Modell editiert wird. Es bietet sich also an, den LMMC-Interpreter direkt in der Java-basierten Validierung von LMM-Ressourcen aufzurufen. Die Constraints werden dann nebenläufig während des Editiervorgangs, beim Öffnen einer LMM-Datei und beim Speichern überprüft und die resultierenden Fehler und Warnungen können an das Validierungs-Framework von Xtext übergeben werden (vgl. Abbildung 28).

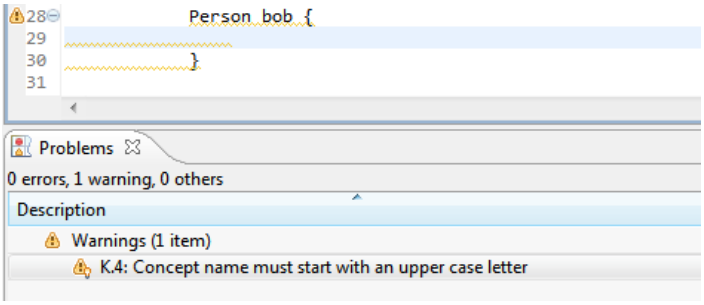


Abbildung 28: Integration in den LMM-Editor: Verletzte Invarianten werden als Warnungen und Fehler angezeigt

5 Anwendungsbeispiele

Anhand von nicht-trivialen Beispielen soll im Folgenden veranschaulicht werden, wie die Constraint-Sprache LMMC eingesetzt werden kann. Dazu wird zunächst ein mit dem LMM dargestelltes Metamodell durch Invarianten verfeinert. Im zweiten Teil wird ein kleiner Ausschnitt aus einer einfachen Semantik für das LMM gezeigt.

5.1 Geschäftsregeln auf LMM-basierten Modellen: Entity-Relationship-Diagramm

Zum Entwurf von Datenbanken werden sogenannte *Entity-Relationship-Diagramme* verwendet, in denen sich das Geschäftsmodell aus Entitäten (engl. *entity*) und Beziehungen (engl. *relationship*) zwischen ihnen zusammensetzt. Die Hierarchie aus einem Metamodell für ER-Diagramme, konkreten ER-Diagrammen und deren Instanzen bildet einen typischen Anwendungsfall für die Metamodellierung mit dem LMM. Um die Präzisierung einer Metamodellhierarchie durch inhaltliche Constraints zu demonstrieren wurde in LMM das Metamodell des ER-Diagramms erstellt. Ein ER-Diagramm beschreibt demnach ein Schema, das Entitäten enthält. Eine Entität verfügt über Attribute, die einen Typ besitzen, Teil des Primärschlüssels sein können (*isPrimaryKeyColumn*), optional sein können (*isNullable*) und einen Wert haben (*value*). Entitäten lassen sich durch Beziehungen einer bestimmten Kardinalität miteinander verknüpfen.

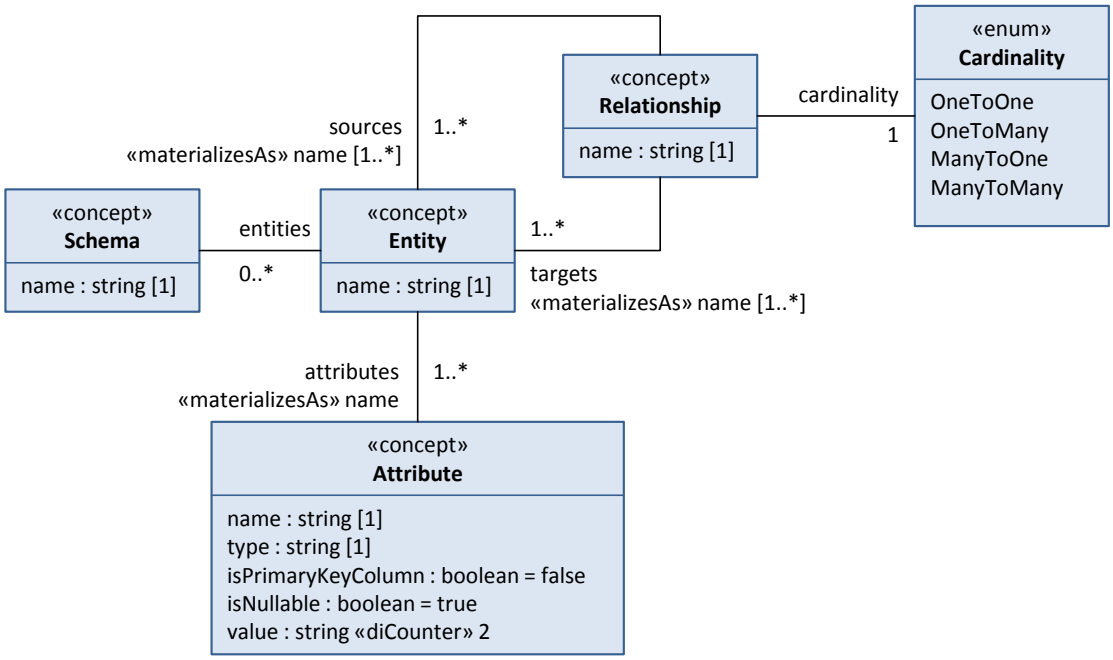


Abbildung 29: Metamodell eines Entity-Relationship-Diagramms

Das Metamodell enthält zwei besondere Modellierungskonstrukte, die sich z. B. in der UML nicht ausdrücken lassen. Eines von ihnen ist das Konzept der *Materialisierung* [55]. So wird z. B. die Beziehung *attributes* zwischen *Entity* und *Attribute* über die Eigenschaft *name* von *Attribute* „materialisiert“. Wenn nun also eine Instanz von *Entity* eine Instanz von *Attribute* referenziert, dann deklariert diese Instanz implizit eine neue Beziehung mit dem Namen des Attributs. Das zweite besondere Konstrukt in diesem Metamodell ist die *Deep Instantiation* [56]. Die Eigenschaft *value* von *Attribute* wurde mit einem Zähler versehen, der mit 2 initialisiert wird. Bei jeder Instanziierung des Konzepts wird dieser Zähler herabgesetzt. Erst, wenn der Zähler den Wert 0 erreicht hat, muss *value* mit einem Wert belegt werden.

Abbildung 30 zeigt eine mögliche Instanziierung des ER-Diagramms. Das Metamodell befindet sich auf Metaebene M2. Da *Customer* auf Ebene M1 das Attribut *CustomerName* mit dem Namen „customerName“ referenziert, kann *Alice* auf Ebene M0 die neue Beziehung *customerName* mit *AliceName* eingehen. Der Wert von *value* muss sinnvollerweise erst auf Ebene M0 gesetzt werden.

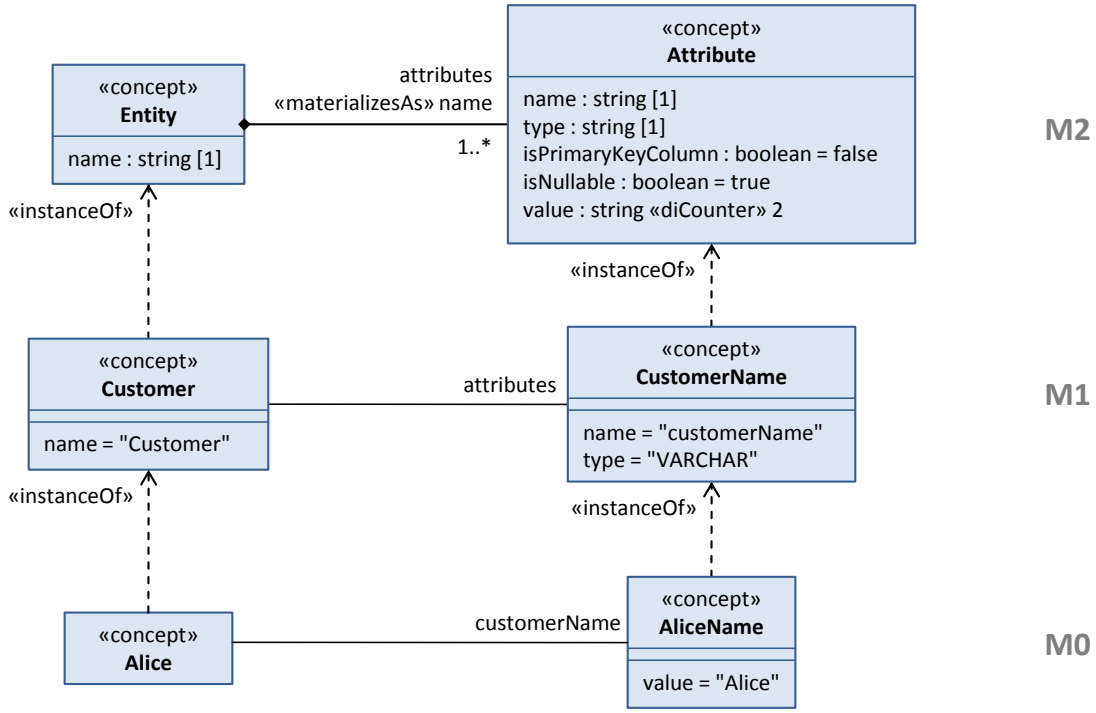


Abbildung 30: Mögliche Instanziierung des ER-Diagramms[4]

Die ER-Modellierung verbietet nun, dass ein Attribut Teil des Primärschlüssels ist und zugleich NULL-Werte enthalten darf. Diese Einschränkung lässt sich mit dem Metamodell nicht durchsetzen, da die Attribute *isPrimaryKeyColumn* und *isNullable* unabhängig voneinander gesetzt werden können. Sie lässt sich wenn überhaupt dann durch ein wesentlich komplexeres Metamodell erreichen.

Durch eine LMMC-Invariante lässt sich diese Regel nun leicht formulieren: wenn ein Attribut Teil des Primärschlüssels ist, dann darf es keine NULL-Werte enthalten:

```
on ER.ER2.ER.Attribute {
  rule ensures isPrimaryKeyColumn implies not isNullable
    causes "A primary key column must not be nullable"
}
```

Eine weitere Einschränkung ist, dass eine Entität nicht über zwei Attribute mit demselben Namen verfügen darf. Die Namen der Attribute lassen sich im Metamodell ohne Einschränkung vergeben, so dass auch diese Randbedingung nicht eingehalten werden kann. Mit den Mitteln des LMM allein ist es nicht möglich eine solche Bedingung zu formulieren.

[4] Zur Darstellung von LMM-Modellen wird eine an die UML angelehnte Pseudo-Notation verwendet. Konzepte werden wie Klassen dargestellt. Das Kompartiment, welches in der UML die Operationen enthält, beinhaltet hier die Wertzuweisungen eines Konzepts.

Folgender Constraint stellt sicher, dass eine Entität keine zwei Attribute mit demselben Namen enthält:

```
on ER.ER2.ER.Entity {
  rule ensures
    not attributes.exists(Attribute a, Attribute b | a.name = b.name)
    causes "There must not be two attributes with the same name in one "
      + "entity"
}
```

5.2 Austauschbare Modellierungsparadigmen: Semantik des LMM

Wie bereits erwähnt soll die Constraint-Sprache LMMC auch dazu dienen, dass LMM selbst einzuschränken, also ein Modellierungsparadigma vorzugeben. Ein solches Paradigma setzt sich aus einer Sammlung von Konventionen und Regeln zusammen. Für das Verständnis der folgenden Beispiele ist der in Abbildung 31 gezeigte Ausschnitt aus dem LMM relevant. Demnach gibt es im LMM Ebenen, die miteinander in Beziehung stehen können. Ebenen enthalten Pakete und diese enthalten wiederum Referenztypen. Ein möglicher Referenztyp ist das Konzept, das andere Konzepte referenzieren kann.

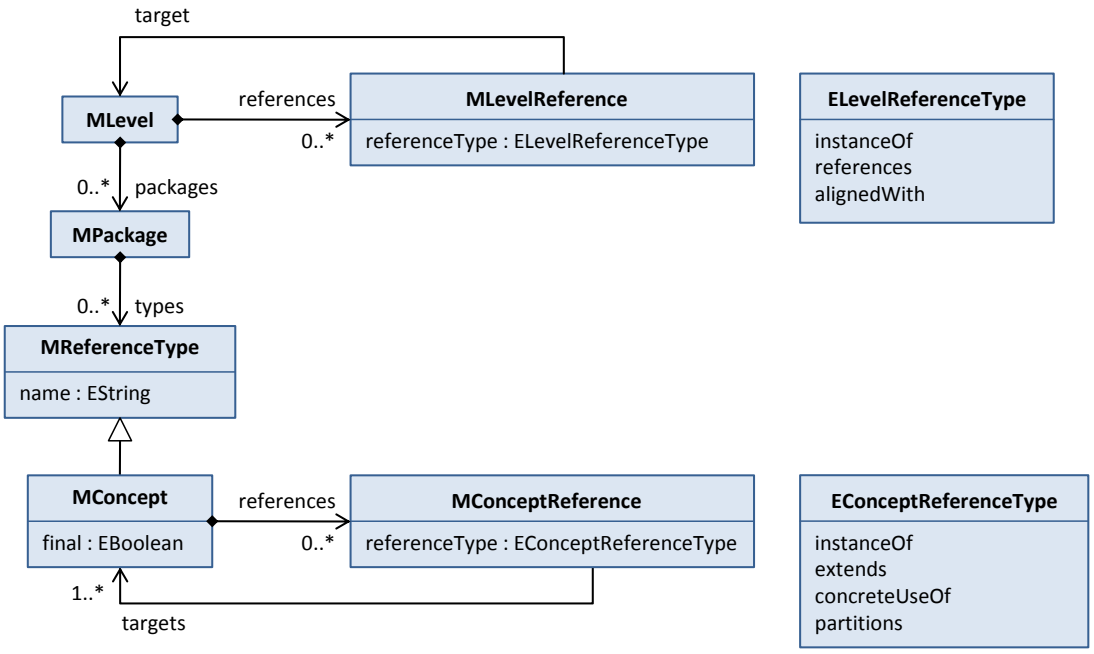


Abbildung 31: Für die Anwendungsbeispiele relevanter Ausschnitt des LMM

5.2.1 Einfache Invarianten

Eine sehr einfache Konvention ist dabei z. B., dass der Name eines Konzepts mit einem Großbuchstaben beginnen muss:

```
on MConcept {
  convention ensures name.firstCharacter().isUpperCase()
    causes "K.4: Concept name must start with an upper case letter"
}
```

Außerdem müssen die Beziehungen, die Konzepte miteinander eingehen können, beschränkt werden. Der folgende Ausschnitt enthält zwei Regeln für die Beziehung zwischen Konzepten. Die erste stellt sicher, dass, wenn es sich um eine Spezialisierungsbeziehung handelt, die Ziele dieser

Beziehung nicht als final markiert sein dürfen. Die zweite Regel gibt vor, dass eine Instanziierungsbeziehung nur ein einziges Ziel haben darf, das heißt, dass ein Konzept nur jeweils ein anderes instanziieren darf.

```
on MConceptReference {
  rule ensures referenceType = EConceptReferenceType.extends
    implies targets.forAll(MConcept co | not co.final)
    causes "ER.4: Concept "
      + targets.select(MConcept co | co.final).any().name
      + " is final and may not be extended"

  rule ensures referenceType = EConceptReferenceType.instanceOf
    implies targets.count() = 1
    causes "ER.8: A concept can only instantiate one other concept"
}
```

5.2.2 Komplexere Semantik: Sichtbare Attribute von Erweiterten Powertypen

Die Constraint-Sprache wird auch verwendet um die Semantik von fortgeschrittenen Modellierungsmustern wie dem *Erweiterten Powertypen* [57] zu definieren. Dieses Muster erlaubt es, die Eigenschaften eines Typs bei der Instanziierung ein- und auszuschalten.

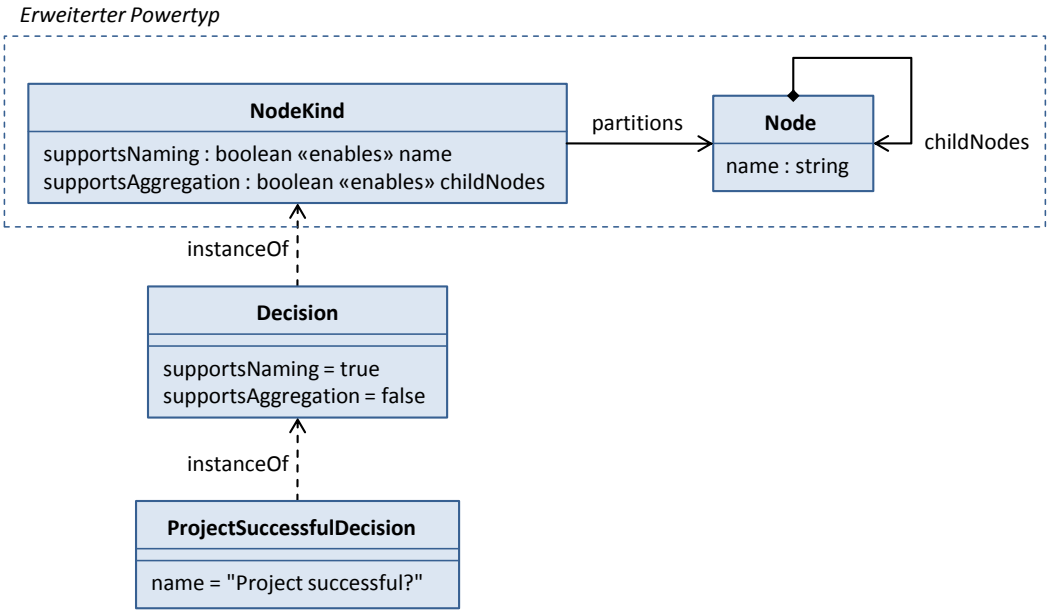


Abbildung 32: Beispielmodell für einen Erweiterten Powertypen

Abbildung 32 zeigt ein einfaches Beispiel für ein solches Konstrukt. Das Konzept *Node* bildet den „partitionierten Typ“, der zunächst alle Eigenschaften deklariert, also *name* und *childNodes*. *NodeKind* partitioniert diesen Typ und deklariert Attribute, die als „Schalter“ für die Eigenschaften von *Node* fungieren. *Decision* instanziiert *NodeKind* und setzt diese Schalter auf geeignete Werte. Eine Entscheidung hat demnach einen Namen, unterstützt aber keine Aggregation. Die drei Konzepte *Node*, *NodeKind* und *Decision* bilden zusammen einen Erweiterten Powertypen. In der Instanz *ProjectSuccessfulDecision* kann nun nur das Attribut *name* gesetzt werden, *childNodes* ist nicht verfügbar.

Im Rahmen der Semantik muss nun unter anderem bestimmt werden, welche Attribute von einem Typ effektiv deklariert werden, d. h. welchen Attributen in einer Instanz dieses Typs Werte

zugewiesen werden können. Dazu gehören auch Attribute, die durch die Instanziierung eines Erweiterten Powertypen verfügbar werden. Im Folgenden soll dazu die Berechnung der explizit aktivierten Attribute beschrieben werden. Um diese Berechnung nachvollziehbar zu gestalten, werden die einzelnen Schritte in Methoden ausgelagert.

Zunächst ermittelt `enabledAttributes` alle Attribute, für die ein Attribut als Schalter fungiert. Diese „enables“-Beziehung wird im LMM über eine *MPowertypeExtend*-Instanz hergestellt (vgl. Abbildung 33).

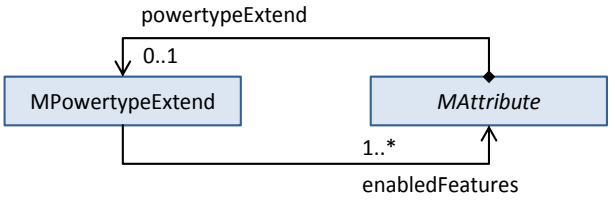


Abbildung 33: Ausschnitt aus dem LMM zur Modellierung einer *enables*-Beziehung zwischen Attributen

Wenn ein Attribut also über eine solche Struktur verfügt, liefert die Methode alle referenzierten Attribute. Wenn nicht, liefert sie eine leere Sammlung:

```
on MAttribute
  0..* MAttribute enabledAttributes() =
    if powertypeExtend = undefined then []
    else powertypeExtend.enabledFeatures
```

Außerdem soll sich zur Vereinfachung jede Wertzuweisung als eine zu einem Boole’schen Attribut behandeln lassen. `booleanValue` liefert dazu mit Hilfe der sicheren Typverengung (vgl. Abschnitt 3.5.3) für eine beliebige Zuweisung deren Wahrheitswert:

```
on MAttribute
  boolean booleanValue() =
    if self is MBooleanAssignment then self.literals.any()
    else false
```

Die Methode liefert im Beispiel für die Zuweisung zu *supportsNaming* und *supportsAggregation* in *Decision* die Werte `true` und `false`. Zunächst müssen nun die von einem Konzept partitionierten Typen ermittelt werden. Dazu werden für ein Konzept die Ziele aller Referenzen vom Typ *partitions* ermittelt:

```
on MConcept
  0..* MConcept partitionedTypes() =
    references.select(MConceptReference cr |
        cr.referenceType = EConceptReferenceType.partitions)
     .collect(MConceptReference cr | cr.targets)
```

Die Methode liefert im Beispiel für *NodeKind* den Wert `[Node]`. Für alle anderen Konzepte liefert sie die leere Sammlung `[]`. Mit Hilfe von `enabledAttributes` und `partitionedTypes` lässt sich nun bestimmen, ob es sich bei einem Konzept um einen Erweiterten Powertypen handelt. Dazu muss überprüft werden, ob es eine *partitions*-Beziehung eingeht und ob es Attribute eines partitionierten Typs aktiviert:

```
on MConcept
  boolean isExtendedPowertype() =
```

```
    references.exists(MConceptReference cr |
      cr.referenceType = EConceptReferenceType.partitions) and
    attributes
      .collect(MAttribute at | at.enabledAttributes())
      .exists(MAttribute at | at.concept in partitionedTypes())
```

Im Beispiel liefert sie nur für *NodeKind* `true`, für alle anderen `false`. Ausgehend davon lässt sich bestimmen, welche Erweiterten Powertypen ein Konzept instanziiert:

```
on MConcept
  0..* MConcept instantiatedExtendedPowertypes() =
    references.select(MConceptReference cr |
      cr.referenceType = EConceptReferenceType.instanceOf)
    .collect(MConceptReference cr | cr.targets)
          .select(MConcept co | co.isExtendedPowertype())
```

Die Methode `instantiatedExtendedPowertypes` liefert im Beispiel für *Decision* den Wert `[NodeKind]`; für alle anderen Konzepte liefert sie `[]`. Nun können alle „Schalter-Attribute“ berechnet werden, denen ein Boole‘scher Wert zugewiesen kann und mit denen sich die Eigenschaften des partitionierten Typs aktivieren bzw. deaktivieren lassen. Die Methode `enablingAttributes` liefert diese Attribute, indem sie in den Erweiterten Powertypen, die ein Konzept instanziiert, die Attribute sammelt, die als Schalter für mindestens ein Attribut fungieren:

```
on MConcept
  0..* MAttribute enablingAttributes() =
    instantiatedExtendedPowertypes()
    .collect(MConcept co | co.attributes)
    .select(MAttribute at | at.enabledAttributes().count() > 0)
```

Sie liefert im Beispiel für das Konzept *Decision* die Attribute `supportsNaming` und `supportsAggregation`. Werden z. B. in *Decision* diesen Attributen nun Werte zugewiesen, lassen sich diese Zuweisungen mit `enablingAssignments` ermitteln:

```
on MConcept
  0..* MAssignment enablingAssignments() =
    assignments
    .select(MAssignment as | as.attribute in enablingAttributes())
```

Im Beispiel bildet *Decision* den Typ für *ProjectSuccessfulDecision*. Welche Attribute nun aus dem Typ als Instanz eines Erweiterten Powertyps schließlich hervorgehen, berechnet `activeAttributes`. Es werden dazu alle Zuweisungen an Schalter-Attribute ausgewählt, in denen `true` zugewiesen wird und zu diesen wiederum die Attribute ermittelt, die dadurch aktiviert werden:

```
on MConcept
  0..* MAttribute activeAttributes() =
    enablingAssignments
    .select(MAssignment as | as.booleanValue() = true)
    .collect(MAssignment as | as.attribute)
    .collect(MAttribute at | at.enabledAttributes())
```

Diese Methode liefert für *Decision* demnach das Attribut *name* aber nicht *childNodes* und drückt damit aus, dass eine Instanz von *Decision* nur dem Attribut *name* einen Wert zuweisen darf.

5.2.3 Ausschluss von zyklischen Referenzen

Die bisher gezeigten Invarianten lassen sich in ähnlicher Form auch in der OCL oder EVL formulieren (vgl. Abschnitte 2.1 und 2.2). Die folgende Regel demonstriert hingegen, wie sich auch komplexe strukturelle Einschränkungen auf prägnante Weise in LMMC formulieren lassen. Sie stellt sicher, dass die Referenzbeziehungen zwischen Metaebenen keine Zyklen bilden. Dazu liefert die Methode `referencedLevels` zunächst alle von einer Ebene direkt referenzierten Ebenen. Innerhalb der Regel wird die Methode dann mit dem „`.+`“-Operator transitiv aufgerufen, das heißt es werden alle direkt und indirekt referenzierten Ebenen geliefert. Ist die betrachtete Ebene Teil eines Zyklus, dann ist sie in dieser Menge enthalten (vgl. Abschnitt 3.7.2). Dieser Fall muss dann entsprechend ausgeschlossen werden:

```
on MLevel {
  0..* MLevel referencedLevels() =
    references.collect(MLevelReference lr | lr.target)

  rule ensures not (self in self.+referencedLevels())
    causes "LO.1: Level " + name + " is part of a cycle"
}
```

Angenommen, das LMM wäre in einen OCL-Interpreter integriert, dann ist folgende Formulierung notwendig um das Ergebnis dieses Constraints in OCL zu erreichen:

```
context MLevel
  def: referencedLevels : Set(MLevel) =
    self.references->collect(
      MLevelReferences r : MLevelReference | r.target)->asSet()

  def: allReferencedLevels(l : Set(MLevel)) : Set(MLevel) =
    if l->includesAll(self.referencedLevels()) then l
    else allReferencedLevels(l->union(self.referencedLevels()))
    endif

  inv: not allReferencedLevels(Set{})->includes(self)
```

Die Operation `allReferencedLevels` ermittelt rekursiv die Hülle über `referencedLevels` und enthält ein entsprechendes Abbruchkriterium. Die Formulierung in der OCL bringt offensichtlich einige entscheidende Nachteile mit sich:

- Die Berechnung der transitiven Hülle muss mit den Mitteln der Sprache selbst implementiert werden. Dabei ist nicht sofort ersichtlich ob die Implementierung korrekt ist und zum gewünschten Ergebnis führt. Außerdem ist nicht sofort ersichtlich, dass überhaupt eine transitive Hülle über den referenzierten Ebenen berechnet wird.
- Statt wie im Modell mit `0..* MLevel` muss die Beziehung zwischen Ebenen jetzt mit `Set(MLevel)` typisiert werden.
- Es lässt sich keine aussagekräftige Meldung mit der Invariante verknüpfen und es ist nicht klar ob ihre Verletzung zu einem Fehler oder nur zu einer Warnung führen soll.
- Für den Aufruf von Eigenschaften (Operationen und Attribute) an einwertigen Objekten wird einer anderer Operator (`.`) als für den Aufruf an mehrwertigen Objekten (`->`) verwendet. Dies führt zu einer unnötig komplexen Syntax.

6 Bewertung und zukünftige Arbeitsfelder

Mit LMMC wurde eine Sprache entwickelt, die es erlaubt Randbedingungen für Modelle zu formulieren. Sowohl Anforderung 1 als auch 2 wird erfüllt, da sich Constraints sowohl auf der Modellierungssprache LMM als auch auf dem Inhalt einer Metamodellhierarchie formulieren lassen. Constraints lassen sich mit einer kontextbezogenen Meldung versehen und in zwei Schweregrade einteilen. Durch die nahtlose Integration in die Metamodellierungsumgebung OMME kann dadurch nachvollzogen werden, welcher Teil eines Modells aus welchem Grund gegen welchen Constraint verstößt. Der Entwurf wird durch eine vollwertige Entwicklungsumgebung unterstützt, die eine Code-Vervollständigung, Validierung und viele weitere Funktionen aufweist. Damit wird auch Anforderung 3 erfüllt. Die Sprache erlaubt eine deklarative Formulierung von Constraints und abstrahiert von Listenverarbeitung. Sie weist außerdem ein Typsystem auf, das sich auf derselben Abstraktionsebene befindet wie die Modellierungssprache und erfüllt dadurch Anforderung 4. Die Constraints werden von einem Interpreter direkt auf dem Modell ausgewertet. Ein Generierungsprozess ist nicht notwendig, wodurch schließlich auch Anforderung 5 erfüllt wird.

6.1 Optimierung

Im beschriebenen Prototyp wurde als Optimierung ein Zwischenspeichern von Methodenergebnissen (Memoisierung) implementiert. Für reine funktionale Sprachen sind darüber hinaus weitere Optimierungstechniken bekannt. Dazu gehört die nicht-strikte Auswertung von Ausdrücken. Abbildung 34 zeigt das Modell des Ausdrucks 5*4+5*4. Bei einer gewöhnlichen Auswertung wird die Multiplikation zweimal ausgeführt obwohl ihr Ergebnis bereits bekannt ist. Die Multiplikation könnte auch eine sehr teure Operation sein, die an jeder Stelle erneut ausgeführt wird, an der sie auftaucht. Eine Möglichkeit solche wiederholten Auswertungen zu vermeiden ist die Reduktion des Ausdrucks und die gemeinsame Nutzung des Ergebnisses. Solche Verfahren werden allgemein als *Graphreduktion* bezeichnet [28].

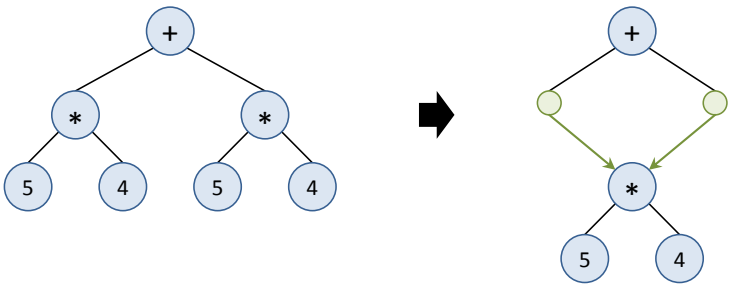


Abbildung 34: Beispiel zur nicht-strikten Auswertung von Ausdrücken

6.2 Debugging

Durch eine Ausführung in Teilschritten, das Setzen von Haltepunkten (engl. *breakpoints*), das Inspizieren von Variablen und das Anhalten und Fortfahren der Ausführung lassen sich Programmfehler aufspüren. Diese Mittel werden als *Debugging* bezeichnet und z. B. durch die *Java Development Tools* für Java-Programme unterstützt [58]. Durch die Integration eines Debuggers für LMMC ließen sich fehlerhafte Constraints leicht eingrenzen, was die Entwicklung insgesamt vereinfachen würde.

6.3 Freie Variablen

Variablen treten im vorgestellten Entwurf von LMMC ausschließlich als Parameter von Methoden oder anonymen Funktionen auf. Durch den Aufruf der Funktion wird die Variable gebunden, das heißt jedem Formalparameter wird ein Aktualparameter zugewiesen.

Freie Variablen hingegen werden nicht explizit gebunden sondern ihr Wert ergibt sich erst in Folge von Einschränkungen. Das folgende Beispiel zeigt eine Invariante wie sie sich in der aktuellen Version der Sprache nicht formulieren lässt:

```
rule ensures self.extendedClasses.contains(e) implies not e.final
```

Wenn also eine Klasse *e* zu den erweiterten Klassen zählt, dann darf sie nicht als final markiert sein. Die Variable ist in diesem Fall nicht gebunden. Um festzustellen ob der Ausdruck gilt, müssen Methoden aus dem Bereich der Constraint-Lösung angewandt werden [59, 60]. Mit Hilfe von freien Variablen ließen sich solche Muster sehr komfortabel formulieren.

6.4 Verbindung von Modellierung und Randbedingungen

LMMC wurde so entworfen, dass sich die Sprache syntaktisch gut an die LMM-Syntax angleicht und für den Benutzer kein Bruch zwischen Modellierung und Constraints entsteht. Die Trennung in Modellierungs- und Constraint-Sprache wie sie bei der UML und der OCL besteht wurde aber für LMM und LMMC prinzipiell beibehalten. Die Modellierungssprache LMM beschreibt zunächst eine strukturelle Grundlage:

```
concept Person {
    attributes {
        concept Person { multiplicity = oneOrMore }
    }
}
```

Durch die Constraint-Sprache LMMC wird diese Grundlage dann weiter präzisiert:

```
on Person {
   rule ensures not self.parents.includes(self)
}
```

Wenn man die Typisierung des Attributs *person* wie in der Sprache Alloy [22] als Mengenangabe versteht, könnte man die Invariante in das Modell integrieren und ein bestimmtes Element aus der Menge ausschließen. In einer hypothetischen Syntax könnte man dies wie folgt formulieren:

```
concept Person {
   parents : 1..* Person \ self;
}
```

Es bleibt zu untersuchen inwieweit sich auch komplexe Constraints in die Modellierungssprache integrieren lassen und ob eventuell vollständig auf eine separate Sprache verzichtet werden kann.

6.5 Erschließen von Reparaturen

Wie in Abschnitt 2.2 gezeigt, bietet die EVL die Möglichkeit für eine Invariante eine Reihe von Reparaturen (engl. *fixes*) zu definieren. Bei der Verletzung eines Constraints werden diese angeboten und verändern das Modell so, dass es den Constraint anschließend erfüllt.

Natürlich kann nicht garantiert werden, dass das Modell den Constraint anschließend tatsächlich erfüllt. Die Reparatur ist lediglich ein Block von Anweisungen, der auf dem Modell ausgeführt wird und in keinem echten inhaltlichen Zusammenhang mit dem Constraint steht.

Zu untersuchen ist, ob ein Zusammenhang zwischen Constraint und Reparatur hergestellt werden und eventuell sogar eine Abbildung gefunden werden kann. Mit Hilfe dieser Abbildung ließen sich dann Reparaturen aus Constraints erschließen.

LITERATUR

[1] Kolovos, D., Rose, L.Paige, R.: *The Epsilon Book.* http://www.eclipse.org/gmt/epsilon/doc/book/ (geprüft am 10.11.2010)

[2] Spivey, M.: *The Z Notation.* A reference manual. Prentice Hall (1992)

[3] Selic, B.: *The Pragmatics of Model-Driven Development.* IEEE Software 20, 19–25 (2003)

[4] Douglass, J.: *Language of Languages for Flexible Development.* ICSE 2010 Workshop on Flexible Modeling Tools (2010)

[5] *Rational Software Architect for WebSphere Software.* http://www-01.ibm.com/software/awdtools/swarchitect/websphere/ (geprüft am 09.02.2011)

[6] Object Management Group: *OMG Unified Modeling Language (OMG UML) Superstructure.* http://www.omg.org/spec/UML/2.3/ (2010)

[7] Fuentes-Fernández, L., Vallecillo-Moreno, A.: *An Introduction to UML Profiles.* UPGRADE - The European Journal for the Informatics Professional 5, 6–13 (2004)

[8] Volz, B., Jablonski, S.: *OMME - A Flexible Modeling Environment.* ICSE 2010 Workshop on Flexible Modeling Tools (2010)

[9] Behrens, H., Clay, M., Efftinge, S., Eysholdt, M., Friese, P., Köhnlein, J., Wannheden, K.Zarnekow, S.: *Xtext User Guide 1.0.1.* http://www.eclipse.org/Xtext/documentation/1_0_1/xtext.pdf (geprüft am 05.01.2011)

[10] Atkinson, C., Kühne, T.: *Concepts for Comparing Modeling Tool Architectures.* Model Driven Engineering Languages and Systems 3713, 398–413 (2003)

[11] Atkinson, C.: *Meta-Modeling for Distributed Object Environments. In:* 1st International Enterprise Distributed Object Computing Conference (EDOC '97), 90–101

[12] *Eclipse Modeling Framework Project (EMF).* http://www.eclipse.org/modeling/emf/ (geprüft am 02.12.2010)

[13] Object Management Group: *Object Constraint Language.* http://www.omg.org/spec/OCL/2.0/ (2006)

[14] Object Management Group: *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification.* http://www.omg.org/spec/QVT/1.0/PDF/ (2008)

[15] Object Management Group: *Production Rule Representation (PRR).* http://www.omg.org/spec/PRR/1.0/PDF (2009)

[16] *Eclipse Documentation - OCL Developer Guide - Creating Metamodel Bindings.* http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.ocl.doc/references/overview/advanced/metamodelBindings.html (geprüft am 02.12.2010)

[17] Kolovos, D., Paige, R., Polack, F.: *On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In:* Rigorous Methods for Software Construction and Analysis, 204–218 (2009)

[18] Vaziri, M., Jackson, D.: *Some Shortcomings of OCL, the Object Constraint Language of UML. In:* Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00) (2000)

[19] Demuth, B., Wilke, C.: *Model and Object Verification by Using Dresden OCL. In:* Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice (2009)

[20] de Lara, J., Guerra, E.: *Deep Meta-Modelling with MetaDepth. In:* Proceedings of the 48th International Conference on Objects, Models, Components, Patterns, 1–20 (2010)

[21] *Xpand Documentation - Xpand / Xtend / Check Reference.* http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.xpand.doc/help/Check_language.html (geprüft am 21.11.2010)

[22] Jackson, D.: *Alloy: A Lightweight Object Modelling Notation.* ACM Transactions on Software Engineering and Methodology (TOSEM) 11 (2002)

[23] White, J.: *GEMS EMF Intelligence Prolog Plug-in Fact Format.* http://wiki.eclipse.org/GEMS_EMF_Intelligence_Prolog_Plug-in_Fact_Format (geprüft am 04.02.2011)

[24] Störrle, H.: *A PROLOG-based Approach to Representing and Querying Software Engineering Models. In:* Proceedings of Visual Languages and Logic 2007 (VLL 07), 71–83

[25] Volz, B.: *A Meta Model for Representing Arbitrary Meta Model Hierarchies.* Symposium on Applied Computing 2010 (SAC 2010) (2010)

[26] Atkinson, C., Kühne, T.: *Meta-level Independant Modeling. In:* International Workshop on Model Engineering (2000)

[27] Fowler, M.Parsons, R.: *Domain-Specific Languages.* Addison-Wesley (2011)

[28] Hudak, P.: *Conception, Evolution and Application of Functional Programming Languages.* ACM Computing Surveys (CSUR) 21, 359–411 (1989)

[29] Armstrong, J.: *History of Erlang.* Proceedings of the third ACM SIGPLAN conference on History of programming languages (2007)

[30] TIOBE Software BV: *TIOBE Programming Community Index for January 2011.* http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (geprüft am 07.02.2011)

[31] Jones, S. P., Blackwell, A., Burnett, M.: *A User-Centred Approach to Functions in Excel.* ACM SIGPLAN Notices (2003)

[32] Chamberlin, D., Boyce, R.: *SEQUEL: A Structured Englisch Query Language.* ACM SIGFIDET (1974)

[33] Novatchev, D.: *The Functional Programming Language XSLT-A proof through examples.* http://alamos.math.arizona.edu/~rychlik/CourseDir/589/Assignments/a3/fp.pdf (geprüft am 18.01.2011)

[34] Goetz, B.: *JSR 335: Lambda Expressions for the Java(TM) Programming Language.* http://jcp.org/en/jsr/detail?id=335 (geprüft am 05.01.2011)

[35] Liberty, J., Xie, D.: *Programming C# 3.0.* O'Reilly (2008)

[36] Meijer, E., Drayton, P.: *Static Typing Where Possible, Dynamic Typing When Needed.* The End of the Cold War Between Programming Languages. In: Proceeding of OOPSLA Workshop On The Revival Of Dynamic Languages (2004)

[37] Goldberg, A., Robson, D.: *Smalltalk-80.* The language and its implementation. Longman Higher Education (1983)

[38] *Python Programming Language.* Official Website. http://www.python.org/ (geprüft am 07.02.2011)

[39] Volz, B., Jablonski, S.: *Towards an Open Meta Modeling Environment. In:* Proceedings of the 10th Workshop on Domain-Specific Modeling (DSM'10) (2010)

[40] Gosling, J., Joy, B., Steele, G., Bracha, G.: *Java Language Specification.* Addison Wesley (2005)

[41] Goetz, B.: *State of the Lambda.* http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-3.html (geprüft am 05.01.2011)

[42] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R.Grose, T.: *Eclipse Modeling Framework: A Developer's Guide.* Addison-Wesley Professional (2003)

[43] Object Management Group: *Meta Object Facility (MOF) Core Specification.* http://www.omg.org/spec/MOF/2.0/ (2001)

[44] Wikipedia contributors: *Concept programming.* http://en.wikipedia.org/w/index.php?title=Concept_programming&oldid=389798199 (geprüft am 09.02.2011)

[45] Parr, T.: *Language Implementation Patterns.* The Pragmatic Bookshelf (2010)

[46] Parr, T.: *The Definitive ANTLR Reference Guide.* Building Domain-specific Languages. Pragmatic Programmers (2007)

[47] Wikipedia contributors: *Filter (higher-order function).* http://en.wikipedia.org/w/index.php?title=Filter_(higher-order_function)&oldid=389387916 (geprüft am 23.02.2011)

[48] Wikipedia contributors: *Map (higher-order function).* http://en.wikipedia.org/w/index.php?title=Map_(higher-order_function)&oldid=411539878 (geprüft am 23.02.2011)

[49] Gamma, E., Helm, R.Johnson, R.: *Design Patterns.* Elements of Reusable Object-Oriented Software. Addison-Wesley Longman (1994)

[50] Clayberg, E., Rubel, D.: *Eclipse.* Building Commercial-Quality Plug-ins. Addison-Wesley (2004)

[51] Völter, M.: *Xtext/TS.* A typesystem framework for Xtext. http://code.google.com/a/eclipselabs.org/p/xtext-typesystem/ (geprüft am 21.01.2011)

[52] Michie, D.: *"Memo" Functions and Machine Learning.* Nature 218, 19–22 (1968)

[53] Søndergaard, H., Sestoft, P.: *Referential Transparency, Definiteness and Unfoldability.* Acta Informatica 27, 505–517 (1990)

[54] Goetz, B.: *Java theory and practice: Thread pools and work queues.* http://www.ibm.com/developerworks/library/j-jtp0730.html (geprüft am 02.02.2011)

[55] Dahchour, M., Pirotte, A., Zimányi, E.: *Materialization and its Metaclass Implementation.* IEEE Transactions on Knowledge and Data Engineering 14, 1078–1094 (2002)

[56] Atkinson, C., Kühne, T.: *The Essence of Multilevel Metamodeling. In:* Proceeding of the 4th International Conferences on the Unified Modeling Language, Modeling Languages, Concepts and Tools, 2185, 19–33 (2001)

[57] Henderson-Sellers, B., Gonzalez-Perez, C.: *A powertype-based metamodeling framework.* Software and Systems Modeling 5, 72–90 (2006)

[58] Aniszczyk, C.Leszek, P.: *Debugging with the Eclipse Platform.* http://www.ibm.com/developerworks/library/os-ecbug/ (geprüft am 21.02.2011)

[59] Barták, R.: *Constraint Programming.* In Pursuit of the Holy Grail. In: Proceedings of WDS'99

[60] Hofstedt, P.: *The Multiparadigm Programming Language CCFL.* Deklarative Modellierung und effiziente Optimierung - dank Constraint-Technologie (MOC 2010) 2010